# Algorithms and Programs:

# An introduction with JavaScript

Colin Pace

MA, The University of Texas at Austin

Table of contents

# Chapter 1

# Introduction

## Who should read this book?

The book is written for people who want to learn algorithms and programs. An algorithm is the steps for a program that with its syntax, another word for grammar, enacts a computation on data. The result of a computation turns into either new data or output in the form of audio and visuals. A few examples of programs include the Apple operating systems, Microsoft Office applications, video games, the Amazon e-commerce website, and the Google Search web page.

I intend for the book to be a guide. It serves as an introduction for those who will study computer science at an institution of higher learning and for those who wish to teach themselves.

It is likely that the reader already has an intuitive sense of an algorithm. This is true even if the term is new to their idiolect. Parallel concepts in common registers include a recipe and a to-do list. An example of the former is a recipe for pasta. A person prepares the ingredients, cooks them, and then combines the result on dishes before serving the combination.

An example of the latter is a to-do list for the creation of a greenhouse. A person purchases the necessary land, equipment, and materials. Then there are the tasks of the construction of building, the organization of the areas, and the population of them with pottery, soil, and plants. In each of the examples, a singular project is analyzed into parts. Then the parts are connected temporally. First this is done. Then that is done.

The conception of an algorithm then can be elaborated. More technical language is introduced. Classifications of size and methods of combination are specified. For the examples of the recipe and

the to-do list, there is the introduction of quantity and quality. The recipe requires a cup of water and a teaspoon of salt. Once the water is boiling, then reduce the heat and add the pasta. Or for the second example, one might decide that the greenhouse needs forty square feet, a hundred pieces of lumber, and two hundred pieces of pottery. Once it is constructed, then the plants are to be placed inside of it. The additional conceptualization of measurement and combination ensures a smoothness of operation once the parts of the machine begin to move.

A program turns an algorithm into computer language. By its vocabulary and grammar, a program writes the operations for a computation. It also describes the course of events that occur during the execution of the operation. A program thereby complements an algorithm as its implementation. The parallel concept of the program in the examples above, is the culinary or constructive act whereby the recipe or greenhouse materialize.

An example of an algorithm and a program is colloquially called the Hello, World! program. The result of the program is an output on a computer screen of the text, Hello, World! . Below in figure 1, is the algorithm for Hello, World!.

Figure 1: An algorithm for Hello, World!

1. Declare the function keyword with the name helloWorld and an empty call signature.
2. Inside the declaration block, declare the const keyword to create a variable with the name str. By the equality operator, assign it the value of a string with the characters Hello, World! .
3. Below the first statement in the declaration block, declare another statement with the return keyword. The return value is the variable from the first line of the declaration block.
4. Below the declaration block, invoke the log method of the console object. Inside the log, invoke the function.

In figure 2, is the program for the algorithm.

Figure 2: A program for Hello, World!

```
1       function helloWorld( ) {
2               const str = “Hello, World!”;
3               return str;
4       }
5       console.log( helloWorld( ) );
```

If the reader has never encountered an algorithm or program before, then it is likely they are unable to comprehend the jargon of either. Despite this, the reader might glimpse the analytical similarities among the Hello, World! program and the examples of the recipe and the to-do list.

Each begins with a goal. For the former it is to display a message on the screen. For the recipe it is to prepare pasta. For the construction, it is to create a greenhouse. After the recognition of the initial requirement, each example then has an algorithm that describes the steps that achieve the goal.

After the algorithm, is the step of the composition of a program. In figure 2, there is a program which has syntax that tells the computer to make this or that computation, in what order, and the organization of the combination in its stages until the result. The other examples have the actions of cooking and building.

The course of the book introduces the reader to the syntax of JavaScript in the second chapter. The knowledge provides the reader with the ability to encode data structures that are learned in the third chapter. Because data structures are the components of algorithms, therefore in the fourth chapter, the reader learns to compose algorithms and programs with data structures in JavaScript. In the chapter, examples of algorithms are given and discussed. The sixth and final chapter is a conclusion. The

conclusion reviews the main points of the book. Then it directs the reader toward new directions for educational material. Please note also that the last sentence of the book has a link to the author's GitHub repository for the algorithms in the book and others. (A repository is an online folder at the Microsoft GitHub website that has files of programs.)

Throughout the book, there are references to other examples of the code that are given in the figures. Programs that contain those references, as well as working examples of the programs given in the book, can be found in the repository. The reader can copy and paste them from the repository into their code editor in order to run and update them. The only difficulty might be finding the appropriate file in the repository. A suggestion is to open the find option in the browser and search for key words that might narrow the search among files. Another suggestion is to look for files that have the suffix .js rather than .java or .cpp, which are for programs written in the Java and C++ programming languages.

**What are the benefits of learning the knowledge of algorithms and programs?**

Knowledge of algorithms and programs enables the development of software. If the reader desires to become a professional software engineer, a video game programmer, or a web developer for computer and mobile applications, the knowledge is elemental.

Consider the curriculum of the degree Computer Science and Engineering from The School of Engineering at The Massachusetts Institute of Technology. Its website lists the required courses of the departmental program. The first requirements are introduction to computer science and a course in mathematics, which probably focuses on algebra, geometry, and calculus. Then a student selects one of the following: robotics, communication networks, medical technology, or interconnected systems.

After these two courses, there are five required courses that form a common core for the students of the department: computation structures, introduction to algorithms, fundamentals of

programming, elements of software construction, and computer systems engineering. Next is a pair of options. Each option is a selection between two alternatives. The first choice is between artificial intelligence and machine learning. The second is between automation and algorithmic design and analysis. Last in the curriculum is series of electives. The electives are advanced undergraduate subjects in mathematics, the natural sciences, and computation.

While automation and artificial intelligence are beyond the technical scope of the material that is covered in this book, the topics of the five required courses in the degree are relevant. Since the information of the curriculum of computer science is often difficult to apprehend on the first encounter, the reader is well served by this book as a primer if a degree in computer science from a university is the course of their education.

At the time of this writing, the Unreal Engine is among the leaders of innovation in the development of video games. The images are clear and realistic. The website of the Unreal Engine company offers documentation for the development of a program with the technology. The Getting Started web page introduces the visitor to the syntax of the programming language C++. Then it focuses on the architecture of programs. The architecture includes modules, classes, functions, objects, strings, arrays, maps, sets, and other data structures. Like the algorithm of the Hello, World! program is implicit in the JavaScript implementation of figure 2 above, so the algorithms for the development of applications with the Unreal Engine are implicit in the programs that are offered in the tutorials of the web page. Again, an introduction to algorithms and programs serves as a stepping stone, here to video game development.

In addition to enterprise software and video games, web development is a burgeoning occupation. One of the highest ranked web development bootcamps in the United States is Flatiron School. The curriculum of the full stack web development program contains programming fundamentals, JavaScript, web frameworks, and front-end frameworks. For the reader who is a tyro

web developer, frameworks are patterns of a programming language. The patterns enable the transmission of data across the Internet and its subsequent storage or display in a web browser like Google Chrome. Flatiron School uses JavaScript in the front-end framework that it teaches, React. Its course on programming fundamentals certainly contains basic data structures.

Knowledge of algorithms and programs is useful in yet other contexts. As a discourse, the literature about algorithms and programs is both technically instructive and relevant for the world of business.

Algorithms and programs require an exactitude of precision and accuracy in logic that rivals mathematics and linguistics. The programmer moves between the ideal realm of algorithms, where they rotate and combine data structures abstractly, to that of programs where syntax delivers the real world implementations of the algorithms. Movement between the two is a technical requirement, as is the creation of a fast program bound by the strictures of syntax.

Whether the reader is an architect, a farmer, or a poet, a study of the fastidious discipline of computer science has the consequence of a greater perception of category and its modulation in other discourses. Consider the common algorithmic component, a statement of control flow. A control flow statement describes the actions to follow at a temporal point in a program.

An example of a control flow statement is the point in the algorithmic recipe when water boils and pasta is then added. The example has one layer of control flow. Imagine the addition of multiple layers of control flow. When the water boils, add salt. If there is no salt, add the pasta. When one minute has passed, stir the pasta. If the pasta becomes softer, stir the pasta. If the pasta is not yet softer, wait for the pasta to become softer.

Rarely does a recipe specify the minutia of each step. To do so would risk tedium. What if, however, the task was not a recipe but instead the organization of documents for taxes or the creation of a financial or market report for a company? The martinet composition of algorithms, becomes a useful

experience to draw logic from.

In addition to preparatory education and methodological instruction, knowledge of algorithms and programs has the benefit of relevance. Increasingly transaction occurs on the World Wide Web across the Internet. Questions about the reliability and ethics of online life proliferate. Less menacingly, the technology that provides the resources like search engines and audio-visual displays incorporate algorithms and programs into their actions.

An example is the Google Search web page. According to their explanation, Google continually sends web crawlers to read the sites and sitemaps of web pages. In doing so, the web crawlers collect information about the sites and also links to other sites. Then the web crawlers visit the other sites and continue the process. After some time, the web crawlers return the information to Google. There programmers write software that searches and organizes the information.

The consequence is that when a visitor to the Google Search web page enters a search request into the search bar, there is a list of results that have an order that prioritizes relevance. By studying the algorithms and programs in this book, the reader gains a basic understanding of the structures by which such search engines operate.

**What is the structure of the chapters?**

The structure of the chapters vary according to the topic of the chapter. The next chapter, about JavaScript, describes the parts of the language as one might describe the parts of the English language: nouns, adjectives, verbs, and so on. The bulk of the chapter is a technical exposition of the syntax. The exposition is intended for the reader who wishes to learn to program JavaScript. Readers already fluent in the language or who are interested instead in data structures or algorithms and programs, might skip this section. The next part of the chapter has examples that illustrate the technical exposition. At the

end of the chapter about JavaScript, there is a summary of JavaScript, and a direction of the reader to resources such as the w3schools and Mozilla Developer Network.

The third chapter of the book considers data structures. There is a range of data structures. A section of the range of data structures, tends to correspond with a type of program. An example of a data structure is a graph. It is often used to determine the distance of a route between two places. Also it is used to find the fastest route between two places. Web development companies that provide the product of a map with directions, often use graphs.

In another example, scientists working with statistical data about a topic like biomolecules, might spend more time working with dimensional arrays that better accommodate the structure of the collection of data, than they will with a graph. It is likely that both the web developer and the scientist are familiar with both types of data structures, even if their occupation tends to implement one more than the other.

The data structures included in the third chapter are those that are described by Google as common topics in their technical interview. After an introduction of each data structure, the chapter has a description of the modification of them. Modification takes place by access of the data that the structure contains. In order to gain access to the data, often it is necessary to iterate the structure to the point of the structure that contains the data. These concepts and others, such as collecting pieces of data during iteration and placing them in new structures, as well as the application of mathematical formulas to them to transform the data, are topics of the discussion of the principles of data structures.

The fourth chapter is about algorithms and programs. A series of examples is given. In each example there is a statement of its prompt, a demonstration of an algorithm that addresses the prompt, a demonstration of a program that provides an answer for the prompt, and a discussion of the asymptotic space and time complexity of the program.

An analysis of the asymptotic complexity of time and space is a metric description of the

efficiency of a program. An asymptote is a mathematical term for a curved line that approaches but never crosses a vertical line on a Cartesian coordinate grid. Different asymptotes are thereby associated with sections of the grid. This association allows programmers to identify time and space complexity by them.

Complexity is a term of conceptual jargon that translates into a ratio in common parlance. The ratio is the rate of increase in the unit of either time or space output, per unit of input. For example, if for each one unit of input, the output is always one, then the complexity is constant. One unit is added, the output is one unit. Two units are added, the output is one unit. Three units are added, the output is one unit. Four units are added, the output is one.

Another example of complexity is the ratio whereby each one unit of input has one corresponding unit of output. One unit of input is added, the output is one unit. Two units of input are added, the output is two units. Three units of input are added, the output is three units. Four units of input are added, the output is four units.

Yet another example of complexity is the ratio whereby each one unit of input has increasing units of output. A quadratic relation is an instance of this type of ratio. One unit of input is added, the output is one unit. Two units of input are added, the output is four units. Three units of input are added, the output is nine units. Four units of input are added, the output is sixteen units.

The measurement of asymptotic time and space complexity allows programmers to create programs that are fast even with big data. The processes of measurement and creation involve the recognition of the structure of data of input and the consideration of what data structures lend themselves to fast and economical storage and iteration.

The sixth and final chapter of the book offers a conclusion. Topics of consideration include a review of the course of the book, a discussion of further material for technical edification, and projects that the reader might pursue in their own programming.

After the conclusion, there are resources for book. There is a bibliography and list of figures.

**What technology is involved?**

Because the subjects of the book include both English descriptions of algorithms and JavaScript implementations of programs, therefore the technology that is involved includes a computer, a text editor, and the JavaScript programming language.

The author writes programs with a MacBook Air and the Visual Studio Code text editor. Another text editor available to download for free, is Sublime Text.

The macOS and Windows each have a JavaScript engine, often referred to as an interpreter. An interpreter is a program that takes the language of another program and translates it into machine code. Machine code is the language in which the computation of a program takes place. The result is then displayed for review. If the reader has a Mac or a computer that runs Windows, then they can run a JavaScript program in their text editor. Readers with other types of computers, can look for tutorials on YouTube and on web forums where there are instructions about set up.

**What is web development?**

Web development is the creation of applications on the Web. An application is any program or piece of software that performs an action. A web browser is an application, as are popular video games. If the video games are online, then they are web applications. Businesses often have web sites that customers can visit. There customers learn about the products that a business offers. Also common is the transaction of business through a website. A customer selects the products or services that they desire to purchase and then offer the website bank account information and a specified amount of

money, as a form of payment. Online banking is an application.

In its archetypal structure, web development involves two pieces that communicate with one another. Common metaphors of the two pieces are client and server and front end and back end. In the structure of a directory or folder, a client contains at least three types of files: Hypertext Markup Language, Cascading Styling Sheets, and JavaScript. Often Hypertext Markup Language is abbreviated as HTML, Cascading Styling Sheets as CSS, and JavaScript as JS.

The HTML files contain information about the structure of a web page. The bricks of the structure are called elements. There are dozens of types of element, each with a purpose. The paragraph element is for paragraphs of text. The hyperlink element is for links to other web pages. The image element is for the inclusion of an image in the web page.

Each element is denoted syntactically by a tag. The punctuation of a tag has carrot symbols < > and letters in between them. Often an element requires two tags. The two tags are identical except for one difference. The second tag has a forward slash, / , after the opening carrot and before the letters. The slash marks the tag as a closing tag.

The elements provide the structure of an HTML document and the CSS imparts style. Style includes location on the page, font, color, and size of text, images, and HTML elements, and also a limited range of interactive features such as action for the events of hover and focus, which occur when a mouse moves over an element or clicks on it.

Stored either in the HTML tag it describes, outside the tag but in the HTML file, or in a CSS file, the CSS of a web page has its own syntax in the form of selector and declaration block. The selector identifies the element to modify, and the declaration block describes the type of modification. An example is the HTML tag for a paragraph with the text, HelloWorld!, Hello, World! , and the CSS selector for the text of a paragraph, which is the singular letter p.

After the selector is a declaration block. The punctuation for the declaration block is curly

braces { }. Each line in the declaration block contains a colon and after the colon, a semi-colon. The description of the type of modification has two parts. The first is a property. The second is a value. The property precedes the colon, and the value follows the colon and proceeds the semi-colon. An extension of the above example of CSS, is the description of the font-size of the paragraph text.

Figure 3: A selector and declaration block of in CSS

```
1       p {
2               font-size: 12px;
3       }
```

In the example, the property is font-size and the value is 12 pixels. The letters px are an abbreviation for pixels.

If HMTL gives the structure of a web page and CSS the style, JavaScript gives the page its responsiveness to visitor interaction. If a visitor clicks on a button to initiate a response, such as to log into Facebook or to upload a picture there, JavaScript is the logic that changes the web page and responds to the actions of the visitor. An ancillary note about JavaScript, is its ubiquity in the World Wide Web. Most modern browsers use JavaScript, as do most web pages.

Together, HTML, CSS, and JavaScript are the basic building blocks of the client of a web application. They are the structure, style, and logic for interaction that a visitor experiences when they visit a web page.

In order to gain the information that it needs to display for a visitor, the client of a web application communicates with its server. A separate and centralized directory of files, the server stores and organizes the information of a website so that if a client of a visitor on a web browser, requests information, the sever then searches the database for the information, collects and organizes it, and

sends the information to the client for display.

In metaphor, the server is the complement of the client. In fact of implementation, however, the server includes a database. So perhaps less poetically but more accurately the relation of a web application is between a client and a server-database.

Servers are written in multiple languages. PHP and JavaScript are two common examples. Typically they receive a request and parse its data for the relevant information. It turns the information into search parameters for the database.

SQL is the eminent language of databases. SQL has variations in mySQL, PostgreSQL, and others. The language allows a programmer to write a request that selects rows and columns of information from different tables. An example of a scheme is the tables for person and employee. The person table might have the columns of name and age. The employee table might have the information of title and identification. If the request is for data from multiple columns or multiple tables, then SQL performs joins of the information into a single response before returning a collection of information to the server.

Out of the database and back in the server with the collection of information, the server iterates the information and selects the data that is relevant for the request from the client. An example is if the client requested authentication of a user name and password for a login to a website, then the collection of information from the database can be either a verification of the veracity of the search parameters or also additional information that might be necessary to direct the visitor on the client to a limited number of web pages rather than to all web pages at the website.

The server creates a response for the client and sends it across the web. Now at the client, the client parses the information of the response from the server. An example is that the verification of an authentication allows a visitor to see their profile on a website. The programmer has the client direct the visitor to that page.

The round trip is summarized as a client request to a server, a server acceptance of the request and subsequent request to a database, the database acceptance of the request and then its response for the server, the server acceptance of the database response and then its response for the client, and finally the client acceptance of the response from the server and an action for the visitor on the browser.

As the basic outline of an algorithm is elaborated and specified with measurement and method, so there are additional aspects of web development that enable the process of dialogue between client and server. The technique of implementation also can vary among traditions. Bootcamps and university departments might privilege the frameworks of Django, Ruby, or Node for example.

**What is the Internet and what is the World Wide Web?**

As described in *Computer Networking,* the Internet is a network of networks that has an edge and a core. The edge of the network is any access point. A domestic wi-fi connection, a smartphone with a data plan, or a business network for office computers are examples. From the edge, a signal travels through the network toward the core. The route along which the signal travels is made of the physical media of communications: overland or underseas copper wire, fiber optic cables, and electromagnetic waves.

The edge networks that connect to the core network are described by geographical extent: local, regional, national, and international networks. In each and among them, are routers where signals enqueue for processing. Network specific software routes the signals among the media toward their destination according to traffic. A Boarder Gateway Protocol conducts movement among the networks. A signal thereby travels from the edge, across the core, and to another machine along the edge elsewhere. An example is a signal from a client browser on a personal computer traversing the network and reaching its server at another point along the edge.

To expedite the journey, the routers break up a signal into pieces. Like the United States interstate highway system allows a group of people who depart from and disembark at the same places to take different roads according to the levels of traffic, so the Internet breaks up a signal into packets and routes it among nodes for better times. There are protocols for each leg of the journey. An example is the Transmission Control Protocol.

The international communications network that today exists as the Internet developed by geographical extent. In the 1960s with military support, universities around the nation developed an internal network of communication. During the next decade, businesses added their own networks. Perhaps they connected to other businesses in their region. By the 1980s, the National Science Foundation began to link regional networks together in a larger network called NSFNET. NSFNET was then commercialized by companies like AT&T and Verizon. Various internet service providers, ISPs, took control of parts of the network. They charge customers for access and provide the service of connection.

A good standard for the speed and size of commercial networking hovers around 5GH for 54 Mbps. In other words, 54 million binary digits are transferred in one billion cycles in one second. A page of text is about 2KB. A YouTube video is about 400 KB.

The World Wide Web is a network that is built on top of the Internet. It is a collection of HTML documents and the links among them. While it is possible to use the Internet without being on the Web, it is impossible to use the Web without being on the Internet.

**What is a computer?**

In a basic model, a computer is a combination of three main units and some ancillary ones. The three main units are the control processing unit, the arithmetic-logic unit, and the random access

memory. The randomness of the access to the memory is that pieces of memory do not need to be stored in contiguous transistors. The pieces of memory for one item are stored in multiple, random locations. They are retrieved from the random locations if the item that they in part specify, is required for a computation.

A computer has hardware. The hardware runs an operating system. The operating system is directed by central processor. A central processor retrieves data from memory and supplies it to the arithmetic-logic unit. The processor then instructs the arithmetic-logic unit about the computation that is necessary. The arithmetic-logic unit performs the computation. The processor takes the response and combines it with others, either in the next step of a computation, or in the creation of a result.

An example of an ancillary unit is the input – output unit. The unit includes the screen, microphone, speakers, keyboard, and mouse. The unit is ancillary not in terms of the proportion of material that it gives to the computer, but in the act of computation.

An example of the speed of an operating system is the 1.8 GH Intel core i5 of the macOS.

With the technology and some technical vocabulary in mind, the reader now is ready to start a study of the syntax of JavaScript.

Chapter 2

JavaScript

**What is a data type?**

JavaScript like other programming languages distinguishes between data and reference types. A data type is a value that can be used in a computation. There are six data types in JavaScript. They are BigInt, Boolean, Number, String, Symbol, and undefined.

BigInt is a value for numbers whose size is limited by memory capacity. It is unlikely that the reader will work with them early in his or her development.

Boolean is a value of either true or false. It is named after George Boole (1815 – 1864). Boole wrote *The Laws of Thought* (1854), which invented Boolean algebra. The branch of algebra concerns the evaluation of networks as either true or false. The mathematics that structures logic gates in the memory of a computer also rely on it. Most programming languages, including Java, C++, and Python, have a Boolean data type.

Number is a value between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. It is the data type that the reader is likely to use if he or she programs with an integer or floating-point number. In this way, JavaScript is unlike Java, C++, and Python which distinguish between integers and floating-point numbers as data types. Java and C++ further distinguish between small and large floating-point numbers.

String is used to represent one or more characters. A string is formed by placing the characters between two quotation marks. The parentheses can be either single or double quotation marks, '' or “ ” . Style guides recommend the second because double quotation marks allow text to have an apostrophe without additional syntax to ensure that the JavaScript interpreter does not misinterpret the

apostrophe as a final, single quotation mark, thereby misreading the string. JavaScript and Python each have a string data type. Unlike JavaScript and Python, Java and C++ have a string as a collection type, not as a data type. A collection is a collection of data types. Java and C++ instead have a character data type, which is then collected in a string.

Symbol is a value that has an anonymous and unique value. It is unlikely that the reader will program with them early in their progress. Anyone interested in the data type is encouraged to read about it in the official documentation of JavaScript. The documentation is called the ECMAScript Language Specification. The version of ECMAScript that specifies the Symbol data type was released in 2015.

Undefined is a data type whose value has not yet been specified. Most other programming languages do not have an undefined data type. It can be used in JavaScript to indicate that a variable is initialized or created but without a value assigned to it. Variables are explored in the next section of this chapter.

In addition to these six data types, there are three reference types: null, object, and function. The references are pointers to addresses in memory. The pointers connect a point of data with other data points. A reference type is therefore more complex than a data type. A reference type points to multiple data types.

Null is a pointer to no memory address, an indication of an absence of data. Like the data type undefined, which is assigned to a variable to indicate it has not yet been assigned a value, null can be assigned to a variable to indicate that its value is non existent.

Object is a collection of data. It has the variations of array, map, set, and others. These structures are discussed momentarily.

A peculiarity about JavaScript is that although a function is derived from an object, nonetheless the typeof operator returns function if it evaluates a function keyword in a console log statement. The

typeof operator is a keyword that prints to the console, the type of data that it is applied to.

The console is another object that has a method called log. Log is the method that prints the evaluation of its call signature. A call signature is the information between parentheses in an invocation. An invocation is a line of code that has a function name and its call signature. The invocation initiates the action of the evaluation of the function. In figure 4, is what the syntax for a log of the console with a typeof operator looks like.

Figure 4: The keyword typeof logs a nameless function to the console

```
1    console.log( typeof function( ) { } ) ;
```

In line one, there is the keyword console.log that access the log method of the console object. In its call signature is the typeof keyword that evaluates a return of the next part of the statement. The next part of the statement is the function keyword with an empty call signature. The curly braces are the declaration block of the function. They are also empty.

Note that the blank space at the beginning and end of the console statement is for legibility. It is used throughout the book. In development, however, the first element of the call signature tends to begin in the first available position and the last element in the last position. The statement in line one could be rewritten as follows, without adjustment to the syntax: console.log(typeof function( ) { }); .

If the reader runs this program in their text editor, he or she sees that the response is function. This is the peculiarity, since the function is a derivation of object. For a comparison, the reader can log to the console the console object and see that the output is object.

Figure 5: The keyword typeof logs the console to the console

```
1    console.log( typeof console );
```

In line one, the keyword of console.log has a call signature with the typeof keyword that evaluates the next part of the statement, which is the console object.

As the reader progresses in the book, it is useful to keep these data types in mind. Also practice the method of the investigation of new data with the keyword typeof.

**What is an operator?**

An operator performs an action on an operand. An operand is either a data or reference type. Common operators include the plus sign + , the minus sign – , the multiplication sign * , and the division sign % . Another common operator, the equality sign = , is used to assign variables and perform algebraic calculation. Inequality operators such as the greater than sign >, less than sign < , and their combinations with the equality operator, >= and <= for greater than or equal to and lesser than or equal to, also are used for algebraic calculation. It is advised to place a space on either side of an operator. An example is that 4>=3 is inappropriate, while 4 >= 3 is advisable and legible. Nonetheless, this is a point of style that the reader can experiment with.

If doubled, the plus sign and minus sign are commonly used to achieve incrementation. A numeric variable followed by ++ increments by one, while a numeric variable followed by - - decrements by one. The incrementation can be modulated by a combination of either a plus or a minus sign and an equality operator, which is followed by a number. An example is that a variable followed by += 2, is incremented by two. A variable followed by – = 20, is decremented by 20.

The equality operator can check the Boolean condition of a statement. It is achievable in two ways. The first checks that the two statements evaluate to the same value. The punctuation of the equality operator is that there are two equality operators == . An example is the statement, 4 == 4. If the

reader were to log this statement to the console, they see the response is true.

A second way to check the Boolean condition of a statement is by the punctuation of three equality operators, === . The triple equality operation checks if the data type and the value of the data type are the same. Like the double equality operation, the triple equality operation returns true for the statement, console.log( 4 === 4 ); .

The difference between the operations of double and triple equality arises if the JavaScript interpreter performs an automatic conversion of type in a comparison. An example is if a string data type has as its characters the integers 10. The string looks like this, “10”. The characters are an integer, but the data type is a string because it has quotation marks.

In a double equality operation, the JavaScript interpreter converts the string to the data type of number before it makes the comparison. If the reader logs to the console “10” == 10 , he or she sees that the response is true. As mentioned, the triple equality operation checks for both data type and value, so the JavaScript interpreter does not convert the string with the characters of integers into a data type of number. Therefore the triple equality operation of the statement, console.log( “10” === 10 ); , returns false.

As the double and triple equal signs achieves the comparison of different types of equality, so an exclamation mark followed by two equal signs, achieves a comparison of inequality. An example is the statement, console.log( 10 !== 10 ); , returns false. Because 10 equals 10, therefore the comparison of their inequality evaluates as false. Another example is the statement, console.log( 10 !== 20 ); . The statement returns true because 10 does not equal 20.

Another operator, \n , returns a new line in in the output of a console log statement. The operator can be included in a string. An example is that the statement, console.log( “The weather is warm.\n”, “I will go outside.” ); . The statement returns the following in the console:

The weather is warm.

I will go outside.

A final note about operators is the symbol // , which is not an operator in a strict sense, but has the quality of one in a practical sense. In a JavaScript file, the double forward slash symbol creates a comment in the line that follows it. The commented line is not run if the program is executed. To comment out a line, is often a useful way to modify and check aspects of a program without having to delete code. It also allows the developer to write comments inside a program that describe parts of a program.

**What is a variable?**

Like an assignment of a value to a variable in algebra, for example x = 4 + 16, also in JavaScript a value is assigned to a variable.

There is a standard format for the declaration of a variable. First is a keyword that defines the type of variable: either let or const. Let is used if the value of the variable might be reassigned during a program. Const is used if the value remains constant throughout the program. The distinction allows constant variables to prevent accidental access to information that is supposed to remain unchanged. Older versions of JavaScript have var as the sole variable keyword. Because it fails to distinguish the changeability of a variable, it is good to recognize but not to use.

After the keyword let or const, is the variable name. Style guides encourage the selection of a variable name that specifies not the data type but the theme of the data as it is used in a program. For example, rather than naming a string with the variable name string, if the string contains letters for example, a better name for the variable is letters. Despite the recommendations of the style, the reader

will see a near transgression of the guidelines. An example is that often a string is given the name str.

A variable name is followed by an equality operator, = . After the equality operator is the value assigned to the variable, whether it is a data or reference type.

The final piece of the declaration of a variable is a semi-colon. In JavaScript, all simple declarations are followed by a semi-colon. The assignment of data types and references are examples of simple declarations.

Figure 6 contains examples of the declaration of variables.

Figure 6: Examples of the declaration of types of variables

```
1    let letters = “abcde”;
2    const numbers = 4321;
3    let isTrue = true;
4    let isTrue = false;
5    var isFalse = false;
```

The first example has the keyword let, the variable name letters, an equality operator that assigns the following value of a string to the variable name, and finally a semi-colon. The second example has the const keyword, the name numbers, an equality operator that assigns the number 4,321 to the name, and a semi-colon. A note here is that in JavaScript, commas are omitted in numbers greater than 999.

The third example has the keyword let, the variable name isTrue, an equality operator that assigns the Boolean value of true to the name, and a semi-colon. The fourth example has the keyword let, the variable name isTrue, an equality operator that assigns the name the Boolean value of false, and a semi-colon. The fifth example has the keyword var, the variable name isFalse, an equality operator that assigns it the value of false, and a semi-colon.

The above examples raise three interesting points. The first is that the variable names of the third and fourth examples are the same, but the values are different. The duplication of the name prompts the question of whether the name is appropriate for either example. A context in which it is appropriate to name a variable isTrue is if a program computes a value that can return a true or false value. So the variable satisfies the requirement of style that it identifies an element of value without naming the value type or reference.

The second question is whether a variable can be declared twice. The answer is no. A variable can be declared only once. After its declaration, if the keyword of the declaration was let, then the variable can be reassigned. But it cannot be declared again. Here is an example of an allowable reassignment of the variable:

Figure 7: An assignment of a new value to a changeable variable

```
1    let isTrue = true;
2    isTrue = false;
```

Now if the reader logs the variable to the console, console.log( isTrue ); , he or she sees true as the response. If the reader logs to the console isTrue from the example of redeclaration, he or she sees a type error. The type error says that there is a duplicate declaration of the variable isTrue. Type errors and others, such as syntax error, are useful tools that the interpreter offers a developer. With them, a developer can pinpoint an error in their program, and thereby begin to redress it.

A last point about the above examples is that a type error is not raised for a redeclaration of a variable if the keyword is var, which has a long and complicated historical explanation. Nonetheless it is as advisable to avoid such a redeclaration, as it is to avoid the keyword var.

At this point, if the reader has followed the material, they will understand the following

examples.

Figure 8: A console log statement of a multiplication expression

```
1	let firstProduct = 10;
2	let secondProduct = 10;
3	console.log( firstProduct * secondProduct );
```

In line one, the keyword let declares a variable named firstProduct and assigns it the value of ten by an equality operator. In the second line, the keyword let declares a variable called secondProduct and assigns it the value of ten by an equality operator. In the third line, the evaluation of the expression of the firstProduct multiplied by the secondProduct, is logged to the console. The result in the console is the number 100.

A variation of the above program is the following:

Figure 9: A console log statement of a variable whose assignment is a multiplication expression

```
1	let firstProduct = 10;
2	let secondProduct = 10;
3	let result = firstProduct * secondProduct;
4	console.log( result );
```

The first two lines are the same as above. The third line is different. In line three, the keyword let declares a variable named result. The equality operator assigns the variable the product of the first two variables. In line four, the result is logged to the console. The result is still 100. The difference between the two programs in figures 8 and 9, is that the former performs the operation of multiplication in the

call signature of the log of the console, while the latter performs the operation and saves the result in the variable. The variable is then logged, and its value evaluates to 100.

**What is a collection?**

A collection is category that includes collections of data types. The name collection is another description for a reference type. Here is a brief list of the types of collections: array, set, object, map, linked list, stack, queue, graph, tree. In this section of the chapter, four of the types are considered: array, set, object, and map. These four are the only collections in the above list that have built-in structures in JavaScript. The other types are considered in the next chapter, where they are renamed data structures and are built from scratch.

