**Month 1: Week 2**

**1.2.1** [**Introduction to Computer Science**](https://thealtschool.com/lessons/introduction-to-computer-science/)

Computers have really allowed us to do some amazing things, things like international commerce, travel, global communications, medicine, online shopping, online dating, e.t.c. Computers have allowed us to do a wide range of things from texting using emoji, facetime, e.t.c but they’re not magical.

**1.2.2** [**What is Computer Science?**](https://thealtschool.com/lessons/what-is-computer-science/)

Theoretically, computer science is the study of computers including the hardware and software. It is a broad field that includes all aspects of computing, including figuring out new ways to solve problems using computers and improving how computers work.

Practically, computer science is simply about solving problems.

**1.2.3** [**Computer Components**](https://thealtschool.com/lessons/computer-components/)

**Learning goals**

At the end of this lecture, you will be in pole position to:

1. Explain what computer system is
2. Differentiate between computer hardware and software
3. Intensively and extensively discuss computer software
4. Differentiate between application software and system software
5. List the examples of application software and system software

**1.2.4** [**What is a Computer?**](https://thealtschool.com/lessons/what-is-a-computer/)

Take a deep breath, and ask yourself. Oh funny right? You think you know? Yes? No? You had known the definition of computer since you were in primary school? Exactly! But, what if I tell you that a computer is a human being? And that the difference between you and the computer is that it listens and does whatever you command it to do?  Would you believe it?  Now let’s get started!

A computer is an electronic device or machine that takes input from the user, processes data according to a set of instructions, outputs the data and temporarily or permanently stores the data internally. Just like human body metabolism, a computer is a combination of interrelated parts or components.

Human beings take-in food to get energy and computers, being an electronic device, require electrical energy to function effectively. A computer cannot perform any of its functions without its components and they are **Hardware** and **Software.**

**1.2.5** [**Computer Hardware and Software**](https://thealtschool.com/lessons/computer-hardware-and-software/)

From the word *Hard,* which means something that can be touched or held. Computer hardware can be defined as the physical components of a computer. Examples of computer hardware are Keyboard, Mouse, Processor, Memory devices, Monitor, Printer, and the Central processing unit etc.

From the word *Soft,* which means something that can neither be touched nor held*.*

Computer software refers to the components of a computer that perform specific tasks on a computer based on a collection of instructions or procedures. Computer software is a compendium of executable code that is written in high or low level programming language. Examples of computer software are Adobe Photoshop, Google Chrome, WhatsApp, Twitter, Slack, Operating System etc.

**1.2.6** [**Computer Software Deep Dive**](https://thealtschool.com/lessons/computer-software-deep-dive/)

You must have been wondering why we’re diving deeper into computer software and not hardware, well who doesn’t like a soft life? So, yeah! Welcome to the Soft life! Now, shall we begin?

The main reason why we are diving deeper into computer software is that it’s what AltSchool Africa School of Engineering is all about. People often think Computer Science as a profession is all about Microsoft Office (MS Word, Excel, etc) and Coreldraw, but I’m here to tell you that it’s more than that and as a matter of fact very broad. Today, with the advancement of technology, computers have become part of our life so much so that we can no longer do without it. So it’s safe to say that **Software Engineering** is from **Computer Software Engineering** and computers can be in the form of Mobile phones (Android, iPhone, etc), Laptop, Smartwatch, Smart TV, etc.  
There are two branches of computer software, which are **System Software** and **Application Software.** Just like computers comprises Hardware and Software, computer software also consists of system and application software. Things are getting broader now right? Don’t be dismayed, but remember, computer science is so broad and we’re only looking at the *Introduction to Computer Science.*

**1.2.7** [**System Software and Application Software**](https://thealtschool.com/lessons/system-software-and-application-software/)

**System Software** acts as an intermediary between the user and the computer. It is designed to use and maintain the computer hardware. For instance, think of a house, with running water and electricity, and well furnished, so System software is like the running water, electricity and furniture in the house and without these things it would be very difficult to live in the house. This means, without System software in a computer, Application software cannot be executed or run successfully.

However, system software is a concept of system programming, which involves the programming of software that accesses and controls the inner workings of a computer hardware. Ipso facto, System programming is another field and in fact, an entirely different course under Computer Science.

Examples of System Software include operating systems (OS), like MacOS, Windows and Linux. Other examples are firmware, device drivers, Utility software, etc.

**Application software** is a computer software that is designed to help users perform a specific task on a computer. Again for instance, think of a house, with running water and electricity, and well furnished but without home appliances such as Television, Home Theatre, Washing machine, etc. So Application software is the home appliances, it brings livelihood and effectiveness to the house. Imagine your Laptop or Phone without Browser, WhatsApp, Slack, Twitter etc, how boring and useless would they be? You get the point now? Okay, nice!

Application programming is a concept of application software and in fact, it involves building and developing the applications (Mobile, Desktop and Web) that run on computers and that’s what Altschool Africa school of engineering is all about and your journey to become an application software engineer begins now!

Examples of application software are Microsoft Office, VLC media player, WhatsApp, Slack, Google Chrome or Firefox, Twitter etc.

**Month 1: Week 3**

**(Introduction to Programming 2)**

**1.3.1 Introduction to Programming**

**1.3.1.1 Getting Started (**[**watch video**](https://thealtschool.com/topic/getting-started/)**)**

**1.3.1.2** [**What is Computer Programming?**](https://thealtschool.com/topic/what-is-computer-programming/)

It’s 10am in the morning, you’ve just woken up and you would like to make a cup of coffee. How would you go about it?

* Stand up
* Walk to your kitchen
* Boil water
* Add coffee and milk, if you like
* Your coffee is ready.

That’s how programming works, but now you’re writing these instructions for a computer to perform. Computer programming, also known as coding, involves writing well structured instructions for a computer to execute. (detailed instructions for a mindless machine).

**Computer Languages**

Language is a mode of communication that is used to share ideas, opinions with each other. For example, if we want to teach someone, we need a language that is understandable by both communicators.  
  
Unlike the steps to make your coffee which are detailed in English, a computer only understands machine language – zeros and ones.

A programming language is a **computer language** that is used by **programmers (developers) to communicate with computers**.  
  
It is a set of instructions written in any specific language ( C, C++, Java, Python) to perform a specific task.  
Computer programs are written in a programming language – but it doesn’t mean a computer can directly interpret that language. Programming languages are divided into 2:

**Low- level languages**

Low-level language is machine-dependent (0s and 1s) programming language. The processor runs low- level programs directly without the need of a compiler or interpreter, so the programs written in low-level language can be run very fast.

**Machine language**

Machine language is a type of low-level programming language. It is also called as machine code or object code.Machine language is easier to read because it is normally displayed in binary or hexadecimal form (base 16) form. It does not require a translator to convert the programs because computers directly understand the machine language programs.

**Assembly language**

Assembly language is also a type of low-level programming language that is designed for specific processors. It represents the set of instructions in a symbolic and human-understandable form. It uses an assembler to convert the assembly language to machine language.

**High level languages**

High-level programming language is designed for developing user-friendly software programs and websites. This programming language requires a compiler or interpreter to translate the program into machine language (execute the program).

**Compilers and Interpreters**

Compilers and Interpreters convert a computer program written in one language into another language, usually machine language which can be processed by a computer.  
  
However, while a compiler translates the entire program at once  an interpreter translates just one statement of the program at a time.  
  
A compiler is generally faster than an interpreter because because it analyses each statement just once, while an interpreter must analyse it each time (for example in a loop).

**1.3.1.3** [**Data and Information**](https://thealtschool.com/topic/data-and-information/)

Data are a collection of values that do not have a specific meaning. These values may be symbols, numbers , letters, facts, etc. An example of data is a list of dates.  
  
There are two types of Data – analog and digital.

Digital data is data that represents other forms of data using specific machine language systems that can be interpreted by various technologies.Devices that use digital data are Computers/Laptops/IPads/smartwatches.  
  
Analog data is data that is represented in a physical way.  e.g, Sound. Analog devices include loudspeakers and sound, some thermometers and amplifiers.

**Information** refers to processed data that is data with meaningful use. When data is interpreted, it provides context with which we can make informed decisions.

For example, collecting rainfall and temperature data over a period of time can help scientists predict weather conditions in a location. To appreciate the importance of the various breakthroughs in the history of the computer industry you will need a basic knowledge of how a computer works, so let us look at how a computer represents information.

**1.3.1.4** [**Bits and Bytes**](https://thealtschool.com/topic/bits-and-bytes/)

Do you Remember when I talked about computer language being ones and zeros? So computers work by manipulating 1s and 0s.  
These 1s and 0s are referred to as binary digits, which are in short called bits. Because these bits (1s and 0s ) are too small to be of much use, they are grouped together into units of 8 bits.  
This 8 bits units is what we now call a byte. Not the bite you are thinking about, I am sure you have heard of kilobyte, megabyte, etc.. so yeah, that byte!

So, a byte is the basic unit of a computer. Remember that a byte is a group of 8 bits. It is because of this fact that  number 8 and its multiples have become important in computing.  
You will specifically come across the numbers 8, 16, 32 and 64 in various computing contexts  and this is usually due to the 8-bit byte being the basic building unit.  
  
The key point to appreciate is that although basing your entire system on only two digits (1s and 0s)  may seem limiting, these two digits can be used to represent almost anything.  
  
This is an example of a byte 10000000 that can be used to represent an instruction or information. For example 10000000 could be an instruction representing “ start a program” which is basically telling the computer that it is the beginning or start of that particular program.

You will also hear people speak of kilobytes, megabytes and gigabytes or often just ‘K’, ‘meg’ and ‘gig’ as in, ‘This computer has 64 gigs of RAM’, or ‘This file is 45 Kb’.  
  
A kilobyte is 1024 bytes, Megabyte is 1024 kilobytes, gigabytes is 1024 megabytes  
It is however common to see 1000 used instead of 1024 in everyday usage.

Fun fact: A nibble refers to 4 bits. So if a nibble refers to a group of 4 bits, what do we call a group of 8 bits? Well, I know you know the answer.

It is important to note that A bit is denoted with a small letter **b**while a byte is denoted with a capital letter B

Bytes (as well as KB, MB, etc) are commonly used when referring to size of data on a computer while bit measurements can be seen more often in describing network speed.

Therefore, when your network speed is 200Mbps (Megabits per seconds), this means it will take about 8 seconds to download a file of 200 Megabytes (since 1 byte is 8 bits).

**1.3.1.5** [**Variables and Data types**](https://thealtschool.com/topic/variables-and-data-types/)

In programming, a variable is a value that can change, depending on conditions or on information passed to the program. Generally,  a computer  program consists of instructions that tell the computer what to do and data that the program uses when it is running.

The data consists of constants or fixed values that never change and variable values (which are usually initialised to “0” or some default value because the actual values will be supplied by a program’s user). Usually, both constants and variables are defined as certain data types. We will talk about data types later, now let us learn more about variables.

Variables are used to store data that is likely to be reused and/or manipulated in a program.  
Variables provide a way of labelling data so our programs can be understood more clearly by the reader and ourselves.Take a moment and think about a program that requires users to input their first name or age.  So a programmer will use variables such as firstName or age as variables because this is not fixed data

You can think of a variable as a data container which you call when you need the data that is in it.

It is important to use descriptive words when naming a variable as it can get difficult to manage unknown variable names in large programs.

**Assigning a value to a variable**

In most programming languages, the equals sign/assignment operator, ‘=’, is used to assign a value to a variable with the variable name on the left side of the operator and the value on the right.

For example in Python:

*name = ‘Altschool Africa’*

Some languages have special keywords that come before declaring a variable like JavaScript:

*const name = ‘AltSchool Africa’*

Whenever we need to access the name in our application, we can easily use the variable name and manipulate it based on the data type of the value assigned to our variable.

**Data Types**

In computer programming, data type is a classification of data which tells the compiler or interpreter how the programmer intends to use the data. Data type also defines a set of values and a set of operations that can be applied on those values. In simple terms, a data type specifies which type of value a variable has.

Imagine we have two values: 4 and ‘Ade Tiger’ – We can easily find the square of 4  but the same operation cannot be carried out on ‘Ade Tiger’.

Some operations that can be applied to values of one data type obviously do not make sense when applied to values of another type.

When we try the wrong operations on data types, an error is thrown either on compilation or execution of our program.

**Various Data Types**

Most programming languages support the following data types:

* Integers: A data type representing whole numbers, including negative ones.
* Floating point numbers: A data type representing numbers  with decimal parts
* Boolean operators: A data type representing logical False or True
* Text/String: A data type representing words containing characters which may be made up of letters, digits, symbols, signs, etc
* NULL: A data type representing unknown/unspecified values

**Type Checking in Programming Languages**

Type checking involves ensuring that operations carried out on values of each particular data type in a computer program are valid. An **type error** is thrown if an invalid operation is carried out.

For example, the code ‘x = 4 \* ‘Ade Tiger’’ will throw an error because the multiplication operator can only be used with numbers.

There are two major methods of type checking: Static and Dynamic.

Static type checking is done at compile time – when a compiler translates the source code, type checking is done and an error is immediately thrown and the code doesn’t run. This is useful because type errors are caught early in the development cycle. Seen in C++, Java, C, Go, Typescript etc

Dynamically typed programming languages do type checking at runtime – The program runs but throws an error when the mismatched type code block is executed. Seen in JavaScript, Python, Ruby. This can make it quite difficult to catch type errors in development especially if the program is not properly tested

**1.3.1.6** [**Data Structures**](https://thealtschool.com/topic/data-structures/)

Often, you’ll deal with large amounts of data that are related in some way and will need an efficient way to organize and structure them. For example, data of students in a class or employees in an organization.

A data structure provides you with a way of organizing data to use them effectively and efficiently. Think of it as a collection of data that are related.

Examples of data structures include arrays, stacks, queues, linked lists, heap, trees, etc.

**Arrays**

An array is a collection of data or items that are stored sequentiallyAn array contains values called ‘elements’ which are of the same data type.  
Each item in an array is indexed starting with zero. You may have heard of the programmer joke that programmers start counting from zero and not 1 like other people.

Think of an array  like a container of similar items, located together, which are ordered by the position of the item within the container for example a bookshelf, does it make sense now?  
A bookshelf consists of books stacked together and the books are ordered by their position in the bookshelf.

Circling back to indexing in arrays, the first book in the shelf will be index 0, the next book index 1 and so on..  
  
Arrays are commonly used as structures for building other, more complicated data structures. They are also used for sorting algorithms.

**Stacks**

A stack works almost exactly as it sounds. It’s like stacking elements within a tall container.  
It is a linear data collection that only allows addition and removal of items in a Last in First Out order (LIFO).  
  
An everyday illustrative example of a stack is  a stack of plates – the last item to be added to the stack is the first item to be removed.  
Stacks are commonly used for  evaluating mathematical expressions and to implement function calls in recursion programming.

**Queues**

A Queue is a similar data structure to stacks but instead follows a First In First Out order (FIFO)  
  
The easiest way to think about a queue is to think of a line of people waiting to enter a building.  
The person at the beginning of the line will enter the building first, while the person at the end will enter last.  
Queues can be applied to the following operations:

* Job scheduling for operations which may take a long time.
* To handle congestion in network requests used to implement priority queuing systems.

**Linked Lists**

A linked list is a linear data structure that includes a series of connected *nodes*. In simple terms, a linked list is a sequence of items arranged in a linear order all connected to each other. This means you must access data in order, so random access to data is not possible.

Each element in a linked list is called a “node,” and each node contains a key and a pointer.  
  
The pointer directs you to the next node, called a “next.” The sequence starts with a “head,” which directs you to the first element within the list.  
  
Therefore, The first node in a linked list is called the head, while the last node points to NULL.

Linked lists are used for symbol table management in switching between programs using Alt + Tab (On a PC).

**1.3.1.7** [**Conditionals, Loops, and Recursion**](https://thealtschool.com/topic/conditionals-loops-and-recursion/)

**Hash Tables**

A hash table is an efficient way to store, modify and retrieve data, especially when a large amount of data is involved.

A hash table stores elements in a key-value pair where the *key* is an integer unique to the hash table for indexing the values and the *value* is the data associated with the key. The integer value is generated by a hash function.

So, in a way, it is like an array that links a key to a specific data value.

A hash function takes the key and generates an integer of fixed size which will then be associated with that key. – used in cryptography, security, etc.

You can think of a hash table as a form of dictionary. The name is the key and the entry is a value.

A hash table can be useful when trying to keep a database of users in an organisation.

**Control Flow**

The control flow of a program is the order in which a program executes. Types of flow include:

* Sequential – This is the default flow of a program, with the code execution happening in the sequence it is written.
* Selection/Conditionals – A statement or set of statements is executed when a particular condition is True and ignored when the condition is False.
* Repetition/Loops – A sequence is repeated multiple times until a condition is met.

**Conditionals**

It is also known as Decision Control. It means a program will do one multiple alternatives based on if a condition resolves to true or false. For example, when making coffee, you can decide to add milk if you like it, else you can leave it.

Some conditional statements you can find in different programming languages include:

* if – continues as normal after the block
* if/else – alternatives follow the else block and then continues as normal
* nested if – an if inside an if. E.g, if (water) {if (boilingWater) {}}
* switch

**Switch statements**

A type of conditional statement – similar to if/else. It is present in programming languages like JavaScript, Java, C, C++, C#, etc.

Switch statements use keywords like switch, case, break, etc.  
  
A typical switch statement looks somewhat like this:

switch( expression ) {  
case value-1:  
Block-1;  
Break;  
case value-2:  
Block-2;  
Break;  
case value-n:  
Block-n;  
Break;  
default:  
Block-1;  
Break;}

**Loops**

Remember our code for making coffee from earlier? Now imagine we want to make cups of coffee for all 90 students in a class. How would we go about it?

We could rewrite our code 90 times, but that wouldn’t be efficient – as computers are designed to be. So we use something called a loop instead.  
A loop is a programming feature that lets us repeat a sequence of instructions until a condition is met – the break condition.

In the example earlier, that condition is the number of students in the class – 90.  
If a break condition is not specified, the loop will continue running for as long as your program is active. This is called an infinite loop.

**Types of Loops**

Most programming languages have at least two types of loops, both identified by their keywords – FOR loops and WHILE loops.

While both loops can be used interchangeably, FOR loops are more commonly used when we know the actual number of times we want the loop to run. (In our coffee example, 90 times).

The structure is usually in the format:

for (initialize; condition; increment/decrement)  
{  
doStuff()}

**Breakdown of a FOR Loop**

A FOR loop consists of a header portion and a body portion with the header typically consisting of 3 parts and the body containing the code to be executed while our condition remains true.

In the first part of the header, we initialize our loop variable.

This is then checked against a condition we declare in the second part. If our condition is met, the code in the body of our loop runs

Our loop variable is then increased or decreased by a value we declare in the third part of the loop header, checked against the loop condition and if it’s met, the loop runs again.

**A WHILE loop**

A while loop runs for as long as a conditional statement passed into it remains true. It is usually used when you do not know how many times your loop will run.

For example, instead of making coffee for 90 people, imagine we were making coffee till we run out of boiled water and we do not know how much water it would take to make a cup of coffee or be wasted in between

A while loop would be helpful in this case.

while (boiledWater) {  
makeCoffee()  
}

Usually, there’d be a way to update the boiledWater variable in our makeCoffe function. So, whenever we makeCoffee, the boiledWater variable is updated.

It becomes falsy when we run out of boiledWater and the while loop stops running

**Recursion**

Imagine reading through a wikipedia entry about an historical event, and then you come across another interesting event, you click on that, find another and keep clicking until you no longer find anything that interests you, and then stop.

A recursion is a program that calls itself until it gets to a base condition. A recursive function can call itself directly or indirectly.

A direct recursive function calls itself, within itself, while an indirect recursive function calls other functions that eventually calls the original function.

It is important to have a base condition else, your function continues to call itself infinitely.

Usually, a recursion problem can be solved with loops and vice versa.

**Practical Recursion Example**

Let’s try to create a recursive function that adds integers from 1 to a given number. (This code is in JavaScript)

function sumRange(range) {  
return range + sumRange(range – 1)  
}

The code above will run infinitely or throw an error as there is no base case, so we should add one.

function sumRange(range) {  
if (range === 1) {  
return 1  
}  
return range + sumRange(range – 1)  
}

This works, but for only positive numbers only. The same error thrown in the first code will be seen here if 0 or a negative number is shown.

function sumRange(range) {  
if (range <= 0) {  
return -1  
}  
if (range === 1) {  
return 1  
}  
return range + sumRange(range – 1)  
}

With a base case and type checking, we now have a working recursive function to find the sum of numbers up to a range.

Recursive functions are used in a lot of sorting algorithms.

**1.3.1.8** [**Big O Notation**](https://thealtschool.com/topic/big-o-notation/)

It is also known as time complexity. It defines the longest time it can take a program to perform an operation.

Some common complexities are:

* O(1) – Constant time complexity: when you try to access a value with its index in an array.
* O(n) – Linear time complexity – Inserting an item into a hash table, or in real life, reading a book with n pages, simple search. Time increases as input increases.
* O(log n) – Logarithmic time complexity. – binary search in a sorted array. As input size grows, operation time increases slowly (linear).
* O(n^2) – Quadratic time complexity – very bad, using nested arrays.

**1.3.1.9** [**Operating Systems**](https://thealtschool.com/topic/operating-systems-2/)

An operating system is the most important software that runs on a computer. It manages the computer’s memory and processes, as well as all of its software and hardware.

It also allows you to communicate with the computer without knowing how to speak the computer’s language.

**Linux**

Linux (pronounced LINN-ux) is a family of open-source operating systems, which means they can be modified and distributed by anyone around the world. This is different from proprietary software like Windows, which can only be modified by the company that owns it. The advantages of Linux are that it is free, and there are many different distributions—or versions—you can choose from.

According to StatCounter Global Stats, Linux users account for less than 2% of global operating systems. However, most servers run Linux because it’s relatively easy to customise.

* Ubuntu, Kubuntu, Debian, Kali, Rufus, Mint

**1.3.2 Programming Paradigms**

**1.3.2.1** [**Introduction to Programming Paradigms & OOP**](https://thealtschool.com/topic/intro-to-programming-paradigms-oop/)

Programming Paradigms refer to a way of solving problems using the features available in a language. In other words, the style of programming

**Imperative programming paradigm**

This uses the imperative mood (making a command or request). It uses a series of commands, which specify what the computer has to do – and when – in order to achieve a desired result. E.g, Java, Basic, Fortran, Assembler, etc

* Procedural programming
* Object Oriented Programming

**Declarative programming paradigm**

This focuses more on what a program should accomplish and less on ‘how’ to go about it. Programs describe their desired results without explicitly listing commands that must be performed. It may seem like ‘magic’ but there’s a lot of work going on under the hood.

* Functional programming

An example is furniture assembly: While imperative programming provides instructions for assembly, declarative programming provides a picture of the finished piece of furniture as a template.

Most modern languages support both paradigms to an extent.

**Procedural programming**

Procedural programming follows a linear, step-by-step approach to creating software. Each procedure is usually a function or a set of functions. Languages that follow procedural programming concepts include Pascal, FORTRAN, etc

An application to use a library, when written in procedural programming format would look somewhat like this:

* function register () {}
* function addBook () {}
* function removeBook () {}
* function findBook () {}
* function borrowBook () {}
* function returnBook() {}

**Advantages of Procedural Programming**

* Very useful in general-purpose programming
* Easy to learn
* Code can be reused in different parts of the program just by calling the function
* Easy to transfer skills to another language

**Downsides to Procedural Programming**

* Does not work for complex application
* The data is exposed to the whole program, hence it is not very safe.
* It can be hard to modify and debug as the application gets managed.
* Difficult to create new data types
* Fails to model real world applications making it difficult to design

**Object Oriented Programming**

In real life, a lot of things are related and share similar properties. All dogs bark, everybody eats. OOP structures a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects

Classes often represent broad categories, like Car or Dog that share attributes. These classes define what attributes an instance of this type will have, like color, but not the value of those attributes for a specific object.

Classes can also contain functions, called methods available only to objects of that type.

For example, say we created a class, Dog, to contain all the properties a dog must have, name and color. We then create an instance of a Dog type object, myDog to represent my dog.

We could then set the value of the properties defined in the class to describe my dog, without affecting other objects or the class template.

We can then reuse this class to represent any number of dogs. The four principles of object-oriented programming are **encapsulation**, **abstraction**, **inheritance**,and **polymorphism**.

**Encapsulation**

This principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class.

Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods.

**Abstraction**

Think — a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button. Applying abstraction means that each object should only expose a high-level mechanism for using it.

This mechanism should hide internal implementation details. It should only reveal operations relevant for the other objects.

Preferably, this mechanism should be easy to use and should rarely change over time. Think of it as a small set of public methods which any other class can call without “knowing” how they work.

**Inheritance**

Objects are often very similar. They share common logic. But they’re not entirely the same. So how do we reuse the common logic and extract the unique logic into a separate class?

One way to achieve this is inheritance.For example, consider a (parent) class Person, that has methods walk and talk. Chris is an object created from class Person but Chris is a teacher. Now a teacher is a person but not every person is a teacher. We can create a (child) Teacher from Person, add a method *introduce* to it, and then create the object Chris from Teacher.

Therefore the common logic remains in Person, which is inherited by Teacher and is therefore present in chris.

When we call chris.walk(), the walk() method moves up the chain of child to parent classes, to find where the walk method is defined.

This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.

**Polymorphism (Many-forms)**

Polymorphism gives a way to use a class exactly like its parent so there’s no confusion with mixing types. But each child class keeps its own methods as they are. Polymorphism allows different types of objects to pass through the same interface. (shapes class, circle, rectangle, etc)

Assume we have classes Circle and Square derived from class Shape, and class Shape as a method calcArea for calculating area of the shape, we can add different implementations of the getArea method to each inherited class and which will then return the area for that particular class. The method can take many forms.

**Advantages of OOP**

* Modularity: Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
* Flexibility: Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.
* Security: Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.
* Reusability: Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.

**Criticisms of OOP**

* Can be complicated to write
* Steep learning curve: The thought process involved in OO programming may not be natural for some people, and it will take the time to get used to it.
* Effort: Object Oriented programs require a lot of work to create. Specifically, a great deal of planning goes into an object oriented program well before a single piece of code is ever written. Else, it can become very hard to manage.

**1.3.2.2** [**Functional Programming**](https://thealtschool.com/topic/functional-programming-2/)

So far, we’ve been talking about imperative programming in which we use a series of commands, which specify what the computer has to do – and when.

With this method, we often have to rewrite a lot of code or create a new application when a drastic change needs to be made in our application due to interdependence between modules. Functional programming is declarative instead – telling the computer what you want to do.

Functional programming removes interdependencies between programs by replacing procedures with pure functions, which requires the use of shared and immutable state. Functional Programming Languages include: Lisp, Erlang, Haskell, Clojure.

**What is a pure function?**

Given the same inputs, always returns the same output and has no side-effects. Doesn’t change the global variable, hence it has no side effects.

**Side Effects**

A side effect is any application state change that is observable outside the called function other than its return value. Side effects include:

* Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain)
* Writing to the screen
* Writing to a file

**What is a shared state?**

Shared state is a variable that can be accessed from more than one function. Often, in object oriented programming, objects are shared between programs and functions by adding properties to other objects.

Functional programming avoids shared state — instead relying on immutable data structures and pure calculations to derive new data from existing data

Another common problem associated with shared state is that changing the order in which functions are called can cause a cascade of failures because functions which act on shared state are timing dependent.

**Immutability**

An immutable object is an object that can’t be modified after it’s created. Conversely, a mutable object is any object which can be modified after it’s created.

Immutability is a central concept of functional programming because without it, the data flow in your program can be lost and and strange bugs can creep into your software.

**Advantages of Functional Programming**

* Functional code tends to be more concise, more predictable, and easier to test than imperative or object oriented code.
* Pure functions always produce the same output and have no external values affecting the end result. Because of this aspect of pure functions, algorithms developed using functional programming are easier to debug.
* Static Variables – Here, the user cannot modify the variables once it has been initiated. So it is secure.
* If we write a bunch of pure functions, we get to leverage referential transparency to abstract away and hide complexity.

**Disadvantages of Functional Programming**

* Steep learning curve
* I/O operations rely on side effects, so they are almost non-functional in FP
* Recursion, which is quite popular with FP is memory intensive which can affect speed.

**1.3.2.3** [**Code Editors**](https://thealtschool.com/topic/code-editors/)

A text editor that is specialized for writing software. A source code editor may be a stand-alone program or part of an integrated development environment (IDE). They make writing and reading the source code easier by differentiating the elements and routines so programmers can more easily look at their code.

Source-code editors have characteristics specifically designed to simplify and speed up typing of source code, such as syntax highlighting, indentation, autocomplete and brace matching functionality.

An IDE is a text editor, a code editor, a debugger, compiler and more all under a single tool belt.

An IDE, on the other hand, is a set of software development tools designed to make coding easier. It combines the different aspects of a computer program into a single GUI, simplifying the whole software development process.

**Notable examples**

* Atom
* Brackets
* Emacs
* Notepad++(Windows)
* Sublime Text
* TextMate (macOS)
* UltraEdit
* vi/Vim
* Visual Studio Code

**Command Line**

The command line is a quick, powerful, text-based interface developers use to more effectively and efficiently communicate with computers to accomplish a wider set of tasks. You can do anything you can do on your GUI on the command line and even more. It is also faster

A command line also comes with your OS but you can download third party ones which sometimes come with more functionality.