An array is an order of data types. The order is specified by integers. The integers increase one by one. Please note that the order is specified by the sequence of the number of integers and not by the data. An order of the data can be imposed on the order of the integers of the array such that both the integers of the array and the data are ordered. But the only requirement for an array is the order of the integers of the array. In jargon, the order is called an index.

The notation of an array is achieved with brackets. Before the first data in an array is an opening bracket, [ , and after the last data is a closing bracket, ] . Each data in an array is separated by a comma. A space after a comma is optional.

An example of an array with integers is, [ 15, 25, 35, 45, 55 ] . An example of an array with strings is, [ “abcd”, “efgh”, “ijkl”, “mnop” ] . An array can have multiple data types. An array with integers and strings is, [ 15, “abcd”, 25, “efgh”, 35, 45, 55, “ijkl”, “mnop” ] . Because this is an array without the additional quality of being sorted, the previous example could be rearranged through all its permutations. An example of one such permutation is, [ 55, “abcd”, 25, 35, 45, “mnop”, “ijkl”, “efgh” ]

. An example of a sorted array with only integers is, [ 1, 2, 3, 4, 5 ] . An example of a sorted array with integers and letters is, [ 1, 2, 3, 4, 5, “a”, “b”, “c”, “d”, “e” ] . Note that if an array contains integers and letters, its sorted order has the integers before the letters.

There is one important and unintuitive aspect of an array. It is that the index begins not from integer one but from integer zero. As mentioned above, the order of an array is specified by integers increasing one by one. The unintuitive aspect of an array is that the beginning data type of an array is located not at index one but instead at index zero. The structure is colloquially called off by one syntax. Note also that strings have off by one syntax. Their first index is zero. An example is the string “Greetings”. Index 0 of the string is the letter “G”.

Reconsider the most recent example of an array, [ 1, 2, 3, 4, 5, “a”, “b”, “c”, “d”, “e” ] . Because the index of the first data type, which is the integer 1, is located at the index zero, therefore the index of the array can be written as in figure 10.

Figure 10: An array above and its index below

```
1       [ 1, 2, 3, 4, 5, “a”, “b”, “c”, “d”, “e” ];
          0  1  2  3  4  5     6     7     8     9
```

Please note that the first index of the array is the integer zero and the first data of the array is the integer one.

An example that is more intuitive has strings as data types. In the following, figure 11, the data type at index zero is the letter “a”.

Figure 11: An array above and its index below

```
1       [ “a”, “b”, “c”, “d” ]
```

0 1 2 3

An explanation of the name off by one, is that the system of indices is off by one from the length of an array. In the example in figure 11, the length of the array is four because it has four data types separated by commas. The final index with the data “d”, is off by one from the length: it is index three. Three is one less than four.

Because the index system is counter intuitive, therefore it takes time and practice to comprehend. The string and the array are common data structures. The reader is advised to acclimatize themselves to the concepts of length, index, and off by one.

A set is an ordered collection of data types. The order of a set is like the order of an array. Integers of the indices, increase one by one. A set is easier to work with than is an array because the iteration of a set can be performed automatically rather than manually. Therefore the reader does not need to pay attention to the length and index of a set as he or she does with an array. The mode of iteration and access of a set is developed later in this chapter.

The only restriction on the data of a set is that each data must be unique. For example the data, 1, 2, 3, 4 , can be placed in a set. If however one of the data were to be duplicated, as in the example, 1 1, 2, 3, 4 , the second integer one would not be accepted by the set because it would violate the uniqueness of the first integer one already included in the set.

A set is achieved by the punctuation of curly braces. Commas separate the items of a set. An example of a set with only numerical data types is { 1, 2, 3, 4 } . An example of a set with only string data types is { “a”, “b”, “c”, “d” } . An example of a set with numerical, string, and Boolean data types is { “a”, “b”, 1, 2, true, false } .

The third type of collection that is considered in this section, after the array and set, is an object. Synonymy risks confusion. Object was above described as a category of reference types: array, object

itself, the etymon of the function derivation, and so on. Object also denotes a specific reference type, what in the previous list is called object. For the beginner, the specific reference type is the likely concept that is indicated by the term object. In the following few paragraphs, it is that concept that is explicated.

The structure of an object is two pieces of data. The first is a key and the second is a value. The key and the value are always associated together, called a key-value pair. The data type of a key is a string. The value can be of any data type. An example is if the key is the string “a” and the value is the integer 1.

Any data type that is not a string, is made a string before becoming the key. An example is if the key is the Boolean true and the value is the integer 100. Another example is if the integer 1 is the key as, 1, and the value is, “a”.

An object is achieved by the punctuation of curly braces and a colon. The key of an object follows the opening curly brace and precedes the colon. The value of an object follows the colon and precedes the closing curly brace. Here are the examples from the previous paragraph, with the punctuation of an object: { “a” : 1 }, { true : 100 }, { 1 : “a” } . Note that although the Boolean value true and the integer 1do not appear with quotation marks as keys in the object, nonetheless their data types are strings. Verification of this is provided later in the chapter.

Please note also that in the previous sentence, the objects are separated by commas, but in the sentence they are not contained by a larger data structure. It is possible to collect them in a larger data structure. For example, one can place them in an array, as in figure 12.

Figure 12: An array with objects in its indices

```
1    const lettersAndIntegers = [ { “a” : 1 }, { true : 100 }, { 1 : “a” } ] ;
```

In line one, there is the keyword const that declares a variable with the name lettersAndIntegers. An equality operator assigns the variable an array with three objects. Each object has one key-value pair. The keys of the objects are the string “a”, the Boolean true, and the number 1.

It is also possible that an object can contain multiple key-value pairs. The pairs are separated by commas. An example is figure 13, which has the key-value pairs from the separate objects of the array in figure 12, placed into the same object.

Figure 13: An object with three key-value pairs

```
1    const lettersAndIntegers = { “a” : 1 , true: 100 , 1: “a” };
```

In line one, a keyword declares the variable lettersAndIntegers. An equality operator assigns it an object. The object has three key-value pairs.

The order of the items in an object is unspecified. The object in Figure 13 can be written { true : 100, “a” : 1, 1: “a” }, { 1: “a”, “a” : “1”, true: 100 }, and so on through its permutations and continue to be the same object.

The keys of an object are unique. An example is that if there is the object { true: 100 } and the reader added the key-value pair, { true: 200 } , then the console log of the object is, { true: 200 } . In the example, the key remains unique. The act of adding another key-value pair in which the key is the same as a key in the object, results in a reassignment of the value of that key, to the value of the new key-value pair.

Please note that while an object and set have similarities in the curly braces and property of uniqueness, nonetheless the difference between them is that an object has an associated value for each of its keys.

A map is an ordered collection of key-value pairs. Like an object and a set, its outer punctuation

is achieved by curly braces. Unlike the object and set, however, its inner punctuation is achieved by an arrow. An arrow is an equality operator that is followed by a greater than sign, => .

Also like an object and set, a map has unique values: the set has unique data, and the object and map have unique keys. Further like a set, but unlike an object, the data of a map can be of any type. Figure 14 has an example of a map.

Figure 14: A map with four key-value pairs

```
1    const let demonstration = { “a” => 1, “b” => 2, 1 => “a”, true => “a” }
```

In line one, the keyword const declares a variable with the name demonstration. An equality operator assigns the variable a map. The map has four key-value pairs. The first key is a string with the letter a. The value associated with the first key is the integer one. The second key is a string with the letter b. The value associated with the second key is the integer 2. The third key is the integer one. The value associated with the third key is a string with the letter b. The fourth key is the Boolean true. The value associated with the fourth key is a string with the letter a.

An interesting point about figure 14, is that because of the principle of uniqueness of the keys, therefore if one were to try to add another key-value pair that has a key already in the map, for example “a”, then the value of the key in the map is reassigned to the value of the new key-value pair. An example is if the map, { “a” => 1, “b” => 2, 1 => “a”, true => “a” } , has, { “a” => 2 } , added to it. The result is the object { “a” => 2, “b” => 2, 1 => “a”, true => “a” } .

The array, set, object, and map are types of the category of collections. The collections have similarities and differences, among punctuation, order, and uniqueness. Taken singularly, a form of a collection is often intuitive, such as an array or object. Combining the collections into a category, the reader is likely to loose track of which collection had what property. It is advised that the reader

familiarize themselves first with the array and then the object. Once he or she is comfortable with them, then the additional study of the set and map are achievable.

**What is control flow?**

As mentioned in the introduction, control flow specifies an action to be taken at a temporal point in a program. JavaScript has multiple types of control flow. In this section, two types of control flow are addressed. The first type is conditional logic. The second is the loop.

Conditional logic is a statement that first specifies a Boolean operation. The Boolean operation specifies a condition that evaluates as either true or false. The program then develops the true and false conditions.

Conditional logic is achieved by a syntax whose basic form is the if-else statement. The keyword if initiates the conditional logic. It is followed by a condition. The punctuation of the condition is achieved by parentheses that contain an operator. The operator can evaluate to either true or false. After the condition a declaration block develops the program if the evaluation of the condition results in true. The declaration block is achieved by the punctuation of curly braces. The keyword else follows the declaration block. After the keyword else is another declaration block also achieved by the punctuation of curly braces. It develops the program for the circumstance in which the condition of the conditional logic evaluates as false.

An algorithmic example of conditional logic is in figure 15.

Figure 15: An algorithm of conditional logic

1. Declare a const variable with the name logicTest and assign it a Boolean value of true.
2. Initiate a statement of conditional logic. The condition should test that the variable is true.

3. If the variable is true, log the Boolean value true to the console.
4. If the variable is false, log the Boolean value false to the console.

An example of the algorithm in a program is in figure 16.

Figure 16: A program of conditional logic

```
1       const logicTest = true;
2       if ( logicTest  === true ) {
3               console.log( true );
4       } else {
5               console.log( false );
6       }
```

In line one, the keyword const declares a variable with the name logicTest. An equality operator assigns it the value true. In line two, the if keyword initiates both the entire statement of conditional logic and also the declaration block for the program if the condition is true. The parentheses contain the test of the condition. The triple equality operator checks whether the type and the value of the two operands are equal. The first operand is the variable named logicTest. The second operand is the Boolean value true. A curly brace marks the beginning of the declaration block of the true case.

In line three, inside the declaration block, the Boolean value true is logged to the console. In line four, a closing curly brace ends the first declaration block. The else keyword initiates the second declaration block. The beginning of the second declaration block is marked by an opening curly brace. In line five, the Boolean value false is logged to the console. On line six, a closing curly brace completes both the second declaration block and the statement of conditional logic.

The program develops both conditions of the evaluation. In the declaration blocks, a Boolean value is logged to the console. In the execution of the program, however, only one of the declaration blocks will be entered. This is because the condition evaluates to a Boolean value. Either the variable named logicTest evaluates to true or to false. If it evaluates to true, then the interpreter enters the first declaration block of the program. If the condition evaluates to false, then the interpreter enters the second declaration block. The movement from the condition to either of the declaration blocks is an example of flow. The direction by the condition to either one or the other declaration blocks is an example of control. Together, they make the control flow.

The reader is encouraged to take the basic form of the if-else statement and experiment with it. Try different operands and operators in the condition. Try different declaration blocks. Put a conditional statement inside each of the declaration blocks. The more the reader experiments with different combinations of the basic form, the more quickly and secure he or she will feel with the syntax. Examples of control flow are discussed below in the section with examples.

The second type of conditional logic that is discussed in this section, is the loop. A loop is a statement that a program runs multiple times or iterations. Each iteration accesses a different piece of data. An example of a statement that a loop can run, is the equality operation. Consider the equality operation of figure 17, which is without a loop.

Figure 17: A console log statement of an equality operator

```
1    const firstOperand = 2;
2    const secondOperand = 2;
3    console.log( firstOperand === secondOperand );
```

In line one, the constant variable firstOperand is assigned a numeric value of two. In line two, the

constant variable secondOperand is assigned the value of two. In the third line, the console logs the equality operation that checks that the two operands, have the same value and data type. If the reader logs this program to the console, they see the result is true.

Imagine that the second operand were not a variable, but an array of integers. Such an array is in figure 18.

Figure 18: An array of strings

```
1       const integers = [ “a”, “b”, “c”, “d” ];
```

With a loop, the statement of the equality operation can be performed on multiple data points. The variable named firstOperand can be checked against each data point in the array named integers. The algorithm for such a program is given in figure 19.

Figure 19: An algorithm of a loop that checks the equality of a variable with the data of an array

1. Create a variable and assign it a string value of a letter.
2. Create an array and populate it with string values of letters.
3. Iterate the array with a loop and compare the variable with each data point in the array.
4. Log to the console the result of the result of the comparison.

The interpreter enters the array named integers with the variable named firstOperand. The interpreter goes to the first index of the array. Remember that the first index of an array has the integer 0. At the first index of the array, which is index 0, the interpreter compares the value of the data, which is the string with the letter a, there to firstOperand. It logs the result of the operation, which is either true or false. Then the interpreter goes to the second index of the array which is index 1. It compares the value

of the data there, which is a string with the letter b, to firstOperand and logs the result to the console. The interpreter goes to the third index of the array which has index 2. It compares the data there, which is a string with the letter c, to firstOperand and logs the result to the console. The interpreter goes to the fourth index of the array which has index 3, compares the data there, which is a string with the letter d, to the operand, and logs the result to the console. Because the fourth index is the last index of the array, the interpreter ends the loop. There are no more comparisons or logs to the console.

The loop allows the singular statement of an operation of a comparison of equality, to be performed at each point of data in the array. The syntax for a loop is achieved through one of two keywords. The first that is considered is the keyword for. After a discussion of the for loop, there is a discussion of the loop of the second keyword, while.

A for loop is a loop that performs an action once for each item of data in the structure that is iterated. An example is the previous figure, figure 19, in which a comparison of equality was performed between a variable and the data structure that was being iterated or looped over, an array.

The syntax of a for loop begins with the keyword for, which initiates the loop. Following the keyword is a parentheses. Inside the parentheses, is a three part statement that specifies the conditions of the loop. The parts of the condition are separated not by commas but by semi-colons.

The first part is a definition of the iterator. The iterator is the locus of movement from one data point to the next. An iterator moves along an iterable. An iterable is another term for any data structure that an iterator can move over.

The second part of the condition inside the parentheses, is an equality operator that specifies the limit of iteration. The third part is a definition of the size of the movement. Figure 20 summarizes the parts of the statement of the conditions of a for loop.

Figure 20: The conditions of a for loop, defined inside a parentheses after the keyword for

1. The assignment of a value to the iterator
2. The limit of the iterable
3. The quantity of movement at each step by the iterator

After the declaration of the keyword for and the parentheses, there is a declaration block. The beginning of the declaration block is marked by an opening curly brace. Inside the declaration block is a development of the program, such as a log to the console of the result of a comparison. The declaration block is then closed by a curly brace, which closes the loop. An example of the syntax of a for loop is given in figure 21.

Figure 21: A program that has a for loop that checks the condition between a variable and the data items in an array

```
1	const firstOperand = “a”;
2	const integers = [ “a”, “b”, “a”, “c”, “d”, “a” ];
3	for ( let i = 0; i < integers.length; i++ ) {
4		console.log( firstOperand === integers[ i ] );
5	}
```

In line one, a constant variable named firstOperand is declared and assigned a number data type with the value 10. In line two, a constant variable named integers is assigned an array with integer values in its indices. In line three, the for loop begins with the keyword for. Next an opening parenthesis initiates the condition of the for loop. The variable keyword let initiates a variable named I, which is the iterator. The equality operator assigns the variable named i the number value of the integer 0, which is the value of the first index of the array. Then a semi-colon separates the first from the second part of the

condition.

The second part of the condition contains an inequality operator that sets the limit of the iterator. The statement indicates that the iterator i is less than the length of the array named integers. Please note here the appendation syntax which evaluates to an integer that is the length of the array. Said another way, the phrase integers.length evaluates to the length of the array named integers. Another semi-colon separates the second from the third part of the condition.

Still in line three and inside the parentheses, the third part of the condition specifies with the double plus sign, ++, that the iterator, i, is to increase incrementally along the iterable which is the array of integers. The closing parenthesis ends the condition. Next an opening curly brace begins the declaration block of the for loop.

In line four, the declaration block has the console log method. Inside the parentheses of the method is an operation of the comparison of equality between two operands. The first operand is the variable named firstOperand, which earlier was assigned the data type of number with the value of the integer 10. The second operand is the data point at the index in the array named integers, where the iterator is currently located. The syntax that achieves the extraction of the data for the comparison is called an indexation of the the iterable by the iterator. Please note that there is an indexation of an index in an iteratble by an iterator.

The indexation achieves the extraction of the data point. The indexation also automatically returns the data after the index. If the operation during indexation changes the data point, then the data that the indexation returns is the changed data point.

The punctuation by which indexation is achieved is an opening and closing bracket, [ ], placed after the iterable: integers[ ] . With the iterator inside the brackets, [ i ], the indexation extracts the data point from that index. If the iterator evaluates to zero, then the data point at integers[ 0 ] evaluates in this example as a string with the letter a, “a”. If the iterator evaluates to one, then the data point at

integers[ 1 ] evaluates to a string with the letter b, “b”. And so on.

Still in line four, after the closing bracket of the indexation, there is a closing parenthesis. The closing parenthesis ends both the comparison of equality and the log to the console.

In line five, a closing bracket ends both the declaration block and the loop. If the reader tries this program in their text editor, they see the result is true, false, true, false, false, true.

Consider what happens during the iteration. At the first index, firstOperand is compared to “a” and evaluates as true. At the second index, firstOperand is compared to “b” and evaluates as false. At the third index, firstOperand is compared to “a” and evaluates as true. At the fourth index, firstOperand is compared to “c” and evaluates as false. At the fifth index, firstOperand is compared to “d” and evaluates as false. And at the sixth and final index, firstOperand is compared to “a” and evaluates as true.

A point worth refocusing on is the off by one indexical structure of the array. Note that the iterator is defined in the condition with the assignment of the integer 0, in the first part of the parentheses in line three. Because the array named integers has off by one indices, resulting in the first index having the integer 0, therefore by assigning the iterator the value 0, the loop begins with an indexation of the array named integers at its first index: integers[ i ], where i = 0.

Another aspect of the off by one indices is the final index and its correspondence with the definition of the limit of the iterable in the second part of the parentheses. The last index of the array named integers has the data point “a” . The index at this point of the array is five. This index is one less than the length of the array, which has six data points. The length of the array is six, and it is also the limit that was set on the iterator: i < integers.length. The comparison of an inequality operation between the iterator and the length of the array, indicates that the iterator continues to increment by one, as long as it is less than the length of the array.

Think about the condition in the second part of the parentheses in line three, with each iteration.

At first the iterator evaluates to 0. 0 is less than the length of the array, which is 6. So the iterator increments, i++. In the second iteration, the iterator evaluates to 1. The length of the array named integers is still 6. 1 is less than 6. So the iterator increments by one.

In the third iteration, the iterator evaluates to 2. 2 is less than 6. The iterator increments by one. In the fourth iteration, the iterator evaluates to 3. 3 is less than 6. The iterator increments by one. In the fifth iteration, the iterator evaluates to 4. 4 is less than 6. The iterator increments by one. In the sixth iteration, the iterator evaluates to 5. 5 is less than 6. The iterator increments by one. In the seventh iteration, the iterator evaluates to 6. 6 is not less than 6. The iterator does not increment. The declaration block is not entered. The loop terminates.

There are other aspects of the for loop that are worth considering, such as additional indexation capability and its consequence for setting the limit of the iterator. These aspects are covered later in the chapter, with examples and discussion that exposit them.

In addition to the for loop, which performs a statement of operation for each item in a collection as long as its condition allows, there is a while loop which performs a statement of operation while a condition is true. The for and while loops are similar, in that each has a condition that governs its operation.

The differences between them are both conceptual and syntactical. The conceptual difference is that the for loop is used if the number of iterations to be performed is known before the loop begins. An example is figure 21, in which the size of the array named integers is known. A while loop is used if the number of iterations is unknown before the loop begins.

The syntax of the while loop begins with a declaration of the iterator. An equality operator assigns the iterator a numerical value. Next the keyword while initiates the loop. The keyword while is followed by an opening parenthesis. Inside the parentheses is the declaration of an equality or inequality comparison that serves to establish the condition which evaluates in order to continue or

discontinue the loop. After the comparison is a closing parenthesis. Next an opening curly brace initiates the declaration block of the loop. The program is developed inside the declaration block. An example is a log to the console of the iterator. Also in the development is the incrementation of the iterator. Then a closing curly brace terminates both the declaration block and loop. An example of a while loop is given in figure 22.

Figure 22: A while loop

```
1       let i = 0;
2       const j = “d”;
3       const integers = [ “a”, “b”, “c”, “d” ];
4       while ( j !== integers[ i ]  ) {
5               console.log( integers[ i ] );
6               i++;
7       }
```

In line one, the variable named i is declared and assigned the value 0, which will serve to begin the iteration of an array from index 0 . In line two the constant variable j is declared with the value of a string with the character d, “d”. In line three, an array named integers is declared that has four data points. The data points are strings, each with one character: a, b, c, or d.

In line four, the while keyword begins the loop. The opening parenthesis allows the beginning of the declaration of the condition that provides the limit of the loop. The variable j is an operand that with an inequality operator, is compared to an indexation of the array named integers at the index of the value of i. A closing parenthesis ends the condition declaration. Next an opening curly brace begins the declaration block.

In line 5, there is a log to the console that evaluates to the data point achieved by the indexation of the array named integers by the iterator during each iteration. In line six, the iterator is incremented by one. In line seven, a closing curly brace ends both the declaration block and the while loop.

If the reader runs this while loop in a text editor, the output is a, b, c . Note that some editors show strings without quotation marks in the log of a console. Consider the iterations. In the first iteration, the variable named 0 evaluates to 0. The variable named j evaluates to “d”. The inequality comparison checks the evaluation of the first operand, “d”, to the value of the data point that is extracted by the indexation of the array at the iterator. The iterator is 0. So the indexation draws out the data point from index 0, which is “a”. “a” does not equal “d”, so the inequality operator evaluates as true. The interpreter then enters the declaration block. The console logs the evaluation of the indexation, a. The iterator i increments by one and becomes the integer 1. From line seven, the while loop returns to line four.

The condition of the loop is checked again. The variable named j, which has the value of “d”, is compared to the evaluation of integers[ i ]. Because i now equals 1, the data point at the first index of integers is extracted for the comparison. Because “b” does not equal “d”, therefore the inequality operation evaluates as true, and the program enters the declaration block. In line five, the console logs the data point of the array that is indexed by the iterator. The data point is “b”, which prints to the console as the letter b. In line six, the iterator is incremented by one and evaluates to the integer 2. In line seven, the iterator is returned to line four to again check the condition of the while loop.

Now in the third iteration, the while loop continues. Inside the parentheses of the conditional of the loop, the constant variable named j evaluates to “d”. It is an operand in an inequality operation with the data point that is extracted from the array named integers by an indexation of the array with the iterator, which currently evaluates to 2. The data point in the array named integers at index 2 is “c”. So the data points “c” and “d” are compared for inequality. Because “d” does not equal “c”, the inequality

comparison evaluates as true. The interpreter enters the declaration block. In line five, the console logs the data point that is indexed by the iterator. The iterator is 2, so the data point is “c”. The console prints c to the text editor. In line six, the iterator is incremented by one. In line seven, the iterator is returned to line four in order to continue the while loop.

In line four, at the fourth iteration with an iterator that evaluates to 3, the condition of the while loop is checked. Inside the parentheses, the constant variable named j evaluates to “d”. It is the first operand of the inequality operation. The second operand is the data point that is extracted with the indexation of the array named integers by the iterator. The iterator currently evaluates to 3. The data point at the fourth index of the array, which because of off by one indexation is denoted by the integer 3, is “d”. Because “d” equals “d”, therefore the inequality comparison evaluates to false. Because the condition of the continuation of the while loop requires an evaluation of true to enter the while loop, therefore the loop ends with the result of a false evaluation of the condition.

That is the basic structure and implementation of a while loop.

The while loop in figure 22 differs from the example of a for loop that was given in figure 21. The limit placed on the iterator in the for loop in figure 21, is the length of the iterable, which is known before the for loop begins. In contrast, the while loop in figure 22 has the limit that a variable is unequal to a data point that is extracted by an indexation of the array named integers by an iterator. As long as that condition is true, the while loop continues. The number of iterations is not known before the loop begins.

In the example of the while loop in figure 22, the limit of the while loop is immediately observable because one can see the variable j an inch away from the data point in the array named integers at index 3. But what if the index that nullifies the while loop is not at index three but instead at one hundred or one million? If the index is so far away, then the number of iterations is unknown at the start of the program. The while loop is suited for this type of loop because, as mentioned, its iteration is

based not on a number of iterations but on an iterable condition. The iterator increments item by item until it extracts a value that returns false in the evaluation of the conditional comparison.

As with the for loop, with the while loop there are additional considerations that are considered in the examples later in the chapter.

**What is a function?**

A function is a type of object that can receive an input, access outside variables, create local variables, perform a computation, and report a result. An example of a function that the reader has already encountered is the Hello, World! program that was demonstrated in figure two and is reproduced here as figure 23.

Figure 23: A reproduction of the Hello, World! program from figure 2

```
1    function helloWorld( ) {
2        const str = “Hello, World!”;
3        return str;
4    }
5    console.log( helloWorld( ) );
```

The syntax of a program can be learned from the example. In line one there is the keyword function that initiates a function. After the keyword, there is the name of the function, helloWorld. At this point, please note this aside: the typography of the name of the function demonstrates what is colloquially called camel case. The upper case letter of any word after the first word, resembles the humps of a camel. Style guides recommend it as an unofficial standard among JavaScript developers.

After the name of the function, there is an empty call signature, ( ). Later in the chapter are examples wherein the call signature receives variables.

After the call signature, an opening curly brace initiates the declaration block of the function. In line two a constant variable named str is assigned by an equality operator a string with the characters Hello, World! .

In line three the keyword return, sends the variable named str outside the function, to wherever the function was invoked.

In line four, the closing curly bracket concludes both the declaration block and the function.

In line five, the invocation of the function named helloWorld, is logged to the console. The invocation is the name of the function and its call signature, helloWorld( ) . Because the invocation is inside the parentheses of the console log statement in line five, its evaluation is logged to the console. The evaluation of the invocation results in the variable named str which is returned by the keyword return in line three. The return statement pushes the return value outside the function to the invocation of the method.

The console logs the value of str, which is a string with the characters Hello, World, as defined in line two.

An advanced point for the beginner is to note that the variable named str is created inside the function and then returned to the invocation of the function that is outside and below the function, on line five. If there were no return statement inside the function, and if the reader logged the variable named str to the console, then a reference error arises.

The technical explanation of the circumstance concerns the concept of scope, which denotes a limitation of access to variables. The limitation occurs by the location of the declaration of the variable. In this example, the variable is declared inside the function. Therefore it cannot be accessed outside the function unless it is transferred there by a return statement. Inside a function is called local scope.

Outside a function is called global scope.

A rule is that variables defined in global scope can be accessed by statements in global scope and by statements in local scope. Variables defined in local scope, however, can be accessed only in local scope, and not in global scope unless transferred there, as mentioned. The reader might store this bit of information in their memory along with the distinction between the variable types const and let, which address a different aspect of scope that was discussed in the section on variables.

Examples of other functions are discussed later in this chapter. For now, the reader is advised to learn the basic structural elements of a function: the keyword function, the name and call signature, and the keyword return inside a declaration block.

**What is a class?**

Like a function, a class is another type of object. It is a collection of attributes and methods. An attribute is a local variable of the class. A method is a local function of the class. Also like a function, the syntax of a class can learned from an example, given below as figure 24.

Figure 24: The basic syntax of a class

```
1       class Addition {
2               constructor( ) {
3                       this.sum = null;
4               }
5
6               add( firstAddend, secondAddend ) {
7                       this.sum = firstAddend + secondAddend;
```

```
8                       return this.sum;
9               }
10      }
11
12      const integers = new Addition;
13      console.log( integers.add( 20, 30 ) );
```

In line one, the keyword class declares a class named Addition. Note both the keyword new and the capitalization of the first letter of the name. The keyword new creates a new instance of the abstract class, for storage in a variable. The class name, unlike a function which has a lower case letter at the beginning of its name, has an upper case letter at the beginning of its name.

After the name of the class is an opening curly brace. The brace begins the declaration block of the class. In line two, the keyword constructor declares the constructor of the class. A constructor is placed at the beginning of the declaration block of a class. It initializes the attributes of the class. A constructor is called every time that the class keyword is invoked, to provide the invocation access to the attributes of the class. After the keyword constructor is an empty parentheses, which is the call signature of the constructor.

Like with the example of the HelloWorld! program in figure 23, the empty call signature here indicates that the constructor receives no input. If either the HelloWorld! program or the constructor were to receive input, it is received in the call signature. An example of the reception of input into a call signature is given in the method of the class, which will be explained momentarily. After the empty call signature is an opening curly brace that begins the declaration block of the constructor. In line three, the attribute sum is assigned the data type null.

The keyword this that precedes the variable named sum, and which is attached to it by the

notation of appendation, this.sum, colloquially called “dot syntax,” gives the variable a class scope despite the fact that the variable is declared in the constructor. If ever the attribute is called inside the class, the this keyword and the notation of appendation also occur.

In the first occurrence of access to the attribute, the attribute evaluates to null. If the occurrence redefines the attribute, then its value in the constructor updates to the reassignment. Otherwise the value remains null. An example of this is given later in the chapter. Please note also that the keyword this, serves in place of either the keywords let or const in the declaration of the attribute.

In line four, a closing curly brace ends the declaration block of the constructor of the class. In line five, there is a blank space, which is given stylistically to separate the constructor from the method. The reader can choose this style, or one with more or fewer spaces.

In line six, the declaration of the word add is the name of a function. As mentioned, a class is a collection of attributes and methods. The attributes are included in the constructor. The methods are a series of functions.

Inside a class, a function is called a method of the class. Unlike a declaration of a function outside a class, a declaration of a method inside a class excludes the keyword function. The declaration of a method begins with the name of the method. In the example in line six, the name of the method is add.

Next is the call signature of the method. The call signature has two variables, one named firstAddend and the other named secondAddend. Inside a call signature of a function, constructor, or method, the variables are called parameters. So in a technical description, one can write that the method add of the class Addition, has two parameters.

The parameters evaluate to whatever value is passed to them from the place where the method is invoked. After the call signature is an opening curly brace that starts the declaration block of the method.

In line seven, the attribute this.sum is accessed from the constructor. Because this is the first time that the attribute this.sum has been accessed inside a method, therefore it evaluates to the value that it was initialized with inside the constructor, which is the data type null. By the equality operator, the attribute this.sum is assigned the value of an addition statement. The operands of the addition statement are the two parameters that were passed to the call signature of the function, in line six. This addition operation redefines the value of the attribute this.sum. The attribute updates in the constructor to whatever is the sum of the addition operation.

In line eight, is a return statement that transfers the value of this.sum to the location where the method was invoked. In order to accomplish the transfer, the return statement accesses the value of this.sum from the constructor, which contains the updated value.

In line nine, a closing bracket ends both the declaration block of the method and the method itself. In line ten, a closing bracket ends both the declaration block of the class and the class itself. In line eleven there is a blank space for style and legibility.

In line twelve, there is a declaration of a constant variable with the name integers. The variable named integers is assigned a value of the keyword new and the class named Addition. Together, the keyword new and the class named Addition, create a new instance of the class that was defined with the keyword class in line one. That new instance of the class Addition is assigned as the value to the variable named integers. Therefore, it is by that variable that the attributes and methods of the class are accessed in subsequent steps.

In line thirteen, the content of its call signature is logged to the console. The content of the call signature includes the variable named integers and a notation of appendation that accesses the method add of the class Addition.

Inside the call signature of the method, which is inside the call signature of the console log statement, are two variables. The first is the integer 20. The second is the integer 30. The variables

inside the call signature of an invocation, are called arguments. The invocation transfers the arguments to the method of the class, where the arguments become parameters of the method. Said another way, the arguments of an invocation of a method become parameters in the call signature of the method.

In line thirteen, a closing parenthesis is the last punctuation of the program.

Please imagine the execution of the program in figure 24. The JavaScript interpreter reads the declaration of the class with its attribute and method. Then the interpreter reads the declaration of the variable named integers. The interpreter creates a new instance of the class named Addition and assigns it to the variable named integers. The instance of the class is isomorphic with the declaration of the class. There is a constructor that initiates an attribute named this.sum with the value of null. There is also the method of the class. These creations are stored inside the variable named integers.

Then the JavaScript interpreter reads the console log statement of the invocation of the add method that since line 12 has been associated with the variable named integers. The JavaScript interpreter runs the program.

The interpreter first constructs an attribute named this.sum with the value of null. Then it transfers the arguments from the invocation to the method of the class. The method add( ) of the class instance Addition inside the variable named integers, receives the arguments from the invocation, and transforms them into parameters.

The interpreter enters the declaration block of the method. There it accesses the value of this.sum, which is null because this is the first time the value has been accessed. The variable this.sum with the value of null is reassigned a value that is the result of an addition operation.

The operands of the addition operation are the variables firstAddend and secondAddend. The variables evaluate the value that they are passed from parameters of the call signature. So firstAddend evaluates to the integer 20, and secondAddend also evaluates to the integer 30. An addition operation is carried out with these addends, with the result of the integer 50. The integer 50 is assigned to this.sum.

The interpreter updates the value of this.sum to 50, in the constructor.

In line 8, the return keyword accesses the value of this.sum, which is now 50, and transfers it to the location of the invocation of the method. The location is the inside the call signature of the console log statement. The method add( ) of the class Addition, evaluates to the integer 50. The console logs the integer 50.

This is the basic structure of a class. There is more to be learned, and more is covered in the following examples. For now, the reader is advised to conceptualize the basic structural elements of a class. There are the keyword and name of the class; the constructor with its name, call signature, and declaration block; and the method with its name, call signature, and declaration block.

Also try to familiarize oneself with the movement of a class. There is the declaration of a class, an assignment to a variable of an instantiation of the class, and a log to the console of the invocation of a method of the instance.

In addition, there is the movement of the variables, from arguments in the invocation of the method, to the parameters of the method. Also there is the movement of the initialization of the attribute, its update by the method, and its access and transfer to the call signature of the invocation, where it becomes the value that is logged to the console.

**What is a built-in method?**

A built-in method is a method that corresponds to a data type. For each type, whether it is a string, array, object { } , set, or map, there are methods that only that type can access from the JavaScript interpreter. Here, a few built-in methods are discussed. Note that there is a reference for the built-in methods of strings and arrays at the w3schools website.

Like other built-in methods, those of the string data type, focus on the access or creation of a

copy of the string with a potential of the transformation of the data points. An example is the built-in method charAt( ) . The syntax for the method charAt( ) is given in figure 25.

Figure 25: The charAt( ) built-in method

```
1    const name = “Francis Bacon”;
2    console.log( name.charAt( 0 ) );
```

In line one, a constant variable named name is assigned a string with the characters Francis Bacon. In line two, the console logs the variable with the built-in method charAt( ). The call signature of the method has the integer zero. The built-in method charAt( ) returns the data point of the string at the index of the value given as an argument in the invocation. Since the data point at index zero of the variable named name, which evaluates to the string “Francis Bacon, contains the letter F, the built-in method returns that data point to the invocation of the method. The console prints the letter F.

The built-in method is useful because it saves a developer the effort of accessing the data point by iteration. The built-in method is therefore economical code.

If the built-in method charAt( ) accesses the data point an an index of a string, the built-in method named includes( ) determines whether a string has the value that is given as the argument of the method. An example of the syntax of the includes( ) built-in method is given in figure 26.

Figure 26: The includes( ) built-in method

```
1    const name = “Francis Bacon”;
2    console.log( name.includes( “B” ) );
3    console.log( name.includes( “S” ) );
4    console.log( name.includes( “i” ) );
```

In line one, a constant variable named name is assigned the value of a string with the characters Francis Bacon. In line two, the evaluation of its call signature, is logged to the console. The call signature contains the variable named name. By appendation, the built-in method includes( ) is activated with the variable. The call signature of the built-in method is a string with the letter B.

In line three, the evaluation of the call signature, is logged to the console. The call signature contains the variable named name. To it, the built in method includes( ) is appended. The call signature of the built-in method has a string with the letter S. In line four, the evaluation of its call signature, is logged to the console. The call signature has the variable named name with the built-in method includes appended to it. The call signature of the built-in method has the argument of a string with the letter i.

What happens when the program executes? The JavaScript interpreter reads the first line and saves the string to the variable. In the second line, the interpreter runs the built-in method on the variable. From the beginning of the string, the interpreter loops along the string and extracts and compares data points with the value of the argument of the invocation, until it finds the letter B. The interpreter finds an affirmative comparison at index 8. The interpreter returns the Boolean value true. The call signature of the built-in method evaluates to the Boolean true. The console logs true.

In line three, the interpreter logs to the console the evaluation of the call signature. The call signature contains the variable named name. Appended to the variable, is the built-in method that has a call signature with a string and the letter S. The interpreter begins to search the variable named name for the letter S from its beginning index.

The interpreter searches the indices one by one, not finding the letter until the interpreter reaches the end of the string. Because the interpreter found no comparison in which both operands are the string with the letter S, therefore the interpreter returns Boolean data type with the value false to the

invocation of the built-in method. The console then logs false.

In line four, the interpreter starts the console log statement. The interpreter looks inside the call signature, and finds the variable named name and with it, the built-in method includes( ). The built-in method has an argument that is a string with the letter i. The interpreter begins to search the variable named name for the letter i. The interpreter finds a positive comparison at index 5. The interpreter returns the Boolean value true to the invocation of the method in the call signature. The invocation evaluates to true. The console logs true.

A few things of interest about the example in figure 26 include the fact that the built-in method has case sensitivity. An example is that if the reader runs the built-in method with a string and the letter s, the result is false. Also note that the built-in method is more economical in code than is an iteration of the string, a comparison at each index with the value of the argument, and a return statement.

An example of a built-in method that changes a string is given below in figure 27.

Figure 27: The replace( ) built-in method for strings

```
1    const name = “Francis Bacon”;
2    console.log( name.replace( “a”, “o” ) );
```

In line one, a constant variable is assigned a string with the characters Francis Bacon. In line two, the evaluation of its call signature, is logged to the console. The call signature contains the variable named name. To the variable, is appended the replace( ) built-in method. The structure of the replace( ) built-in method is that it accepts two variables as arguments. The method then finds the first instance of the first argument in the string. The method then replaces that data point with the second argument.

In line two, the first argument is a string with the letter “a”. The second argument is a string with the letter “o”.

When the program is executed, the interpreter reads in line one that the variable named name is assigned a string with the characters Francis Bacon. In line two, the interpreter begins a log to the console. The interpreter checks inside the call signature and accesses the replace built-in method, running it on the value of the variable named name.

From the first index of the string, the interpreter begins to check the data point against the second argument of the invocation. For each comparison in which the comparison is negative, the interpreter increments by one to the next index, until it reaches the end of the string. In this example, the interpreter finds a positive comparison at index 2, where “a” === “a” . The interpreter replaces the value at the index with the second argument from the call signature of the invocation of the method.

Then the interpreter returns the string with the replacement, to call signature of the invocation of the method. The call signature evaluates to the new string. The console logs the string Froncis Bacon.

In figure 27, the built-in method exemplifies the type of built-in method that modifies the string that it accesses, rather than merely returning information about it to the call signature. Another note is that the modification of the string takes place in a local scope. The built-in method returns the updated string to the invocation of the call signature, but the method does not update the value of the variable named name.

The difference distinguishes the relationship between the built-in method and the variable seen here in figure 27, from the method that updated the attribute in figure 24. In the example in figure 27, the built-in method produces a new string with the replacement data point, rather than updating the existing variable. A final point, is that the new string can either be logged to the console, as it is in figure 27, or it can be saved to a variable and then developed further. An example is given in figure 28.

Figure 28: The assignment of a variable of a new string from the built-in method replace( )

```
1    const name = “Francis Bacon”;
2    const alteration = name.replace( “a”, “o” );
3    console.log( name );
4    console.log( alteration );
```

In line one, a constant variable named name is assigned a string with the characters Francis Bacon. In line two, a constant variable named alteration is assigned the evaluation of the built-in method replace( ) that is appended by dot syntax to the variable name. The call signature of the method contains two string arguments, the first with the letter a and the second with the letter o. In line three, the variable named name, is logged to the console. In line four, the variable named alteration, is logged to the console.

When the program is executed, in line one, the interpreter assigns the string to the variable named name. In line two, the interpreter assigns the evaluation of the built-in method to the variable named alteration. From index 0 in the string of the named name, the interpreter loops along the indices making comparisons between the first argument and the indexed data point, until it finds a positive comparison.

The interpreter finds a positive comparison at index 3. The interpreter creates a new string with the data point at index three replaced by the second argument of the call signature of the built-in method. The interpreter returns the new string as the value to assign to the variable named alteration. In line three, the interpreter logs to the console the value of the variable named name, which is Francis Bacon. In line four, the interpreter logs to the console the value of the variable named alteration, which is Froncis Bacon.

Figures 27 and 28 show the flexibility of JavaScript. The language can control the results of built-in methods by either logging them to the console or saving them in variables for further

development. Examples of the further development of built-in methods that are saved to variables, are included in the examples section of this chapter.

A final example of the built-in method with strings is given in figure 29. The method is the concat( ) method, which joins two strings together.

Figure 29: The built-in method concat( ) with two strings

```
1    const name = “Francis Bacon”;
2    const profession = “ was a statesman.”;
3    const sentence = name.concat( profession );
4    console.log( sentence );
```

In line one, the constant variable named name is assigned by an equality operator a data type with the value o string with the characters Francis Bacon. In line two, the constant variable named profession is assigned by an equality operator a string with the characters was a statesman. Note the space at index 0 in the value of the variable named profession. The space allows the combination of the strings with the appropriate spacing in the English sentence.

In line three, the constant variable sentence is assigned by the evaluation of the concat( ) built-in method that is appended to the variable named name. The call signature of the method has the variable named profession as an argument. In line four, the variable named sentence is logged to the console.

When the program executes, the interpreter assigns the string to the variable named name in line one. In line two, the interpreter assigns the string to the variable named profession. In line three, the interpreter assigns the evaluation of the method to a variable named sentence. The interpreter enters the call signature of the built-in method concat( ) and finds the variable named profession. The interpreter then combines the variable name which it accesses via appendation notation, with the variable named

profession, located in the call signature. In the combination, the variable name is first and the variable profession is second. The interpreter then assigns the value of that string to the variable named sentence. In line four, the interpreter logs the value to the console: Francis Bacon was a statesman.

Note that in addition to the concat built-in method, another syntax for the achievement of concatenation is the plus sign between two strings. For example, if line three of figure 29 were instead to read, const sentence = name + profession, the result is the same. Both the concat built-in method and the plus sign accept strings or variables whose values are strings. For example, line three of figure 29 could also read, const sentence = “Francis Bacon” + “ was a statesman.”, or, const sentence = “Francis Bacon”.concat( “ was a statesman.”), without a change to the output in the console.

Because the built-in methods for the arrays vary from those of the strings more in syntax than in concept, and because the practice with the methods on the w3schools is the way to achieve familiarity with them, therefore a review of the built-in methods for arrays is omitted here. In its place, note that arrays differ from strings in that the array contains a sequence of elements whose data points can be any data type, while a string has only the string data type. Otherwise, the built-in methods for arrays focus on the identification of the data it contains or the creation of a copy array with a transformation of some data point or points.

**Examples**

In this section are numerous examples and commentary about them. The topical order follows that of the chapter: data type; operator; variable; collections by string, array, and object; conditional logic by if – else statements, for loops, and while loops; function; class; and built-in method. The aim of this section is to give the reader exposure to and explanation of variations of the models that are set out earlier for each topic. Practice with changes achieves a subtler comprehension and often one more

deft at the traversal and manipulation of data structures.

Data types

Below, figure 30 has an example of the keyword typeof applied to different types of data in the call signature of a log to the console.

Figure 30: Data types and their display: string, number, Boolean, undefined, and null

```
1     const name = “Francis Bacon”;
2     console.log( typeof name, name );
3
4     const age = 59;
5     console.log( typeof age, age );
6
7     const scientist = true;
8     console.log( typeof scientist, scientist );
9
10    const bankAccount = undefined;
11    console.log ( typeof bankAccount, bankAccount );
12
13    const nickName = null;
14    console.log( typeof nickName, nickName );
```

In line one, a constant variable named name is declared and assigned a string with the letters Francis

Bacon. In line two, a console statement logs both the type of variable named name, and the variable named name, to the console of text editor of the reader. Line three is left empty for legibility. In line four, a constant variable named age is declared and assigned the number 59.

In line five, a console statement logs both the type of the variable named age, and the variable, to the console. Line six is left empty for legibility. In line seven, a constant variable named scientist is declared and assigned the Boolean value true. In line eight, a console statement logs both the type of the variable named scientist, and the variable, to the console. Line nine is left empty for legibility.

In line ten, a constant variable named bankAccount is declared and assigned the data type undefined. In line eleven, a console statement logs both the type of the variable named bankAccount, and the variable, to the console. Line twelve is left empty for legibility. In line thirteen, a constant variable named nickName is declared and assigned the data type null. In line fourteen, a console statement logs both the type of the variable named nickName, and nickName.

When the program runs, the interpreter assigns the keyword typeof to each of the variables in the console statements: in lines 2, 5, 8, 11, and 14. The result of the evaluation is that the console prints not the value of the variable in each instance, but instead first the type of variable in each instance. The type is followed by the value of the variable. The output is string Francis Bacon, number 59, Boolean true, undefined undefined, and object null.

The output of line 14 differs from the others in that the data type does not match the evaluation by the keyword typeof. In JavaScript, the null data type evaluates to an object. Null is considered its own data type nonetheless.

Multiple ways exist to convert strings to numbers and numbers to strings. An example is given below, in figure 31.

Figure 31: The conversion of a string to a number

```
1       const stringedInteger = “40”
2       console.log( typeof stringedInteger, stringedInteger );
3
4       const integer = Number( stringedInteger );
5       console.log( typeof integer, integer );
6
7       const alsoInteger = parseInt( stringedInteger );
8       console.log( typeof alsoInteger, alsoInteger );
```

In line one, a constant variable named stringedInteger is declared and assigned a string with the number 40 inside its parentheses. In line two, both the type of the variable named stringedInteger, and the variable itself, are logged to the console. In line four, a constant variable named integer is declared. The assignment is of a value that evaluates to the return statement of a global function called Number( ).

Note that a global function in JavaScript is like a built-in method, but also different in both concept and syntax. The difference in concept is that the members of the class of global functions are called in the global scope. The difference in syntax is that rather than being appended by dot syntax to a variable that it then modifies, instead a global function receives a variable as an argument in its call signature. An example is in line three, where the global function Number( ) is called, and it is assigned the variable named stringedInteger. In line five, the console statement logs both the type of the variable named integer, and the integer.

In line seven, a constant variable named alsoInteger is declared and assigned the value of an evaluation of the global function parseInt( ). Inside the call signature of the global variable parseInt( ) is the variable named stringedInteger. In line eight, both the type of the variable named alsoInteger, and the variable itself, are logged to the console.

When the program is executed, in line one, the interpreter assigns the variable named stringedInteger the value of a string with the number 40. In line two, the console logs the type of the variable and the variable: string and 40. In line four, the interpreter assigns the evaluation of the global function number with stringedInteger as its argument. The evaluation converts the string into a number. In line five, the interpreter logs to the console both number and 40.

In line seven, the interpreter assigns the evaluation of the global function parseInt( ) and its argument stringedInteger. The function parseInt( ) parses a string and returns the value as new number. The evaluation of parseInt( stringedInteger ) is assigned to the variable. In line eight, the interpreter logs to the console both the type of the variable named alsoInteger and its value.

When the program runs, the interpreter performs the various assignments and evaluations, and logs to the console the following: string 40, number 40, number 40. Note that the global functions Number( ) and parseInt( ) both convert the data type from string to number. The way in which each function does so is different. The Number( ) function converts the data type. The parseInt( ) function parses the string and returns a numerical copy of the number that the string contains. Despite the different routes, however, the result is the same.

The global variable that converts a string to a number is given in the following figure, figure 32.

Figure 32: The conversion of a number to a string

```
1    const integer = 50;
2
3    const stringedInteger = String( integer );
4    console.log( typeof stringedInteger, stringedInteger );
```

In line one, the constant variable integer is declared and assigned the numerical value 50. In line three,

a constant variable named stringedInteger is declared and assigned the evaluation of the global function String( ) with the argument in its call signature. The argument is the variable named integer. In line four, the console logs both the type of the variable named stringedInteger, and the variable stringedInteger.

When the program runs, in line one, the interpreter assigns creates the variable named integer and assigns it the value of the number 50. In line three, the interpreter creates the variable named stringedInteger and assigns it the evaluation of the global function String( ) and its argument integer. The global method String( ) converts the data type from number to a string and includes the characters 50. In line four, the console logs both the type of the variable named stringedInteger and the variable stringedInteger. The result is, string 50.

In addition to noting the type of a data type with the keyword typeof, and also the conversion of data types from one to another by global functions, a last point in this section is the consideration of automatic data type conversion that the interpreter performs during string concatenation.

String concatenation is the technical description of a concept that the reader encountered in the previous section, with the built-in method concat( ) and the plus operator + . While achieving string concatenation, the interpreter automatically converts numbers to strings if they can be included in the string. An example is given in figure 33.

Figure 33: An example of the automatic conversion of number to string by concatenation

```
1    const firstPhrase = “The grade on my test is a ”;
2    console.log( typeof firstPhrase );
3
4    const secondPhrase = 100;
5    console.log( typeof secondPhrase );
```

```
6
7       const thirdPhrase = firstPhrase + secondPhrase;
8       console.log( typeof thirdPhrase, thirdPhrase );
```

In line one, a variable named firstPhrase is declared and assigned a string data type with the characters The grade on my test: . Note the blank space after the colon at the end of the string. In line two, the console logs the type of the variable named firstPhrase. In line four, a variable named secondPhrase is declared and assigned a number data type with the value of 100. In line five, the console logs the type of the variable named secondPhrase.

In line seven, a variable named thirdPhrase is declared and assigned the evaluation of an addition operation. Because the operands are strings, so the plus sign concatenates the two strings together. Then the concatenated string is assigned to the variable named thirdPhrase. In line eight, the console logs the type of the variable named thirdPhrase and the variable to the console.

When the program executes, the interpreter makes the necessary evaluations and assignments. The output is the following: string, number, string The grade on my test is a 100. Note that the variable secondPhrase is of the number type. During concatenation in line seven, the interpreter incorporates the number into the string. Then in line eight, the number is identified by the keyword typeof as part of the string, and the data type of the variable thirdPhrase is string.

## Operators

As mentioned above, the plus and minus operators, + and – , can be used to achieve an incrementation of a numeric variable. If the operators are doubled, ++ and - - , the incrementation occurs by the integer one in either direction, an increase or decrease. If either of the operators are

paired with an equality operator, = , then the amount of incrementation is defined by the number that is assigned after the equality operator. An example of different types of incrementation with addition are given below in figure 34.

Figure 34: Variations in the syntax of incrementation

```
1       let integer = 1;
2
3       integer ++;
4       console.log( integer );
5
6       integer += 1;
7       console.log( integer );
8
9       integer += 2;
10      console.log( integer );
11
12      integer += 100;
13      console.log(integer);
```

In line one, a changeable variable named integer is declared and assigned a number data type with the value of 1. In line three, the double plus sign increments the value of the variable named integer by 1. In line four, the result is logged to the console. The output is 2. In line six, the variable named integer is incremented by one. The syntax of the line demonstrates that incrementation by one is achievable also with the plus – equality operator.

In line seven, the result of the incrementation is logged to the console. The output is 3. In line nine, the variable named integer is increase by an incrementation of two. In line ten, the value of the variable is logged to the console. The output is 5. In line twelve, the variable named integer is increased by an incrementation of 100. In line thirteen, variable is logged to the console. The output is 105.

Note that the variable is updated with each incrementation. The update of the variable is allowed with in the scope of the variable, which is defined in the keyword let declaration in line one. Note also that the program in figure 34, could have the incrementation decrease rather than increase. To do so, the plus signs are changed to minus signs. The reader is encouraged to experiment with the program by altering the incrementation from an increase to a decrease, and also by changing the numerical value of each incrementation.

Another variation of operation is observable in a statement that has multiple operations. A few examples of a statement with multiple operations, along with statements of single operations for comparison, are given in figure 35.

Figure 35: Statements with multiple operations

```
1       let integer = 1;
2       console.log( integer < 2 );
3       console.log( 0 < integer && integer < 2 );
4       console.log( integer < 2 && integer > 0 );
5       console.log( integer < 2 || integer > 10 );
6       console.log( integer !== 2 );
7       console.log( integer );
```

At this point, the reader is familiar with the declaration and assignment operations that have been

exposed to in the fastidiously explicated figures above. From here on in the book, such explications are omitted. The reader is advised to think them to themselves, as they read along. Instead of a discussion of the declarations and assignments, the new points of interest are highlighted and explained.

In line two, the evaluation has two operands and one comparison operator. The comparison evaluates to the Boolean true, which is what is logged to the console. In the example on line three, by contrast, the comparison has four operands and two comparison operators, separated by the and operator, which is explained momentarily. The interpreter evaluates the first comparison and then second. Only if both are true, does the interpreter return true. If either are false, then the interpreter returns false.

Since the variable named integer evaluates to 1, since the first comparison checks if 1 is greater than 0, which it is, and since the second comparison checks if 1 is less than 2, which it is, therefore both comparisons are true, and the interpreter returns true. The reader is advised to change one of the operands to make one of the statements false, as an experiment.

In lines four and five, two new operators are introduced. The first is the operator named and. Its symbol is &&. The second is the operator named or. Its symbol is || . The and operator checks that statements on either side of it are true. In line four, the and operator checks that integer is less than two and greater than zero. The interpreter learns from the and operator that both statements evaluate to true. So the interpreter returns true. In line five, the or operator checks that one of the two operands on its sides is true. If either one of them is true, then the return value is true, even if the other operand evaluates to false.

An example of this is the operation in line five. The evaluation of the comparison that integer is less than two returns true, but the evaluation of the comparison that integers is greater than ten returns false. The interpreter then checks the or operator checks to see if a comparison on either side returns true. The interpreter learns that the first comparison returns true, and so the interpreter evaluates the

statement to true. The console logs true.

Lines six and seven contain additional review that is included so that the reader can try out different syntax variations with the results as output.

The last set of examples about operators that is considered in this section concerns the modulus operator. The symbol of the modulus operator is % . Its process is that it divides a number by another and returns the remainder. Examples are given in figure 36.

Figure 36: The modulus operator

```
1    const integer = 100;
2    const integer_ = 101;
3
4    console.log( integer % 2 );
5    console.log( integer_ % 2 );
6
7    console.log( integer % 3 );
8    console.log( integer_ % 3 );
9    console.log( integer % 4 );
10   console.log( integer_ % 4 );
```

In line two, the variable named integer_ has an underscore at the end of its characters. An underscore at the end of a variable that otherwise has the same name as another variable, is a stylistic device that achieves a recognition of a difference within a similarity between the two variables. Here the similarity is that they are variables that are going to be used in the same way: as a dividend. The difference is that their values differ. The underscore can be included to differentiate function and class names as well.

In line two, the evaluation is a modulus operation on the variable named integer. The interpreter divides 100 by 2 and returns the remainder. The remainder is 0. In line three, the same operation is performed, but the dividend is 101. Since the remainder is 1, the modulus operator returns 1 to the invocation in the log statement. The output is 1. The other lines of the program demonstrate that the divisor in a modulus operation is adjustable.

## Variables

A brief section, the examples that elaborate variables contains a discussion of two declaration types, seen in figure 37.

Figure 37: Variations in the declaration of a variable

```
1    let i = 0, j = 0;
2    console.log( i, j );
3
4    i = j = 1;
5    console.log( i, j );
```

In line one, is a review of a declaration that economizes the keyword declaration by a sequential order of those variables that share the keyword. The reader is encouraged to add more variables that are declared with the keyword let.

In line four, is an example wherein the variables economize the similarity in the value of the assignment. Note that this is not a statement of declaration and assignment. It is a statement only of assignment. The variables have already been declared, in line one. In line four, each variable is

assigned the number data type with a value of 1.

Strings

As mentioned above, strings like arrays have an off by one indexation. As a result, the first index of a string is denoted by the integer 0. Because of this, the last index of a string, is not equal to the length of the string, but instead to the length of the string minus one. This fact is used to access the data points in a string during iteration. Examples of the technique are given in figure 38.

Figure 38: Variations in the indexation of a string

```
1    const instrument = "Circuit board";
2    console.log( instrument[ 0 ] );
3    console.log( instrument[ instrument.length – 1 ] );
4    console.log( instrument[ instrument.length – 2 ] );
5    console.log( instrument[ Math.floor( instrument.length / 2 ) ] );
```

In line two, the indexation achieves the evaluation of the first data point in the string that is assigned to the variable named instrument. The data point at index 0, is C. The letter is logged to the console.

In line three, a new syntax is given. The syntax demonstrates the way to achieve an indexation of the final index of a string. The notation of appendation achieves the evaluation of the length of the variable that it follows. From that length, the interpreter subtracts the integer 1, to accommodate off by one indices.

The integer that is now inside the brackets [ ] that follow the variable named integer, evaluated to 12 because the length of integer is 13. One is subtracted from the length. Index 12 of the variable

named integer is its last index, and the data point there is extracted and logged to the console. Lines four and five demonstrate variations in the mathematical operations that can be used to index parts of a string. In line four, the subtraction is a of a greater number.

In line five, the operation is not subtraction but division, which results in the middle character of the string. The Math.floor( ) built-in function returns the lower of the two integers if the quotient is a floating-point number. For example, Math.floor( 5 / 2 ), returns 2. Math.ceil( ) is its complement, that returns the higher of the two integers. For example, Math.ceil ( 5 / 2 ), returns 3.

Arrays

Like the indexation of a string, that of an array has variations that enable the access of the data points at different indices. The syntax for the indexation is similar to that of the string. Both place the index in brackets after the variable. The index can be either an integer or a statement that evaluates to an integer. Example of the variations in the indexation of an array are given in figure 39.

Figure 39: Variations in the indexation of an array

```
1    const integers = [ 1, 2, 3, 4, 5 ];
2    console.log( integers[ 0 ] );
3    console.log( integers[ integers.length – 1 ] );
4    console.log( integers[ Math.floor( integers.length / 2 ) ] );
5    console.log( typeof ( integers + integers ), integers + integers );
```

If the reader followed the examples from the previous figure, figure 38, then the examples in figure 39 are clear except for the example in line five. In line five, the mathematical operation that is performed

in the call signature of the keyword typeof is an addition whose operands are the variable named integers. The JavaScript interpreter considers the addition of two arrays a context that results in string concatenation. Therefore the output of line five is the string, 1,2,3,4,51,2,3,4,5 .

Because the string is not an integer, therefore it cannot be used to index a value in the variable named integers. The example is given in order to demonstrate to the reader that the evaluation of the operation statement inside the brackets must fall within the range of indices of the iterable. If the range of indices is either indexed by a string or at an index that is outside its range, then the result is undefined. If the indexation achieves string concatenation, then the result is a string.

Objects

A discussion of an object begins after the examples in figure 40.

Figure 40: The population of an object

```
1       const storage = {};
2
3       storage[ "Francis" ] = 1;
4       console.log( storage );
5
6       storage[ "Francis" ]++;
7       console.log( storage );
8
9       const storage_ = { };
10      storage_[ "Francis" ]++;
```

```
11      console.log( storage_ );
```

In line three, the string “Francis” is assigned as the key to the storage object. The number data type of integer 1 is assigned as the value of the key. In line four, storage displays as { “Francis”: 1 } . In line six, the indexation of the object by the key “Francis” access the value of the key. Once accessed, the value is then incremented by the double plus operator. In line seven, the updated storage object is logged to the console: { “Francis”: 2 } .

Lines nine and ten, demonstrate a requirement that a key-value pair is assigned to an object before its value can be incremented. The output from line eleven is NaN or not a number. This result indicates that an error occurred, which is that in line ten, the interpreter attempted to increment a non existent value.

There is a compact, syntactical form that allows either the insertion of a key into an object or the incrementation of its value if the key already exists. The form makes use of the or operator. An example of the syntax is given in figure 41.

Figure 41: An economical syntax to achieve the population of an object

```
1       const items = {};
2       items[ "Francis" ] = ( items[ "Francis" ] || 0 ) + 1;
3       console.log( items );
4
5       items[ "Francis" ] = ( items[ "Francis" ] || 0 ) + 1;
6       console.log( items );
```

In line two, the assignment is of a statement of addition. The statement evaluates to the value of the

contents of the parentheses plus one. Inside the parentheses, there is the object named items. There, the object named items is indexed by the string “Francis”. After the string is the or operator whose other operand is the integer 0.

When the interpreter encounters the statement inside the parentheses, the interpreter first checks if the object named items contains a key of a string with the letters Francis. The interpreter learns that no such key exists in the object name items. The interpreter goes to the other operand, and learns that the evaluation of the parentheses is therefore to be the integer 0. Next the interpreter adds 0 to 1 and assigns the sum as the value for the key “Francis”.

In line five, the statement is exactly the same as in line two. The difference is that during the evaluation of the parentheses, the interpreter finds that the updated object named items contains a key that is a string with the letters Francis. Therefore the interpreter accepts this operand as the evaluation of the parentheses. The evaluation of the indexation of the object named items by the key “Francis” is the value of the key, which is 1. The interpreter adds the value 1 to the integer 1 that is outside the parentheses. The result is that the value of the key “Francis” in the object named items is incremented by 1. The output that is seen from line six, shows the updated object: { “Francis”: 2 } .

Note the stages of the development of the object named items. In line one, it is declared. In line two, it is updated with a key-value pair. In line three, it is updated again, with the value incremented. Note also that the syntax that achieves the initial population of the object with the key-value pair and the incrementation of the value, is the same.

The concise syntax shown in lines two and five of figure 41 thereby contrast with that in line ten of figure 40 that attempts to increment a non existent value. With the or operation, the syntax in lines two and five of figure 41, can update either an object without the key-value pair that is used in the indexation, or an object that already has the key-value pair.

If-then statements

As discussed above, an if-then statement is a type of conditional logic that achieves a control flow in a program. A basic if-then statement is reviewed in figure 42.

Figure 42: An if-else statement

```
1       const integer = 20;
2       if ( integer < 10 ) {
3               console.log( "Less than 10" );
4       } else {
5               console.log( "More than 10" );
6       }
```

The if-then statement in figure 42 checks whether the variable named integer is less than 10 and then logs to the console whether if is less than or more than ten. The reader is familiar with the syntax that achieves the control flow.

The program can be elaborated by the introduction of the keyword else if . The keyword else if makes another comparison and offers a declaration block, before the else keyword and its declaration block. An example of the program in figure 42 with the addition of an else if statement, is given in figure 43.

Figure 43: An if, else-if, else statement

```
1       const integer_ = 20;
2       if ( integer < 10 ) {
```

```
3               console.log( "Less than 10" );
4       } else if ( integer < 100 ) {
5               console.log( "Less than 100" );
6       } else {
7               console.log( "More than 100" );
8       }
```

The program in figure 43 is the same as that in figure 42 except for the lines four, five, and seven. In line four, an else if statement follows the closing bracket of the declaration block of the if statement. In the parentheses, the else-if statement compares whether the variable named integer is less than the integer 100. Since the variable named integer has the assignment of the integer 10, the comparison returns true. Because the comparison is true, the interpreter enters the declaration block of the else-if statement and logs a message to the console.

Like the if-else statement, the if, else-if, else statement enters the first declaration block that evaluates to true. In the composition of if-else statements, therefore the reader is to write mutually exclusive conditions among the parentheses of the keywords if and else or if, else if, and else. Note that it is easy to confuse if-else statements with the else-if statement: the former is a combination of an if statement and and else statement. The latter is a conditional statement that mediates and if statement and and else statement. In this book, a hyphen is used in the phrases if-else statement and else-if statement, but not in the phrase of the keyword else if .

There is a compact syntax for writing the declaration block if the declaration block is one line and short. The syntax is to omit the curly braces and to write the declaration block after the closing parenthesis of the if and else-if statements and after the else keyword in the else statement. An example is given in figure 44.

Figure 44: An if-else statement with the curly braces omitted

```
1       const integer = 1;
2       let result = null;
3
4       if ( integer !== 0 ) result = 0;
5       else result = 4;
6       console.log( result );
```

In lines four and five, the declaration blocks of the keywords if and else, are included in the same lines as the keyword. Guides suggest to follow this style as long as the declaration block is short in length.

Above, the reader was recommended to practice his or her own if-else statements by, in one suggestion, the incorporation of an if-else statement inside the declaration blocks of an if-else statement. An example of this program is given in figure 45.