* Windows: Mintty, Git Bash
* MacOS: zsh, powershell, iTerm

**Linux**

* Zsh, powershell

**CMD Commands**

* ls
* cd
* mv
* mv “filename” “path/to/new/file/location”
* cp “filename” “newfilename”
* touch create file
* mkdir
* rm -R “/path/to/root/directory”
* sudo – superuser privilege
* clear – clear terminal
* whatis “command – Get one-line description for a command
* exit – close out the current session in the Terminal

**1.3.3 Algorithms**

**1.3.3.1** [**What is an Algorithm?**](https://thealtschool.com/topic/what-is-an-algorithm/)

To cook a new recipe, one reads the instructions and steps and execute them one-by-one, in the given sequence. The result thus obtained is the new dish cooked perfectly. An algorithm is a set of instructions for solving a problem or accomplishing a task

As humans we can easily solve everyday problems without thinking about them due to our experiences and memorisation but it is very important to be able  break our thought process down into its individual steps and translate that to what computers can do.

Breaking a problem into individual parts will help you become a much better programmer and problem-solver.

A good Algorithm should be:

* Clear and Unambiguous – One meaning
* Inputs and input types should be clear – integers not numbers, same with outputs
* Practical – should be applicable and able to be created
* Should be interpretable in any language

**Let’s write an algorithm**

* Let’s write an algorithm to find the average of 2 numbers and print the result:  
  + STEP 1: Declare 3 variables a,b,sum as integers
  + STEP 2: Get input a and b from user
  + STEP 3: sum=(a+b)/2
  + STEP 4: print sum
* Easy to understand and write in a language of your choice.

**Popular Algorithms** – Binary Search

Suppose you’re trying to find your friend Sam (who is 5’5’’) in a single-file line of people that have been ordered by height from left to right, shortest to tallest. It’s a really long line, and you don’t have time to go one-by-one through the whole thing, comparing everyone’s height to Sam’s. What can you do?

* You select the person in the very middle of the line, and measure their height. They’re 5’7’’. So right away you know that this person, along with everyone to their right, is not Sam.
* Now that you’ve cut your problem in half, you turn your attention to the remainder of the line and pick the middle person again. They’re 5’4’’.
* So you can rule out that person and anyone to their left, cutting the problem in half again. And so on.
* After just five or six of these splits, you quickly home in on Sam. In doing so, you have followed the binary search algorithm.

**How to get better at writing algorithms**

* Algorithms and Data structures questions regularly come up at interviews and a deep understanding and their application can help you get to the top of your field in good time.
* Study popular algorithms
* Write your own algorithms, start simple
* Master Data Structures
* Spaced Repetition
* Practice

**1.3.3.2** [**Closing the Curtain**](https://thealtschool.com/topic/closing-the-curtain/)

To cook a new recipe, one reads the instructions and steps and execute them one-by-one, in the given sequence

**Month 1: Week 4**

**(Introduction to Programming 2)**

**1.4.1** [**Introduction to Programming Languages 2**](https://thealtschool.com/lessons/month-1-week-4-introduction-to-programming-languages-and-tools-of-the-trade/)

This week, we are putting our focus on consolidating all the knowledge we grabbed from last week’s learning content on Introduction to Programming. We would need to have a strong grasp on the fundamental concepts we got introduced to last week. Take a look at our slides from last week’s live classes by clicking on the link below:

<https://docs.google.com/presentation/d/16-2s_RHJGzHpbAfU6VtnIiW3Ovg9beqg_2lXtTV4NBw/edit?usp=sharing>

**Important accounts to open:**

1. Github
2. Medium
3. FreeCodeCamp.org
4. CodeAcademy
5. W3Schools

**Useful Resources:**

* Stack Overflow
* Mozilla Documentation Network
* FreeCodeCamp forums
* Stack Exchange
* Reddit r/programming, r/webdev, r/learnprogramming

**Further Research:**

1. <https://www.freecodecamp.org/news/introduction-to-computer-programming-and-computer-science-course/amp/>
2. <https://levelup.gitconnected.com/a-brief-introduction-to-programming-and-computer-science-d0195195c2d5>
3. <https://www.freecodecamp.org/news/a-gentler-introduction-to-programming-1f57383a1b2c/amp/>

**Test Your Knowledge of Week 3:**

<https://forms.gle/ydn1aRxfDELQwm6LA>

**Month 2: Week 1**

**(Welcome to Frontend Engineering)**

**2.1.1 Building Blocks**

**2.1.1.1** [**Building Blocks of Frontend Engineering**](https://thealtschool.com/topic/building-blocks-of-frontend-engineering/)

* [**HTML**](https://developer.mozilla.org/en-US/docs/Web/HTML) – is the language used to describe and define the content of a web page.
* [**CSS**](https://developer.mozilla.org/en-US/docs/Web/CSS) – is used to describe the appearance of web content.
* [**JavaScript**](https://developer.mozilla.org/en-US/docs/Web/JavaScript) –  is a programming language used to add interactivity to a website.

HTML, CSS and JS are the basic web technologies.

For example, with a registration page, we have a form, defined by HTML, and if centred and has colors, this is done by CSS. And to collect and save somewhere the information provided by a user, we use JS.

**2.1.1.2** [**WEB**](https://thealtschool.com/topic/web/)

**Now let’s look at how the web really works. When you type an address of a website in a browser tab, and press enter, these are the things that are involved.**

* The client which is the (the browser).
* The web server which is (where a web page or a resource is stored or retrieved from)

[**https://altschoolafrica.com**](https://altschoolafrica.com)

Look at this example web address – also called a URL – a short form for a Uniform Resource Locator. A URL is used to locate or request for resources on the internet. Each url points to a unique resource. This address retrieves the Alt School website’s home page.

**Resources requested by a browser using a url could be;**

* Web Pages
* Images
* Video and audio files
* Fonts

When you visit a web address;

* **The browser** through the URL (Universal Resource Locator) sends a request message to a web server to provide or retrieve resource stored at the given web address or URL. It could be a home page of a given website.
* The transfer occurs via **HTTP(S)** – which is a language the client and the server use to communicate.

**HTTP(S): Hypertext Transfer Protocol – Secure**

Let’s look at what happens when requesting for a web page as a resource via a URL.

**HTML Page Request**

Once the browser or the client sends an HTML page request to the server, it constructs what we call a DOM from the server response sent via HTTPS – resulting into a web page – which is a collection of HTML Elements that might contain content such as images, text, links and so on. You will see this in action later.

The **Document Object Model (DOM)** is the data representation of the objects that comprise the structure and content of a document on the web. You do not need to worry about this at all at this time.

And if a page contains other resources such as images and fonts – each have an address or a URL – the browser sends parallel requests via HTTPs to fetch the resources. When all resources are fetched, renders(displays) the web page.

With this introduction on how the web works, we can  now dive into HTML, CSS and JS.

**2.1.2 Introduction to HTML**

**2.1.2.1** [**Introduction**](https://thealtschool.com/topic/introduction-2/) **(**[**watch video**](https://thealtschool.com/topic/introduction-2/)**)**

**2.1.2.2** [**What is HTML**](https://thealtschool.com/topic/what-is-html/) **(& History of HTML)**

HTML is a markup language that defines the structure of your content. HTML consists of a series of [elements](https://developer.mozilla.org/en-US/docs/Glossary/Element), which you use to enclose, or wrap, different parts of the content to make it appear a certain way, or act a certain way.

* In 1989,  Tim Berners-Lee invented the Web with HTML as its publishing language. The language has evolved from its version 1.0 with limited tags and capabilities to now HTML5 which has more capabilities that we will discuss later on.
* **More on this** [link](https://www.w3.org/People/Raggett/book4/ch02.html)

**2.1.2.3** [**The Basics of HTML (Hypertext Markup Language)**](https://thealtschool.com/topic/the-basics-of-html-hypertext-markup-language-2/)

Now let’s jump into learning more about HTML as a building block of the web.

This is what we are going to cover in this lesson about HTML.

* Anatomy of an HTML Element
* Nesting Elements
* Empty Elements
* Anatomy of an HTML Document
* Comments in HTML
* Images
* Marking up text in HTML
* Links
* Forms
* Tables
* In HTML, If we wanted the content above to be a stand-alone paragraph, we would put it in a paragraph element as shown.
* The web browser will then display “Hello HTML” when the web page with the content is opened it.
* Now, let’s learn more about an HTML Element in the next section.

**2.1.2.4** [**Anatomy of an HTML Element**](https://thealtschool.com/topic/anatomy-of-an-html-element/)

The main parts of an HTML element are as follows:

* **The opening tag:** This consists of the name of the element (in this case, p), wrapped in opening and **closing angle brackets**. This states where the element begins or starts to take effect — in this case where the paragraph begins.
* **The closing tag:** This is the same as the opening tag, except that it includes a forward slash before the element name. This states where the element ends — in this case where the paragraph ends. Failing to add a closing tag is one of the standard beginner errors and can lead to strange results.
* **The content:** This is the content of the element, which in this case, is just text.
* **Attributes**: This contain extra information about the element that won’t appear in the content. In this example, the **class** attribute is an identifying name used to target the element with style information.
* **The element:** The opening tag, the closing tag, and the content together comprise the element.

**Element Attributes**

Attributes contain **extra information** about the element that you don’t want to appear in the actual content (Placed in the opening tag).

Here, **“class”** is the attribute name and **“text”**  is the attribute value. The class attribute allows you to give the **element a non-unique identifier** that can be used to target it (and any other elements with the same class value) with style information and other things.

An attribute should always have the following:

* A space between it and the element name (or the previous attribute, if the element already has one or more attributes).
* The attribute name is followed by an equal sign.
* The attribute value is wrapped by opening and closing quotation marks.

Please note the “id” attribute here is a **unique identifier** while the class attribute for instance is **a non-unique identifier**

**2.1.2.5** [**Properties of an Attribute – HTML Tags**](https://thealtschool.com/topic/properties-of-an-attribute-html-tags/)

An attribute should always have the following:

1. A space between it and the element name (or the previous attribute, if the element already has one or more attributes).
2. The attribute name followed by an equal sign.
3. The attribute value wrapped by opening and closing quotation marks.

Please note the “id” attribute here is a **unique identifier** while the class attribute for instance is **a non-unique identifier**

**2.1.2.6** [**Nesting Elements**](https://thealtschool.com/topic/nesting-elements/)

You can put elements inside other elements too — this is called nesting.

**Empty Elements**

Some elements have no content and are called empty elements.

Looking at the example above: This contains two attributes, but there is no closing </img> tag and no inner content. This is because an image element doesn’t wrap the content to affect it. Its purpose is to embed an image in the HTML page in the place it appears.

**2.1.3 Let’s build our first HTML Document**

**2.1.3.1** [**HTML Document**](https://thealtschool.com/topic/html-document/)

Individual elements are combined to form an entire HTML page. An HTML Page is a collection of nested HTML Elements.

Here, we have the following:

* **<!DOCTYPE html>** — doctype. It is a required preamble. They don’t do much and are basically just needed to make sure your document behaves correctly. That’s all you need to know for now.
* **<html></html>** — the [<html>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/html) element. This element wraps all the content on the entire page and is sometimes known as the root element.
* **<head></head>** — the [<head>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/head) element. This element acts as a container for all the stuff you want to include on the HTML page that isn’t the content you are showing to your page’s viewers. This includes things like [keywords](https://developer.mozilla.org/en-US/docs/Glossary/Keyword) and a page description that you want to appear in search results, CSS to style our content, character set declarations, and more.
* **<meta charset=”utf-8″>** — This element sets the character set your document should use to UTF-8 which includes most characters from the vast majority of written languages. Essentially, it can now handle any textual content you might put on it.
* **<title></title>** — the [<title>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/title) element. This sets the title of your page, which is the title that appears in the browser tab the page is loaded in. It is also used to describe the page when you bookmark/favorite it.
* **<body></body>** — the [<body>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body) element. This contains all the content that you want to show to web users when they visit your page, whether that’s text, images, videos, games, playable audio tracks, or whatever else.
* Let’s see this in action

**2.1.3.2** [**Marking up Text**](https://thealtschool.com/topic/marking-up-text-2/)

* Headings
* Paragraphs
* Lists

This section will cover some of the essential HTML elements you’ll use for marking up the text.

**Headings**

* Allows you to specify that certain parts of your content are **headings** or **subheadings**.
* heading levels, [<h1>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements)–[<h6>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements), although you’ll commonly only use 3 to 4 at most. In the same way that a book has the main title, chapter titles, and subtitles, an HTML document can too. HTML contains 6
* Let’s see this in action.

**Paragraphs**

* The [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p) elements are for containing paragraphs of text.
* In the same way that a book has the main title, chapter titles, and subtitles, an HTML document can too. HTML contains 6 heading levels, [<h1>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements)–[<h6>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements), although you’ll commonly only use 3 to 4 at most.
* Let’s see this in action.

**Lists**

**We have two commonly used list types in HTML** – The most common list types are ordered and unordered lists:

* **Unordered lists** are for lists where the order of the items doesn’t matter, such as a shopping list. These are wrapped in a [<ul>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ul) element.
* **Ordered lists** are for lists where the order of the items does matter, such as a recipe. These are wrapped in an [<ol>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ol) element.

**Links**

* Links are very important — they are what makes the web a web! To add a link, we need to use a simple element — [<a>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a) — “a” being the short form for “anchor”.
* Let’s see this in action.

**Download**

* The ***download*** attribute only works for same-origin URLs. You can only download files that belongs to that website.
* Let’s see this in action.

**2.1.3.3** [**Handling Media**](https://thealtschool.com/topic/handling-media/)

* Audio Element
* Video Element

In this bid, we are going to look at how we can handle media in HTML. We are going to learn how to add audio and video to an HTM Document.

**Audio Element**

We use the ***<audio>*** element to add audio to an HTML document.

* The [src](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video#attr-src) attribute – In the same way as for the [<img>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img) element, the src (source) attribute contains a path to the video you want to embed.
* The [controls](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video#attr-controls) attribute – Allows users to control audio playback.
* To autoplay our audio in this case, we can use the ***autoplay*** attribute on the audio element.
* Let’s see this in action.

**Video Element** – Allows you to embed a video

We use the **<video>** element to add video to an HTML document.

* The [src](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video#attr-src)  attribute – contains a path to the video you want to embed.
* The [controls](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video#attr-controls) attribute – allows users to control video playback.
* Let’s see this in action.

**HTML Comments**

Anything in HTML between <!– and –> is an HTML comment. The browser ignores comments as it renders the code.  
Let’s see this in action.

**Images**

We use the img tag to embed an image into our page in the position it appears.

* **Source Attribute (src) :**  Contains the path to our image file.
* **The Alternative Attribute (alt) :**  you specify descriptive text for users who cannot see the image => the text should not contain **“image or img or picture”** words.
* Images could be of any format like png, jpg or jpeg.
* Let’s see this in action.

**2.1.3.4** [**Data Collection and Listing**](https://thealtschool.com/topic/data-collection-and-listing/)

**Data collection and listing**

* Form Element
* Table Element

In this section, we are going to look at how to add forms and tables in an HTML document.

**Form Element**

* This is how we define the form element in HTML. Forms allow us to collect data from users interacting with it. For example, think of a sign up page for Alt School.
* Inside the form, we have labels and inputs fields. Labels allow us to display more information about a form element for instance, an email input.
* Let’s see this in action.

The <input> [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element is used to create interactive controls for web-based forms in order to **accept data from the user.**

* Text
* Number
* Button
* Submit
* Email
* Color
* File
* Password
* Radio
* Reset

The <select> [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element represents a control that provides a **menu of options**.

<label for="pet-select">Choose a pet:</label>

<select name="pets" id="pet-select">

<option value="">Select Pet</option>

<option value="dog">Dog</option>

<option value="cat">Cat</option>

</select>

**Table Element**

The <table> [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element represents tabular data — that is, information presented in a two-dimensional table comprised of rows and columns of cells containing data.

The **<td>** [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element defines a cell of a table that contains data.   
The **<th>** [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element defines a cell as header of a group of table cells.  
The **<tr>** [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element defines a row of cells in a table.

**2.1.4** [**Semantic HTML**](https://thealtschool.com/topic/semantic-elements/)

**2.1.4.1 Semantic Elements**

The [<h1>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements) element is a semantic element, which gives the text it wraps around the role (or meaning) of “a top level heading on your page.”

**A great deal of web content can be made accessible just by making sure the correct Hypertext Markup Language  elements are used for the correct purpose at all times.**

By default, most browser’s [user agent stylesheet](https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade#user-agent_stylesheets) will style an [<h1>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/Heading_Elements) with a large font size to make it look like a heading (although you could style it to look like anything you wanted).

**Some of the benefits from writing semantic markup are as follows:**

* Search engines will consider its contents as important keywords to influence the page’s search rankings (see [SEO](https://developer.mozilla.org/en-US/docs/Glossary/SEO))
* Screen readers can use it as a signpost to help visually impaired users navigate a page (accessibility)
* Finding blocks of meaningful code is significantly easier than searching through endless divs with or without semantic or namespaced classes
* Suggests to the developer the type of data that will be populated
* Semantic naming mirrors proper custom element/component naming

These are some of the roughly 100 semantic [elements](https://developer.mozilla.org/en-US/docs/Web/HTML/Element) available:

* <article>
* <aside>
* <details>
* <figcaption>
* <figure>
* <footer>
* <header>
* <main>
* <mark>
* <nav>
* <section>
* <summary>
* <time>

**2.1.4.2** [**Let’s Build our First Semantic HTML**](https://thealtschool.com/topic/lets-build-our-first-semantic-html/)

Here is what we need;

* Install a **text editor**, in this case, VS Code is recommended.
* Install **live server extension** from VS Code Marketplace.
* Install **prettier extension** – allows us to format our HTML code, make it look nice.



<header>  
  <nav>  
    <!– main navigation in here –>  
  </nav>  
</header>  
<main>  
  <article>  
    <!– article content in here –>  
  </article>  
  <aside>  
    <!– aside content in here –>  
  </aside>  
</main>  
<footer>  
  <!– footer content in here –>  
</footer>

**2.1.5 Limitations of HTML**

**2.2.5.1** [**Limitations of HTML**](https://thealtschool.com/topic/limitations-of-html/)

* You can not create dynamic content
* It has limited designing capabilities
* Syntax errors are not identified or displayed by HTML
* Any type of calculations can not be done in HTML
* You can not create interactive web pages with HTML
* Complex HTML code is hard to read and understand

**2.2.5.2** [**Closing Curtains**](https://thealtschool.com/topic/closing-curtains/)

**Month 2: Week 2**

**2.2.1 Introduction to CSS**

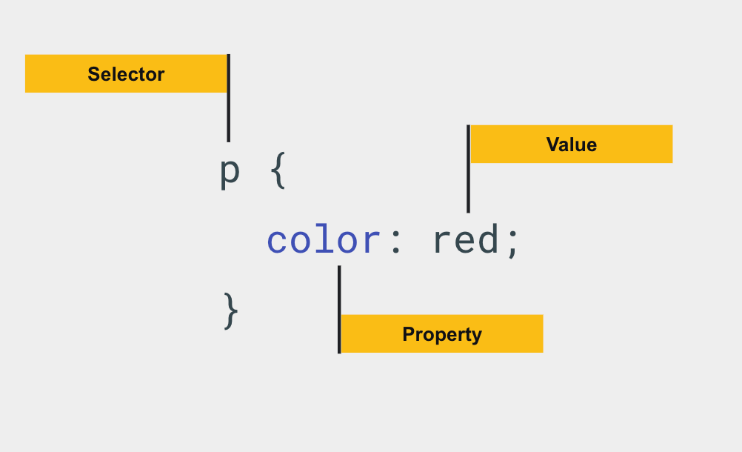
**2.2.1.1** [**CSS Basics**](https://thealtschool.com/topic/css-basics/)



**A paragraph with a red text color**

This CSS selects paragraph text, setting the color to red. The  **“p”** is the **selector** and **“color”** is the **rule property**, and then the **“red”** is the **value of the property**. The whole structure is called a **ruleset**.

**CSS Ruleset**



**The whole structure is called a ruleset;**

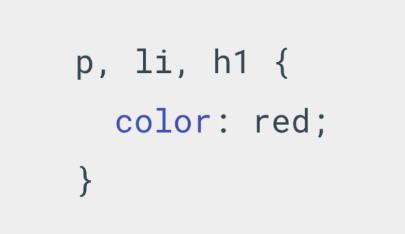
* **Selector** – defines the element(s) to be styled (in this example, [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p) elements). To style a different element, change the selector.
* **Declaration** – This is a single rule like color: red;. It specifies which of the element’s properties you want to style.
* **Property** – A way in which you can style an HTML element. (In this example, color is a property of the [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p) elements.) In CSS, you choose which properties you want to affect in the rule.
* **Property value** – To the right of the property—after the colon—there is the property value

**2.2.1.2** [**CSS Syntax**](https://thealtschool.com/topic/css-syntax/)



**CSS Syntax**

* Each ruleset must be wrapped in **curly braces**. ({})
* You must use a colon (:) to separate the property from its value or values in each declaration.
* You must use a semicolon (;) to separate each declaration from the next one.



**Selecting multiple elements**

You can also select multiple elements and apply a single ruleset to all of them. Separate multiple selectors by commas.

**2.2.1.3** [**CSS Selectors**](https://thealtschool.com/topic/css-selectors/)

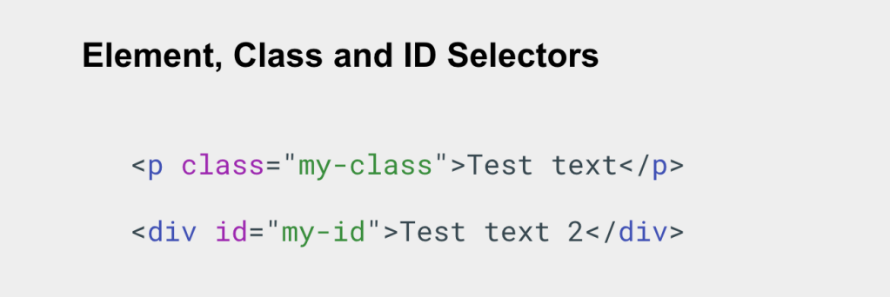
There are many different types of selectors. The examples above use element selectors, which select all elements of a given type. But we can make more specific selections as well;

* **Element selector** – selects all elements of the given type for <p>
* **ID selector** – selects an element with a given id provided in the id attribute
* **Class selector** – selects the element(s) on the page with the specified class.
* **Attribute selector** – selects the element(s) on the page with the specified attribute.
* **Pseudo-class selector** – selects the specified element(s), but only when in the specified state. (For example, when a cursor hovers over a link.)

**Element , ID & Class Selectors**



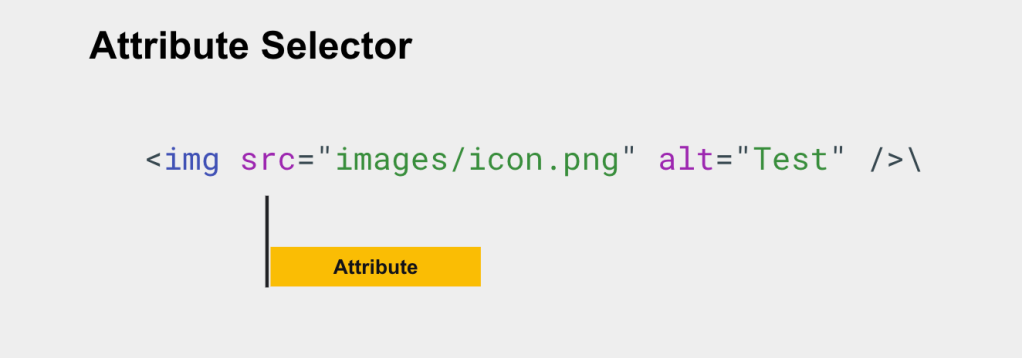
**Element , ID & Class Selectors**



**Attribute & Pseudo-class selector**



**Attribute & Pseudo-class selector**



**Attribute Selector**

**2.2.1.4** [**Adding CSS to HTML**](https://thealtschool.com/topic/adding-css-to-html/)

To make the code work, we still need to apply this CSS (above) to your HTML document. Otherwise, the styling won’t change the appearance of the HTML. We have to ways to achieve this.

* **Inline Styles** – Allows you to add styles to specific elements.
* **Internal Styles –** Allows you to include CSS Code in the **<head/>** section of your HTML Page.
* **External Styles** – Allows you to link external style sheets from CSS files saved with a .css extension.

**Inline Styles**



**Inline Styles**

A few things to know about an **Inline style** are;

* It uses the **style attribute**. It’s important to note inline CSS styling is not a good practice.
* It is difficult to manage and update. Additionally, combining HTML with CSS leads to messy and difficult-to-read code.

**Internal Styles**



**Internal Styles**

A few things to know about an **internal style** are;

* It contains CSS rules for the entire page.
* It uses the **<style>** element –  which contains a block of styling rules and is placed in the **<head>** element of HTML files.

**External Styles**



A few things to know about an **external style** are;

* It contains CSS rules for the given page that links it.
* It uses the **<link>** element put in the **<head>** element –  which has an **href attribute** with the address or link or path to the **external .css** file

**CSS Tips**

Here a few tips to note:

* The best practice for styling HTML documents is the external stylesheet. It lets you set CSS rules for several files and is very easy to update.
* Use inline CSS to make minor modifications to your elements.
* When you specify more than one CSS rule to an element, the style for that item follows the cascading rule : inline > internal + external > browser default.
* You can use any text editor to create an external stylesheet. This process is similar to creating an HTML file.

**2.2.1.5** [**CSS Comments**](https://thealtschool.com/topic/css-comments/)

**2.2.1.6** [**Let’s write our first CSS Styles**](https://thealtschool.com/topic/lets-write-our-first-css-styles/)

Here is what we need;

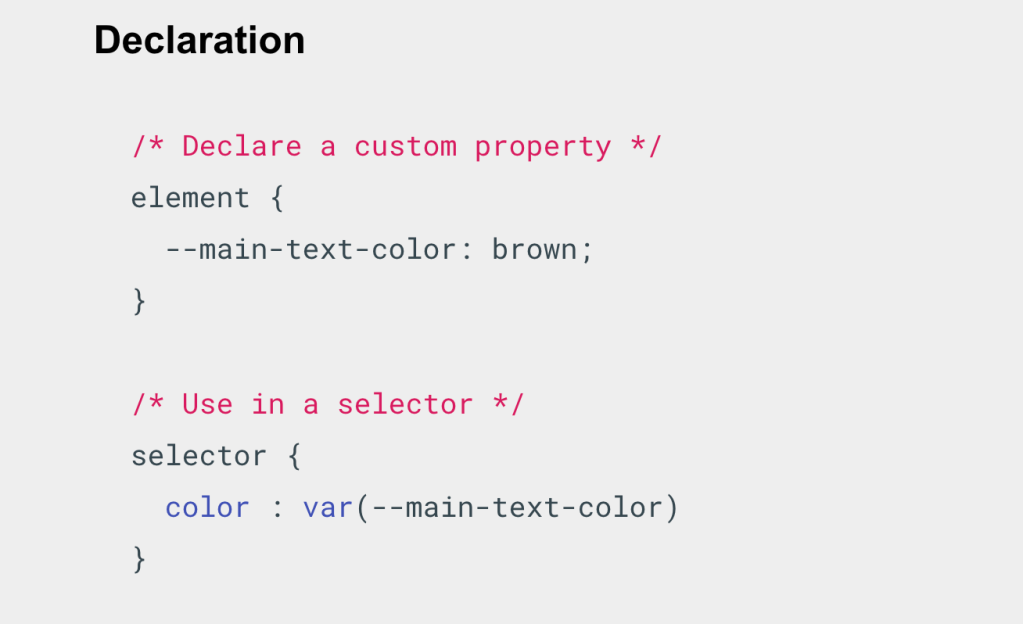
* Install a **text editor**, in this case, VS Code is recommended.
* Install **live server extension** from VS Code Marketplace.
* Install **prettier extension** – allows us to format our HTML code, make it look nice.

**2.2.1.7** [**CSS Variables**](https://thealtschool.com/topic/css-variables/)

Custom properties (sometimes referred to as CSS variables or cascading variables) are entities defined by CSS authors that contain specific values to be **reused throughout** a document.

They are set using **custom property notation.**

* Declaring a custom property is done using a custom property name that begins with a double hyphen (–), and a property value that can be any valid CSS value.
* To use a given CSS variable –  set the property value by using the (var) and the variable name in brackets with double hyphen preceding it.