Figure 45: Nested if-else statements

```
1       const television = "On";
2       const show = "Sports";
3       const energySource = "Plugged in";
4
5       if ( television === "On" ) {
5               if ( show !== "Sports" ) {
6                       console.log( "Not sports" );
7               } else {
```

```
8                       console.log( "Sports" );
9               }
10      } else {
11              if ( energySource !== "Plugged in" ) {
12                      console.log( "Plug in the television" );
13              } else {
14                      console.log( "Off" );
15              }
16      }
```

Both the first declaration block, from lines five to ten, and the second declaration block, from lines ten to sixteen, include another if-else statement. The inclusion of a type or structure within the same type or structure, is called a nest. The elements are said to be nested. In the next chapter, about data structures, the reader will encounter nested data structures and nested control flow statements.

Loops

In this section, are a demonstration and discussion of the variations of the basic structures of the for loop and while loop. The focus is on two topics: the limitation of the iterator and the sub indexation of the iterable. To begin the analysis, consider figure 46.

Figure 46: A for loop that indexes each element in a string

```
1       let instrument = "Circuit board";
2       const l = instrument.length;
```

```
3
4       for ( let i = 0; i < l; i++ ) {
5               console.log( instrument[ i ] );
6       }
```

The reader has seen the syntax before. In line one, a variable is created. In line three, its length is assigned to the variable l. Note that the variable l is the letter l and not the integer 1. In line four, the keyword for begins a for loop. The first part of the parentheses, before the first semi-colon, declares a variable named i and assigns it the value of zero. The variable is the iterator. In the second part of the parentheses, the iterator is limited by the inequality operator. In the third part of the parentheses, the iterator is incremented by one. In line five, the iterator indexes the iterable to achieve a log to the console of the data point at the index of the iterator.

When the program runs, the iterator with its incrementation moves one by one along the iterable and extracts the data points at its current index. The iterator does this as long as it is less than l, which is the length of the iterable. When the iterator is incremented to the length of l, then the inequality comparison evaluates to false, and the for loop ends before entering the declaration block. So far this is a review.

To elaborate the review, consider the for loop of the program in figure 47, below.

Figure 47: A for loop with sub indexation

```
1       let instrument = "Circuit board";
2       const l = instrument.length;
3
4       for ( let i = 0; i < l - 1; i++ ) {
```

```
5               console.log( instrument[ i ], instrument[ i + 1 ] );
6       }
```

Note the limit of the iterator in the second part of the parentheses in line three: i < l – 1 . As the limit of the iterator in the previous example in figure 46 stopped the iteration once the iterator was equal to the length of the iterable, so the example in figure 47 has an iterator that stops once it is equal to the length of the iterable minus one. Said another way, once instrument[ i ] is equal to the string “d” at the end of the loop, then the iteration stops and the declaration block is not entered.

The reason that the loop in figure 47 stops one index earlier than the loop in figure 46, is that in line four in figure 47, the second value that is logged to the console is a sub index of the iterator. The sub index evaluates to the integer of the iterator plus one. In the example, the sub index thereby achieves an access to the final letter of the string, “d”, when the iterator is at the index of one less than the final letter of the string, which is the index with the data point “r”.

If the program were written without the limitation of the iterator as one less than the final index of the iterable, and if the sub index were kept, then the result is that during the iteration of the final index with the data point “d”, then the sub index attempts to access the index at one plus the index of “d”. Because the index of one plus the index of “d” is undefined, so the result is undefined.

The principle to learn from this example and to keep in mind as the reader composes for loops with sub indexation, is to note the distance of the sub index, and to limit the iterator to those indices of the iterable that are that much away from the final index that the iterator approaches. To include an undefined result in a program, is considered bad practice.

Here are some more examples of the limitation of an iterator to achieve sub indexation.

Figure 48: A for loop with sub indexation of two indices

```
1       let instrument = "Circuit board";
2       const l = instrument.length;
3
4       for ( let i = 0; i < l - 2; i++ ) {
5               console.log( instrument[ i ], instrument[ i + 1 ], instrument[ i + 2 ]  );
6       }
```

In line three, the limit of the iterator is the comparison of inequality between it and the length of the iterable minus two. The limitation of the iterable to this length, allows the sub indexation of the iterable to the length of two. A sub indexation is made in line five: instrument[ i + 2 ] .

Figure 49: A for loop and sub indexation with an inequality operation

```
1       let instrument = "Circuit board";
2       const l = instrument.length;
3
4       for ( let i = 0; i < l - 1; i++ ) {
 5              console.log( instrument[ i ] !== instrument[ i + 1 ] );
6       }
```

The limit of the iterator that is defined by the inequality operator in line four, is familiar to the reader. Also familiar is both the syntax on line five, which indexes and sub indexes the iterable by the index of the iterator and the index of the iterator plus one; and also the syntax of the inequality operation between the two operands.

With familiar syntax, the example demonstrates a new concept. The concept is a comparison

between each index and the next to see if there is a duplication of the data point at the indices. Such a comparison highlights the fact that sub indexation can be used to access the data points at indices during iteration and also to perform calculations on them once they are extracted. In the chapters on data structures and algorithms and programs, comparisons like this are common.

In figure 50, is another example that shows variation in the sub indexation during for loops. Instead of a sub index ahead of the iterator, the sub index is behind it.

Figure 50: A for loop with a sub index behind the iterator

```
1       let instrument = "Circuit board";
2       const l = instrument.length;
3
4       for ( let i = 1; i < l; i++ ) {
5               console.log( instrument[ i – 1 ], instrument[ i ] );
6       }
```

Note that in the first part of the parentheses in line four, the iterator is defined with an assignment of the integer 1. The result is that at the start of iteration in the for loop, the iterator begins at index 1 of the iterable. Because of off by one indexation, index 1 of the iterable is not the first index of the iterable, which is index 0. So in this example, the iterator starts at the second index which has the data point “i”.

Note also that the limit of the iterator is defined by the inequality operator in line four: i < l . The limit is set to the length of the iterable because the sub indexation occurs not on the index after the iterator but before it.

By starting in the for loop at index 1, the iterator is able to reach behind it during the first iteration to achieve sub indexation, without returning an undefined value. Then as it loops along the

iterable, with each iteration the interpreter first logs the iterator as it sub indexes the index behind it, and then the index where the iterator is. The result is the output: Ci, ir, rc, and so on, but displayed on new lines in the text editor without commas.

The next example shows a variation in limitation of the iterator, seen below in figure 51.

Figure 51: A variation in the definition of the limit of an iterator

```
1	let instrument = "Circuit board";
2	const l = instrument.length;
3
4	for ( let i = 0; i <= l - 1; i++ ) {
5		console.log( instrument[ i ] );
6	}
```

In line four, the second part of the parentheses, which is between the two semi colons, limits the iterator to a value less than or equal to the length of the iterable minus one. The result of this definition is that when the iterable is the length of the iterator minus one, it enters the declaration block. The value of l – 1 is the data point d. Because there is no sub indexation, therefore the iterator is able to log the data point to the console without an undefined data type.

Consider, however, if the limit were instead i <= l , without the subtraction of 1 from l. Then the iterator would enter the declaration block when it is at the length of the iterable, which because of off by one indexation is an undefined value. The index is an undefined value after the final letter of the string.

The point to learn from this example, is that if the limit is defined by a compound operation such as less than or equal to <=, then the case of equality is to be considered as the limit. If the index at

the limit is undefined, then the comparison operator needs to be modified by a subtraction of the length until a defined index of the iterable is found.

Said another way, both of the following syntaxes for limitation achieve an indexation without an undefined value: i < l and i <= l – 1 . Note that these syntaxes pertain to this example that does not have sub indexation. The reader is encouraged to introduce sub indexation as a practice problem, and try to set the limits of the iterator appropriately.

As an iterator is able to iterate the iterable in a loop that moves from index 0 to the last index, so an iterator can iterate in the opposite direction, from the final index to index 0, as in the example in figure 52.

Figure 52: An for loop that iterates the iterable from the last to first index, index 0

```
1       let instrument = "Circuit board";
2       const l = instrument.length;
3
4       for ( let i = l - 1; i >= 0; i-- ) {
5               console.log( instrument[ i ] );
6       }
```

In line four, the iterator is declared in the first part of the parentheses. The iterator is assigned the value of the integer that is the length of the iterable minus one. The result is that during iteration, the iterable begins at the final index of the iterable, in this case the index with the data point “d”. The limit of the iterator is that it is greater than or equal to zero. The result is that the final iteration that enters the declaration block is that which has the iterator at index 0, with the data point “C”. The incrementation of the iterator is a decrement by one, achieved by the two minus signs, -- .

Figure 53: A for loop that iterates the iterable from the last to first index, and sub indexes before it

```
1	let instrument = "Circuit board";
2	const l = instrument.length;
3
4	for ( let i = l - 1; i >= 1; i-- ) {
5		console.log(instrument[ i ], instrument[ i – 1 ] );
6	}
```

In the example in figure 53, the iterator is declared and assigned the value of the length of the iterable minus one. The iterator is then limited to the condition that it is greater than or equal to one. The result is that the final iteration that enters the declaration block, occurs when the iterator is at the index with data point “i”. Next the iterator is decremented by one.

In line five, both the data point at the index of the iterable and the data point at the index of the iterable minus one, are logged to the console. Because the iterator loops from the end to the beginning of the iterable, therefore the index that is at the iterator minus one is ahead of the iterator. The result is that the output with each iteration is the letter at the index of the iterator and then the letter at the index one before the iterator. In the first iteration, the output is the following: d r . Because the iteration is limited to the second index, which is denoted by the integer 1 and contains the data point “i”, therefore the iterator never sub indexes an index that lies beyond the iterable: there is never an undefined data type logged to the console.

With a for loop, an iterator can iterate and object as an iterable, just as it can a string or an array. In the following example in figure 54, an object is iterated.

Figure 54: The iteration of an object with a for loop

```
1	let lettersAndIntegers = { "a": 1, "b": 2, "c": 1, "d": 4 };
2	for ( let i in lettersAndIntegers ) {
3		console.log(i, lettersAndIntegers[ i ] );
4	}
```

In line one, a changeable variable is declared and assigned the value of an object that contains four key-value pairs. In line two, the syntax for the iteration of an object is introduced. There is the keyword for, which starts the loop. Next, inside the parentheses is a statement that declares the changeable variable i. After the declaration is the keyword in. They keyword in specifies that the following term is the iterable object that the iterator is to iterate. In this example, the iterable is the object lettersAndIntegers. The parentheses close and an opening curly brace marks the beginning of the declaration block of the for loop.

In line three, a console log statement logs two items. The first is the iterator, which evaluates to the key of each iteration: a, b, c, d. The second item is an indexation of the object by the iterator, which evaluates to the value of the key-value pair of that is at the location of the iterator. Therefore the output has the key and the value of each iteration: a 1, b 2, c 1, d 4 .

Like the limitation of a for loop can be adjusted in order to allow a sub indexation of a string or array without a return of an undefined data type, so that of a while loop can be adjusted for the same result. An example is given in figure 55.

Figure 55: A while loop with sub indexation

```
1	let instrument = "Circuit board";
2	const l = instrument.length;
```

```
3       let i = 0;
4
5       while ( i < l – 1 ) {
6               console.log(instrument[ i ], instrument[ i + 1 ] );
7               i++;
9       }
```

The limit is defined inside the parentheses in line two. The while loop enters the declaration block as long as it is less than the value of the length of the variable named instrument, minus one. The limit to one less than the length of the iterable allows the sub indexation of one greater than the iterator, which occurs in the second item logged to the console, in line six.

Loops like if-else statements, can be nested. Often a nested loop is used to achieve the iteration of a two dimensional array. The concept of a two dimensional array and a nested for loop are introduced in figure 56.

Figure 56: The iteration of a two dimensional array with a nested loop

```
1       const integers = [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ], [ 9, 10 ] ];
2       const l = integers.length;
3       for ( let i = 0; i < l; i++ ) {
4               console.log( integers[ i ] );
5       }
6
7       for ( let i = 0; i < l; i++ ) {
8               const l_ = integers[ i ].length;
```

```
9               for ( let j = 0; j < l_; j++ ) {
10                      console.log( integers[ i ][ j ] );
11              }
12      }
```

In line one, a constant variable named integers is declared and assigned a value of an array. Inside the array, each index has another array. An inside each array at each index, there are two integers. Because each array has an array, and because each array at each index contains not another array, but instead integers, therefore the array is two dimensional.

Here is another example of the same array, without the integers: const integers = [ [ ], [ ], [ ], [ ], [ ] ]; . A three dimensional array is if the first array contains an array at each index, and if each array at each index contains an array. Here is an example: const threeDimensionalArray = [ [ [ ] ], [ [ ] ] ] . There are only two indices in the first array and one index in each array at each index. Dimensionality is determined not by the number of arrays contained at a level, but instead by the number contained by the depth, which is consistent across levels.

The for loop in lines three through five, demonstrate the value that is extracted by the iterator. The output is each index, which is an array with integers: [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7,  8 ], [ 9, 10 ] . Note that a single for loop without sub indexation can access only the index of the first dimension of the two dimensional array. In order to access the second dimension, a nested for loop is required. The nested for loop in lines seven through twelve of figure 56 demonstrate the syntax that achieves the indexation of the second dimension.

In line seven, an iterable is declared, limited, and incremented. Note that the use of the letter i for the iterable is not a redeclaration of the variable. Even though the same letter was used in the first for loop in line three as is used in the second for loop in line seven, nonetheless because both

declarations have local scope, therefore the variables are limited to the respective for loops.

In line eight, a constant variable named l_ is declared and assigned the value of the evaluation of the length of the array at the current index of the iterator. In line nine, a second iterator is declared, limited, and incremented. In line ten, the a two dimensional indexation is logged to the console. In the call signature, the phrase, integers[ i ][ j ] , is the point that achieves the two dimensional indexation.

The indexation of integers[ i ] evaluates to the array at each index. In the first index, the data is the following: [ 1, 2 ] . The indexation of this value by the iterator j begins at index 0 of the value and continues until the iterator j is one less than the length of the array, denoted by l_ . The syntax evaluates to the integers 1 and 2, which are returned to the call signature of the console for logging. The iterator i then increments, selects the array at the next index, which is [ 3, 4 ], and the iteration of the second dimension by the iterator j begins again.

The reader is advised to practice the creation and iteration of arrays with varying dimensions. A common data storage format is called JSON. JSON is an acronym for JavaScript Object Notation. Although the format is not discussed here, nonetheless it is a common format of objects, arrays, and strings, with minor syntactical differences. Often JSON has deep nests for the structure of its storage. Looping through dimensions becomes a required capability.

The final example in this section is a nested loop. The outer loop is a for loop and the inner loop is a while loop. The example is given in figure 57.

Figure 57: A while loop nested in a for loop

```
1    const integers = [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ], [ 9, 10 ] ];
2    const l = integers.length;
3
4    for ( let i = 0; i < l; i++ ) {
```

```
5               const l_ = integers[ i ].length;
6               let j = 0;
7               while ( j < l_ ) {
8                       console.log( integers[ i ][ j ] );
9                       j++;
10              }
11      }
```

The example in 49 is like the example in 48, in terms of the indexation of the second dimension of the array. The reader is already familiar with the syntax of while loops. So the syntax of this example is a review, even if it is in an example that has a new combination.

Data transfer among a string, array, and object

In this section, examples are given to demonstrate ways to transfer data among a string, array, and object. The first example, given in figure 58, transfers data from a string to an array and then from a string to an object.

Figure 58: The transfer of data from a string to an array and to an object

```
1       const instrument = "Circuit board";
2       const l = instrument.length;
3
4       const items = [ ];
5       for ( let i = 0; i < l; i++ ) {
```

```
6               items.push( instrument[ i ] );
7       }
8       console.log(items);
9
10      const items_ = { };
11      for ( let i = 0; i < l; i++ ) {
12              if ( items_[ instrument[ i ] ] ) {
13                      items_[ instrument[ i ] ]++;
14              } else {
15                      items_[ instrument[ i ] ] = 1
16              }
17      }
18      console.log( items_ );
```

In line four, an empty array is declared, In lines five through eight, a for loop is used to populate the empty array with the indexed data points from the iterable that is a string. The built-in method push( ) is used to achieve the transfer. The method push( ) is appended to the array. The call signature of the push( ) method contains the statement whose evaluation is pushed to the array. In this example, the call signature contains an indexation of the string.

Note also that the push( ) method pushes the evaluation of its call signature to a new, last index in the array to which the method is appended. In this example, during the first iteration the array is empty, so the push( ) method pushes the first letter of the string to index 0 of the array. During the second iteration, the call signature evaluates to the letter “i” and pushes it to index 1 of the array.

In line ten, an empty object is declared. In line eleven, a for loop is declared to populate the

object with the items of the string. The if-else statement in lines twelve through sixteen, demonstrate another way to populate an object that can be empty during the first iteration and not empty during the subsequent iterations. It thereby complements figure 41 above.

In figure 58, in lines twelve and thirteen, the condition of the if statement is a two dimensional indexation. Note that unlike the previous example of two dimensional indexation, in which the each iterable was an array, in this example, the first iterable is an object and the second is a string.

The first indexation has the object as the iterable and the instrument index by the letter i as the iterator. The second indexation has the variable named instrument as the iterable and the letter i as the iterator. When the program runs and reaches this point, the interpreter first evaluates the nested indexation in line twelve.

The variable string is indexed by the iterator, and whatever data point that is stored at that index is the result of the evaluation. That data point then becomes the iterator for the first dimensional indexation. If the object contains the iterator, then the interpreter enters the declaration block. If the object does not contain the iterator, then the interpreter enters the declaration block of the else statement.

In line fifteen, in the declaration block of the else statement, assigns a key-value pair to the object. The interpreter enters the declaration block of the else statement because the object is empty and does not have any keys to match the evaluation in the conditional comparison.

Then in subsequent iterations, the object might contain the value of the result of the second dimensional indexation, in which case the declaration block of the if statement is entered, and the value is incremented by one. In this example, there are two instances in which the object already contains the key. They are at index 5 with the letter “i” and index 11 with the letter “r”. Note that each letter had been encountered by the iterator in a previous iteration, when it was entered into the object by the declaration block of the else statement.

The transfer of data from an array to a string and then from an array to an object are given in figure 59.

Figure 59: The transfer of data from an array to a string and to an object

```
1	const integers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
2	const l = integers.length;
3
4	let items = "";
5	for ( let i = 0; i < l; i++ ) {
6		items += integers[ i ];
7	}
8	console.log( items );
9
10	let items_ = {};
11	for ( let i = 0; i < l; i++ ) {
12		items_[ integers[ i ] ] = ( items_[ integers[ i ] ] || 0) + 1;
13	}
14	console.log( items_ );
```

The concepts and syntax of the examples in figure 59 are mostly familiar to the reader. Lines four through eight transfer the data from the integer to an empty string. Line six contains a new syntactical form. The combination of a plus with an equals sign, has been seen before, in the incrementation of an integer. By contrast, here the combination of punctuation achieves string concatenation.

In line twelve, two compact forms of syntax for a two dimensional indexation and a population

of an object, achieves the transfer of data from the string to the object.

The last figure in the section, figure 60, contains an example of the transfer of data from an object to first a string and then from an object to an array.

Figure 60: The transfer of data from an object to a string and to an array

```
1       const lettersAndIntegers = { "a": 1, "b": 2, "c": 1, "d": 4 };
2
3       let items = "";
4       for ( let i in lettersAndIntegers ) {
5               items += i;
6               items += lettersAndIntegers[ i ];
7       }
8       console.log( items );
9
10      const items_ = [];
11      for ( let i in lettersAndIntegers ) {
12              items_.push( i );
13              items_.push( lettersAndIntegers[ i ] );
14      }
15      console.log( items_ );
```

The concepts and syntax of figure 60 are familiar to the reader. The application of them to achieve the transfer of data is new. Lines four through seven transfer the data from the object to the string. Lines eleven through fourteen transfer the data from the object to the array. If the reader did not read the

commentary for figure 58, and is confused about the push( ) method, check there for an explanation.

## Functions

In this section, two functions are considered. The first function, like Hello, World!, is popular. The function is called fizzBuzz, and it is introduced in figure 61.

Figure 61: fizzBuzz

```
1       function fizzBuzz( integer ) {
2       const result = [ ];
3
4       for ( let i = 1; i <= integer; i++ ) {
5               if ( i % 3 === 0 && i % 5 === 0 ) {
6                       result.push( "Fizz buzz" );
7               }  else if ( i % 3 === 0 ) {
8                       result.push( "Fizz" );
9               } else if ( i % 5 === 0 ) {
10                      result.push( "Buzz" );
11              } else {
12                      result.push( i );
13              }
14      }
15       return result;
16      }
```

```
17
18      console.log( fizzBuzz( 100 ) );
```

The prompt of fizzBuzz is to print the integers from one until a given integer with three provisions. The first provision is that if the integer is divisible by three, then in place of the integer, the word fizz is printed. The second provision is that if the integer is divisible by five, then in place of the integer, the word buzz is printed. The third provision is that if the integer is divisible by three and five, then in place of the integer, the word fizzBuzz is printed.

The modulus operator, which was introduced in figure 36, is used to achieve a determination of the conditions. To check the divisibility of the integer by three and five, an and operator is used in line five. The rest of the syntax and concepts are familiar to the reader.

The order of the condition that checks for the divisibility of three and five must precede the other two, whose relative order with each other is variable, but whose combined order with respect to the last condition, the else block, is immutable. The order of the program is so fixed because the conditions that check for single divisibility, either by 3 or 5, would accept those numbers that evaluate to true, but also that evaluate to true for the combined condition.

Said another way, all the integers of 3 that are also divisible by 5 would be added to the result by the declaration blocks of either 3 or 5 instead of the declaration block that contains both of them.

If the condition for the console log of the integer preceded the conditions of evaluation for divisibility, then the declaration block for the integer would be entered with each iteration, bypassing the those declaration blocks that could satisfy the requirements of the prompt.

One other note is that often the fizzBuzz prompt asks that the program log the result to the console with each iteration. A principle about programs, however, is that they are to have a return value. Therefore, the method that is chosen here, is to use the push( ) method to collect the results of

the iteration in an array and then to return the array. The evaluation of the invocation of the function, in line eighteen, is logged to the console.

The second example of a function, which is covered in this section, demonstrates that a function can call another function during its execution. Often the other functions are colloquially called helper functions. It is also common that the helper function has its name in common with the function that it helps, with the addition of an underscore at the end of the name.

An example of a program with a function and a helper function is given in figure 62.

Figure 62: A function and helper function

```
1       const input = 100;
2
3       function minusTwo( ) {
4               const integer = input - 1;
5               return minusTwo_( integer );
6       }
7
8       function minusTwo_( integer_ ) {
9               const result = integer_ – 1;
10              return result;
11      }
12
13      console.log( minusTwo( ) );
```

While the syntax of the example is familiar to the reader, the steps of the execution are new and are

explicated here. The program begins to run on line thirteen with the invocation of the function oneLess( ) by the call signature of the console log statement.

The function oneLess( ) receives zero arguments as parameters. The interpreter enters the declaration block of the function. In line four, the interpreter creates a constant variable named integers and assigns it the evaluation of a subtraction statement. The interpreter accesses the global variable input to evaluate its value, which is 100. The interpreter then subtracts 1 from 100 and assigns 99 to the variable named integer.

In line five, there is the keyword return that returns an invocation of the helper function, which has the variable integer as an argument. In line eight, the helper function minusTwo_ receives the argument integer as a parameter. Note that during the acceptance of a parameter, the name of the parameter can be changed to a name that is different from the name that is in the argument. In this example, a minor change is made with the addition of an underscore at the end of the parameter name, to distinguish it from the argument but also show the consistency of value that obtains in the argument to parameter transformation.

In line nine, a variable named result is declared, in local scope. The variable is assigned the value of the variable named integer_ minus one. Because the value of integer_ is 99, therefore the value that is assigned to result is 98. In line ten, the return keyword returns the variable result to the location of the invocation of the function. The location is the calls signature of the invocation in line five. The call signature now evaluates to 98. The function minusTwo( ) then returns the variable named result with the value 98 to the location of the call signature of its invocation, which is in line thirteen. The call signature now evaluates to the variable result with the value of 98. The console logs 98 as output.

Note that an invocation of a helper function might not follow the keyword return. Instead it might be the assignment for a variable, or also as another action that has a purpose without concern for the first function.

In this example, however, if the keyword return were omitted in line five, then the return value result from the function minusTwo_( ) would remain there and the console log statement evaluates to the undefined data type.

The principle is to focus on the transmission of data from function to function, with attention given to the action that is to be taken on the data at that point such as by a keyword or variable assignment.

Class

In this section, a demonstration of a program with a class, is followed by commentary. The class is the longest program yet discussed. The syntax if familiar to the reader. As with the previous example, the commentary focuses not on an explanation of the syntax, but instead on the process of the program as it runs. The example is given in figure 63.

Figure 63: A class with four methods

```
1       class Subtraction {
2               constructor( integer ) {
3                       this.minuend = integer;
4               }
5
6               subtractByLoop( ) {
7                       const result = [ ];
8                       for ( let i = this.minuend; i > 0; i-- ) {
9                               result.push( i );
```

```
10              }
11              return result;
12      }
13
14      subtractByInput( data ) {
15              const result = this.minuend - data;
16              return result;
17      }
18
19      subtractAndAdd( subtrahend, addend ) {
20              const result = [ ];
21
22              const difference = this.minuend – subtrahend;
23              result.push( difference );
24
25              const sum = this.subtractAndAdd_( addend );
26              result.push( sum );
27
28              return result;
29      }
30
31      subtractAndAdd_( addend_ ) {
32              const result = this.minuend + addend_;
33              return result;
```

```
34              }
35      }
36
37      const integers = new Subtraction( 100 );
38
39      console.log( integers.subtractByLoop( ) );
40      console.log( integers.subtractByInput( 20 ) );
41      console.log( integers.subtractAndAdd( 50, 100 ) );
```

When the program is run, the interpreter reads the abstract class into the memory of the computer from lines one to thirty-five. In line 37, the interpreter assigns a new instance of the class to the variable named integers. The class instantiated inside the variable named integers is initialized with the number 100 as a value assigned to the attribute this.minuend.

A side note is that the pieces of a subtraction problem are called minuend, subtrahend, and the difference. For example, in the statement, 5 – 4 = 1 , 5 is the minuend, 4 is the subtrahend, and 1 is the difference.

In line 39, the evaluation of its call signature, is logged to the console. Inside the parentheses of the call signature, is the variable named integers that has the new Subtraction class assigned to it. By dot syntax, the method subtractByLoop( ) is invoked.

The method is located in line six. It has no parameters, so the interpreter enters the declaration block. In line seven, the interpreter creates a new variable and assigns it an empty array. Lines eight through ten are a for loop that populates the array with integers beginning from the minuend, which has the value of 100, to the limit of the variable i, which is that it is greater than zero. The result of the limit is that the final iteration that evaluates as true in the condition, is when the value of i is 1. When the

value of i is 0, the condition evaluates as false, and the loop ends before entering the declaration block.

In line eleven, after the loop, the result is returned to the location of its invocation, in the call signature of the console log statement in line 39. The evaluation of the call signature evaluates to the value of the variable result, which is an array with the integers from 100 to 1 in a decreasing order by one.

In line forty, the interpreter invokes the subtractByInput( ) method of the variable integers that still has the class Subtraction assigned to it. The call signature of the method has a number data type with the value of the integer 20. The interpreter takes the argument to the location in the class where the method is declared, in line fourteen.

In line fourteen, the argument is accepted as a parameter named data. Then the interpreter enters the declaration block of the method. The interpreter creates a variable named result. The assignment to the variable named result is the evaluation of a subtraction statement between the class attribute this.minuend, which is still 100, and the input data, which is 20. The interpreter assigns the variable result a number data type with the value of 80. In line sixteen, the interpreter returns the result to the location where it was invoked, in line forty. The call signature now evaluates to 80. The console logs 80.

In line forty-one, the interpreter enters the call signature of the console log statement. Inside the parentheses of the call statement, the interpreter finds the integers variable with a method appended to it. The method is subtractAndAdd( ). The method subtractAndAdd( ) has two arguments, 50 and 100. The interpreter takes the arguments to line nineteen.

In line nineteen the arguments are accepted. The argument 50 becomes the parameter subtrahend. The argument 100 becomes the parameter addend. The interpreter enters the declaration block of the method. In line twenty, a variable named result is declared and assigned an empty array. In line twenty-two a variable named difference is declared and assigned the value of a subtraction

statement with the minuend as the class attribute this.minuend, which still evaluates to 100, and the variable named subtrahend, which evaluates to 50, as the subtrahend. The difference of 100 – 50 is 50, which is assigned to the variable named difference.

In line twenty-three, the interpreter pushes the variable named difference to the array that is the variable named result.

In line twenty-five, the interpreter declares a variable named sum and assigns it the evaluation of a statement that is a helper function named subtractAndAdd_ .

Note that the keyword this is prepended by dot syntax to the helper function. In principle, whenever a method of a class is invoked within another method of the class, the keyword this is prepended by dot syntax in order to allow the current method access to the class methods, from which it specifies the next method by the appendation in dot syntax.

In line twenty-five still, the helper function subtractAndAdd( ) has an argument that is the parameter named addend. The interpreter takes the argument to the location where the helper function subtractAndAdd_ is declared, in line thirty-one.

In line thirty-one, the addend argument becomes a parameter named addend_ . Next the interpreter enters the declaration block of the method. In line thirty-two, a constant, local variable named result is declared and assigned a number data type with the value of a statement of addition. In the statement of addition, one of the addends is the class attribute this.minuend, which still evaluates to 100, and the other addend is the parameter named addend_ , which evaluates to the integer 100. Since 100 + 100 is 200, the interpreter assigns a number data type with the value of the integer 200 to the variable named result. In line thirty-three, the interpreter returns the variable named result to the location where it was invoked, which is the call signature of the helper function in line twenty-five.

In line twenty-five, the interpreter assigns the evaluation of the helper function subtractAndAdd_( ), which is 200, to the variable named sum. In line twenty-six, the interpreter pushes

the variable named sum into the variable named result. Since the variable named result already had the variable named difference pushed into it on line 23, so the variable named sum is pushed into index 1.

In line twenty-eight, the interpreter returns the variable named result to the location of the invocation of the method, which is the call signature of the console log statement in line forty-one. The call signature now evaluates to an array with two elements, [ 50, 100 ] . The interpreter logs the array to the console.

Because the reader recognizes the syntax and concepts of the class, therefore it is advised that he or she pays attention to the locations of invocation and the methods that receive arguments as parameters and return variables to the invocation. The recognition of those dynamics, enables a comprehension of the class with its attributes and methods.

Built-in methods

This section focuses on built in methods that achieve the transition of data between a string and an array. The first set of examples concerns the transformation of data from a string to an array, seen in figure 64.

Figure 64: Variations of the split( ) built-in method to transfer data from a string to an array

```
1    const sentence = "Mars is the fourth planet from the Sun.";
2
3    const sentence_ = sentence.split( );
4    console.log( sentence_ );
5
6    const letters = sentence.split( "" );
```

```
7       console.log( letters );
8
9       const words = sentence.split( " " );
10      console.log( words );
```

In line three, the interpreter declares a variable named sentence_ and assigns it the evaluation of the variable named sentence with the built-in method split( ) appended on. The call signature of the built-in method is empty. The evaluation of the built in method is the entirety of the string. The split( ) method then pushes the new string into an array at the index 0 of the array. The interpreter then assigns the array with the string that contains the sentence to the variable named sentence_ . The output is logged to the console in line four.

In line six, the interpreter creates a variable named letters and assigns it the evaluation of the variable named sentence with the split( ) method appended on. In this example, the call signature of the split( ) method has a pair of quotation marks without a space in between them. The evaluation of the built-in method split( ) is an array with each letter from the variable named sentence as an index in the array with the data type string. The result of the evaluation is assigned to the variable named letters. In line seven, the interpreter logs letters to the console.

In line nine, the interpreter creates a variable named words and assigns it the evaluation of the variable named sentence with the split( ) method appended on. The call signature has a pair of quotation marks that have a space between them. The evaluation of the built-in method is that each word of the variable sentence is push to a new array as an index. The interpreter then assigns that evaluation to the variable named words. The interpreter logs the variable named words to the console in line ten.

A brief summary is that the built-in split( ) method acts on a string to convert the data into

strings in an array. If the call signature is empty, the entire string is push to index 0. If in the call signature there are quotation marks without a space, then letters of the sentence are pushed to the array. If the quotation marks in the call signature have a space, then words of the sentence are pushed to the array.

As with the string, there is a set of built in methods to convert the data of an array to a string. A demonstration of the built-in methods join( ) with variations in its call signature, are contained in figure 65.