**Declaration**

**Global Scope**



**Global Scope**

Note that the selector given to the ruleset defines the scope that the custom property can be used in. A common best practice is to define custom properties on the [:root](https://developer.mozilla.org/en-US/docs/Web/CSS/:root) pseudo-class, so that it can be applied globally across your HTML document

**Advantages of using CSS Variables**

Here are a few advantages of using CSS variables;

* Easier reading, no repetition
* Great for creating custom color themes
* Easy to get started

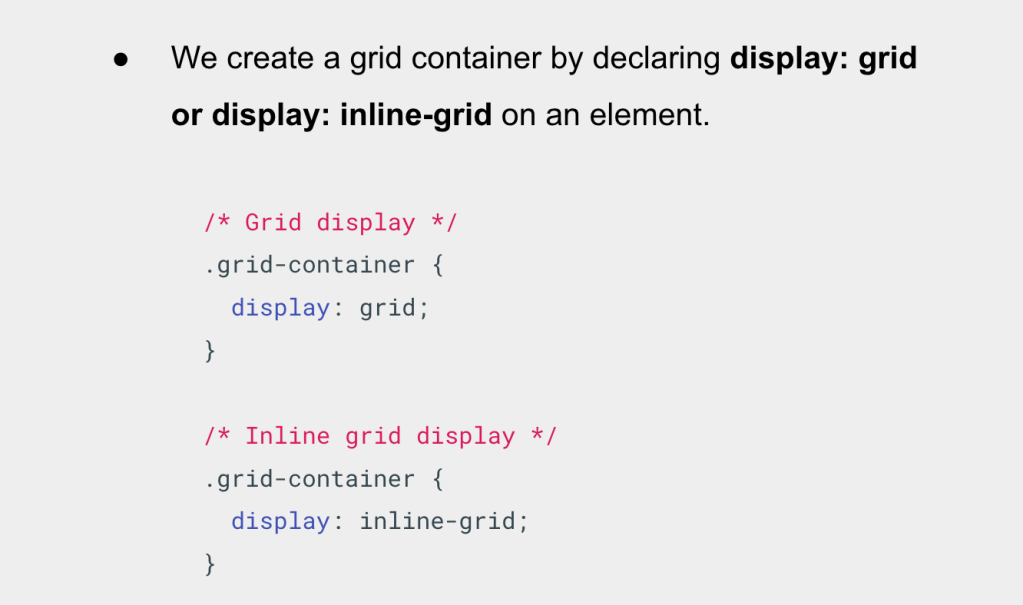
**Month 2: Week 3**

**2.3.1 Introduction to CSS**

**2.2.1.1** [**Block and Inline**](https://thealtschool.com/topic/block-and-inline/)

**2.2.1.2** [**Grid Layout**](https://thealtschool.com/topic/grid/)

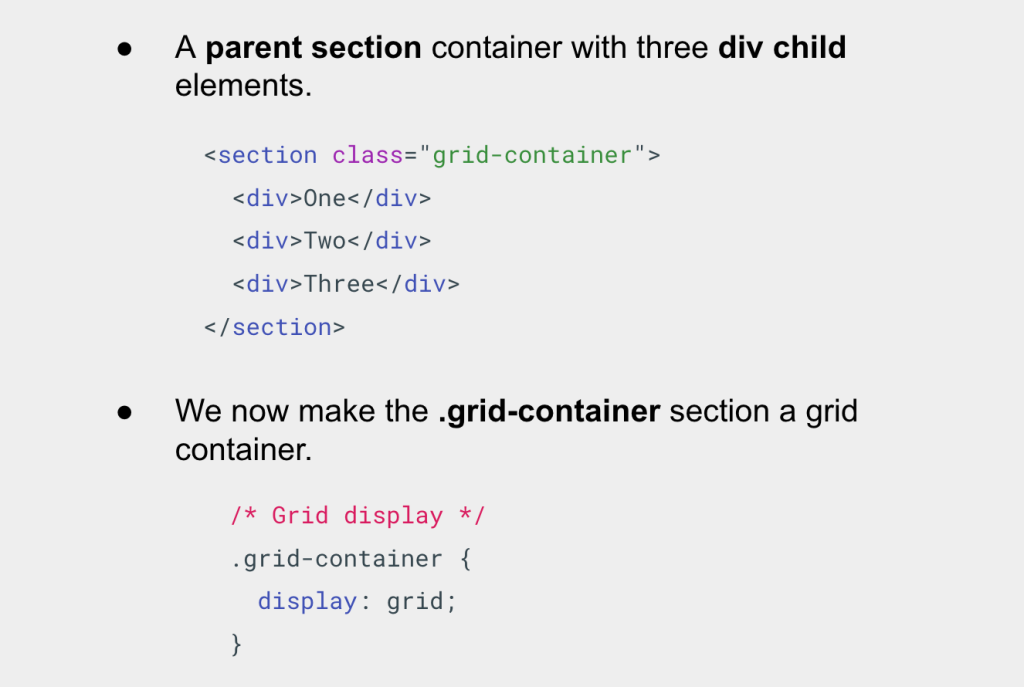
**Grid Layout**



**Grid Layout**

We create a grid container by declaring display: grid or display: inline-grid on an element.

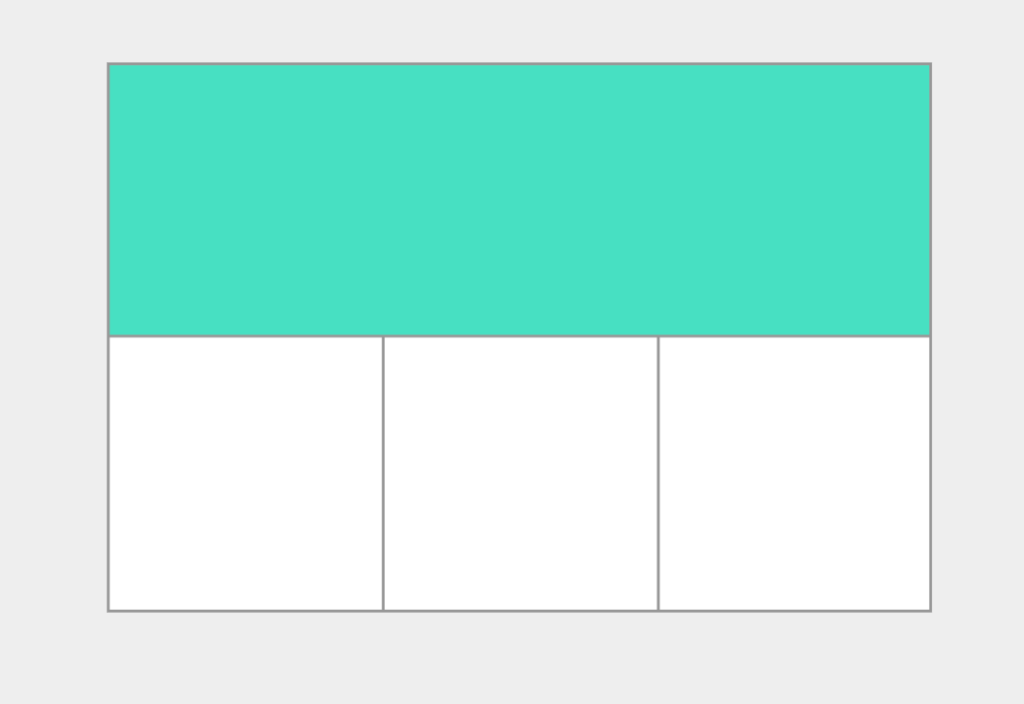
**Grid Styles**



**Grid Styles**

As you learn and then work with the CSS Grid Layout this **Browser DevTools** will give you a better idea of what is happening with your grids visually.

**Grid Tracks**

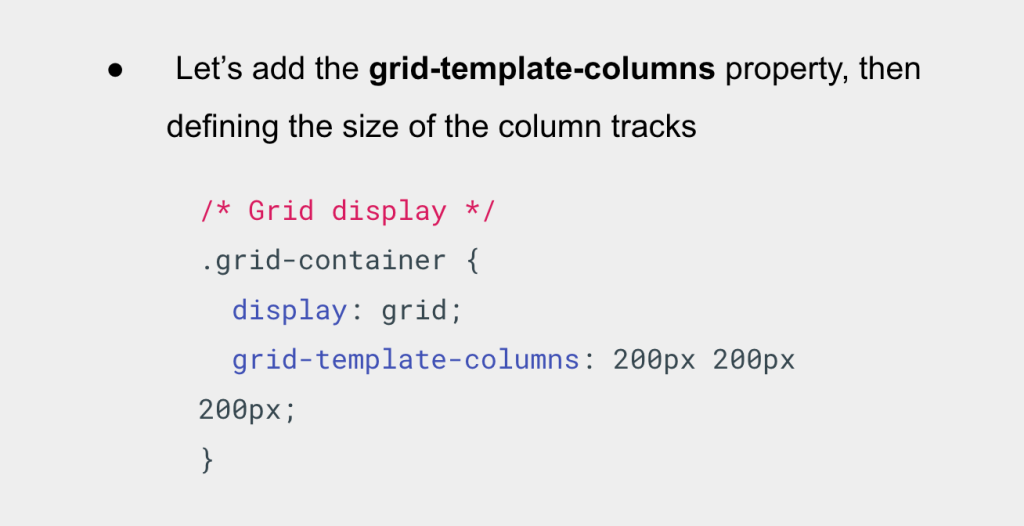


**Grid Tracks**

**So what’s a grid track?**

* A grid track is the space between any two lines on the grid. In the  image you can see a track highlighted – this is the first row track in our grid.
* We define rows and columns on our grid with the [grid-template-columns](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-template-columns) and [grid-template-rows](https://developer.mozilla.org/en-US/docs/Web/CSS/grid-template-rows) properties.
* Let’s revisit our earlier example and add the grid-template-columns property, then defining the size of the column and row tracks as 200px;

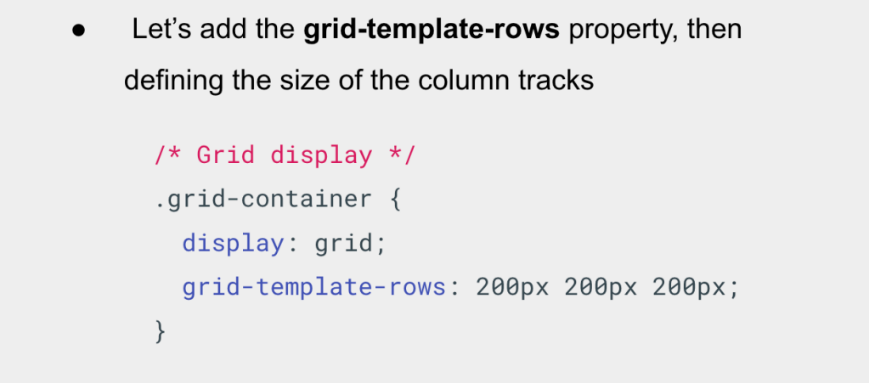
**Grid Columns**



**Grid Columns**

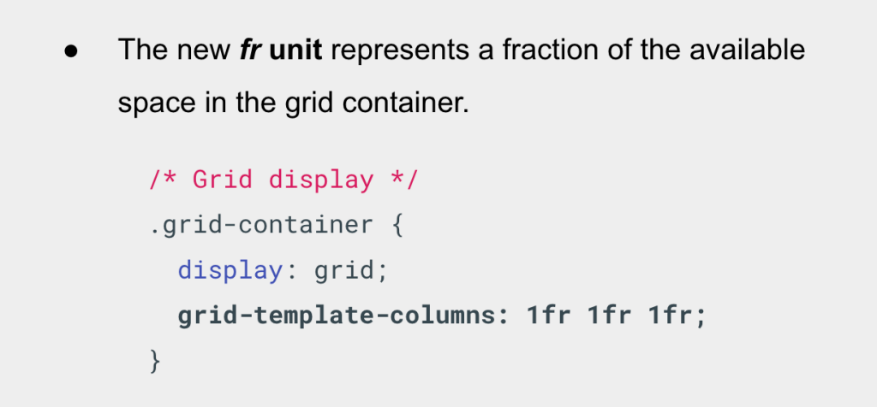
Let’s head over to our html page and see this in action.

**Grid Rows**



**GRID ROWS**

**The *fr Unit***



**The *fr Unit***

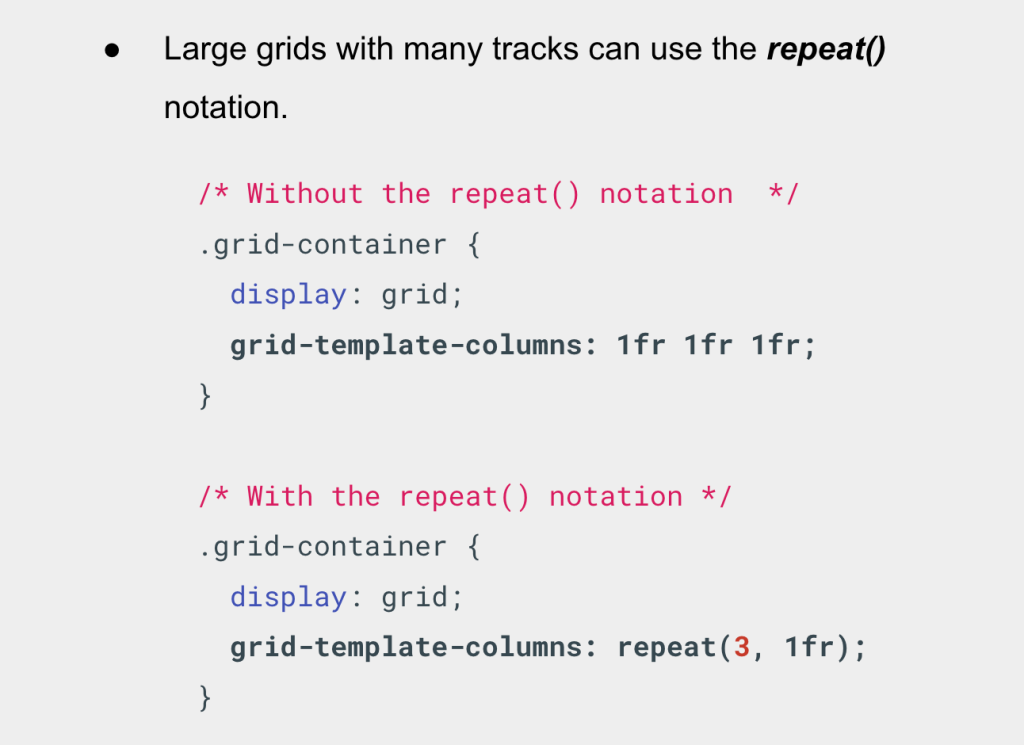
* Tracks can be defined using any length unit. Grid also introduces an additional length unit, ***the fr unit***, to help us create flexible grid tracks.
* The next grid definition would create three equal width tracks that grow and shrink according to the available space.