Figure 65: Variations of the join( ) built-in method to transfer data from an array to a string

```
1       const sentence = [ "Fortran", "was", "the", "first", "modern", "programming", "language" ];
2
3       const commas = sentence.join( );
4       console.log( commas );
5
6       const noSpaces = sentence.join( "" );
7       console.log( noSpaces );
8
9       const sentence_ = sentence.join( " " );
10      console.log( sentence_ );
```

The reader is familiar with the syntax and concepts in figure 65, which closely follow those of figure 64. Instead of an explanation of each line, here is a summary of the result of the operations and logs to the console. A built-in join( ) method with an empty call signature creates a string with a comma between each item that was at a separate index in the array. A built-in join( ) method with a pair of

quotation marks that do not have a space between them, creates a string with every index placed one after another without a comma in between the items. A built-in join( ) method with a pair of quotation marks and a space between them, as is seen in line nine, results in a string that has a space between what had been every index in the array.

At this point, the reader knows built-in methods for data transfer, and also ways to achieve transfer without built-in methods, as seen in figures 64 and 65.

**Conclusion**

JavaScript is a language that was created in 1995 to facilitate the interaction of a visitor with a web browser. Since then, the language has undergone transformations and acquired new characteristics. The grammar that is given above, considers many but not all of the characteristics of the contemporary language. Readers can learn more about the developments at websites like the w3schools and the Mozilla Developer Network.

Topics of further study include arrow functions, the this keyword, and other forms of control flow such as async-await and try-catch. The reader also might study the traversal and manipulation of the document object model, which is a web development paradigm that allows JavaScript to respond to activities on a web page, such as the submission of an input that can contain, for example, a user name and password.

Yet other topics to research include dozens of other built in methods and keywords. Also, in terms of the structure of the language, the reader might look into the prototype chain. It is the backbone of JavaScript that allows inheritance among objects. So too it differentiates the language from others such as C++ and Java that have class inheritance.

The intent of this chapter was not a comprehensive review of the syntax of JavaScript, but

instead a review of its crystalline structure. With this knowledge, the reader is able to read other documentation and learn new syntax and concepts.

The next chapter introduces data structures, which although a substantial subject in its own right, nonetheless is another preparatory chapter for the fourth chapter, about algorithms and programs. The fourth chapter combines this chapter, about JavaScript, and the next chapter, in terms of syntax and concepts.

Chapter 3

Data structures

**What is a data structure?**

A data structure is a container of information. The types of data structures include string, array, object, linked list, stack, queue, graph, and tree. Because the structure and iteration of a string, array, and object were addressed in the previous chapter, therefore they are not discussed in this chapter other than the creation of a hash table, which is the first example below. Instead, the other data structures are the focus of the chapter. In the next chapter, about algorithms and programs, the data structures are utilized in the solutions of technical problems.

**What is a hash table?**

A hash table is synonymous with object { } . Another name for the data structure is dictionary, which is popularized by the Python programming language. Hash table is name that is used in the Java language. Because many of the languages have data structures in common, but call them different names, the reader is advised to familiarize himself or herself with the synonyms so that during discussion with programmers who use other languages, the reference is clear.

As is noted above, in JavaScript the word object has multiple definitions. An object is a data structure { } . It is also the form that generates many other forms. It can be said that a function is an object, a class is an object which evaluates in the typeof operator as function, and so on. The multiplicity of references in JavaScript gives the synonym hash table the allure or referential clarity. In this section, a hash table is demonstrated and discussed. After this section, however, the data structure

whose punctuation is { } , is again referred to as an object, in order to maintain discursive normality within JavaScript.

The reader is already familiar with the quality of a hash table. Above, figure 13 demonstrates the basics of its syntax: unique keys paired with values, separated by commas and between two curly braces. The reader also knows the process of the population of a hash table by assignment, as well as the access of its content by iteration and indexation.

In this section, a hash table is constructed from scratch, using arrays and strings. A class is the data structure that holds the attributes and methods that create the hash table. At first this idea is perplexing. How can a class be the form of a hash table, if a hash table is a hash table? To conceptualize the perplexity, consider the hash table not as a set of punctuation marks but instead a set of capabilities.

If the hash table is given a key that is not stored, then it can store the key with its value. If the hash table is given a key that it already has, then it can re-write the value of that key. If the hash table is indexed by an iterator, it can return either a key or a value.

JavaScript gives its object { } these capabilities and allows a programmer to utilize them by the JavaScript syntax. Such utilization, however, does not prevent a programmer from developing the capabilities from scratch, using a class form in order to hold the attributes and methods. It is by those methods that the procedures of population of the hash table and retrieval of data, are performed.

In figure 66, there is an example of a hash table.

Figure 66: A hash table

```
1       class HashTable {
2               constructor( size = 200 ) {
3                       this.size = size;
```

```
4                   this.array = [ ];
5           }
6
7           hash( str ) {
8                   let resultString = "";
9                   const l = str.length;
10                  for ( let i = 0; i < l; i++ ) {
11                          resultString += str.charCodeAt ( i );
12                  }
13                  return Number( resultString ) % this.size;
14          }
15
16          set( key, value ) {
17                  const index = this.hash( key );
18                  const item = [ key, value ];
19                  const subArray = this.array[ index ];
20
21                  if ( !subArray ) this.array[ index ] = item;
22                  else {
23                          let keyExists = false;
24                          const l = subArray.length;
25                          for ( let i = 0; i < l; i++ ) {
26                                  if ( subArray[ i ] === key ) {
27                                          keyExists = true;
```

```
28                                      subArray[ 1 ] = value;
29                                  }
30                              }
31
32                              if ( !keyExists ) {
33                                  subArray.push( item );
34                              }
35                          }
36                      }
37
38                      get( key ) {
39                          const index = this.hash( key );
40                          const subArray = this.array[ index ];
41
42                          if ( !subArray ) return null;
43                          else {
44                              const l = subArray.length;
45                              for ( let i = 0; i < l; i++ ) {
46                                  if ( subArray[ i ] === key ) {
47                                      return subArray[1];
48                                  }
49                              }
50                              return null;
51                          }
```

```
52              }
53      }
54
55      let hashTable = new HashTable( );
56
57      hashTable.set( “Fortran”, 12345 );
58      hashTable.set( “Fortran”, 678910 );
59
60      console.log( hashTable );
61      console.log( hashTable.get( “Fortran” ) );
```

A brief overview of the hash table in figure 66 is that it is a class with a constructor that has two attributes. The class also has three methods. In line two, is a piece of new syntax for the reader. The parameter has an equality statement.

The reader recalls that a constructor initializes the attributes of a class whenever a new instance of the class is declared and assigned to a variable, as happens in line fifty-five. The equality operation in line two, automates the value of the attribute this.size to the number data point with a value of 200. This occurs unless the invocation of the creation of a new instance of a class passes as an argument another value. For example, if line fifty-five instead had this statement, let hashTable = new HashTable( 400 ); , then the value of the attribute this.size would be 400. Because no such argument is passed, the equality statement evaluates to 200 and determines the value of the attribute this.size.

Another piece of new syntax is the built-in method charCodeAt( ) in line eleven. The built-in method charCodeAt( ) returns a numerical value for the value of a character in a string, whether it is a letter or a number. The schema that underlies the correspondence is called ASCII. The correspondence

between the value of string characters and numbers allows the developer advantages, such as in this method. Having populated the variable named result with the numerical values of the variable named str, then a modulus operator reduces the value of the string to a remainder by the attribute this.size. In the hash table, this number is used as the index of storage for a key-value pair.

With the pieces of new syntax and basic concept of the hash table outlined, now here is a walk-through of the program.

The interpreter reads lines one through fifty-four into memory. In line fifty-five, the interpreter creates a new instance of the class named HashTable and assigns it to a variable named hashTable. In line fifty-seven, the variable named hashTable has its set method appended on with a call signature that contains two arguments, the first a string and the second an integer.

The interpreter takes the arguments from the invocation of the method into the class of the new instance in the variable hashTable. In the class, the interpreter finds the start of the method declaration, in line sixteen. The arguments are turned in parameters. The key is “Fortran”, and the value is 12345. In line 17, the interpreter finds the method invocation of this.hash( ), which is the assignment for the declaration of the variable named index. The interpreter takes the variable named key to the method declaration of this.hash( ), in line seven.

In line seven, the variable key is accepted as a parameter named str. In lines eight through thirteen, a result string is populated with the character codes of the values of the variable str and is divided by the modulus operator to give a remainder value that is returned as the result of the method. The value of the return statement is 72. The interpreter takes that result to the location of the invocation of the method, which becomes the evaluation of the assignment side of the equality operation in line seventeen.

In line seventeen, the interpreter assigns the value of the return statement to the variable named index. In line eighteen, a variable named item is created as an array with two indices: the first is the

parameter key and the second is the parameter value. In line nineteen, the interpreter creates a variable named subArray, which is the indexation of the attribute this.array by the variable named index.

Line twenty-one assigns the variable named item to the attribute this.array at the index of the variable named index, if the variable subArray is null. The statement, ( !subArray ), uses an exclamation mark ! to negate the variable. If the variable named subArray is null, then the negation of the variable evaluates as a true statement and the declaration block of the if statement is entered. This process is what happens at this time in this example.

After the insertion of the variable named item in the attribute named this.array, which is indexed by the variable named index, the interpreter returns to the location of the invocation of the method, in line fifty-seven. Before continuing with the program, note the attribute this.array is indexed by the variable named index. Because the variable named index was assigned a value of 72 in the method this.hash( ), therefore the location of the insertion of the variable named item is at index 72 of the the attribute this.array. The index of insertion is different than the variable named index: the index of insertion is an index of the array, while the variable named index is a variable that has the index integer assigned to it so that it can be used to index the index of insertion. If ever the reader becomes confused in a program when a variable has the name of some other aspect of the program, consider the data type and use of the variable in relation to the aspect.

Also note that the data type of the variable named item is an array with two data points, each a different type: a string and an integer. Therefore, at this point in the program, the class attribute this.array is a two dimensional array with one data point at index 72.

In line fifty-eight, the interpreter finds another set method of the variable hashTable. The call signature contains two arguments, the string “Fortran” and the integer 678910. Already the reader might curious about the key. Since a hash table is a data structure with unique keys, therefore the reader knows that the repetition of the key “Fortran” indicates not the submission of a new key-value pair, but

a redefinition of the value of the key “Fortran” already in the hash table. The value will be updated from 12345 to 678910.

The interpreter takes the arguments from the call signature of the invocation of the method, into the class that is instantiated in the variable named hashTable. In line sixteen, the arguments become parameters. The reader is familiar with the processes in lines seventeen through twenty-one. One exception is that instead of entering the declaration block of the if statement in line twenty one, instead the interpreter sees the evaluation of the condition as false and proceeds to the else declaration block in line twenty-two.

In line twenty-three, a variable named keyExists is declared and assigned the Boolean value false. In lines twenty-four through thirty, the variable subArray is iterated. During the iteration, a comparison between the indexed data point and the parameter named key is carried out. In this example, the comparison evaluates to true on the first iteration since the only key-value pair at the index is [ “Fortran”, 12345 ] . In the declaration block of the conditional statement, in line twenty-seven, the value of the variable named keyExists is set from false to true. In line twenty-eight, index 1 of the array at the index 72 of this.array is indexed, and its value changes from 12345 to 678910.

The interpreter goes to line thirty-two and finds that the condition of the else statement evaluates to false. The interpreter completes the method and returns to the invocation of the method in line fifty-eight.

In line sixty, the interpreter logs the variable named hashTable to the console so that the reader can observe its structural characteristics and content there.

In line sixty-one, the interpreter finds a console.log( ) statement that has in its call signature the variable named hashTable with its get method appended. The method has a call signature with one argument, which the interpreter takes into the instance of the class to find the declaration of the method, in line thirty-eight.

In line thirty-eight, the argument is accepted as the parameter key. In line thirty-nine, the interpreter first takes the parameter key to the class method this.hash( ) and returns with a value that is assigned to the variable index. In line forty, a variable named subArray is created by the indexation of the class attribute this.array by the index. The reader knows that the value of the variable named subArray, at this time is another array with the data points [ “Fortran”, 678910 ] .

In line forty-two, the interpreter finds that the evaluation of the condition of the if statement is false. The interpreter does not enter the declaration block in line forty-two, but instead goes to line forty-three and enters the declaration block of the else statement. During the first iteration, the interpreter finds that the comparison operation between the index of subArray[ i ] and key evaluates to true. In line forty-seven, the interpreter returns the indexation of subArray[ 1 ] to the invocation of the method.

Note the counterfactual circumstance in which the interpreter completes the iteration of the subArray and does not find a comparison that evaluates to true. In that hypothetical case, then the interpreter continues to line fifty and returns null to the invocation of the method.

In the example, however, the interpreter does find a comparison that evaluates to true. Now in line 61, the value that is returned by the interpreter from the method evaluates to 678910. The call signature is logged to the console and the program terminates.

In addition to having storage capability, the hash table also has constant lookup time. This is an aspect of the hash table data structure that makes it useful in the composition of algorithms with fast asymptotic time complexity.

**What is a linked list?**

A linked list is a data structure that comprises nodes and links. The nodes contain both data and

a link that points to the next node. Typically, a linked list is a linear structure. The first node is called the head of the linked list. The last node is called the tail of a linked list. The nodes in the middle are called the middle nodes of a linked list. Here is a picture that might help the reader conceptualize a linked list, with the data in quotation marks and the link as a pointer:

head, “a” → middle node, “b” → middle node, “c” → tail, “d”

The head of the linked list stores a data point of a string with the character a. The head links to the first middle node. The first middle node stores the string “b” and points to the second middle node. The second middle node stores the string “c” and points to the tail. The tail stores the string “d” and has no link because it is the end of the linked list.

As the hash table was encoded by a class, so too is the linked list. It is common to use a class for the creation of data structures because the class holds attributes in common among methods. Each method can specify a task, such as adding or removing an item. The structures required for a linked list are the nodes, data, and pointers. In the example in this section, two classes are used: a class to encode the nodes and a class to encode the linked list with data and pointers.

The program for a linked list is given in figure 67.

Figure 67: A linked list with add( ) and remove( ) methods

```
1       class Node {
2               constructor( data, next ) {
3               this.data = data;
4               this.next = next;
5               }
```

```
6       }
7
8       class LinkedList {
9               constructor( ) {
10                      this.head = null;
11                      this.tail = null;
12              }
13
14              add( data ) {
15                      if ( !this.head ) {
16                              this.head = new Node(data);
17                              this.tail = this.head;
18                      } else {
19                              let node = new Node( data );
20                              this.tail.next = node;
21                              this.tail = this.tail.next;
22                      }
23              }
24
25              remove( value ) {
26                      let slow = this.head;
27                      let fast = this.head.next;
28                      while ( fast ) {
29                              if ( slow.data === value ) {
```

```
30                                  this.head = slow.next;
31                                  return slow = null;
32                              } else if ( fast.data === value ) {
33                                  return slow.next = fast.next;
34                              } else {
35                                  fast = fast.next;
36                                  slow = slow.next;
37                              }
38                          }
39                  }
40      }
41
42      const list = new LinkedList;
43
44      list.add( 1 );
45      list.add( 2 );
46      list.add( 3 );
47      list.add( 4 );
48      list.add( 5 );
49
50      list.remove( 4 );
51
52      console.log( list.head.next.next.next.data );
```

A summary of the program is that there are two classes. The first class is named Node and has two attributes in its constructor, data and next. The second class is named LinkedList and has two attributes in its constructor, this.head and this.tail, as well as two methods: add and remove.

There is no piece of new syntax in this program, although the appendation notation that links variables to the next node is in a new appearance of dot syntax. For example, in line twenty, the statement, this.tail.next = node; , assigns the value of node to the attribute of next that belongs to this.tail. Another way to think about the assignment, is that the nodes named this.tail and node exist before the assignment in line twenty, but are unconnected. The syntax of connection is the assignment of node to the next attribute of this.tail.

| | | | |
|---|---|---|---|
| Before assignment: | this.tail | | node |
| After assignment: | this.tail | → | node |

A brief walk-through of the program starts now. When the program runs, the interpreter assigns the classes and their variables to memory. The interpreter reaches line forty-two, and creates a new instance of the class named LinkedList and assigns it to a variable named list. The instantiation creates two class attributes, this.head and this.tail, each of which is assigned the value of null.

In line forty-four, the interpreter finds the variable named list with its add( ) method appended via dot syntax. The method has an argument, the integer 1. The interpreter takes the argument into the new instance of the class that is assigned to the variable named list.

In line fourteen, the integer 1 is accepted as a parameter named data. The interpreter enters the declaration block and in line fifteen finds that the condition of the if statement evaluates to true. Because the value of this.head is null, therefore the negation of it in a condition evaluates to true. The interpreter enters the declaration block of the if statement. The interpreter declares a variable named

node and assigns it the value of a new node that is created by the Node class. In line seventeen, the attribute this.tail has its value reassigned from null to this.head.

The reader is to know that the reassignment prepares the node this.tail for the occurrence of the method in which the if statement is bypassed and the else statement is entered. The process will be addressed momentarily. For now, note that the reassignment is strategic and not haphazard. At this point in the program, the linked list looks like this:

node, 1

this.head

this.tail

The vertical alignment indicates that the two attributes, this.head and this.tail, are pointed at the same node. The first node can now be accessed by either of the pointers. At the moment, the first node is also accessible by the variable named node. Once the invocation of this method ends, however, the variable named node will no longer be available to access the node.

The interpreter completes the declaration block of the if statement and returns to the invocation of the method in line forty-four. In line forty-five, the interpreter finds the variable named list with the add( ) method appended on. The method has an argument that is the integer 2. The interpreter takes the argument into the instance of the class, to line fourteen.

In line fourteen, the integer 2 becomes the parameter named data. In line fifteen, the condition of the if statement evaluates to false because the attribute this.head exists and is equal to the value of the node that was assigned to it during the previous invocation of the method. Because the condition of the if statement evaluates to false, therefore the interpreter enters the declaration block of the else statement.

In line nineteen, the interpreter declares a variable named node and assigns it the value of the creation of a new node that has its attribute this.data assigned the value of the parameter data. At this point in the program, the linked list looks like this:

node, 1 node, 2

this.head

this.tail

In line twenty, the attribute this.tail has its pointer assigned to the variable named node. This is the strategic aspect that was mentioned above: Because the attributes this.head and this.tail are currently occupying the same node, which was created during the previous invocation of the method, therefore by assigning the link of the attribute this.tail to the new node that was created during this invocation of the method, a link is thereby established between the head node and the first middle node.

In line twenty-one, the attribute this.tail is moved from the node that also is the assignment of this.head, to the node that was created during this invocation of the method. At this point in the program, the linked list looks like this:

node, 1 → node, 2

this.head this.tail

The interpreter completes the declaration block of the add( ) method and returns to the invocation, in line forty-five. Lines forty-six through forty-eight repeat the process that was accomplished in line forty-five. A few differences include the value of the argument and also the status of the linked list, which looks like this as it grows:

node, 1→ node, 2 → node, 3

this.head this.tail

node, 1→ node, 2 → node, 3 → node, 4

this.head this.tail

node, 1→ node, 2 → node, 3 → node, 4 → node, 5

this.head this.tail

In line fifty, the interpreter finds the variable named list with its remove( ) method appended on. The call signature of the remove( ) method has the integer 4, which the interpreter takes into the instance of the class the variable variable named list, to line twenty-five.

In line twenty-five, the interpreter converts the argument into a parameter and enters the declaration block. The interpreter creates two variables in lines twenty-six and twenty-seven. These variable will be used to find the data point in a node that is to be removed by an iteration of the linked list. They are in a staggered position, one after the other.

The use of two pointers in a staggered position to iterate a linked list is called the runner technique. There is a fast runner and a slow runner. The fast runner is the one ahead of the slow runner. The fast runner is currently assigned to this.head.next . The idea is that the fast runner encounters the node to be removed before the slow runner does. During iteration, the slow runner is kept one node behind the fast runner so that when the fast runner encounters the node to be removed, then the slow runner can assign its pointer to the node that follows the fast runner. Because the fast runner has no access the node behind it, without the slow runner, the node that the fast runner is on could not be

removed. Here is a picture of the process:

Step one:

node, 1 → node, 2 → node, 3 → node, 4 → node, 5

this.head this.tail

slow fast

Step two:

node, 1 → node, 2 → node, 3 → node, 4 → node, 5

this.head this.tail

slow fast

Step three:

node, 1 → node, 2 → node, 3 X node, 4 → node, 5

this.head this.tail

slow fast ^

→ → → → → →

Result:

node, 1 → node, 2 → node, 3 → node, 5

this.head this.tail

Note that in step three, the upper case letter X replaces the arrow that represents the link between the node with data 3 to the node with data 4. In step four, a link is created from the node with data 3 to the node with data 5. Also in step four, neither the node with data 4 nor its link to the node with data 5 is shown, although neither were erased in the remove method. Because an iterator has no access to a node behind it, therefore the fact that a node with a pointer remains does not affect the traversal. In order to achieve a greater efficiency in memory and more organized code, the node could be reassigned the value of null. The syntax to achieve this, is an additional line added after line 33. The remove statement would be moved from line 33 to line 34, followed by the phrase, fast = null; . After finishing the walkthrough of the example in figure 67, the reader is encouraged to implement the program with the reassignment.

In line twenty-eight, a while loop begins the iteration of the linked list. The condition that is set for the continuation of the while loop is that the variable named fast evaluates to true. The syntax of the condition is the inclusion of the variable name inside the parentheses, while ( fast ) . Whenever fast evaluates to false, then the condition evaluates to false, and the loop ceases to iterate. In the example, the condition evaluates to true, however, and the interpreter enters the declaration block of the while loop.

In lines twenty-nine through thirty-seven is an if, else-if, else statement whose conditions compare the data of one of the runners to the data at the node where they are currently located. Lines twenty-nine through thirty-one check if the first node of the linked list contains the data to be removed. If it does, then the head of the linked list is set to the second node and the first node is set to null, thereby removing the node from the linked list. That is not the case in the example, however, and so the interpreter checks the second condition, in the else-if statement in line thirty-two.

In line thirty-two, the condition evaluates to false because the value of the the node where the fast runner is, does not match the value of the parameter data, which identifies the value of the node to

be removed. The interpreter skips the declaration block of the else if statement, and continues to the declaration block of the else statement in line thirty-four. In lines thirty-five and thirty-six, the two runners are incremented along the linked list to their respective next nodes, and the while loop returns to the condition to check if there is another iteration.

In line twenty-eight, the interpreter finds that the evaluation of the condition is true because fast has a value: fast is at the second middle node which has the value of 3. The interpreter enters the declaration block of the while loop. Neither the if nor the else-if condition evaluates to true, so the interpreter enters the declaration block of the else statement and increments the runners along the linked list before returning to the condition of the while loop to check for another iteration.

In line line twenty-eight again, the interpreter finds that the evaluation of the condition is true and enters the declaration block of the while loop. The condition of the if statement is false. The condition of the else-if statement, evaluates to true because the value of the node where fast is located is compared and found equal to the value of the parameter data. The interpreter enters the declaration block of the else if statement.

In line thirty-three, the pointer of the slow runner is reassigned to the node that follows the fast runner. In effect, this deletes the node of the fast runner from the linked list. The interpreter returns the fact of the reassignment to the invocation of the method in line fifty. The result of the return statement is not logged, so the interpreter continues to the next line of the program.

In line fifty-two, the interpreter finds a console log statement whose call signature contains the variable named list which has been assigned the new instance of the LinkedList class. Appended to the variable by dot syntax, is the head of the linked list and a series of pointers before the attribute data. The result is the access of the data that is at the node that many pointers away from the head of the linked list. Here is what the linked list looks like now, updated by the removal of the node with the data attribute 4.

Result:

node, 1 → node, 2 → node, 3 → node, 5

this.head this.tail

The statement in the call signature of the console log statement, accesses the node where head node is assigned, the node with data 1, and then adds three pointers. Imagine a pointer named marker. Marker starts at the node where this this.head is located and then proceeds for three pointers.

list.head:

node, 1 → node, 2 → node, 3 → node, 5

this.head this.tail

marker

list.head.next:

node, 1 → node, 2 → node, 3 → node, 5

this.head this.tail

marker

list.head.next.next:

node, 1 → node, 2 → node, 3 → node, 5

this.head this.tail

marker

list.head.next.next.next:

node, 1 → node, 2 → node, 3 → node, 5

this.head this.tail

marker

After the pointers, the marker is told to access the data of the node. The marker now evaluates to 5. The call signature evaluates the marker, which evaluates to 5, and logs the result to the console.

The log is proof that the remove( ) method worked correctly. Without the remove( ) method, the data of the node that is three pointers after the head node, is 4. When the node with data 4 is deleted, then the node that is three pointers away from the head node becomes the node with data 5.

If there were duplicate data points in the linked list, then the remove( ) method would remove the first node that contains a value that matches the parameter data. Imagine that the remove( ) method were to remove all the instances of nodes whose data matches the parameter. A possible solution is to iterate the list with the runner technique, removing each node that matches the parameter data, until the fast runner reaches the end of the list. This method differs from the current method in that the current method returns after the first instance of the node that matches the parameter data is removed.

Another consideration is that the remove( ) method might be called on a linked list that no nodes or only one node. Either before the remove( ) method or in the first lines of the remove( ) method, conditional statements that check for the presence of nodes, can be used in order to prevent an invalid assignment of the runners to non existent nodes. Such an assignment returns an error.

The reader is encouraged to restore the original linked list by commenting out the remove( ) method in line fifty. Then check the data at three pointers from the head of the linked list. Other experiments that the reader is encouraged to attempt, are the commenting in of the remove( ) method with a substitution of the data point that is to be deleted. Try all the data points. Try inserting and

removing alphabetical data points. Try removing a node by its place in the linked list rather than by the data it contains. Try to add a conditional statement to ensure that the assignment of the runners is unproblematic. In order to achieve these different tasks, the constructor of the linked list can be modified to include other attributes.

After the reader develops a comfort with the modification of the data points and the logging of the linked list to the console, try modifying the runners and their next attributes. These are pieces that can be rearranged for different purposes. The linked list can be elaborated, such as with double pointers and a cycle. These are advanced concepts that are omitted here so that the reader focuses on the basics of the structure and traversal.

**What is a stack?**

A stack is a data structure that, like a linked list, is linear and has nodes with data and pointers. The difference between a stack and a linked list, is that the stack has a removal order built into its structure. After the creation of a stack, when the method to remove a node is called, which is commonly named pop( ), then the removal of nodes begins with the most recently added node and sets the next most recent node as the next to be removed. Another common description of a stack is that its structure is last in and first out. The last node that is pushed into the stack, is the first node that is removed.

Here is a picture of a stack that has the items 1, 2, 3, and 4 pushed onto it. The name push( ) is commonly used to indicate the insertion of a value into a stack. The picture of the stack is set vertically:

node, 4 this.head

v

node, 3

v

node, 2

v

node, 1 this.tail

When remove is called, then the most recently added node is popped from the stack. Here is the stack after the pop( ) method:

node, 3 this.head

v

node, 2

v

node, 1 this.tail

The process is repeated until there are no more nodes in the stack:

node, 2 this.head

v

node, 1 this.tail

Then:

node, 1 this.tail, this.head

And finally:

```
this.head = null, this.tail = null;
```

Figure 68 demonstrates the implementation of a stack in a program.

Figure 68: A stack with two methods, push( ) and pop( )

```
1   class Node {
2       constructor( data, next ) {
3           this.data = data;
4           this.next = next;
5       }
6   }
7
8   class Stack {
9       constructor( ) {
10          this.head = null;
11      }
12
13      push( item ) {
14          if ( !this.head ) {
15              let node = new Node( item, null );
```

```
16                          this.head = node;
17                  } else {
18                          let node = new Node( item, this.head );
19                          this.head = node;
20                  }
21          }
22
23          pop( ) {
24                  if ( !this.head ) {
25                          return null;
26                  } else {
27                          let node = this.head;
28                          this.head = node.next;
29                          return node;
30                  }
31          }
32  }
33
34  let integers = new Stack;
35
36  integers.push( 2 );
37  integers.push( 1 );
38  integers.push( 4 );
39  integers.push( 3 );
```

```
40      integers.push( 5 );
41
42      integers.pop( );
43      integers.pop( );
44
45      console.log( integers );
```

Both the syntax and the concepts are familiar to the reader. Try to walk-through the program of the stack, as was done with the program of the linked list in figure 67.

Please also note that the pop( ) method iterates the stack in order to find the next node to pop. During the iteration, the iterator does not delete the temporary node that is defined in line 27. Once the method is complete, the node is no longer accessible because there is no pointer to it. Because the temporary node is inaccessible, therefore it does not affect the iterator and therefore the functionality of the stack. The remaining, inaccessible nodes nonetheless unnecessarily require memory.

A more efficient algorithm would be to delete the temporary node during iteration. This can be achieved by storing the data of the temporary node in a variable, assigning a null value to the temporary node, and returning the variable for the function call. Please note that this is an important concept and should be learned, but that the less efficient and easier method is used throughout this book. The basic functionality of the stack is easier to learn with shorter code. And the process of recognizing and improving inefficiencies in speed and memory is good practice for the reader. Finally please note that the above note about the inefficiency of the stack, holds true for the inefficiency of the queue data structure, given below.

**What is a queue?**

A queue is a data structure that is like a stack in structure and procedure, except that its remove( ) method removes elements from the queue in the order not of last in, first out, but first in and first out. Here is a picture of a queue that has the items 1, 2, 3, and 4 enqueued. The name enqueue is commonly used to indicate the insertion of a value into a stack. The name dequeue is commonly used to indicate the removal of the item that was entered before the others. The picture of the queue is set vertically:

node, 4 this.head

^

node, 3

^

node, 2

^

node, 1 this.tail

When remove is called, then the most recently added node is popped from the stack. Here is the stack after the pop( ) method:

node, 4 this.head

^

node, 3

^

node, 2 this.tail

The process is repeated until there are no more nodes in the stack:

node, 4 this.head

^

node, 3 this.tail

Then:

node, 4 this.tail, this.head

And finally:

this.head = null, this.tail = null;

Figure 69 has a program with an implementation of a queue.

Figure 69: A queue with enqueue( ) and dequeue( ) methods

```
1       class Queue {
2               constructor( ) {
3                       this.head = null;
4                       this.tail = null;
5               }
6
7               enqueue( data ) {
```

```
8                   if ( !this.head ) {
9                           this.head = new Node( data );
10                          this.tail = this.head;
11                  } else {
12                          let node = new Node( data );
13                          this.tail.next = node;
14                          this.tail = node;
5                   }
16          }
17
18          dequeue( ) {
19                  let node;
20                  if ( !this.head ) return null;
21                  else if ( this.tail === this.head ) {
22                          node = this.head;
23                          this.head = null;
24                          this.tail = null;
25                          return node;
26                  } else {
27                          node = this.head;
28                          this.head = this.head.next;
29                          node.next = null;
30                          return node;
31                  }
```

```
32              }
33      }
34
35      const letters = new Queue;
36
37      letters.enqueue( "a" );
38      letters.enqueue( "b" );
39      letters.enqueue( "c" );
40      letters.enqueue( "d" );
41      letters.enqueue( "e" );
42
43      letters.dequeue( );
44      letters.dequeue( );
45
46      console.log( letters );
```

As with the stack, the reader is familiar with both the concepts and syntax of the queue class. Try to walk-through it. Then make modifications to achieve different results, such as enqueueing integers or even saving the dequeued result in a variable, accessing its data, and performing a computation that can be logged to the console.

If the reader began the chapter with the section “What is a queue” and did not read the section “What is a stack”, please read the final paragraph of the section “What is a stack” for a note about algorithmic efficiency.

**What is a graph?**

A graph is a data structure that comprises nodes and links, like the linked list, stack, and queue. What differentiates a graph from the others is that the graph can be non linear. There is a particular type of graph, a tree, that is considered in the next section. In this section, a basic graph is addressed, as are the methods for traversal of the graph.

There are two common types of traversal of a graph. The first is called a breadth first search. The second is called a depth first search. A breadth first search begins at a node and checks each of its adjacent nodes before moving onto another node in the traversal. A depth first search, by contrast, checks a single adjacent neighbor of each level in a graph until it runs into a node either that has already been visited or has no other adjacent nodes. Then the depth first search backtracks to the most recently visited node and initiates another depth first search until it again reaches a node that either has no adjacent nodes other than the one from which it was reached, or has adjacent nodes that have already been visited by the depth first search. The depth first search then backtracks to the most recently visited node and continues the process. If the most recently visited node has no other options for the continuation of the depth first search, then the search backtracks to the previous level that had most recently been visited. The search continues the process until all nodes have been visited.

The concepts are difficult and wont be comprehended by the reader immediately. With a few walk-throughs in examples and then practice, however, they are accessible.