**The *fr Unit***

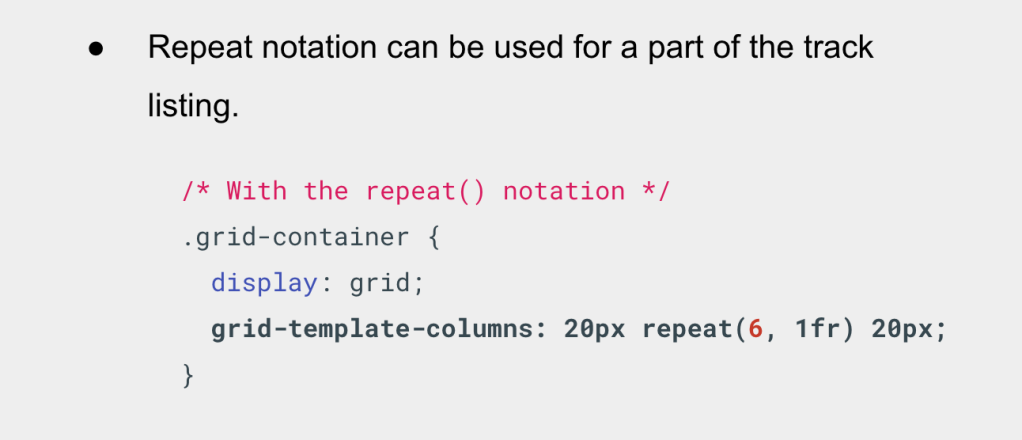
* In this final example, we mix **absolute sized tracks with fr units**. The first track is 200 pixels, so the fixed width is taken away from the available space.
* The remaining space is divided into two and assigned in proportion to the two flexible tracks.

**The *repeat()***



The ***repeat()***

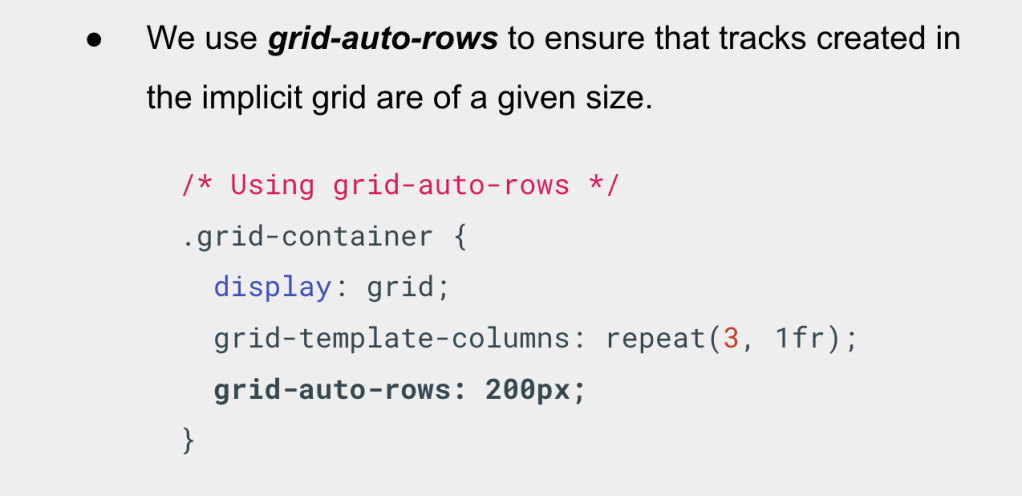
Large grids with many tracks can use the ***repeat()*** notation, to repeat all or a section of the track listing.



The ***repeat()***

Repeat notation can be used for a part of the track listing. In this next example we create a grid with an initial 20-pixel track, then a repeating section of 6 1fr tracks then a final 20-pixel track.

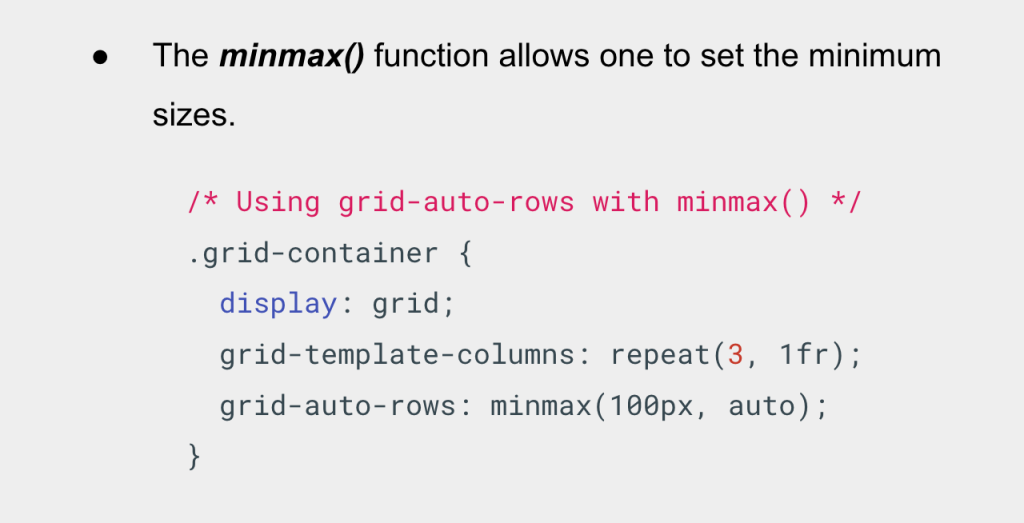
**Grid Rows**



**Grid Rows**

We use grid-auto-rows to ensure that tracks created in the implicit grid are of a given size. In this case **200px.**

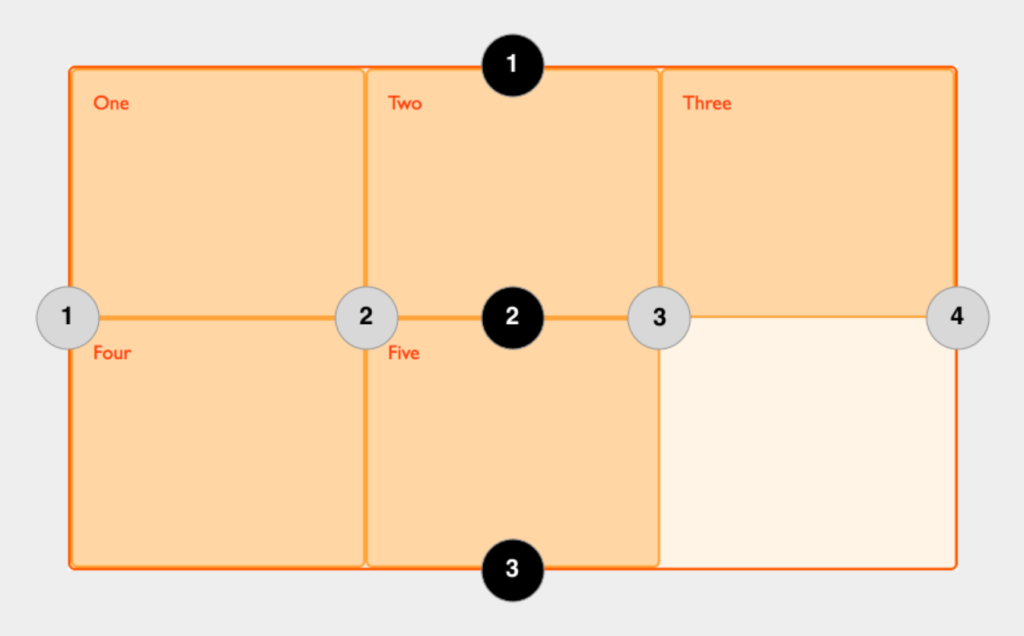
**Grid *Minmax()***



**Grid *Minmax()***

* In some cases, I may want my rows to never collapse smaller than 100 pixels, but if my content stretches to 300 pixels in height, then I would like the row to stretch to that height.
* This means automatically created rows will be a minimum of 100 pixels tall, and a maximum of auto.
* Using **auto** means that the size will look at the content size and **will stretch to give space for the tallest item in a cell**, in a given row.

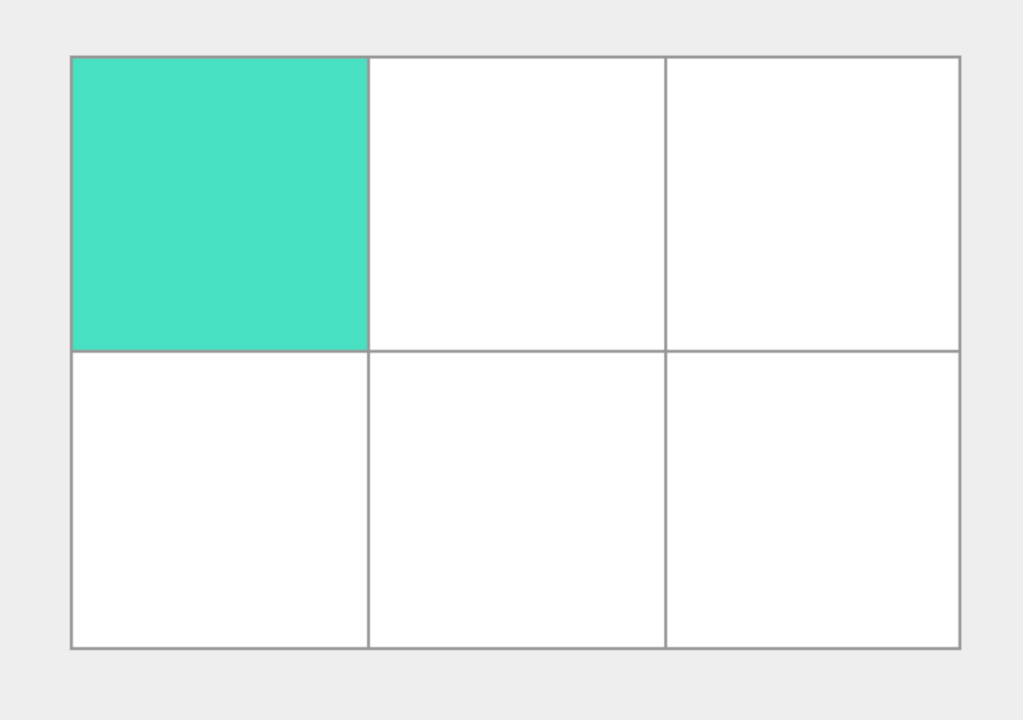
**Grid Lines**



**Grid Lines**

* When we define a grid we define the grid tracks, not the lines. Grid then **gives us numbered lines** to use when positioning items.
* In our **three column, two row grid,** we have **four column lines** and **three row lines.**

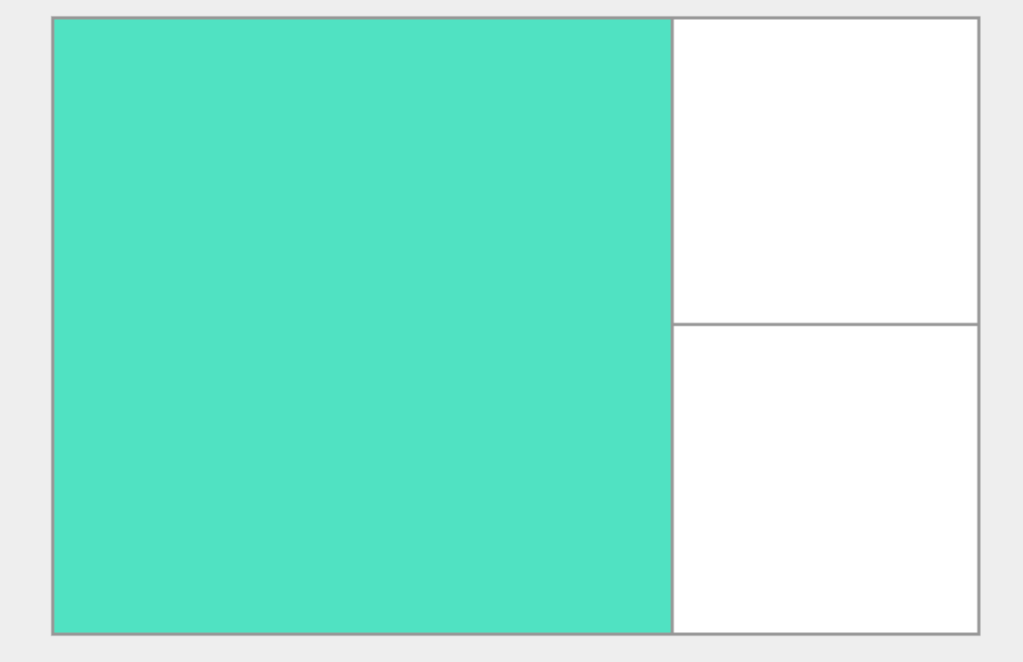
**Grid Cells**



**Grid Cells**

A *grid cell* is the smallest unit on a grid.

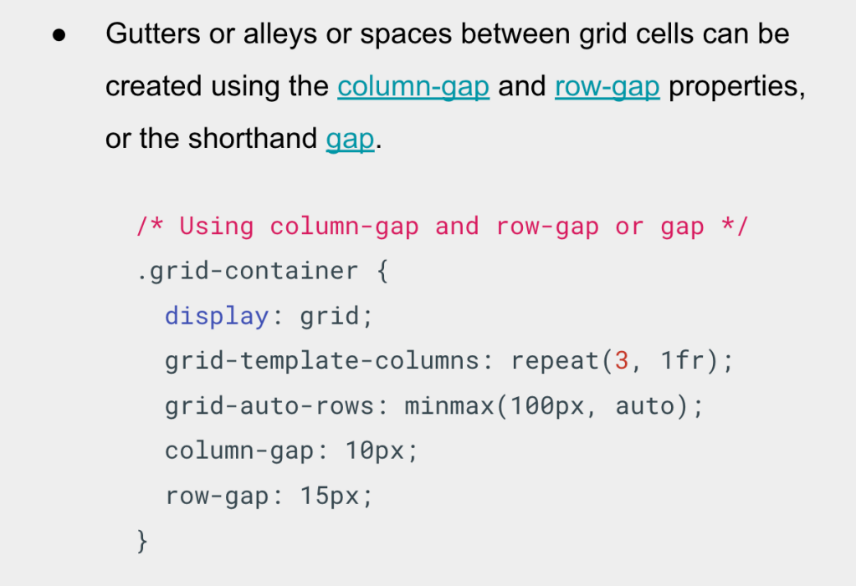
**Grid Areas**



**Grid Areas**

Items can span one or more cells both by row or by column, and this creates a grid area. Grid areas must be rectangular.

**Grid Gutters**



**Grid Gutters**

* *Gutters* or *alleys or spaces* between grid cells can be created using the [column-gap](https://developer.mozilla.org/en-US/docs/Web/CSS/column-gap) and [row-gap](https://developer.mozilla.org/en-US/docs/Web/CSS/row-gap) properties, or the shorthand [gap](https://developer.mozilla.org/en-US/docs/Web/CSS/gap).
* In the  example, I am creating a 10-pixel gap between columns and a 15-pixel gap between rows.