What is a breadth first search?

A breadth first search is a search of a graph that traverses each of the adjacent nodes of the current node, before moving to one of the adjacent nodes for a continuation of the traversal. Typically,

a breadth first search is achieved with the help of another data structure, one already encountered: a queue. JavaScript developers sometimes use an array instead of a queue because the array allows the procedure of a queue, first in and first out, by its push( ) and shift( ) built-in methods. Because this chapter focuses on data structures, therefore the example below instead of an array, incorporates a queue into the binary search. Figure 70 has a the example.

Figure 70: A breadth first search with a queue

```
1       class QueueNode {
2               constructor( data, next ) {
3                       this.data = data;
4                       this.next = next;
5               }
6       }
7
8       class Queue {
9               constructor( ) {
10                      this.head = null;
11                      this.tail = null;
12              }
13
14              enqueue( data ) {
15                      let node = new QueueNode( data );
16                      if ( !this.head ) {
17                              this.tail = node;
```

```
18                          this.head = this.tail;
19                  } else {
20                          this.head.next = node;
21                          this.head = node;
22                  }
23          }
24
25          dequeue( ) {
26                  if ( !this.head ) return null;
27                  else {
28                          let node = this.tail;
29                          if ( this.tail === this.head ) {
30                                  this.tail = null;
31                                  this.head = null;
32                          } else this.tail = this.tail.next;
33                                  node.next = null;
34                                  return node;
35                          }
36                  }
37          }
38
39  class GraphNode {
40          constructor( data, adj ) {
41                  this.data = data;
```

```
42                      this.adj = adj;
43              }
44      }
45
46      const a = new GraphNode( “a” );
47      const b = new GraphNode( “b” );
48      const c = new GraphNode( “c” );
49      const d = new GraphNode( “d” );
50      const e = new GraphNode( “e” );
51
52      a.adj = [ b ];
53      b.adj = [ a, c, d ];
54      c.adj = [ b, e ];
55      d.adj = [ b, e ];
56      e.adj = [ c, d ];
57
58      function route( node ) {
59              if ( !node ) return null;
60              else {
61                      const visited = [ ];
62                      const order = [ ];
63                      queue.enqueue( node );
64                      while ( queue.head ) {
65                              const node_ = queue.dequeue( ).data;
```

```
66                              order.push( node_.data );
67                              const adjacents = node_.adj, l = adjacents.length;
68                              for ( let i = 0; i < l; i++ ) {
69                                      if ( !visited.includes( adjacents[ i ] ) ) {
70                                              visited.push( adjacents[ i ] );
71                                              queue.enqueue( adjacents[ i ] ) ;
72                                      }
73                              }
74                              if ( !visited.includes( node_ ) ) visited.push( node_ );
75                      }
76                      return order;
77              }
78      }
79
80      const queue = new Queue;
81
82      console.log(route( a ) );
```

A summary of the program in figure 70 is that there are three classes, a graph declaration by an adjacency list in lines forty-six through fifty, and a function that traverses the graph in a breadth first search. A walk-through of the program will elucidate the breadth first search for the reader.

The text editor runs the program. The interpreter reads the three classes into memory: class QueueNode, class Queue, and class GraphNode. In lines forty-six through fifty, the interpreter creates new variables with the letters and assigns them the value of a graph node with the data as the letters.

Note that the reader is to keep the two distinct. The variable name could be anything as could the data point. It is for ease of consideration that they correspond, and it is not out of requirement. After the walk-through, when the reader practices modulation of the graph traversal, try assigning different variable names and data points to the graph nodes.

In lines fifty-eight through seventy-eight the interpreter reads the function named route( ) into memory. In line eighty, the interpreter creates a variable named queue and assigns it the value of a new instance of the class named Queue. In line eighty-two, the interpreter finds a console log statement that logs to the console the evaluation of the invocation of the route function with a call signature that contains the variable a, which had been assigned the graph node in line forty-six.

The interpreter takes the argument inside to the declaration of the function in line fifty-eight. In line fifty-eight, the interpreter converts the argument into a parameter named node and enters the declaration block of the function. The interpreter finds the evaluation of the condition in line fifty-nine is false. Node has a value, which is the assignment of the graph node a. In line sixty, the interpreter enters the declaration block of the else statement.

In lines sixty-one and sixty-two, the interpreter declares new variables, visited and order, and assigns each an empty array. The variable named visited is used to keep track of which nodes are visited during the traversal. It prevents the interpreter from visiting the same node multiple times. The variable named order is used to keep track of the order of the traversal, which later can be returned to the invocation of the method.

In line sixty-three, the interpreter access the queue that had been declared in line eighty. The reader recognizes that the variable named queue has global scope because it is declared outside the function. Also in line sixty-three, the interpreter enqueues the parameter named node.

In line sixty-four, the interpreter declares and begins a while loop whose condition for iteration is that the statement that the head attribute of the queue evaluates as true. Said another way, when the

queue has a head whose value is false, the while loop ends its iteration. The interpreter enters the declaration block of the while loop.

In line sixty-five, the interpreter declares a variable named node_ and assigns it the evaluation of the data of the node that is dequeued from the queue. Note the syntax of the assignment: queue.dequeue( ).data . Appended to the queue is the dequeue( ) method, whose invocation requires empty parentheses. Then, appended to the dequeue( ) method is the data attribute of a QueueNode. The evaluation first dequeues a node and then accesses the data of that node. The data is a graph node that had been assigned to the queue node. The data of the graph node is the letter a. It is for this reason that the syntax in line sixty-six, which pushes the data of the node, achieves the access of the letter a. Pause over this and the previous paragraph and comprehend the principles before moving on to the next paragraph.

In line sixty-seven, two variables are declared and assigned values. The first variable is named adjacents whose value is the adjacency list of the variable named node. Recall that the adjacency list of each node is declared in lines forty-six through fifty. The second variable that is declared in line sixty-seven, named the letter l, is assigned the value of the length of the variable named adjacents. The variable named adjacents is an array whose length can be accessed by such notation of appendation.

In line sixty-eight, a for loop begins whose iterator is limited as less than the length of the variable named l. During the for loop, the adjacent nodes of the current node dequeued, are compared to those nodes in the variable named visited. This is achieved by the negation of the built-in method seen in line sixty-nine, .includes( ) . If the iterator indexes a node in adjacents that is not included in the variable array named visited, then the interpreter pushes the indexed node to visited in line seventy and enqueues the node in line seventy-one.

In line seventy-four, the interpreter pushes the current node to the variable array named visited, if it is not already included. Then the interpreter returns to line sixty-four to check if the node at

queue.head is existent.

At this point in the program, the graph node named a, has been enqueued and dequeued and pushed to visited and order. The adjacent node of node a, defined in line fifty-two, is enqueued. Therefore the current node that the attribute this.head is located at in the queue, is the adjacent node that had been enqueued during the for loop in lines sixty-eight through seventy-three.

In line sixty-four, the interpreter finds that the condition for the for loop evaluates to true because the graph node b is the current head of the queue. The interpreter enters the declaration block of the while loop. Lines sixty-four through seventy-five repeat the process that had occurred for the graph node a.

First the interpreter dequeues the current head node of the queue and pushes the data of the graph node that it contains to the variable array named order. Then it begins the for loop with the adjacent elements of the current node.

A difference between the current and the previous iteration is that the current iteration considers more adjacent nodes. The graph node b has three adjacent nodes: a, c, d. In line sixty-nine, the condition that visited does not include the current adjacent node, evaluates as false for the first adjacent node, graph node a, of the current node which is graph node b. The evaluation fails because during the previous iteration of the while loop, graph node a had been pushed to the variable named visited in line seventy-four. Therefore, for graph node b during the iterations of its for loop, the queue enqueues only those nodes that are not already in visited: graph nodes c and d.

In line seventy-four, the interpreter finds that visited already includes graph node b. It had been pushed to the array during the previous iteration in line seventy.

The interpreter returns to line sixty-four in order to check the condition that the queue has a current attribute this.head. The interpreter finds that the condition evaluates to true because the current head of the queue is its eldest element, a queue node whose data is a graph node with the data c. The

interpreter enters the declaration block of the while loop, dequeues the queue node and accesses the data of the graph node so that it can push the data to the variable array named order. Then the interpreter defines the variables named adjacents and length and begins the for loop, performing the breadth first search on the adjacents of the current node.

Declared in line fifty-four, the adjacents of graph node c are the graph nodes b and e. The evaluation of the comparison between graph node b and those values that are in the variable array named visited, results in the Boolean value false. Therefore only the graph node e is enqueued in line seventy-one. In line seventy-four, the interpreter finds that graph node c is already in the array named visited. So the interpreter returns to line sixty-four in order to check the condition for a continuation of the while loop.

In line sixty-four, the interpreter finds that the graph currently has a head node, which is the queue node whose data is the graph node with the data “d”. The graph node with the data “d” had been enqueued during the iteration of graph node b. The queue currently holds the graph node d as the head and the graph node e after it. The interpreter enters the declaration block of the while loop.

In lines sixty-five and sixty-six, the interpreter dequeues the current head node, assigning its data to the variable named node_ and also pushes the data of that variable to the variable array named order, which now has the values “a”, “b”, “c”, and “d” . In line sixty-seven, the interpreter creates the variables adjacents and l, and assigns them the adjacents of graph node d that is declared in line fifty-five. Its length is two. Both the adjacents are already included in the variable array named visited. So the for loop ends.

In line seventy-four, the interpreter finds that the variable array named visited already includes graph node d, so the interpreter returns to the beginning of the while loop to check the condition that allows the next iteration.

In line sixty-four, the interpreter finds that the queue has a head node and evaluates the

condition as true. The interpreter enters the declaration block of the while loop. The interpreter dequeues the head node and accesses the data of the queue node, assigning the resultant graph node to a constant variable named node_ in line sixty-five. The interpreter pushes the value of that node, graph node with the data e, to the variable named order in line sixty-six.

In line sixty-seven, the interpreter makes a list of the adjacents of the node and assigns the variable named l the length of the the adjacency list, which is an array with two elements as defined in line fifty-six. Both elements result in a false evaluation in line sixty-nine because they both had been pushed the variable array named visited. The for loop stops. The interpreter goes to line seventy-four and finds that the variable array named visited already includes the data of the current node. The interpreter therefore returns to line sixty-four to check for another iteration of the while loop.

In line sixty-four, the interpreter finds that the condition of the if statement evaluates to false because there is no head node in the queue.

In line seventy-six, the interpreter returns the variable array named order to the invocation of the method, which is in the call signature of the console log statement in line eighty-two. The interpreter converts the invocation of the method to the variable array named order. The call signature of the console log statement evaluates to the variable array named order, which is logged to the console: [ a, b, c, d, e ]. The program terminates.

A long and complex program with multiple moving parts, the example given in figure 70 demonstrates for the reader the traversal style of the breadth first search. As mentioned above, the breadth first search begins by a search of each adjacent node of the current nodal level before moving to the next level to search the adjacent nodes there. Here is a picture of the graph that was traversed by the program in figure 70, where a carrot pointing at a node indicates a link between the adjacent node and itself:

a

∧

v

b < > c

∧ ∧

v v

d < > e

What if a marker that makes the traversal with the breadth first search? The movement of the marker is the following:

step one:

a marker

∧

v

b < > c

∧ ∧

v v

d < > e

step two:

a

∧

v

marker b < > c

∧ ∧

v v

d < > e

step three:

a

∧

v

b < > c marker

∧ ∧

v v

d < > e

step four:

a

∧

v

b < > c

∧ ∧

v v

marker d < > e

step four:

```
a
^
v
b       < >     c
^               ^
v               v
d       < >     e       marker
```

This is a breadth first search. Try to think instead of what the order would be in a depth first search. It is the topic of the next section.

What is a depth first search?

A depth first search is a traversal of a graph that visits the adjacent nodes of the next level before the adjacent nodes of the current level. Unlike a breadth first search, a depth first search does not use a queue. Instead, the depth first search is often achieved by recursion, which is a new concept to the reader. Recursion is the invocation of the function that the interpreter is already in, often with a new piece of data as the argument. The concept is difficult, so be patient with the study of it. Figure 71 exemplifies a depth first search with recursion.

Figure 71: A depth first search of a graph

```
1       class Node {
2               constructor( data ) {
3                       this.data = data;
```

```
4                       this.adj = [ ];
5               }
6       }
7
8       const a = new Node( “a” );
9       const b = new Node( “b” );
10      const c = new Node( “c” );
11      const d = new Node( “d” );
12      const e = new Node( “e” );
13
14      a.adj = [ b ];
15      b.adj = [ a, c, d ];
16      c.adj = [ b, e ];
17      d.adj = [ b, e ];
18      e.adj = [ c, d ];
19
20      const visited = [ ];
22      const order = [ ];
23      function route( node ) {
24              if ( !node ) return null;
25
26              else {
27                      visited.push( node );
28                      order.push( node.data );
```

```
29                      const adjacents = node.adj, l = adjacents.length;
30                      for ( let i = 0; i < l; i++ ) {
31                              if ( !visited.includes( adjacents[ i ] ) ) route( adjacents[ i ] );
32                      }
33                      return order;
34              }
35      }
36      console.log( route( a ) );
```

A summary of the program is that there is a class named Node, a declaration of nodes in lines eight through twelve, a declaration of nodal adjacencies in lines fourteen through eighteen, a declaration of two variables, and then a declaration of a function named route and its invocation in a console log statement in line thirty-six.

When the program executes, the interpreter reads all the declarations into memory and then evaluates the call signature of the console log statement in line thirty-six. The interpreter takes the argument, the graph node named a, to the declaration of the function name in line twenty-three.

In line twenty three, the argument becomes a parameter named node. The interpreter enters the declaration block of the function. In line twenty-four, the interpreter finds that the condition of the if statement evaluates to false. The condition is posed so that if the recursive call has an undefined value, then the function returns and continues the depth first search on defined nodes.

In line twenty-seven, the interpreter pushes the node with the data point “a” to the array named visited. In line twenty-eight, the interpreter pushes the data from the node into the array named order.

In line twenty-nine, two variables are declared. The first accesses the adjacency list of the node and the second is assigned the length of the adjacency list, which becomes the limit of the iterator in the

next line. In line thirty, a for loop begins and ends that makes a recursive call with each element in the adjacency list. In this instance, the only element in the adjacency list is the graph node “b”. The interpreter takes graph node “b” to line twenty-three in the first recursive iteration.

In line twenty three, graph node “b” is converted into the parameter named node. The interpreter enters the declaration block. Both conditions of the if and else-if statements evaluate as false, so the interpreter enters the declaration block of the else statement. The variable named visited pushes the node, the variable named order pushes the data of the node, and two variables are created, the first with the adjacency list of the node and the second with the length of the adjacency list. In this iteration, the adjacency list of graph node b is [ a, c, d ] . The first adjacent node, graph node a, evaluates as false in the condition of the if statement in line thirty. The interpreter continues the for loop with the next adjacent, graph node c.

In line twenty-three, the interpreter converts graph node c to the parameter node. The interpreter enters the declaration block. The conditions of the if and the else-if statements in lines twenty-four evaluates as false because the node exists and for the second test, the variable named visited contains only the graph nodes a and b, but not graph node c. The interpreter enters the declaration block of the else statement in line twenty-six.

In line twenty-seven, the variable array named visited pushes the node. In line twenty-nine, the variable named order pushes the data of the node, the string “c”. In line thirty the interpreter sets the iterable and limit of the iterator. Also in line thirty, the interpreter enters the for loop with the first adjacent element, graph node b. In line thirty-one, graph node b is found to be already included in the variable array named visited. The for loop continues with the second adjacent node, graph node e. The adjacent graph node e, is not in the variable array named included. The interpreter makes the recursive call to the function route( ) with graph node e as the argument.

In line twenty-three, the interpreter converts the argument to a parameter named node. Because

there is a node and because it is not included in the variable named visited, therefore the conditions in lines twenty-four evaluates as false. The interpreter enters the declaration block of the else statement. In line twenty-seven, the interpreter pushes the variable named node to the variable named visited. In line twenty-eight, the interpreter pushes the data of the variable named node to the variable array named order. In line twenty-nine, the interpreter declares two variables for the for loop. In line thirty, the interpreter enters the for loop with graph node e.

The first of the adjacents of graph node e, which are graph nodes c and d, is found to already be included in the variable array named visited. The loop continues with graph node d and recurses with it.

The recursive call with graph node d is found to contain adjacent elements that are already included in the array named visited. Therefore the recursive call unwinds until the previous graph node, which had called graph node d: graph node e. The interpreter finds that the for loop during the iteration of the adjacents for graph node e has reached its limit, so the interpreter backtracks again to the node which had recursively called graph node e: graph node c.

Again, the limit of the iteration is reached, and the recursive call backtracks to the node that had invoked the function with graph node c: the iteration with graph node b. The next node in the for loop of graph node b, is graph node d.  In the final iteration of the for loop in line thirty, the interpreter compares the data of graph node d to the array named visited and finds that it is already included in the variable.

The recursive call backtracks again. The call is now the first which was made with graph node a. The iteration is at its limit.

The loop ends. The interpreter exits the for loop. In line thirty-three, the interpreter returns to the invocation of the function, the variable name order.

In line thirty six, the call signature of the console log statement evaluates to the return statement, which is the variable named order. The variable named order is an array with the data points

of the nodes in the graph that were traversed in the depth first search: [ “a”, “b”, “c”, “e”, “d” ] . The console logs the array. The program terminates.

The program in figure 71 differs from that in figure 70 in that the program in figure 71 is a depth first search of the graph that searches the nodes of the next level before the nodes of the current level. Another difference is that the depth first search uses recursion rather than a queue to facilitate its procedure. Note that the variables named visited and order are declared outside the function named route. This is a requirement because it allows the variables to received input with each recursive call. Said another way, each recursive call has an up to date version of the variable arrays named visited and order, rather than an empty array that is created during the recursive call.

The reader is encouraged to modify the graph and function in order to experiment with the syntax. Learn which piece of syntax achieves what procedure in the data structure. Also try to imagine a variable named marker as it iterates along the graph during the depth first search. Sketch the graph and the marker. This will prepare the reader for the next chapter, when data structures are used in order to achieve the solution of various technical problems.

**What is a tree?**

A tree is a type of graph. A tree has requirements that make it a specific type of graph. There are no cycles. There is a root node. The relationship between the root node and its adjacent nodes, like the adjacent nodes with their adjacent nodes, is thought of as a relationship between parent and child or super and sub. There are many types of trees, including binary trees, binary search trees, and others.

The reader can read *Cracking the coding interview*, chapter four of the 2015 edition, for a review of the types of trees. Also, consult a video by the freeCodeCamp called “Binary Search Tree,” which is found on YouTube. The code for a binary search tree can be found there.

In this section, a binary tree is considered. A binary tree is a tree whose nodes have two or fewer child nodes. As with a graph, a binary tree is typically traversed in two ways: a breadth first search or a depth first search. The depth first search is further subdivided into three categories: in order, pre order, and post order. The order refers to the sequence in which the root of a sub tree is visited. It is difficult to apprehend in the abstract, so and example of each type of search is given in the program that is in figure 72.

Figure 72: The breadth first search and depth first search in a binary tree

```
1       class QueueNode {
2               constructor( data, next ) {
3                       this.data = data;
4                       this.next = next;
5               }
6       }
7
8       class Queue {
9               constructor( ) {
10                      this.head = null;
11                      this.tail = null;
12              }
13
14              enqueue( data ) {
15                      if ( !this.head ) {
16                              this.head = new Node( data );
```

```
17                      this.tail = this.head;
18                  } else {
19                      let node = new Node( data );
20                      this.tail.next = node;
21                      this.tail = node;
22                  }
23              }
24
25              dequeue( data ) {
26                  let node;
27                  if ( !this.head ) return null;
28                  else if ( this.head === this.tail ) {
29                      node = this.head;
30                      this.head = null, this.tail = null, node.next = null;
31                      return node;
32                  } else {
33                      node = this.head;
34                      this.head = this.head.next;
35                      node.next = null;
36                      return node;
37                  }
38              }
39      }
40
```

```
41  class Node {
42      constructor( data, left, right ) {
43          this.data = data;
44          this.left = left;
46          this.right = right;
47      }
48  }
49
50  class BinaryTree {
51      constructor( ) {
52          this.root = null;
53      }
54
55      inOrder( ) {
56          if ( !this.root ) return null;
57          else {
58              const result = [ ];
59              const traversal = function( node ) {
60                  node.left && traversal( node.left );
61                  result.push( " " + node.data );
62                  node.right && traversal( node.right );
63              }
64              traversal( this.root );
65              return result;
```

```
66                  }
67          }
68
69          preOrder( ) {
70                  if ( !this.root ) return null;
71                  else {
72                          const result = [ ];
73                          const traversal = function( node ) {
74                                  result.push( " " + node.data );
75                                  node.left && traversal( node.left );
76                                  node.right && traversal( node.right );
77                          }
78                          traversal( this.root );
79                          return result;
80                  }
81          }
82
83          postOrder( ) {
84                  if ( !this.root ) return null;
85                  else {
86                          const result = [ ];
87                          const traversal = function( node ) {
88                                  node.left && traversal( node.left );
89                                  node.right && traversal( node.right );
```

```
90                                  result.push( " " + node.data );
91                          }
92                          traversal( this.root );
93                          return result;
94                  }
95          }
96
97          levelOrder( ) {
98                  if ( !this.root ) return null;
99                  else {
100                         const order = [ ];
101                         queue.enqueue( this.root );
102                         while ( queue.head ) {
103                                 const node = queue.dequeue( ).data;
104                                 order.push( " " + node.data );
105                                 if ( node.left ) queue.enqueue( node.left );
106                                 if ( node.right ) queue.enqueue( node.right );
107                         }
108                         return order;
109                 }
110         }
111 }
112
113 const a = new Node( "a" );
```

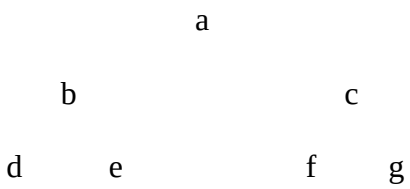
```
114     const b = new Node( "b" );
115     const c = new Node( "c" );
116     const d = new Node( "d" );
117     const e = new Node( "e" );
118     const f = new Node( "f" );
119     const g = new Node( "g" );
120
121     a.left = b;
122     a.right = c;
123     b.left = d;
124     b.right = e;
125     c.left = f;
126     c.right = g;
127
128     const queue = new Queue;
129     const binaryTree = new BinaryTree;
130
131     binaryTree.root = a;
132
133     console.log( "In order: " + binaryTree.inOrder( ) );
134     console.log( "Pre order: " + binaryTree.preOrder( ) );
135     console.log( "Post order: " + binaryTree.postOrder( ) );
136     console.log( "Level order: " + binaryTree.levelOrder( ) );
```

A summary of the program is that there are four classes: QueueNode, Queue, Node, and BinaryTree. In lines 113 through 119, the nodes of the binary tree are declared and assigned a value. Note that like with the graph, each node of the binary tree is arbitrarily assigned an alphabetical value that matches the name of the variable that defines it.

In lines 121 through 126, the structure of the binary tree is defined by a declaration of the left and right children of each node. In lines 128 and 129, two classes are instantiated in the variables named queue and binaryTree. In line 131, the variable named binaryTree is assigned a root node. The node named a is assigned as the value of the root.

The syntax used in line 60 is new to the reader. The statement, node.left && traversal( node.left ); , first checks that node.left is an existent node. If the evaluation is true, then the traversal is called on it. Otherwise the reader is familiar with the syntax of the program.

Consider the following picture, which illustrates the binary tree that is created in the above example:




The root of the tree is the node a. The left node of a is b. The node b is the root node of the sub tree that is to the left of node a. The node b has a left node, d, and a right node, e. The right node of a is c. The node c is the root node of the sub tree that is on the right of node a. Node c has a left node, f, and a right node, g.

Because the nodes d, e, f, and g do not have children, therefore they are called leaf nodes as well as the children of their parents.

The three types of traversal approach a tree in three different ways. The first way is the in order traversal. The sequence of an in order traversal is left, root, right. At the start of the traversal, the interpreter enters the tree at the root, which is node a. During an in order traversal, the interpreter follows the pattern of left, root, right. So from node a, the interpreter goes to the leftmost node, d. Because this is the leftmost node at the beginning of the traversal, the interpreter pushes the data of the node to the variable named order.

Next the interpreter visits the root of the subtree of the leftmost node, b. The interpreter pushes the data of the node b to the variable named order. Finally it visits the right node, e. The interpreter takes the subtree as the left node of the entire tree.

Because all the left nodes of the entire tree are visited, the next in order node to visit is the root, which is also the root of the entire tree in this moment. The interpreter pushes the data of the root node to the variable array named order.

After the left and root nodes have been visited, the interpreter next goes to the right. At node c, the interpreter finds that it can go to the leftmost node of the subtree, f. The interpreter goes to that node and pushes its data to the variable named order. Next the interpreter visits the root node of the sub tree, c, and pushes its data to the variable named order. The last node that interpreter visits is the right node, g. The interpreter pushes the data of the node g to the variable named order. Then the interpreter ends the in order traversal.

Now that the reader has a walk-through of an in order traversal, and since he or she is familiar with the other concepts and syntax in the program, try to walk-through the pre, post, and level order traversals within the execution of the program.

The pre order traversal follows the pattern, root, left, right. The post order traversal follows the pattern, left, right, root. The level order traversal follows the pattern of moving from the top to the bottom and the left to the right on each level. The reader also has the program that gives an output, if he

or she becomes confused and needs reference material.

# Chapter 4

## Algorithms and Programs

An algorithm is sequence of acts that change an input in order to achieve an output. In technical problems, typically a prompt is given that contains a data structure with data points, and the request is made that from the data, some result is returned. An example is if a prompt gives an array of integers, [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ] , and requests that the integer with the middle value is returned. The reader is then asked to devise an algorithm and encode it in a program that delivers the solution.

The reader is to start by considering what are colloquially called edge and corner cases. A corner case is a complicated edge case. These are situations in which the directives of the prompt are to be questioned for clarification. An example with the above prompt is that the middle element in the array is instead two elements, the integers 5 and 6 at indices 4 and 5. Should the reader return an average of the two middle elements if the length of the array is even? Or should he or she return both or only one of the middle elements? Another point of clarification is the question of whether the the array is in a sorted order. For example, if the above array were instead ordered [ 3, 2, 5, 4, 1, 7, 6, 9, 10, 8 ] , then the middle two elements are 1 and 7. Does the prompt intend that the reference of the phrase middle value specify a location in the array or the middle of the range of values of the elements?

After a few points of clarification have been considered, then the reader is ready to compose an algorithm. Because an algorithm is to be an ordered sequence of acts that transforms the input into a requested output, therefore assume for this example that the points of clarification suggested that middle value is not about the range of values of the elements but instead about the location of the array. Assume also that the request is for the two middle values in the array, if the length of the array is even.

To begin the composition of an algorithm, consider the solution to the prompt without the syntax of a programming language. What would the reader do if they could indicate the answer with a

pencil and paper? One option is to find the length of the array. If the the length is odd, then find the middle element by starting from the beginning and counting the number of elements until it is reached. Then offer the middle value as the answer to the prompt. If the length is even, then also start from the beginning and count the number of elements until the first of the two middle values is reached. Then offer the first middle value and the integer after it, as the answer to the prompt.

With a basic idea in mind of the sequence of acts that could deliver a solution if it were a prompt with pencil and paper, now rephrase the sequence of acts with some technical language, to create a structured algorithm. An example is given in figure 73.

Figure 73: An algorithm for finding the middle element in an array

1. Declare a changeable variable named count and assign half the length of the array as its value.
2. Declare a changeable variable named result and assign it the data type null.
3. Iterate the array in a for loop.
4. During the iteration, decrement the count variable.
5. When the count variable equals one, stop the iteration.
6. If the length of the array is odd, access the data point at the current index.
7. Assign the variable named result the data point as its value.
8. If the length of the array is even, access the data points at the current index and the one after it.
9. Assign the variable named result the data points as its value, while iterating the array.
10. Return the variable named result to the invocation of the function.

With a structured algorithm defined, as in figure 73, then the reader is ready to encode the algorithm in a programming language, in the composition of a program. An example of the above algorithm in a program is given in figure 74.

Figure 74: A program for finding the middle element in an array

```
1       function findMiddleValue( integers ) {
2               let count = Math.floor( integers.length / 2 );
3               let result = null;
4               const len = integers.length;
5
6               let i = 0;
7               while ( i < count – 1en ) i++;
8
9               if ( l % 2 === 0 ) result = [ integers[ i ], integers[ i + 1 ] ];
10              else result = integers[ i + 1 ];
11
12              return result;
13      }
14
15      const input = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
16      const input_ = [ 3, 5, 7, 9, 1, 2, 4, 6, 8 ];
17
18      console.log( findMiddleValue( input ) );
19      console.log( findMiddleValue( input_ ) );
```

The program works in most cases. Is the reader able to think of additional edge cases that might cause a bug in the program? Examples of additional edge cases that need to be considered are if the input is

either a data type of null, a data type that is not either an array or a string, or a data type with a length of one. Conditional logic to check the type of the input and its length could be put in, and the appropriate return statements could be added. An example of the updated program is given in figure 75.

Figure 75: A program for the algorithm in figure 73 with additional edge cases considered

```
1       function findMiddleValue( integers ) {
2               const len = integers.length;
3               if ( !Array.isArray( integers ) ) return false;
4               else if ( len === 0 ) return false;
5               else if ( len === 1 ) return integers;
6               else {
7                       let count = Math.floor( integers.length / 2 );
8                       let result = null;
9
10                      let i = 0;
11                      while ( i < count – 1 ) i++;
12
13                      if ( len % 2 === 0 ) result = [ integers[ i ], integers[ i + 1 ] ];
14                      else result = integers[ i + 1 ];
15
16                      return result;
17              }
18      }
19
```

```
20      const input = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
21      const inputTwo = [ 3, 5, 7, 9, 1, 2, 4, 6, 8 ];
22      const inputThree = { "a": 1, "b": 2, "c": 3, "d": 4 };
23
24      console.log( findMiddleValue( input ) );
25      console.log( findMiddleValue( inputTwo ) );
26      console.log( findMiddleValue( inputThree ) );
```

The syntax in the condition in line three, is an array method that checks whether the value in the call signature evaluates to an array. It is a more specific test than the keyword typeof, which returns a string named object for an array because the data type of an array is a derivation of an object. The additional input that is introduced in line twenty-two, exemplifies a case that would evaluate as true in a condition that used the keyword typeof, but one that evaluates as false with the more specific test in the array method isArray( ).

A final consideration of the program in figure 74, is its asymptotic time and space complexity. The reader recalls that time complexity considers the ratio of the magnitude of an input in relation to the magnitude of its output. Possibilities include constant, logarithmic, linear, and exponential complexity, among others.

In the program in figure 75, four variables are declared: len, count, result, and i. Each of the variables is assigned a number data type. Regardless of the size of the input, the assignment to each variable is only one number data type. So the space complexity of the program is constant.

The time complexity of a program is determined by determining the maximum time class that is involved in its operations. For example, in figure 75, the operation in line two is constant, regardless of the size of the input. The operations in lines three, four, five, seven, eight, and ten are also constant. So

far in the program, the maximum class of time complexity is constant.

In line eleven, the iterator is instructed to iterate half the length of the array. This operation is more than constant because it increases with the size of the input. The task is to determine the magnitude of the increase. If about half the input is left out of the operation, then the time complexity is in the class of logarithmic time. Logarithmic time is more than constant time, so the new maximum class of time complexity is logarithmic.

In lines thirteen, fourteen, and fifteen, the operations are in constant time. By the end of the program, the maximum time class is logarithmic. The asymptotic time complexity of the algorithm is logarithmic.

Time and space complexity are important when an algorithm executes with a large input, in the size of hundreds of thousands or millions. Nonetheless, a small input serves as an instructive experience for the analysis of time and space complexity so that the reader can begin to consider them during the composition of algorithms and programs. The rules for the assignment of each are set out in the introductory chapter of *Cracking the Coding Interview* (2015).

In this chapter, a series of prompts are discussed. The prompts require the composition of an algorithm and a program. For each program, the asymptotic time and space complexity is analyzed and discussed. The algorithms and programs can make use of any of the data structures that were covered in the previous chapter.

Example one: Frequency difference

Another example of a prompt of a technical problem is to find the difference between the count of the most frequent and the least frequent alphabetical characters, given a string of characters. The first step in the composition of an algorithm and then a program, is to conceptualize the problem as if the

solution were with a pen and paper. A possible solution is to read the string and keep a count of each letter, increasing the count for repetitions of a letter. Then after reading the string, read the list of letters and select both the letter with the highest count and the letter with the lowest count. Then subtract the lowest count from the highest count as the answer to the prompt.

The next step is to compose an algorithm with technical language that describes the pen and paper solution in a sequence of acts that could then be translated into code for a program. At this point, consider what types of data structures could be used to accomplish the task. Also consider possible edge and corner cases. In this example, an object presents itself as a viable option that can hold the count of each character in the string. Common edge cases are if the input is null or undefined, if the input does not facilitate the algorithm, such as an input of one unit when at least two are required, and if there are multiple options for the maximum and minimum frequencies.

An example of an algorithm for the prompt is given in figure 76.

Figure 76: An algorithm to find the difference between the highest and lowest character counts

1. Declare an empty object named characters.
2. Iterate the string and assign the object named characters key value pairs of the letters.
3. If the object named characters already has the data of the iterator, increment its count.
4. Declare min and max variables.
5. Iterate the object named characters and assign min and max values to the variables.
6. The min variable now has the minimum value of the key-value pairs.
7. The max variable now has the maximum value of the key-value pairs.
8. Declare a variable named result and assign it the difference between min and max.
9. Return the variable named result.

With the algorithm and edge cases in mind, write a program that gives as output the requirement

of the prompt. An example of a program is given in figure 77.

Figure 77: A program to find the difference between the highest and lowest character counts

```
1       function frequencyDifference( str ) {
2               const len = str.length;
3               let result;
4
5               if ( !str || len === 1 || len === 2 ) return null;
6               else {
7                       const characters = { };
8
9                       for ( let i = 0; i < len; i++ ) {
10                              characters[ str[ i ] ] = ( characters[ str[ i ] ] || 0 ) + 1;
11                      }
12
13                      let min = Infinity, max = 0;
14                      for (let i in characters) {
15                              if ( characters[ i ] < min && i !== " " ) {
16                                      min = characters[ i ];
17                              }
18                               if ( characters[ i ] > max && i !== " " ) {
19                                      max = characters[ i ];
20                              }
21                      }
```

```
22
23                      result = max - min;
24                      return result;
25              }
26      }
27
28      const input = "Ada Lovelace wrote the essay 'Sketch of the Analytical Engine invented by
                        Charles Babbage' in the year 1843. It was the first program.";
29      const input_ = "Ad";
30
31      console.log( frequencyDifference( input ) );
32      console.log( frequencyDifference( input_ ) );
```

The only piece of new syntax for the reader is the first statement of equality in line thirteen. The assignment to the variable named min is the value Infinity. In JavaScript, Infinity is a data point that can be accessed and used in assignment or evaluation. Often its use is for a situation in which a variable is to be assigned a minimum value. Infinity is useful in this situation because any value that is compared against the variable named min when it is assigned Infinity, will be less than the current value. This way, the edge case in which one underestimates the largest value of the input, is avoided.

In line five, three equality operations test for conditions that would invalidate the answer. The declaration block of the else statement, lines seven through twenty-four, constitute the actions of the iteration of the string, population of the object named characters, iteration of the object name characters, population and comparison of the variables, and the return of their difference.

The asymptotic space complexity of the program is linear because the maximum class that is

achieved is linear with the iteration of the parameter named str in line nine.

The asymptotic time complexity of the program also is linear. The cause is again the iteration of the parameter named str. Each of the indices are visited. That the object named characters is also iterated, is negligible because it does not change the complexity class of the program. In a more technical, academic analysis of the time complexity, each variable and operation can be assigned a value in an algebraic equation. For the purposes of the beginner, however, such analysis is beyond the scope of an introduction.

Another edge case that is not addressed in an algorithm, but one that is worth consideration, is if the length of the input is appropriate, but every letter in the str is the same. Try to devise a conditional test to check for this possibility. One idea is to add a for loop in line 12 that iterates the object named characters. If the length of the object is one, return null. Otherwise, continue with the program.

The reader is advised to change the above prompt and redesign the program in order to find a solution for the prompt. An example is rather than find the difference of the most and least frequent alphabetical characters in the input, find their sum, or the sum of the two characters with an average frequency.

Consider more edge and corner cases. First sketch out a pencil and paper answer to the prompt. Then write an algorithm with technical language. Compose a program an test it with varying types of inputs. Finally, analyze the time and space complexity of the program. Try to change the data structures and operations of the program to find the optimal asymptotic time and space complexity.

Example two: Alternative data points

Suppose that there is another prompt. The reader is given a singly linked list. He or she is asked to return an array that includes the data point of every other node in the list, starting from the last and

moving toward the first. Iterate the linked list with recursion.

Start with a consideration as would happen with pen and paper. Sketch a linked list that could help focus the algorithm.

| | | | | | | |
|---|---|---|---|---|---|---|
| node, 1 | → | node, 2 | → | node, 3 | → | node, 4 |
| this.head | | | | | | this.tail |

Reconsider the prompt. The reader is to return the data of every other node, beginning from the tail node. In this example, the data of the tail node is 4. The node before that has the data point 3. The antepenultimate node has the data point 2. Before that, the head node has data point 1. If a singly linked list were traversable backward, from the tail to the head, then the reader could start at the tail, push the data point 4 to an array named result, iterate to the next node and iterate again, push the data point 2, and then return the array.

A problem with this idea is that a singly linked list is not traversable backward. To find a solution for this problem, consider the structure of the data. The length of the linked list provides a structural feature that can be used in order to achieve an appropriate extraction of data during a forward traversal. In this instance, since the length of the linked list is even, the reader can start the iteration of the linked list at the first node, and start the extraction of data at the second node during the traversal. Then if the reader follows the guidelines of the prompt and selects the data at every other node, the result is the array [ 2, 4 ] .

The specificity of the prompt is to be reconsidered. The required result has the elements from the end of the linked list to the beginning of the linked list. Therefore the order of the current result array needs to be reversed.

Consider the edge case in which the length of the linked list is not even but odd.

| | | | | |
|---|---|---|---|---|
| node, 1 | → | node, 2 | → | node, 3 |
| this.head | | | | this.tail |

The change indicates that the iteration of the linked list and the extraction of data from alternative nodes of the linked list, are both to begin at the first node. Once the result array is populated, like with that of the linked list that has an even length, the result array is to be reversed to comply with the requirement of the prompt.

Note also that regardless of the length of the linked lists, the odd versus even distinction allows an appropriate extraction of the data from the linked list.

At this point, the reader has a sketch with pen and paper that illustrates the principles to be arranged for the solution to the prompt. Try to compose and algorithm by arranging the principles. An example of an algorithm is given in figure 78.

Figure 78: An algorithm to collect every other data point of a linked list, from the tail to head

1. Iterate the linked list with recursion and increment a count of the number of nodes.
2. If the length is even, iterate the list again and collect data from the second node.
3. During the iteration, collect data from every other node until the last node is reached.
4. If the length is odd, iterate the list, collecting data from the first node and then every other.
5. Reverse the array that contains the data collected from the recursive iteration.
6. Return the reversed array as the solution for the prompt.

With a sketch and an algorithm, the reader is ready to create a program. An example of a program for the algorithm, is given in figure 79.

Figure 79: A program to collect every other data point of a linked list, from the tail to head

```
1       class Node {
2               constructor( data, next ) {
3                       this.data = data;
4                       this.next = next;
5               }
6       }
7
8       class LinkedList {
9               constructor( ) {
10                       this.head = null;
11                      this.tail = null;
12                      this.count = 0;
13              }
14
15              alternativeDataPoints( ) {
16                      if ( !this.head ) return null;
17                      else if ( this.head === this.tail ) return this.head.data;
18                      else {
19                              let result = [ ];
20                              const len = this.findLength_( );
21                              let isEven = ( len % 2 ) === 0;
22                              let collect = true;
```

```
23
24                  const collectData = function( node ) {
25                      if ( !node.next ) {
26                          if ( collect ) result.push( node.data );
27                          return;
28                      } else {
29                          if ( collect ) {
30                              collect = false;
31                              result.push( node.data );
32                              collectData( node.next );
33                          } else {
34                              collect = true;
35                              collectData( node.next );
36                          }
37                      }
38                  }
39
40                  if ( isEven ) collectData( this.head.next );
41                  else collectData( this.head );
42
43                  const temp = [ ];
44                  const len_ = result.length;
45                  for ( let i = len_ - 1; i >= 0; i-- ) temp.push( result[ i ] );
46                  result = temp;
```

```
47
48                              return result;
49                      }
50              }
51
52              findLength_( ) {
53                      let result;
54                      const searchLength = function( node, count ) {
55                              if ( !node.next ) {
56                                      count++;
57                                      return count;
58                              } else {
59                                      count++;
60                                      return searchLength( node.next, count );
61                              }
62                      }
63                       result = searchLength(this.head, this.count);
64                      return result;
65              }
66      }
67
68      const integers = new LinkedList;
69
70      const a = new Node( 1 );
```

```
71      const b = new Node( 2 );
72      const c = new Node( 3 );
73      const d = new Node( 4 );
74      const e = new Node( 5 );
75      const f = new Node( 6 );
76      const g = new Node( 7 );
77      const h = new Node( 8 );
78      const i = new Node( 9 );
79      const j = new Node( 10 );
80
81      a.next = b;
82      b.next = c;
83      c.next = d;
84      d.next = e;
85      e.next = f;
86      f.next = g;
87      g.next = h;
88      h.next = i;
89      i.next = j;
90
91      integers.head = a;
92      console.log( integers.alternativeDataPoints( ) );
```

A brief summary of the program in figure 79, is that there are two classes. The second class has two

methods, the second of which is a helper method. In line sixty-eight, an instance of the linked list is created and assigned to a declaration for a variable named integers. Lines seventy through seventy-nine declare the nodes for a test case. Lines eighty-one through eighty-nine define the links between nodes. Line ninety-one assigns the variable named “a” as the head node of the linked list. In line ninety-two, the evaluation of the alternativeDataPoints( ) method, is logged to the console.

The helper method this.findLength_( ) is called in line twenty. It iterates the linked list and returns a variable whose assignment is the count after the traversal.

After the invocation of the helper method in line twenty, a few variables are declared to prepare for the iteration that collects the data from the linked list. A function expression exists in lines twenty-four through thirty-eight. This is a piece of new syntax for the reader.

A function expression is a function that is assigned to a variable. The function proceeds as if it were a normal function, with one or two differences which are not important for the moment, but the reader is encouraged to learn more about the differences between the declaration and expression of a function in resources such as the Mozilla Developer Network and w3schools.

The invocation of the function expression collectData( ) happens either in line forty or forty-one. The function expression receives an argument that it accepts as a parameter named node. The contents of the declaration block, from lines twenty-five through thirty-seven, is an if-else statement. The if statement checks for the condition that there is not a link to a next node, with the syntax, ( !node.next ) . The condition evaluates as true when the iterator is at the final node. In that instance, after incrementing the count, the iterator returns.

The else statement contains a nested if-else statement. The condition of the nested if statement is that the variable named collect evaluates as true. When the variable named collect evaluates as true, then the interpreter first reassigns the variable named collect the value of false for the next iteration and then extracts the data point from the node that the iterator is at and pushes a copy of the data to the

array named result. When the variable named collect evaluates as false, the interpreter reassigns the variable the value of true for the next iteration, and then iterates the linked list.

The method of iteration is recursion. The recursive call is made in the lines 32 for the instance with collection and 35 for the instance without collection. When the iterator is at the end of the list, as mentioned, instead of recurring, it returns which ends the recursive call.

In lines forty through forty-eight, the variable array named result is reversed and the result is returned to the invocation of the function in line ninety-two. The interpreter then logs the result to the console.

The asymptotic space complexity of the program is linear because only half the data of the linked list is stored in the array named result. The other considerations of space are variables whose sizes do not grow with the size of the linked list. The linked list itself is a data structure that is given in the prompt, so it is not included in the analysis of space complexity. If the prompt specified that the linked list is included in the analysis of space complexity, then the space complexity of the program becomes linear, with the size of the linked list in its nodes and pointers. An advanced concept that concerns space complexity is the call stack. The reader is advised to learn more about the call stack to help in the analysis of the space complexity of recursive program.

The asymptotic time complexity of the program is linear because each of the nodes of the linked list is iterated. In fact, each of the nodes of the linked list is iterated twice: once in order to find the length of the linked list and the second time during the collection of data from the nodes. The fact that it occurs twice, does not change with an increasing input. If the length of the linked list is ten, then the linked list is iterated twice. If the length of the linked list is one hundred, then the linked list still is iterated only twice. This is a constant operation and so does not change the maximum class of the time complexity.

The reader is encouraged to create other linked lists and to modify the program for

experimentation. Possible variations include the extraction and copy of data from every third node from the beginning or end, or only nodes whose data is either even or odd. Try to encode a doubly linked list and test prompts on it. There are online resources with basic doubly linked list structures to use as a model for development.

Example three: Create and merge two sorted linked

If the prompt asks to create and merge two sorted linked lists, then the reader can begin by thinking about the procedure for the merger of two linked lists on a piece of paper with a pencil. First, draw two sorted linked lists.

List one:

| | | | | |
|---|---|---|---|---|
| node, 1 | → | node, 2 | → | node, 3 |
| this.head | | | | this.tail |

List two:

| | | | | |
|---|---|---|---|---|
| node, 1 | → | node, 1 | → | node, 3 |
| this.head | | | | this.tail |

The creation requirement is satisfied.

Next think about the process of merging them. Check the data at the first index of each list. In this example, the data are equal, 1 and 1. So there is no merge step. The data at the second node of each linked list, is not equal. But the data of the second node of the second linked list is equal to the data of

the first node of the second linked list. So the reader can merge the first two nodes of the second linked list into the first linked list, after the first node and before the second node. The result is given below.

List one:

node, 1 → node, 1 → node, 1 → node, 2 → node, 3

this.head this.tail

List two:

node, 3

this.head

this.tail

There is still a node in the second linked list. It can be merged either before the last node or after it since its value is equal to the value of the last node in the first linked list. The result is given below.

Merged list:

node, 1 → node, 1 → node, 1 → node, 2 → node, 3 → node, 3

this.head this.tail

It is apparent that edge cases will include the starting point of each list. Duplicate data points can also be considered. In principle, the structure of the lists is to be investigated during the composition of the

algorithm and program. The first and last data points, duplicated and frequent data points, the length of the lists, and so on. Any structural characteristic might be advantageous to use in the algorithm and program, and therefore worth even brief consideration. An example of an algorithm for the prompt is given in figure 80.

Figure 80: An algorithm that creates and merges two sorted linked lists

1. Declare a node class with attributes.
2. Declare a linked list class with a constructor and three methods.
3. Write an add( ) method that creates a sorted linked list.
4. Write a merge( ) method that merges two sorted linked lists.
5. Write a collection method that displays the data points on the nodes of the merged linked list.

The algorithm given in 72 is less specific than it could be. It is up to the reader whether having a skeletal structure or one more detailed is a preferable starting point for the creation of a program. An example of a program for the prompt is given in figure 81.

Figure 81: A program that creates and merges two sorted linked lists

```
1       class Node {
2               constructor( data, next ) {
3                       this.data = data;
4                       this.next = next;
5               }
6       }
7
8       class LinkedList {
9               constructor( ) {
```

```
10                  this.head = null;
11                  this.tail = null;
12                  this.result = null;
13          }
14          add( data ) {
15                  if ( !this.head ) {
16                          this.head = new Node( data );
17                          this.tail = this.head;
18                  } else {
19                          let node = new Node( data );
20                          this.tail.next = node;
21                          this.tail = this.tail.next;
22                  }
23          }
24
25          merge( list, list_ ) {
26                  let fast, runner;
27
28                  if ( !list && !list_ ) return null;
29                  else if ( !list && list_ ) return list_;
30                  else if ( list && !list_ ) return list;
31                  else if ( !list.head.next && !list_.head.next ) {
32                          if ( list.head.data === list_.head.data ) {
33                                  list.head.next = list_.head;
34                                  list.tail = list_.head;
35                                  this.result = list;
36                                  return list;
37                          } else if ( list.head.data < list_.head.data ) {
38                                  list.head.next = list_.head;
39                                  list.tail = list_.head;
40                                  this.result = list;
```

```
41                                      return list;
42                              } else {
43                                      list_.head.next = list.head;
44                                      list_.tail = list.head;
45                                      this.result = list_;
46                                      return list_;
47                              }
48                      } else if ( !list.head.next && list_.head.next ) {
49                              fast = list.head;
50                              runner = list_.head;
51                              if ( fast.data < runner.data ) {
52                                      fast.next = runner;
53                                      list_.head = fast;
54                                      this.result = list_;
55                                      return list_;
56                              } else if ( fast.data === runner.data ) {
57                                      fast.next = runner;
58                                      list_.head = fast;
59                                      this.result = list_;
60                                      return list_;
61                              } else {
62                                      while ( runner.next && runner.next.data <= fast.data ) {
63                                              runner = runner.next;
64                                      }
65                                      if ( !runner.next ) {
66                                              runner.next = fast;
67                                              list_.tail = fast;
68                                              this.result = list_;
69                                              return list_;
70                                      } else {
71                                              fast.next = runner.next;
```

```
72                                          runner.next = fast;
73                                          this.result = list_;
74                                          return list_;
75                                  }
76                          }
77                  } else if ( list.head.next && !list_.head.next ) {
78                          fast = list.head;
79                          runner = list_.head;
80                          if ( runner.data < fast.data ) {
81                                  runner.next = fast;
82                                  list.head = runner;
83                                  this.result = list;
84                                  return list;
85                          } else if ( runner.data === fast.data ) {
86                                  runner.next = fast;
87                                  list.head = runner;
88                                  this.result = list;
89                                  return list;
90                          } else {
91                                  while ( fast.next && fast.next.data <= runner.data ) {
92                                          fast = fast.next;
93                                  }
94                                  if ( !fast.next ) {
95                                          fast.next = runner;
96                                          list.tail = runner;
97                                           this.result = list;
98                                          return list;
99                                  } else {
100                                         runner.next = fast.next;
101                                         fast.next = runner;
102                                         this.result = list;
```

```
103                                         return list;
104                                     }
105                                 }
106                             } else {
107                                 fast = list.head;
108                                 runner = list_.head;
109                                 if ( list.tail.data <= list_.head.data ) {
110                                     list.tail.next = list_.head;
111                                     list.tail = list_.tail;
112                                     this.result = list;
113                                     return list;
114                                 } else if ( list_.tail.data <= list.head.data ) {
115                                     list_.tail.next = list.head;
116                                      list_.tail = list.tail;
117                                     this.result = list_;
118                                     return list_;
119                                 } else {
120                                     if ( fast.data === runner.data ) {
121                                         while ( runner ) {
122                                             while ( fast.next && fast.next.data <= runner.data ) {
123                                                 fast = fast.next;
124                                             }
125                                             const node = new Node( runner.data );
126                                             node.next = fast.next;
127                                             fast.next = node;
128                                             runner = runner.next;
129                                             fast = fast.next;
130                                         }
131                                         this.result = list;
132                                         return list;
133                                     } else {
```

```
134                                         if ( fast.data < runner.data ) {
135                                             while ( runner ) {
136                                                 while ( fast.next && fast.next.data <= runner.data ) {
137                                                     fast = fast.next;
138                                                 }
139                                                 const node = new Node( runner.data );
140                                                 node.next = fast.next;
141                                                 fast.next = node;
142                                                 runner = runner.next;
143                                                 fast = fast.next;
144                                             }
145                                             this.result = list;
146                                             return list;
147                                         } else {
148                                             while ( fast ) {
149                                                 while ( runner.next && runner.next.data <= fast.data ) {
150                                                     runner = runner.next;
151                                                 }
152                                                 const node = new Node( fast.data );
153                                                 node.next = runner.next;
154                                                 runner.next = node;
155                                                 fast = fast.next;
156                                                 runner = runner.next;
157                                             }
158                                             this.result = list_;
159                                             return list_;
160                                         }
161                                     }
162                                 }
163                             }
164                         }
```

```
165
166                 collectNodalData( list ) {
167                         const result = [ ];
168                         let runner = list.head;
169                         while ( runner ) {
170                                 result.push( runner.data );
171                                 runner = runner.next;
172                         }
173                         return result;
174                 }
175         }
176
177         const input = [ 1, 1, 2 ];
178         const input_ = [ 0, 1, 1, 2, 4 ];
179
180          // const input = [ -1, 1, 1, 2, 5 ];
181         // const input_ = [ 0, 1, 1, 2, 4 ];
182
183         // const input = [ 1, 1, 2, 5 ];
184         // const input_ = [ 1, 1, 2, 4 ];
185
186         // const input = [ 1, 1, 2 ];
187         // const input_ = [ 1, 2, 4, 6 ];
188
189         const linkedList = new LinkedList;
190         const linkedList_ = new LinkedList;
191
192         input.forEach( integer => linkedList.add( integer ) );
193         input_.forEach( integer => linkedList_.add( integer ) );
194
195         linkedList.merge( linkedList, linkedList_ );
```

```
196
197     const resultList = linkedList.result;
198
199     console.log( linkedList.collectNodalData( resultList ) );
```

A summary of the program in figure 81 begins with the note that size of the font was reduced in order that each statement could fit on one line. There are two classes. The second class has a constructor and three methods. The first method creates a linked list. The second method merges the linked lists. The third method collects the information from the nodes of the merged list.

In lines 192 and 193, is a piece of new syntax for the reader. The symbol => is called an arrow. The two call signatures of the lines are arrow functions with the .forEach( ) built-in method. The forEach( ) method takes each index of the array it is appended to, starting from the beginning and moving to the end, and calls the add( ) method that is appended to the variable named linkedList, which contains an instance of a new linked list that was created from the class.

In the call signature of the method is the variable that represents the value of the index during the iteration by the forEach( ) method. An arrow function is a compact piece of syntax that the reader will increasingly encounter if they continue to study JavaScript, especially in web development.

After lines 192 and 193, two linked lists are created. One is saved in the variable named linkedList and the other in the variable named linkedList_ . The nodes of the first linked list have the data points of the variable array declared in line 177. The nodes of the second linked list have the data points of the variable array declared in line 178.

In line 195, the merge( ) method of the linkedList instance is invoked. The arguments are the variables linkedList and linkedList_ . Note that the merge( )method of the variable named linkedList can take the variable linkedList as an argument.

Since there is no other new syntax, the reader can walk-through the example.

A basic description is that the edge cases of whether the linked lists are populated and to what degree serve as the conditions for the control flow. At first, the lists can either be appended to one another or a single runner can be used to traverse the longer list.

Once the context is established that each linked list has more than two indices, then the iteration of each list begins in line 106. A condition that tests for equality serves as the control flow that increments the iteration of each list.

The final method of the class is used to collect the data of the merged list. Before the return of each list in the merge( ) method, the class attribute this.result is set to the merged list so that the collectNodalData( ) method can begin at the head of the merged list for its traversal.

A few examples of different data sets that are edge cases are given in lines 180 through 187.

The asymptotic space complexity of the program is linear because the maximum class is determined by the linear creation of nodes with data from each input.

The asymptotic time complexity of the program is also linear because each linked list is iterated once, and then the final list is iterated once. That happens regardless of the size of the lists. The traversals are limited to the size of the lists.

There is a simpler way to merge two, sorted linked lists. It is to create a third linked list. Then iterate each of the first two linked lists and add to the third, the lesser of the nodes of the first two linked lists. If there is a correspondence between the data of the two linked lists, the first can be chosen each time. This approach contrasts with the above method, which combined the two linked lists into either of the first two. It is a more difficult and involved problem. The reader is encouraged to try to write an algorithm and program for it.

Example four: Validate parentheses

If the reader is given an array whose elements are strings of either empty or closed opening or closing parentheses, ( ) , brackets, [ ], or braces { } , validate the string. The array is validated by the rule that each punctuation sign is opened and closed, and that it is opened and closed in the correct order by its own type of punctuation.

The reader can begin to analyze the problem by considering examples of what a valid and invalid array look like. An example of a valid array is [ “ [ ”, “ ( ”, “ ) ”, “ ] ” ] , because the opening bracket is closed by a closing bracket and the opening parenthesis is closed by a closing parenthesis. Here is an example of an invalid array with the same characters: [ “ ( ”, “ [ ”, “ ) ”, “ ] ” ] . The array is invalid because although there is a closing parenthesis for every opening parenthesis and a closing bracket for every opening bracket, nonetheless the order is incorrect.

It is instructive to consider the moment at which the array becomes invalid. If the reader moves their pencil along the array, the status of the validity of the array at the first index is valid. At the second index, the array is also valid. At the third index, the array becomes invalid because it is at the third index that an incongruity between the type of opening and closing punctuation occurs.

Try to think of another example of an invalid array, but instead of mismatching punctuation, find another edge case. Here is an example: [ “ [ ”, “ ( ”, “ ) ”, “ ] ”, “ } ”, “ { ” ] . The example is incorrect because although there is a matching closing punctuation symbol for every opening one, the braces do not close one another. If the braces were inverted, the array is valid. Another edge case to consider is if there is an uneven number of a type of the punctuation. It is invalid because one punctuation remains unclosed.

Now that the reader has an idea of the nature of the problem, consider what type of data structure is suited to assist in the computation of the validity of input arrays. A stack lends itself well to the problem. During an iteration of the array, the stack can push each index that is an opening punctuation mark. If the iterator finds a closing punctuation mark, the stack can pop its most recent

element. If there is a match in type, then the principle of validity is maintained and the iteration can continue. If the iterated data is mismatched by type, then the program can return that the array is invalid. Figure 82 has an algorithm that describes the process.

Figure 82: An algorithm for validate punctuation

1. Iterate the input array.
2. For each element, check if it is opening or closing.
3. If the element is opening, push it to the stack.
4. If the element is closing, pop the most recent element from the stack.
5. Compare the popped element to the currently iterated element in the array.
6. If the popped element is an opening type and the iterated element that is closing, continue.
7. If the comparison in step 6 evaluates as false, return that the array is invalid.
8. If the iteration reaches the end of the string, if the stack is empty, return true.
9. If the stack is not empty, return false.

Figure 83 has a program for the algorithm in figure 82.

Figure 83: Validate parentheses

```
1       class Node {
2               constructor( data, next ) {
3                       this.data = data;
4                       this.next = next;
5               }
6       }
```

```
7
8       class Stack {
9               constructor( ) {
10                      this.head = null;
11              }
12
13              push( data ) {
14                      if ( !data ) return null;
15                      else if ( !this.head ) this.head = new Node( data );
16                      else {
17                              const node = new Node( data );
18                              node.next = this.head;
19                              this.head = node;
20                      }
21              }
22
23              pop( ) {
24                      if ( !this.head ) return null;
25                      else {
26                              let node = this.head;
27                              this.head = this.head.next;
28                              node.next = null;
29                              return node;
30                      }
```

```
31              }
32
33              validate( array ) {
34                      if ( array.length === 0 ) return null;
35                      const l = array.length;
36                      for ( let i = 0; i < l; i++ ) {
37                              if ( array[ i ] === ' ( ' || array[ i ] === ' [ ' || array[ i ] === ' { ' ) {
38                                      this.push( array[ i ] );
39                              } else {
40                                      const node = this.pop( );
41                                      if ( array[ i ] === ' ) ' && node.data === ' ( ' ) continue;
42                                      else if ( array[ i ] === ' ] ' && node.data === '  [ ' ) continue;
43                                      else if ( array[ i ] === ' } ' && node.data === ' { ' ) continue;
44                              else return false;
45                              }
46                      }
47                      if (!this.head) return true;
48              }
49      }
50
51      const inputOne =  [ '(', ')' ];
52      const inputTwo = [ '(', ')', '[', ']', '{', '}' ];
53      const inputThree = [ '(', ']' ];
54      const inputFour = [ '(', '[', ')', ']' ];
```

```
55      const inputFive = [ '{', '[', ']', '}' ];
56
57      const parentheses = new Stack;
58
59      console.log( parentheses.validate( inputOne ) );
60      console.log( parentheses.validate( inputTwo ) );
61      console.log( parentheses.validate( inputThree ) );
62      console.log( parentheses.validate( inputFour ) );
63      parentheses.pop( );
64      console.log( parentheses.validate( inputFive ) );
```

A summary of the program is that there are two classes. The second class has three methods: push( ), pop( ), and validate( ). Lines fifty-one through fifty-five have variable declarations with test cases as assignments. Line fifty-seven instantiates a new stack. Lines fifty-nine through sixty-four invoke the validation method with the variables that are assigned the test cases.

The reader is familiar with the syntax of the program and the concepts of a stack, so a walk-through is omitted here. Note instead that in lines forty, a constant variable node is popped from the stack. In lines forty-one through forty-three the attribute named data of the node, is accessed by dot syntax. The nodal attribute named data is the location of the storage of the strings that contain the punctuation. The access is necessary for the operations of comparison.

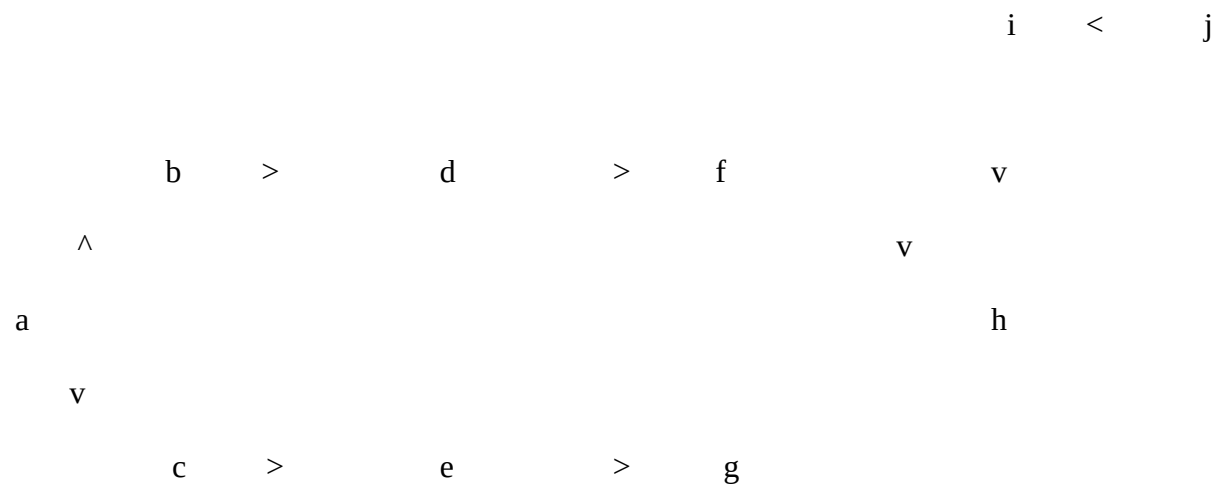
The asymptotic space complexity is for a valid array is linear because each opening punctuation is pushed to the stack and requires a node for storage. Invalid arrays return early, unless the invalidating index is the final index. Therefore invalid inputs perform better than do valid inputs, if both valid and invalid are the same length. Because asymptotic space complexity considers the worst case scenario as

its standard, therefore the characterization of linear is preferable.

The asymptotic time complexity for a valid array is linear because an iteration of each index is required to validate the array.

Example five: Route between graph nodes

Another prompt is to find if a route exists between two nodes in a graph. As with the other prompts, the reader is advised to begin by sketching a picture of a graph and thinking about the requirements of the prompt in relation to the structural features of the picture. Here is an example of a picture of a graph:




The graph has ten nodes. The carrots between the graph nodes indicate the direction of possible movement between the nodes. For example, from node “a”, the nodes “b” and “c” can be visited. From “b”, “d” can be visited. The node “f” can be visited from “d”. Node “h” can be visited from nodes “f” and “i”. But node “i” can be visited only from node “j”, which cannot be visited from any node.

After a picture is drawn, reconsider the prompt in relation to the picture. The prompt asks that it

is determined whether a route exists between two nodes. To generate ideas, list instances in which routes exist between nodes. One can also say that node “g” is visitable from node “a”, a > c > e > g, as is node “h”: a > b > d > f > h . In fact, any of the nodes that are in a series of pointers that give a visitable path, have a route to any other node in that path. There are paths between “a” and “c”, “a” and “e”, “c” and “e”, “c” and “g”, “a” and “b”, and so on.

Also list instances in which a route does not exist. The graph shows that a route does not exist between two nodes if there is not a series of traversable nodes between them. An example is that there is no path from “h” to another node. But other nodes can visit “h”. This raises an edge case. The prompt does not specify whether either of the nodes has to be designated as the start node and the other as the end node. The lack of specification enables the opportunity to try a route from each node to the other. For example there is no route between nodes “h” and “a”, if the traversal begins at “h”. If the traversal begins at “a”, however, there is a route between the nodes “a” and “h”.

Assume that the edge case is clarified that it is necessary to specify a start and an end node.