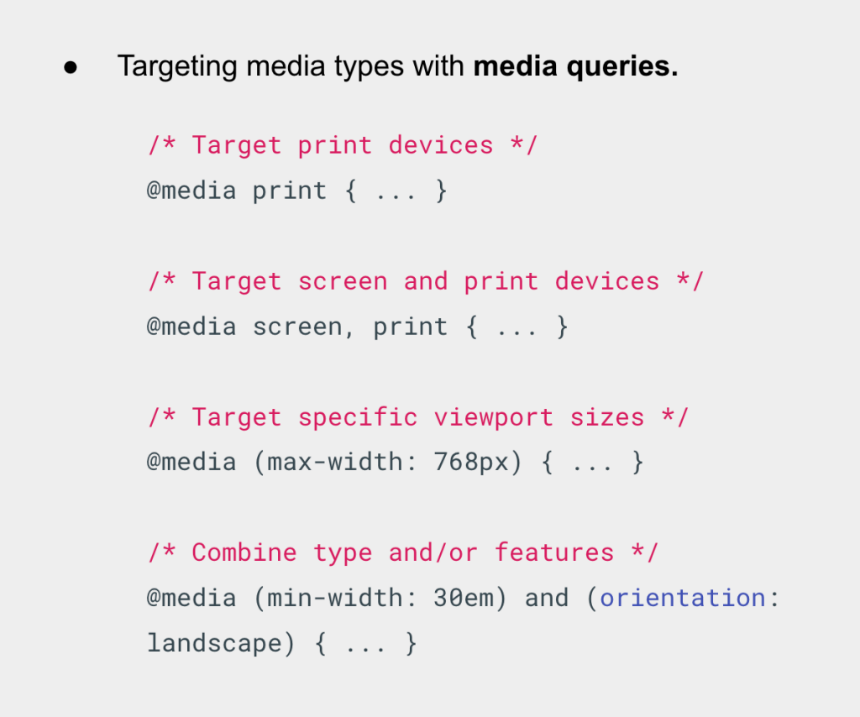
**Month 2: Week 4**

**2.2.4 More CSS**

**2.2.4.1** [**Media Queries**](https://thealtschool.com/topic/media-queries/)

**Media queries** are useful when you want to modify your site or app depending on a device’s device’s general type (such as print vs. screen).  Here are some of the most common use cases.

* To conditionally apply styles based on features like device types or screen sizes.
* Building responsive websites.



**Media Queries**

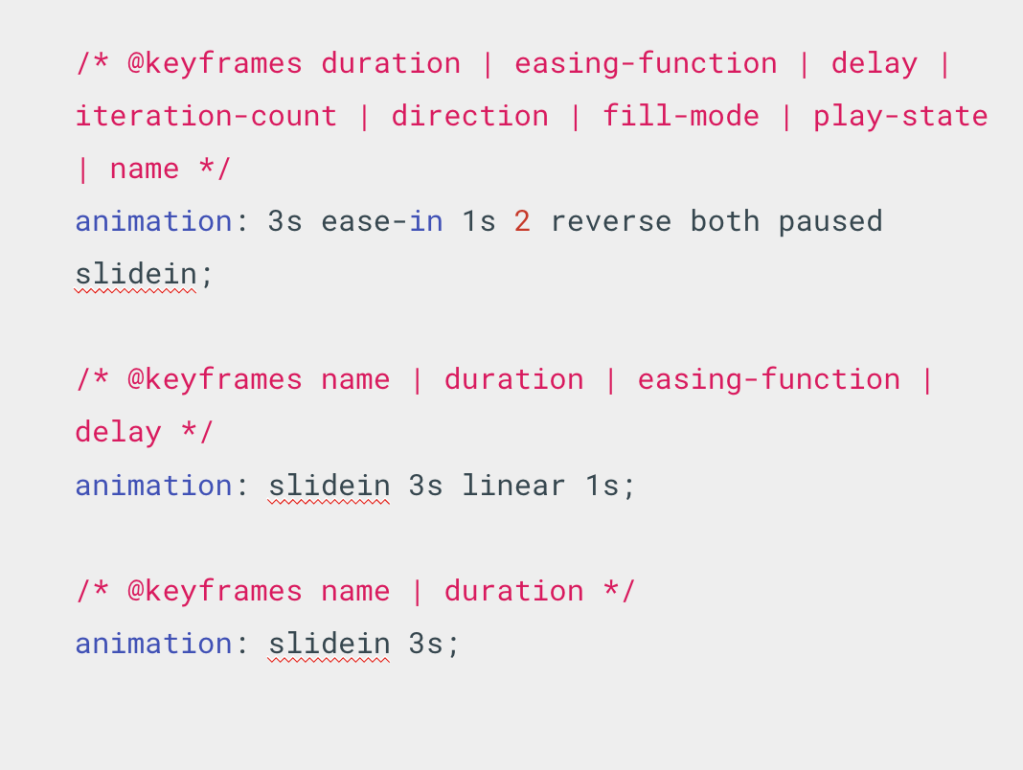
* [Media types](https://developer.mozilla.org/en-US/docs/Web/CSS/@media#media_types) define the broad category of device for which the media query applies: ***all, print, screen***.
* The**type**is optional (assumed to be all) except when using the **not**or **only** logical operators.

**2.2.4.2** [**CSS Animation**](https://thealtschool.com/topic/css-animation/)

**Animations make it possible to animate transitions from one CSS style configuration to another.**

There are a few key advantages to CSS animations over traditional script-driven animation techniques:

* They’re**easy to use**for simple animations; you can create them without even having to know JavaScript.
* The **animations run well,** even under**moderate system load**. Simple animations can often perform poorly in JavaScript.



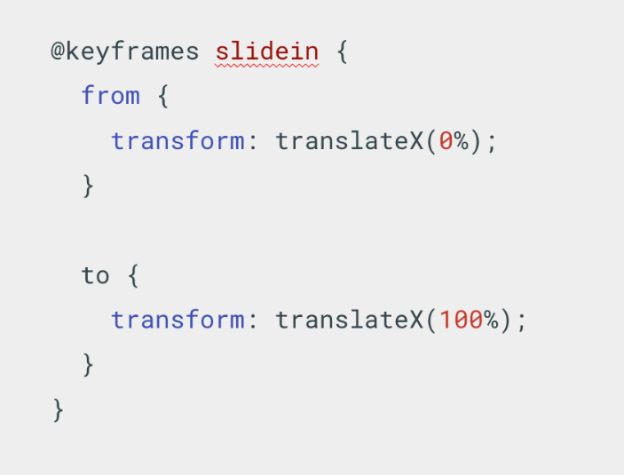
CSS Animation

***Shorthand properties*** are CSS properties that let you set the values of multiple other CSS properties simultaneously.

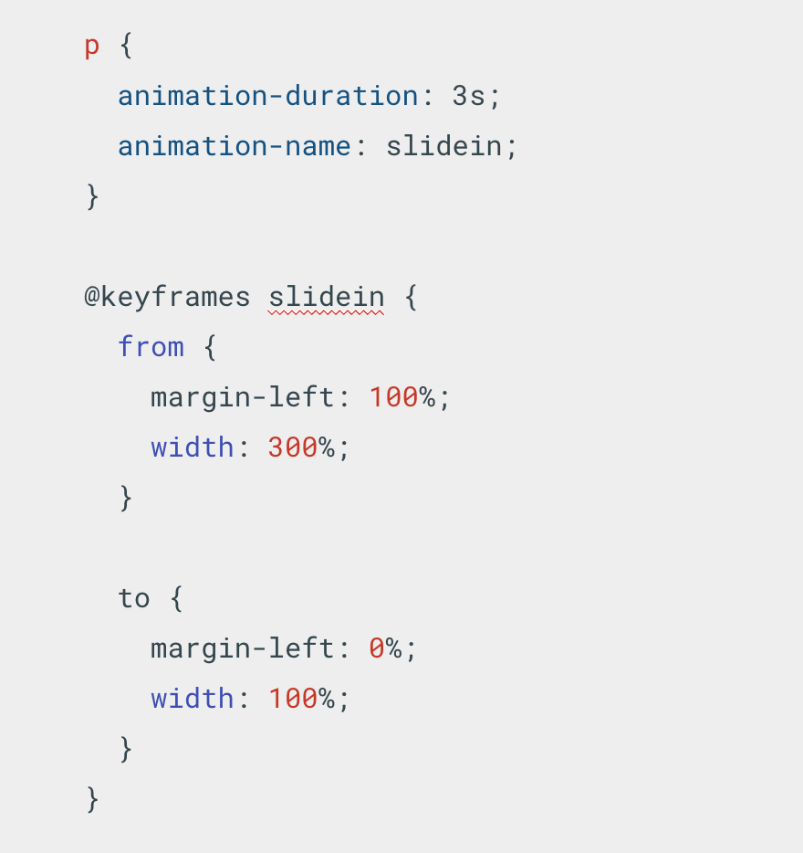
The ***animation*** [shorthand](https://developer.mozilla.org/en-US/docs/Web/CSS/Shorthand_properties) css property applies an animation between styles.

The ***animation property*** is specified as one or more single animations, separated by commas.

* At-rules are [CSS statements](https://developer.mozilla.org/en-US/docs/Web/CSS/Syntax#css_statements) that instruct CSS how to behave. They begin with an at sign, ***‘@’***.
* The ***@keyframes*** CSS [at-rule](https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule) controls the intermediate steps in a CSS animation sequence.
* We can’t animate anything if we don’t have the***@keyframes*** specified.



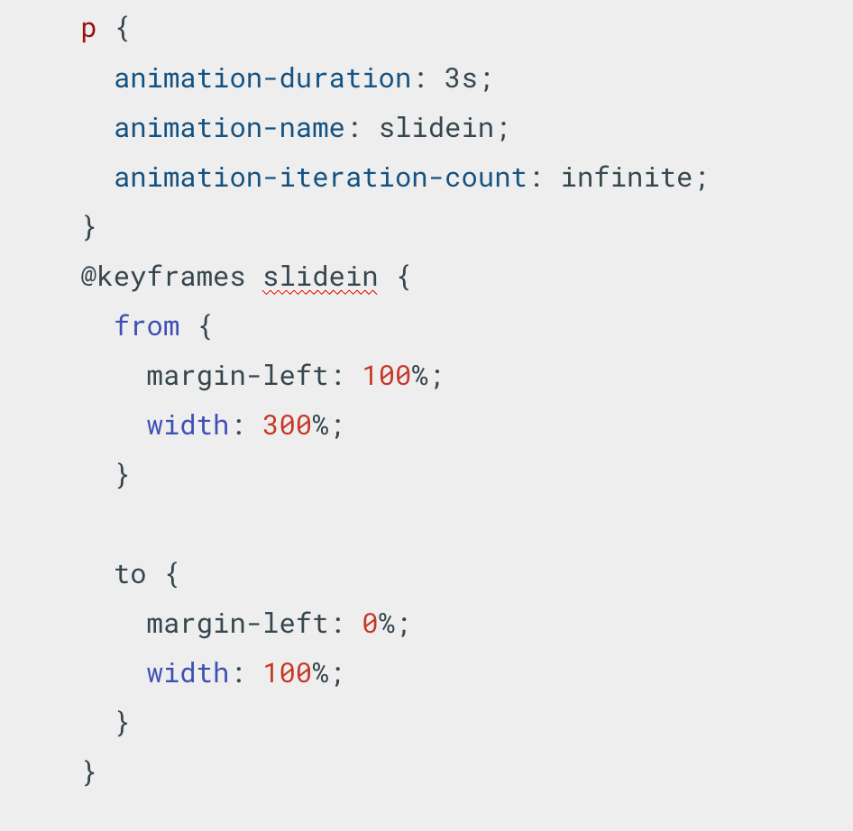
* [<custom-ident>](https://developer.mozilla.org/en-US/docs/Web/CSS/custom-ident) – a name identifying the keyframe list.
* **from** – a starting offset of 0%.
* **to** – an ending offset of 100%.
* [<percentage>](https://developer.mozilla.org/en-US/docs/Web/CSS/percentage) – a percentage of the time through the animation sequence at which the specified keyframe should occur.



**Making text slide across the browser window**

* In this example the style for the [<p>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p) element specifies that the animation should take 3 seconds to execute from start to finish, using the [animation-duration](https://developer.mozilla.org/en-US/docs/Web/CSS/animation-duration) property, and that the name of the [@keyframes](https://developer.mozilla.org/en-US/docs/Web/CSS/@keyframes) at-rule defining the keyframes for the animation sequence is named “slidein”.

**Making it repeat**



**Making it repeat**

* To make the animation repeat itself, use the [animation-iteration-count](https://developer.mozilla.org/en-US/docs/Web/CSS/animation-iteration-count) property to indicate how many times to repeat the animation.

**Month 3: Week 1**

**3.1.1 Introduction to JavaScript**

**3.1.1.1** [**Introduction to JavaScript**](https://thealtschool.com/topic/introduction-to-javascript/)

**3.1.1.2** [**What is JavaScript?**](https://thealtschool.com/topic/what-is-javascript/)

* A lightweight, interpreted programming language.
* Programs written in JavaScript are called scripts.
* JavaScript allows you to implement complex interaction on Web pages.
* It is the third layer of standard web technologies (HTML, CSS, JS).
* **JavaScript is different from Java programming language**.

**3.1.1.3** [**Fundamentals of JavaScript**](https://thealtschool.com/topic/fundamentals-of-javascript/)



**Relationship between HTML, CSS, & JS**

* HTML gives the web page structure and purpose
* CSS makes it looks aesthetic with styling
* JS makes the page dynamic and interactive

**Uses of JavaScript**

* Adds interactivity to web pages
* Run codes in response to certain events
* JavaScript has a powerful **Application Programming Interface (APIs)**, hence access to numerous complex features

What is even more exciting however is the functionality built on top of the client-side JavaScript language. So-called Application Programming Interfaces (APIs) provide you with extra superpowers to use in your JavaScript code.

APIs are ready-made sets of code building blocks that allow a developer to implement programs that would otherwise be hard or impossible to implement. They do the same thing for programming that ready-made furniture kits do for home building.

There are two types of APIs available:

* Browser APIs: Built and shipped with Browsers
* Third party APIs: Not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web. For example: Google maps API, Twitter API

**JavaScript can run on:**

* Browsers, using the **JavaScript Virtual Machine (JVM)**.
* Servers, using runtimes like Nodejs
* Any devices that have the Javascript engine

**JS Execution**

JavaScript can execute not only in the browser, but also on the server, or actually on any device that has a special program called the JavaScript engine.

The browser has an embedded engine sometimes called a “JavaScript virtual machine”. Different engines have different “names” for their JVM.

The terms above are good to remember because they are used in developer articles on the internet. We’ll use them too. For instance, if “a feature X is supported by V8”, then it probably works in Chrome, Opera and Edge.

**Different platforms have different “names” for their JVM**

* **V8** – in Chrome, Opera and Edge.
* **SpiderMonkey** – in Firefox.
* **Chakra**” for IE
* **Nitro**” and “**SquirrelFish**” for Safari, etc.

**Embedding JS**

JavaScript is applied to your HTML page in a similar manner to CSS. Whereas CSS uses <link> elements to apply external stylesheets and <style> elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the <script> element.