With a picture in mind and a consideration of an edge case, the reader is to consider what type of traversal will be used. Typically for the establishment of a route between two nodes, a breadth first search of the graph is used with a queue.

Now the reader is ready to compose an algorithm. An example algorithm is given in figure 84.

Figure 84: An algorithm to find if a route exists between two graph

1. Declare classes for a queue node and queue.
2. Declare a class for graph nodes and create a the nodes.
3. Create the graph with a declaration of the adjacency lists of the graph nodes.
4. Write a function that finds if a route exists between two nodes.
5. The function is a breadth first search.

6. Before enqueueing an adjacent node, check if the adjacent node is the end node.
7. Return true if the end node is found from a traversal of the start node.
8. Return false if the traversal completes without locating the end node.

Figure 85: A program to find if a route exists between two graph nodes

```
1       class QueueNode {
2               constructor( data, next ) {
3                       this.data = data;
4                       this.next = next;
5               }
6       }
7
8       class Queue {
9               constructor( ) {
10                      this.head = null;
11                      this.tail = null;
12              }
13
14              enqueue( data ) {
15                      let node = new QueueNode( data );
16                      if ( !this.head ) {
17                              this.tail = node;
18                              this.head = this.tail;
19                      } else {
```

```
20                          this.head.next = node;
21                          this.head = node;
22                  }
23          }
24
25          dequeue( ) {
26                  if ( !this.head ) return null;
27                  else {
28                  let node = this.tail;
29                  if ( this.tail === this.head ) {
30                          this.tail = null;
31                          this.head = null;
32                  } else this.tail = this.tail.next;
33                          node.next = null;
34                          return node;
35                  }
36          }
37  }
38
39  class GraphNode {
40          constructor( data, adj ) {
41                  this.data = data;
42                  this.adj = adj;
43          }
```

```
44      }
45
46      const a = new GraphNode( "a" );
47      const b = new GraphNode( "b" );
48      const c = new GraphNode( "c" );
49      const d = new GraphNode( "d" );
50      const e = new GraphNode( "e" );
51      const f = new GraphNode( "f" );
52      const g = new GraphNode( "g" );
53      const h = new GraphNode( "h" );
54      const i = new GraphNode( "i" );
55      const j = new GraphNode( "j" );
56
57      a.adj = [ b, c ];
58      b.adj = [ c, d ];
59      c.adj = [ e ];
60      d.adj = [ f, g ];
61      e.adj = [ f ];
62      f.adj = [ h ];
63      g.adj = null;
64      h.adj = [ f ];
65      i.adj = [ h ];
66      j.adj = [ i ];
67
```

```
68  function route( start, end ) {
69      if ( !start || !end ) return null;
70      else {
71          const visited = [ ];
72
73          queue.enqueue( start );
74          while ( queue.head ) {
75              const node = queue.dequeue( ).data;
76
77              if ( node.adj === null ) continue;
78              const adjacents = node.adj, l = adjacents.length;
79              for ( let i = 0; i < l; i++ ) {
80                  if ( !visited.includes( adjacents[ i ] ) ) {
81                      if ( adjacents[ i ] === end ) return true;
82                      else {
83                          visited.push( adjacents[ i ] );
84                          queue.enqueue( adjacents[ i ] );
85                      }
86                  }
87              }
88              visited.push( node );
89          }
90          return false;
91      }
```

```
92      }
93
94      const queue = new Queue;
95      console.log( route( a, h ) );
```

A summary of the program in figure 85, is that queue is instantiated in line 94 from its earlier class declaration. In line 95, a console log statement invokes the function named route( ) with two arguments, nodes “a” and “h” that serve as the start and end nodes for the search.

The reader is familiar with a breadth first search, so a walk-through of the program is omitted. Note instead that in line eighty-one, a comparison is made between the indexed adjacent node of the currently iterated node in the graph, with the parameter named end. The declaration block of the if statement returns true because a true evaluation of the condition indicates that a traversable path exists between the current node and the end node, and the current node is reachable from the start node either directly or through a path of traversable nodes.

In line ninety, is the return false statement that indicates the completion of the search without finding the end node.

The asymptotic space complexity of the program is linear because the array named visited contains a copy of every node that it visits, which increases and decreases per node with the size of the graph. In its worst case performance, the iteration visits every node. The average performance will statistically be better, but perhaps not logarithmic.

Note that a common way to express the complexity of graphs is in terms of V and E, where V is the number of vertices in a graph and E is the number of edges. In such notation, which is discussed in *Cracking the Coding Interview*, the answer to the example is O(V).

The asymptotic time complexity of the program is also linear. The time increases and grows

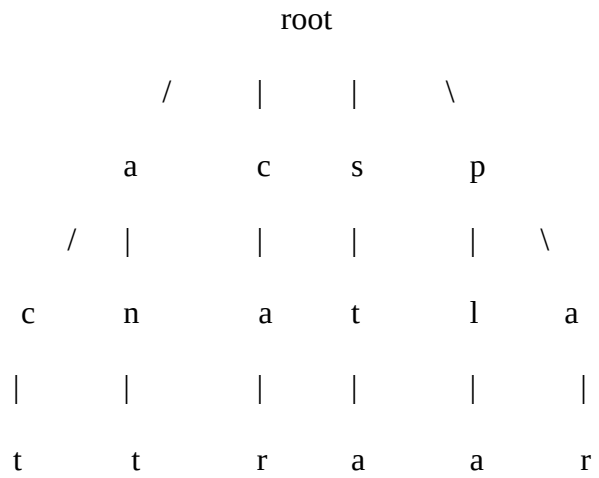
with the number of nodes and edges in the graph, all of which are visited in the worst case. The operations of comparison are compared in constant time and space and do not change the maximum class of the complexity analysis. In the preferable notation for graphs, the analysis of asymptotic time complexity is O(V + E).

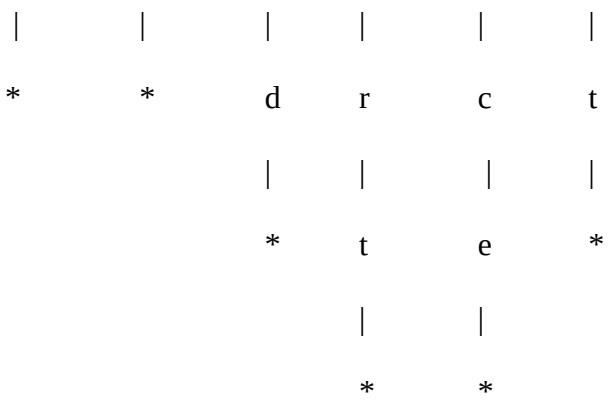
The reader is encouraged to modify the search in line ninety-five by changing the arguments. Also try experimenting with the structure of the graph, either by changing, introducing, or removing nodes as well as the connections among them.

Example six: A sentence to trie

The final example of the chapter has the prompt to transfer the words of a sentence into a trie. A trie is a new data structure for the reader. A trie is a type of tree with additional requirements. The requirements are that it begin with a root node, that each node that the root node connects to contains the data of a unique letter, and that the principle of unique connection continues along each branch of the tree until the leaf node of the branch, which should be an asterisk, * .

Here is a picture of an example of a trie:






The trie in the picture contains the words act, ant, card, start, place, and part. The root node has adjacent nodes whose data points are unique. Each node in a branch has adjacent child nodes that have unique data points. There are two splits in the branches: the node with the data point letter a connects to its child nodes with the letters c and n; the node with the data point letter p connects to its child nodes with the letters l and a. Note that the branches can occur at any level of the tree. The leaf node of each branch has the data point of an asterisk.

The prompt asks that the words of a sentence are transferred into a trie. Edge cases to be considered include the case of the letters, the characters in the sentence that are not letters but instead punctuation marks like commas, hyphens, and apostrophes, and the repetition of words.

An example of an algorithm that describes a solution to the prompt is given in figure 86.

Figure 86: An algorithm that describes putting the words of a sentence into a trie

1. Iterate the string and place its words into an array of strings.
2. Exclude all punctuation, except for hyphens in-between words.
3. Change upper case letters to lower-case letters.
4. Create the root node of a trie.
5. Iterate the array of strings and for each word, add it to the trie.

6. If a word already exists in the trie, return without an update to the trie.

An example of a program for the trie is given in figure 87.

Figure 87: A trie for the storage of a sentence

```
1       class Node {
2               constructor( data, adj ) {
3                       this.data = data;
4                       this.adj = [ ];
5               }
6       }
7
8       class Trie {
9               constructor( ) {
10                      this.root = null;
11              }
12
13              stringPreparation( sentence_ ) {
14                      const len = sentence_.length;
15                      let lowerCase = "";
16                      for ( let i = 0; i < len; i++ ) {
17                              lowerCase += sentence_.toLowerCase( i );
18                      }
19
```

```
20                  let temp = "";
21                  for ( let i = 0; i < lowerCase.length; i++ ) {
22                          const integer = lowerCase.charCodeAt( i )
23                          if ( 65 <= integer && integer <= 90 ) {
24                                  temp += lowerCase[ i ];
25                          } else if ( 97 <= integer && integer <= 122 ) {
26                                  temp += lowerCase[ i ];
27                          } else if ( integer === 32 ) {
28                                  temp += lowerCase[ i ];
29                          } else if ( integer === 39 ) {
30                                  temp += lowerCase[ i ];
31                          } else if ( integer === 45 ) {
32                                  temp += lowerCase[ i ];
33                          }
34                  }
35
36                  const result = temp.split( " " );
37                   return result;
38          }
39
40          add( data ) {
41                  if ( !data ) return null;
42                  const letters = data.split( " " );
43                  letters.push( '*' );
```

```
44          const len = letters.length;
45          let i = 0, marker = this.root;
46
47          if ( !this.root ) this.root = new Node( 'root' );
48          marker = this.root;
49
50          while ( i < len ) {
51              const adj = marker.adj, len_ = adj.length;;
52              if ( !adj.length ) {
53                  adj.push( new Node( letters[ i ] ) );
54                  marker = adj[ 0 ];
55
56              } else {
57                  for ( let j = 0; j < len_; j++ ) {
58                      if (adj[ j ].data === letters[ i ] ) {
59                          marker = adj[ j ];
60                          break;
61
62                      } else if ( j === len_ - 1 ) {
63                          adj.push( new Node( letters[ i ] ) );
64                          marker = adj[ adj.length – 1 ];
65                      }
66                  }
67              }
```

```
68
69                          i++;
70                  }
71          }
72
73          search( data ) {
74                  if ( !data || !this.root ) return null;
75                  else {
76                          const letters = data.split( " " );
77                          letters.push( '*' );
78                          const len = letters.length;
79                          let i = 0, marker = this.root;
80
81                          while ( i < len ) {
82                                  const adj = marker.adj, len_ = adj.length;
83                                  for ( let j = 0; j < len_; j++ ) {
84                                          if ( letters[ i ] === '*' && adj[ j ].data === '*' ) return true;
85                                          else if ( letters[ i ] === adj[ j ].data ) {
86                                                  marker = adj[ j ];
87                                                  break;
88                                          } else if ( j === len_ - 1 ) return false;
89                                  }
90
91                                  i++;
```

```
92                          }
93                  }
94          }
95
96          startsWith( data ) {
97                  if ( !data || !this.root ) return null;
98                  else {
99                          const letters = data.split( " " );
100                         const len = letters.length;
101                         let i = 0, marker = this.root;
102
103                         while ( i < len ) {
104                                 const adj = marker.adj, len_ = adj.length;
105                                 for ( let j = 0; j < len_; j++ ) {
106                                         if ( i === len - 1 && letters[ i ] === adj[ j ].data ) return true;
107                                         else if ( letters[ i ] === adj[ j ].data ) {
108                                                 marker = adj[ j ];
109                                                 break;
110                                         } else if ( j === len_ - 1 ) return false;
111                                 }
112
113                                  i++;
114                         }
115                 }
```

```
116                 }
117     }
118
119     const trie = new Trie;
120
121     let sentence = "Immediately, and according to custom, the ramparts of Fort Saint-Jean were covered with spectators; it is always an event at Marseilles for a ship to come into port, especially when this ship, like the Pharaon, has been built, rigged, and laden at the old Phocee docks, and belongs to an owner of the city.";
122
123     const sentence_ = trie.stringPreparation( sentence );
124
125     sentence_.forEach( word => trie.add( word ) );
126
127     console.log( trie.search( " " ) );
128     console.log( trie.search( 'custom' ) );
129     console.log( trie.search( 'ship') );
130     console.log( trie.search( 'city' ) );
131     console.log( '\n' );
132     console.log( trie.startsWith( 'always' ) );
133     console.log( trie.startsWith( 'rig' ) );
134     console.log( trie.startsWith( 'own' ) );
135     console.log( trie.startsWith( 'even' ) );
136     console.log( trie.startsWith( 'incorrect' ) );
```

A summary of the program, is that there are two classes. The second class has four methods: stringPreparation, add, search, startsWith. In line 119, a new trie is instantiated and assigned to the variable named trie. In line 121, a variable named sentence is declared and assigned a string with characters. Note that the sentence is from the first page of the novel *The Count of Monte Cristo* by Alexander Dumas.

In line 123, a variable named sentence_ is declared and assigned the result of the variable named trie and its appended method stringPreparation, which has the variable named sentence as its argument. In line 125, an arrow function puts each word of the variable array named sentence_ into the trie. Lines 127 through 136 contain console log statements whose call signatures contain invocations of the class methods search( ) and startsWith( ). The arguments of the call signatures are tests for the words in the sentence.

Because the trie is a new data structure for the reader, therefore a walk-through of the program is included. When the program starts, the interpreter reads the classes into memory. It finds the instantiation of the class in line 119, and assigns it to the variable named trie.

In line 123, the interpreter finds the stringPreparation( ) method appended to the variable named trie. The call signature of the method is the variable named sentence. The interpreter takes the argument from the invocation into the instance of the class, to the line of the declaration, line 13.

In line 13, the argument named sentence is accepted as a parameter named sentence_. The interpreter enters the declaration block. In lines 14 through 18, a for loop iterates the parameter named sentence_. For each index, the data point at the index is reduced to a lower case character by the built-in method toLowerCase( ). The character is then put into a variable string named lowerCase.

In lines 21 through 34, the interpreter iterates the string named lowerCase and compares the ASCII value of its character to the range of acceptable characters: the lower and upper case letters from

a to z. Acceptable characters are added to a variable string named temp.

In line 36, the variable string named temp is transformed into an array with the built-in method split( “ ” ) and assigned to a variable named result. In line 37, the interpreter returns the result the invocation of the method, in line 123.

In line 123, the value of the variable named result, which was returned from the method stringPreparation( ), is assigned to a variable array named sentence_ . In line 125, the built-in method forEach( ) is called on each index of the array. Since each index contains a string of prepared letters, that is the data that is in the call signature of the invocation of the method add( ), which is appended to the variable named trie. The first iteration of the forEach( ) method, finds at index 0 of the variable array named sentence_ , the string data point “immediately” . The interpreter takes the argument into the instance of the class that is assigned to the variable named trie, to the location of the declaration of the method, in line 40.

In line 40, the argument named word is accepted by the method as a parameter named data. In lines 41 through 45, preparation for the creation and iteration of a trie are made: the status of data is checked, the data type of the parameter is changed into an array named letters, an asterisk is pushed to the end of the array, and an iterator named i and a variable for its limit are declared.

In line 47, the interpreter checks the instance of a class for a root of the trie. The interpreter finds that no root exists. The interpreter creates a root node with the data point “root”. In line 48, a variable named marker is assigned as a pointer to root.

In line 50, a while loop is declared with the condition that the iterator is less than the length of the variable named len, which is equal to the length of the array named letters. In line 51, a variable named adj is declared and assigned the adjacent elements of the node that marker is currently at. In this example, the marker is at the root node. At the moment, root has no adjacent nodes, so adj is undefined and len_ is undefined.

Lines 52 through 67, contain an if-else statement. The condition of the if statement is declared in line 52. The condition is the negation of the length property of the variable named adjacent. Because the current value of adjacent is undefined, the evaluation of the negation of its length evaluates as true. The interpreter enters the declaration block.

In line 53, the currently iterated letter of the variable named letters is assigned to a new node that is then pushed into the adjacency list of the currently visited node. Note that because the variable named adj is an extension of the node where marker is currently pointing, therefore the assignment of the adjacent node is, in this instance, to the root node. In line 54, the variable named marker is iterated to the only node in the adjacency list of the node it currently points at.

At this point in the program, there is a root node with one adjacent node. The adjacent node has the data point that is the first letter of the first word in the parameter, the letter i .

In line 69, the iterator named i increments by one. The iterator returns to the beginning of the while loop in line 50. There it checks the condition that the iterator is less than the variable named length. The interpreter finds that the condition evaluates as true. The interpreter enters the while loop in its second iteration.

Lines 51 through 54 repeat the process that happened during the first iteration, except for the difference that the variable named marker is currently pointed at the only adjacent node in the adjacency list of the root node. Therefore the adjacency list of the only adjacent node to the root node, the node that has the data point “i”, has added to it, a node with the data point “m”. In line 69, the iterator increments again and then returns to line 50 to check for another iteration of the while loop.

In line 50, the interpreter finds that the evaluation of the while loop is true. It enters the declaration block. In line 51, the variable named adj is assigned the adjacency list of the variable named marker, which is currently pointed at a node with the data point “m”. The adjacency list of the node with the data point “m” is empty. So the condition of the if-statement evaluates to true, and the

adjacency list of the node with the letter “m” has pushed into it a node also with the letter “m”.

Note that the duplication of the letter does not cause a sub branch from the branch. The reason is that a branch can have duplicate letters. Said another way, for the first word of a trie, there is only one branch. This principle can be extended: during the iteration of a new branch on the root node, there are no sub branches. The sub branches can begin with the second iteration of a branch of the root node, when the new input can begin with the node of an established branch and then differ at some point in the iteration of the branch.

The process continues as has been described for the remainder of the variable named letters. When the condition of the comparison operation in line 50 evaluates as false, the interpreter exists the while loop and finds that there are no other instructions in the method. The interpreter returns to the invocation of the method, in line 125.

In line 125, the interpreter finds that the forEach( ) built-in method, has another string as an argument for the method named add( ). In this iteration of the forEach( ) method, the string evaluates to “and”, which is the second word of the sentence. The interpreter takes the argument into the declaration of the method in line 40.

Note that since the first letter of the argument is different from the first letter of the argument during the first iteration of the forEach( ) method, therefore the population of the trie with the current input will be a new branch without sub branches.

The first iteration of the while loop finds that the adjacency list of the node that variable named marker is currently pointing at, which is the root node, is not empty. The else block is entered in line 56 and the for loop begins in line 57. In line 58, the comparison operator evaluates as false for the only adjacent neighbor, because the letters i and a are different. The interpreter enters the else-if block in line 62. In line 63, a new node with the data point “a” is pushed to the adjacency list of the root node. The marker is assigned the index of the adjacency list of the most recently pushed node. In line 69 the

iterator increments and returns to the beginning of the while loop to check for the condition that either prompts its iteration or termination.

In line 50, the condition evaluates as true, and the iteration continues.

In each subsequent iteration of the while loop, after the first iteration, the adjacency list of the marker is empty as the iterator moves along the “a” branch of the trie. The if statement in line is followed in each iteration. The else statement in line 56 is never followed. For this reason, the remainder of the walk-through for this data point is omitted is omitted. The reader is encouraged to walk-through each step for themselves.

After the population of the second data point, the trie looks like this, now pictured horizontally:

```
                i – m – m – e – d – i – a – t – e – l – y – *
        /
root
        \
                a – n – d – *
```

Next the interpreter returns to line 125, to check if there is another input from the sentence.

In line 125, the interpreter finds that the forEach( ) method has another input, the argument “according” . Already the reader can perceive that because the first letter of this input matches the first letter of the input during the previous iteration, therefore this input will be the first instance during the execution of the program when the trie receives a sub branch. For this reason, there is a walk-through of the data point until just after the creation of the sub branch. After the creation of the sub branch, the population of the branch proceeds along familiar path.

The interpreter takes the argument to line 40, where the argument is transformed into the

parameter named data. In the first few lines of the method, the parameter is transformed into an array named letters with an asterisk appended to it, and preparation is made for the iteration of the trie.

In line 50, the condition for the while loop evaluates as true. The interpreter enters the declaration block of the while loop. In line 51, the variables adj and len_ are assigned according to the features of the node that the variable name marker is currently pointing at. The variable named marker is at the root node, so the adjacency list adj has nodes with the data points “i” and “a”.

The condition of the if statement evaluates as false in line 52, so the declaration block of the else statement is entered in line 56. In line 57, a for loop begins that iterates the adjacency list that is assigned to the variable named adj. The conditions of the if, else-if statement in lines 58 and 62 both evaluate as false, so the for loop enters its second iteration.

In the second iteration, the first condition evaluates as true, when “a” is found equal to “a”. The declaration block of the if statement is entered, in line 58. In line 59, the marker is updated to point at the node that has the data which was found in the comparison, which in this instance is the node with the data point “a” that is adjacent to the root node. In line 60, the for loop is broken.

The keyword break is a piece of new syntax for the reader. The keyword break stops the current iteration of a loop and prompts the start of the next iteration. In this instance, it is not required that the for loop break because the loop is already on its final iteration. In the future, however, there could be iterations during which the iteration would continue and the evaluation of the else-if condition would return a false positive. The condition evaluates as true, but the continuation of the syntax distorts the algorithm.

After the keyword break in line 62, the for loop ends. The iterator named i increments by one and return to line 50 to check for another iteration of the while loop. The condition evaluates as true, and the interpreter enters the for loop. In line 51, the variables are set to the node which has the data point “a”. The adjacency list is not empty, so the condition of the if statement in line 52 evaluates as

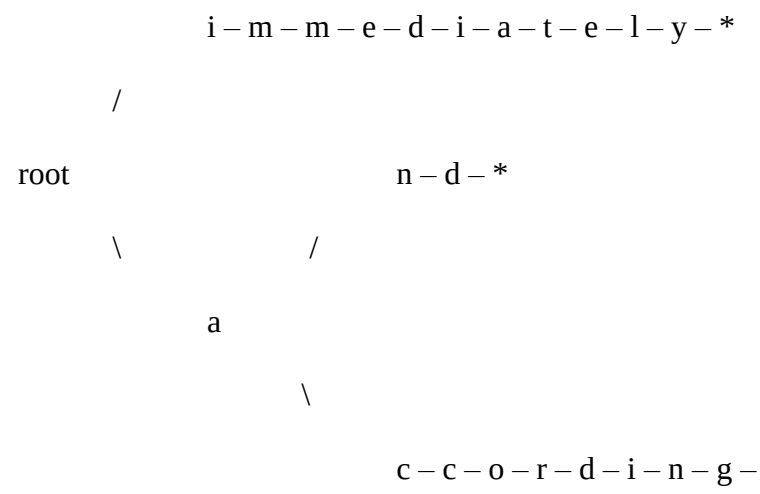
false. The interpreter enters the declaration block of the else statement.

In line 57, a for loop of the adjacency list of the node with the data point letter “a”, begins. The only element in the adjacency list is the letter “n”. The comparison evaluates as false in line 58. In line 59 the evaluation is true because the iterator named j has the value of one less than the length of the adjacency list: the index 0 is one less than the length 1.

The condition block of the else if statement is entered. A new node is added to the adjacency list of the node with the data “a”. The node now has two adjacent nodes, with the data points “n” and “c”. In line 64, the marker is reassigned the value of the most recently added node to the adjacency list. The declaration block of the else-if statement ends in line 67. This was the first sub branch of the trie.

In line 69, the iterator named i increments and returns to the beginning of the while loop to check for the condition of another iteration. The remainder of the insertion of the input into the trie, follows a path that the reader has already encountered. The adjacency list of each node along the branch for the input “according” is empty. For this reason, a walk-through of the rest of the insertion of the input is omitted. The reader is encouraged to make the walk-through themselves.

At this point in the program, the trie looks like this:



The paths of code in the add( ) method have been covered. So the reader could walk-through the remainder of the insertions. The other two methods of the class follow a pattern of iteration similar to that of the add( ) method. Differences exist on the point and value of return. But none of the syntax is new, so the reader should be able to review the methods and comprehend their processes. For this reason, a walk-through of the methods is omitted.

Note the variation in the types of test cases presented to the methods search( ) and startsWith( ) in the lines 127 through 136. The result of the tests is a Boolean value. The tests demonstrate that the trie was successfully created. Another way to test the try is to log it to the console, console.log( trie.root.data ) , and also, console.log( trie.root.adj ); . Inspection of the structure and contents of the trie can be made. Further iteration of the trie is achieved by indexing an element of the adjacency list. For example, console.log( trie.root.adj[ 0 ] ) . The adjacency list of that node can also be accessed: console.log( trie.root.adj[ 0 ].adj ); . It can be indexed: console.log( trie.root.adj[ 0 ].adj[ 0 ] ). This way the structure of the trie can be explored.

The space complexity of the add( ) method in figure 87 is linear because one node is created for each one unit of input, a letter of a word.

The time complexity of the add( ) method is also linear because in the worst case, all nodes in the trie are traversed to add a new node. In practice, the adjacency list of the root node and those of the nodes in the branches result in a faster iteration: not all nodes in the trie are iterated with each new input.

The reader is encouraged to modify the inputs of the trie and also the parameters of string preparation to allow the storage of different types of data.

## Summary

In this chapter, algorithms and programs were composed in order to find solutions for prompts. The algorithms and programs make use of data structures like strings, arrays, objects, linked lists, stacks, queues, graphs, and trees in order to achieve their computation. In addition to data structures, the language that encodes the program has many built-in methods and pieces of syntax that facilitate the computation.

In addition to the algorithms and programs, a walk-through was offered for the types of programs that are unfamiliar to the reader. In each example, an analysis of the the asymptotic time and space complexity were given. As metrics of the efficiency of a program, the asymptotes of complexity indicate the amount of resources that are needed to enact a computation.

In the next chapter, the conclusion, a summary of the course of the book is offered. Then there are suggestions of material that the reader can pursue in a continuation of their study of algorithms and programs.

# Chapter 5

## Conclusion

A summary of the course of the book begins with the introduction that describes a basic, intuitive sense of an algorithm and program. Examples of cooking and building are given. The introduction elaborates the algorithm as a sequence of acts that incorporate technical language in the description of a series of operations among data structures that create an output for an input. An example is given in the program Hello, World!, which uses JavaScript syntax.

After the introduction of an algorithm and program, the introduction briefly discusses asymptotic time and space complexity as a metric of the analysis of a program. The introduction also mentions the basics of web development, the Internet and World Wide Web, and the components of a computer.

In chapter two, the syntax of JavaScript is explained in a series of figures. The main topics of consideration are data type, operator, string, array, object, conditional logic, function, class, and built-in method. After an explanation of the topics, a set of examples is given for each topic. The examples illustrate the variations that each piece of the syntax often takes in a program. An example is the for loop with is sub indexation of an iterable such as an array. The specifications of off by one indexation and the limit of the iterator in the declaration of the condition of the for loop, are discussed.

In chapter three, data types are examined. The types include a hash table, linked list, stack and queue, and graph and tree. The operations in the figures consider the creation, access, and in some cases removal of elements from the data structure. The concept of node is introduced, as is that of helper function. In some of the topics, a walk-through of the program that exemplifies the execution of the program, is given so that the reader understands the procedure of the JavaScript interpreter. Please

note that if the reader did not read the section “What is a stack?”, and instead continued with other data structures, then they might read the final paragraph of the section “What is a stack?” for a note about the improvement of the efficiency of memory. A less efficient model is given throughout the rest of the book whenever a stack, and also a queue, are used in other data structures.

In chapter four, the topics of the previous two chapters are brought together in the composition of an algorithm and program in response to a prompt. Six examples are given whose data structures are an object, two linked lists, a stack, a graph, and a trie. Prompts include the determination of the middle elements of a string, the determination of alternative nodal points in a linked list, the combination of two sorted linked lists by merging them, the validation of an array whose indices have punctuation as data points, the determination if a route exists between two nodes, and the transference of data from a string to a trie.

The process in each example is a consideration of the prompt with a pencil and paper sketch, a composition of an algorithm, and then an encoding of the algorithm in a program. A discussion of the program in each example includes a walk-through of unfamiliar data structures, an explication of pieces of new syntax, and an analysis of the asymptotic time and space complexity of the programs.

If the reader has followed the course of the book, then he or she is versed in the syntax of JavaScript and has a conceptual understanding of data structures, algorithms, and programs. At this point, the reader is ready to encounter more and higher level syntax, algorithms, and programs.

The new material can take the form of a specification of the type of programming that interests the reader. For example, there is scientific computation and data visualization or alternatively the creation of video games. Also the reader can investigate the discourse of the genre which includes many luminous works such as *Cracking the Coding Interview* (2015) and instruction at the freeCodeCamp and on YouTube.

The reader is also encouraged to join websites like LeetCode and HackerRank in order to see

the types of prompts that are currently in consideration in the field of software engineering. Most positions of employment require some sort of technical interview. A cottage industry of preparatory instruction has emerged in response to the interest for such eminent positions.

There are many standard algorithms in the field of computer science. Although this book has algorithms as a preface to their elaboration in programs, often in the discussion of technical problems among computer scientists who know multiple languages, the algorithm is the focus of discussion. Implementation details are ancillary considerations because of the aptitude of the computer scientists in the composition of programs. The reader can start their investigation into this field of classic algorithms with examples such as binary search or merge and quick sort. These are important algorithms not covered in this book.

If the reader continues to develop with JavaScript, then he or she can look for examples of implementations of the common algorithms in a Google Search. The results are varied, and not all of the contributors are as equally fastidious with the efficiency of the program and the presentation of an analysis of it. At this point the reader is in a good position to be able to judge the work of others and also to create their own analyses of new programs that they encounter.

An implementation of most of the programs in the book is included in a repository on the GitHub page of the author: https://github.com/Colin-Pace/Data-structures-and-algorithms .

Bibliography

freeCodeCamp. https://www.freecodecamp.org/.

HackerRank. https://www.hackerrank.com/.

Kurose, James and Ross, Keith (2016). *Computer Networking*. Pearson.

LeetCode. https://leetcode.com/.

McDowell, Gayle Laakman (2020). *Cracking the Coding Interview*. Career Cup.

List of figures

23. The program HelloWorld, again

24. The basic syntax of a class

25. The charAt( ) built-in method

26. An example of the includes( ) built-in method

27. An example of the replace( ) built-in method for strings

28. The assignment to a variable of a new string from the built-in method replace( )

29. The built-in method concat( ) with two strings

30. The data types logged to the console

31. The conversion of a string to a number

32. The conversion of a number to a string

33. Automatic conversion of a number to a string by concatenation

34. Variations in the syntax of incrementation

35. Statements with multiple operations

36. The modulus operator

37. Variations in the declaration of a variable

38. Variations in the indexation of a string

39. Variation in the indexation of an array

40. The population of an object

41. An economical syntax for the population of an object

42. An if-else statement

43. An if, else-if, else statement

44. An if-else statement with the curly braces omitted

45. Nested if-else statements

46. A for loop whose iterator indexes each element of a string

47. A for loop with sub indexation

48. A for loop with sub indexation of two indices

49. A for loop with sub indexation and an inequality operation

50. A for loop with a sub index behind the iterator

51. A variation in the definition of the limit of an iterator

52. A for loop that iterates the iterable from the last to first index

53. A for loop that iterates from the last to first index with a sub index before it

54. The iteration of an object with a for loop

55. A while loop with sub indexation

56. The iteration of a two dimensional array with a nested loop

57. A while loop nested in a for loop

58. The transfer of data from a string to an array and to an object

59. The transfer of data from an array to a string and to an object

60. The transfer of data from an object to a string and to an array

61. fizzBuzz

62. A function and a helper function

63. A class with four methods

64. Variations in the built-in method split( )

65. Variations in the built-in method join( )

66. A hash table

67. A linked list with add( ) and remove( ) methods

68. A stack with push( ) and pop( ) methods

69. A queue with enqueue( ) and dequeue( ) methods

70. A breadth first search with a queue

71. A depth first search of a graph

72. A breadth and depth first search of a binary tree

73. An algorithm for finding the middle element in an array

74. A program for finding the middle element in an array

75. Another example of finding the middle element in an array

76. An algorithm to find the difference between the highest and lowest character counts

77. A program to find the difference between the highest and lowest character counts

78. An algorithm to collect every other data point in a linked list, starting from the tail

79. A program to collect every other data point in a linked list, starting from the tail

80. An algorithm to merge two sorted linked lists

81. A program to merge two sorted linked lists

82. An algorithm to validate punctuation

83. A program to validate punctuation

84. An algorithm to find if a route exists between two nodes in a graph

85. A program to find if a route exists between two nodes in a graph

86. An algorithm for a trie

87. A program for a trie