**Adding JavaScript to your Webpage**

* **Internal JavaScript:** JS code added inside HTML files
* **External JavaScript:**External JS code linked in HTML file

1. **Internal Javascript**

* Write and run JS inline HTML by saving the file and loading in the browser.
* Order matters, so if you add the script before the button, you get an error. ***Show an example here***
* Show an example of  the **“DOMContentLoaded”** event. Order does not matter here, as script code is executed after HTML content is loaded.

1. **External Javascript**

* This is generally a good thing in terms of organizing your code and making it reusable across multiple HTML files. Plus, the HTML is easier to read without huge chunks of script dumped in it.
* Note the **defer** option? I does the same thing as the “DOMContentLoaded”
* Order matters when loading scripts.***Show examples***.

**Statements are syntax constructs and commands that perform actions.**

* We’ve already seen a statement, alert(‘Hello, world!’), which shows the message “Hello, world!”
* We can have as many statements in our code as we want. Statements can be separated with a semicolon.
* A semicolon may be omitted in most cases when a line break exists.
* JavaScript interprets the line break as an “implicit” semicolon. This is called an automatic semicolon insertion.

**Comments**

**Comments are used to describe code**.

* As time goes on, programs become more and more complex. It becomes necessary to add comments which describe what the code does and why.
* Comments can be put into any place of a script. They don’t affect its execution because the engine simply ignores them.

**Variables**

**A variable is a “named storage” for values or data**

* Use real-life analogy of a named ingredients in a kitchen
* JavaScript application needs to work with information. E.g Catalogue list holding shopping items, Chat application holding messages, names etc
* Create a variable using the let and const keyword. We’ll take about their difference later.
* Can declare multiple variables in single line separated by comma
* Talk about var and its availability in older script. Discourage it’s usage

**Variables (Naming conventions)**

* The name must contain only letters, digits, or the symbols $ and \_ . ***Required***
* The first character must not be a digit. ***Required***
* Use [camelCase](https://en.wikipedia.org/wiki/CamelCase) when variables contain multiple words ***Optional***
* Case matters: Variables named apple and AppLE are two different variables.
* Reserved Names/Keyword cannot be used. E.g let let = 5; let return = 5;

**Variables (const)**

* Variables declared with **const** do not change and cannot be re-assigned
* It is a convention in JS to **CAPITALIZE** const variable names
* Separate multiple words with **under\_score**

**Variables (let)**

* Variables declared with **let** can change and be re-assigned
* Variable names should be descriptive and it’s intention should be clear
* It is recommended to use camelCase for long variables
* A variable name should have a clean, and meaning obvious.
* Variable naming is one of the most important and complex skills in programming.
* Some good-to-follow rules are:
  + Use human-readable names like **userName** or **shoppingCart**.
  + Rarely use abbreviations or short names like a, b, c
  + Make names maximally descriptive and concise. Examples of bad names are data and value..
  + Agree on terms within your team and in your own mind. If a site visitor is called a “user” then we should name related variables currentUser or newUser instead of currentVisitor or newManInTown.

**3.1.2 JavaScript Fundamentals**

**3.1.2.1** [**Javascript Datatypes 1**](https://thealtschool.com/topic/javascript-datatypes-1/)

**Data Types**

* A value in JavaScript is always of a certain type. e.g, string, number.
* There are eight (8) basic data types in JavaScript, these are:
  + number
  + bigInt
  + string
  + boolean
  + null
  + undefined
  + object
  + symbol
* We can put any type in a variable. For example, a variable can at one moment be a string and then store a number: let message = “hello”; message = 123456;
* Programming languages that allow such things, such as JavaScript, are called “dynamically typed”, meaning that there exist data types, but variables are not bound to any of them.

**Data types (number)**

* The number type represents both integer and floating point numbers.
* The following operations work on numbers:  multiplication \*, division /, addition +, subtraction -.
* There are “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN.
* Infinity represents the mathematical Infinity ∞. e.g divide by 0
* NaN represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance: console.log(“name” / 2)
* NaN is sticky. Any further mathematical operation on NaN returns NaN
* So, if there’s a NaN somewhere in a mathematical expression, it propagates to the whole result (there’s only one exception to that: NaN \*\* 0 is 1).
* Doing maths is “safe” in JavaScript. We can do anything: divide by zero, treat non-numeric strings as numbers, etc. The script will never stop with a fatal error (“die”). At worst, we’ll get NaN as the result.

**Data types (BigInt)**

* The bigInt type represents extremely large or small integers which cannot be represented by the number type.
* The maximum and minimum **size for a number type is (253-1) and -(253-1)**
* **This number type is enough for many purpose, so the bigInt type is rarely used.**
* **A BigInt is created by adding n to the end of a number or using the BigInt constructor**

**3.1.2.2** [**JavaScript Datatypes 2**](https://thealtschool.com/topic/javascript-datatypes-2/)

**Data types (Boolean)**

* The boolean type has only two values: **true** and **false**.
* This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”.
* Boolean values are mostly gotten from comparisons:

**Data types (null)**

* The value null represents the intentional absence of a value
* The null type is a special value which represents “nothing”, “empty” or “value unknown”.
* Note that type of null is object for legacy reasons
* An unassigned variable is automatically set to type null
* Use null to assign an “empty” or “unknown” value to a variable

**Data types (undefined)**

* The value undefined means a value is not assigned
* A declared variable without assigned values becomes undefined by default
* **Show example of type of undefined**

**Data types (object)**

* Objects are one of the most important data type in JavaScript
* Objects are used to store keyed collections of data and more complex entities.
* All other types are called “**primitive**” because their values can contain only a single value
* Objects are associative arrays with several special features. They store properties (key-value pairs), where:
* You can retrieve values using dot notation or with square brackets
* Objects are store any valid JavaScript type including objects themselves
* delete a property: delete obj.prop.

**3.1.2.3** [**JavaScript Datatypes 3**](https://thealtschool.com/topic/what-is-javascript-3/)

**Data types (String)**

* **A String type is used to represent and manipulate a sequence of characters.**
* A string type must be surrounded by quotes or backticks.
* Double and single quotes are “simple” quotes. There’s practically no difference between them in JavaScript.
* Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in ${…}
* Other quotes don’t have this embedding functionality.
* Strings are special types, and therefore have extra functionalities built on it.
* You can call methods on strings to do special things like the examples below
* You can use special characters in strings, like the new line character (**\n**)
* Other special characters are \’, \”, \\, \t, \u{1F60D}.  Show some examples in console
* Access single characters in strings using subseting a[0]
* Strings are immutable, you can’t change a character
* Get substrings with slice
* Check the mozilla docs to see more helpful methods on strings : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/String

**3.1.2.4** [**Basic Operators**](https://thealtschool.com/topic/basic-operators/)

**Terminologies**

* An operator is a reserved syntax consisting of punctuation or alphanumeric characters that carries out built-in functionality. e.g addition operator (“**+**“), subtraction operator (“**–**“).
* An operator can be **unary** or **binary**.
* An operand – is what operators are applied to.

**Math operators**

* The following math operations are supported in JavaScript:
  + Addition +,
  + Subtraction -,
  + Multiplication \*,
  + Division /,
  + Remainder %,
  + Exponentiation \*\*
* Remember If an expression has more than one operator, the execution order is defined by their [precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence), or, in other words, the default priority order of operators.
* **binary + is applied to strings, it merges (concatenates) them:**
* We know many operators from school. They are things like addition +, multiplication \*, subtraction -, and so on. **Show examples**
* Remember If an expression has more than one operator, the execution order is defined by their [precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence), or, in other words, the default priority order of operators.
* Parentheses override any precedence, so if we’re not satisfied with the default order, we can use them to change it. For example, write (1 + 2) \* 2.
* Let’s meet features of JavaScript operators that are beyond school arithmetics. So operators like **addition** works for strings. **Show example**
  + **if the binary + is applied to strings, it merges (concatenates) them:**

Next, let’s talk about JavaScript-specific operators, not covered by school arithmetic.

**3.1.2.5** [**Comparison Operators**](https://thealtschool.com/topic/comparison-operators/)

**Comparison operators**

* Comparison operators are used to apply boolean logic to operands. E.g
  + Greater than: **a > b**
  + Less than:**a < b**
  + Greater or equals: **a >= b**
  + Less than or equals: **a <= b**
  + Equals: **a == b, Strict Equality a === b**e.g 0 === false
  + Not equals: **a != b**
* All comparison operators return a boolean value
* To see whether a string is greater than another, JavaScript uses the so-called “dictionary” or “lexicographical” order. In other words, strings are compared letter-by-letter.
* Please note the double equality sign == means the equality test, while a single one a = b means an assignment.
* To see whether a string is greater than another, JavaScript uses the so-called “dictionary” or “lexicographical” order. In other words, strings are compared letter-by-letter.
* When comparing values of different types, JavaScript converts the values to numbers. alert( ‘2’ > 1 )
* For boolean values, true becomes 1 and false becomes 0.
* A regular equality check == has a problem. It cannot differentiate 0 from false: 0 == false, ” == false.
* These are JavaScript false values. Use a strict equality check instead. (0 === false ); // false, because the types are different

**Other operators**

* The assignment operator (“=”) is used to assign values to a variable
* The increment (“++”) and decrement (“–“) operators
* Bitwise operators  : AND ( & ), OR ( | ), XOR ( ^ ), NOT ( ~ ), LEFT SHIFT ( << ), RIGHT SHIFT ( >> ), etc
* Assignment **=** is also an operator. It is listed in the precedence table with the very low priority.
* Bitwise operators are used very rarely, when we need to fiddle with numbers on the very lowest (bitwise) level.
* Let’s note that an assignment = is also an operator. It is listed in the precedence table with the very low priority of 2. That’s why, when we assign a variable, like x = 2 \* 2 + 1, the calculations are done first and then the = is evaluated, storing the result in x.
* Increasing or decreasing a number by one is among the most common numerical operations. So, there are special operators for it:
* Bitwise operators are used very rarely, when we need to fiddle with numbers on the very lowest (bitwise) level. We won’t need these operators any time soon, as web development has little use of them, but in some special areas, such as cryptography, they are useful. You can read the Bitwise Operators chapter on MDN when a need arises.

**3.1.2.6** [**Conditionals**](https://thealtschool.com/topic/conditionals/)

**Conditional branching (if)**

* Sometimes, we need to perform different actions based on different conditions. To do that, we can use the if statement and the conditional operator
* The if(…) statement evaluates a condition and, if the result is true, executes a block of code.
* We recommend wrapping your code block with curly braces {} every time you use an if statement, even if there is only one statement to execute. Doing so improves readability.
* The if (…) statement evaluates the expression in its parentheses and converts the result to a boolean. if (0) { // 0 is falsy..} works because 0 is falsy

**Conditional branching (if-else)**

* The **else** clause may contain an optional block if code that evaluates when the condition is falsy

**Conditional branching (else-if)**

* Multiple **else-if** clause can be chained to test numerous conditions
* The final else is optional. When all else fails, it gets executed

**Conditional operator ( ? )**

* The conditional operator (? ) called the ternary operator can be used to assign values based on a condition
* The operator is represented by a question mark ?. Sometimes it’s called “ternary”, because the operator has three operands. It is actually the one and only operator in JavaScript which has that many.

**Conditional branching (switch)**

* A **switch** statement can replace multiple **if** checks. It gives a cleaner way of comparing a value with multiple variants. It has the following syntax:
* The switch has one or more case blocks and an optional default.
* The case check is a strict equality
* If the equality is found, switch starts to execute the code starting from the corresponding case, until the nearest break (or until the end of switch).
* If no case is matched then the default code is executed (if it exists).

**Logical operators**

* Logical operators available in JavaScript are: **||**(**OR**),**&&**(**AND**),**!**(**NOT**)
* Although they are called “logical”, they can be applied to values of any type, not only boolean. Their result can also be of any type.
* OR==> If any of its arguments are true, it returns true, otherwise it returns false. Show examples with if statements
* If an operand is not a boolean, it’s converted to a boolean for the evaluation. For instance, the number 1 is treated as true, the number 0 as false: if (1 || 0) {…}

**3.1.2.7** [**Loops**](https://thealtschool.com/topic/loops/)

Loops are used to perform repeated actions based on a condition.

In JavaScript there many types of loops:

* The **“while”** loop
* The “do…while” loop
* The “for” loop
* The for…of loop
* The for…in loop

**Loops ( while )**

* The while loop has the following syntax
* While the condition is truthy, the code in the while loop body is executed.
* A single execution of the loop body is called an iteration. The loop in the example above makes three iterations.
* If i++ was missing from the example above, the loop would repeat (in theory) forever. In practice, the browser provides ways to stop such loops, and in server-side JavaScript, we can kill the process.
* Any expression or variable can be a loop condition, not just comparisons: the condition is evaluated and converted to a boolean.

**Loops ( do…while )**

* The do…while loop has the following syntax
* This is similar to while loop. The condition check is just moved below the loop body using the do..while syntax:
* This means the code is run at least once before the loop start.

**Loops ( for )**

* The for loop has the following syntax
* The for loop is more complex, but it’s also the most commonly used loop.
* If you are new to loops, it could help to go back to the example and reproduce how it runs step-by-step on a piece of paper.
* The “counter” variable i is declared right in the loop. This is called an “inline” variable declaration. Such variables are visible only inside the loop.
* we can force the exit at any time using the special break directive.
* The continue directive is a “lighter version” of break. It doesn’t stop the whole loop. Instead, it stops the current iteration and forces the loop to start a new one (if the condition allows).

**Loops ( for…of )**

* The **for…of** loop is used to loop over iterable properties of an object. It has the following syntax
* The values of an iterable object can be iterated over, meaning they can counted. E.g Arrays, Maps, Set, Strings  (We will talk deeply about these in Data Structure section)
* On each iteration a value of a different property is assigned to variable. variable may be declared with const, let.

**Loops ( for…in )**

* The **for…in** has the following syntax
* The for…in statement iterates over all enumerable properties of an object that are keyed by strings (ignoring ones keyed by Symbols), including inherited enumerable properties.
* Given that for…in is built for iterating object properties, not recommended for use with arrays, and options like Array.prototype.forEach() and for…of exist, what might be the use of for…in at all?
* It may be most practically used for debugging purposes, being an easy way to check the properties of an object

**3.1.2.8** [**Functions**](https://thealtschool.com/topic/functions-2/)

* Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.
* You’ve seen lots of built-in functions like: alert, console.log, prompt, etc
* Quite often we need to perform a similar action in many places of the script. For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.
* Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

**Functions Variables**

* Local variables: These are variables declared inside a function is only visible inside that function.
* Outer variables: These are variables declared outside a function.
* The function has full access to the outer variable. It can modify it as well.
* An outer variable is only used if there’s no local one. If a same-named variable is declared inside the function then it shadows the outer one.
* Variables declared outside of any function, such as the outer userName in the code above, are called global.
* Global variables are visible from any function (unless shadowed by locals). It’s a good practice to minimize the use of global variables.

**Functions Parameters**

* Parameters are data passed to a function. These are passed in the function’s parenthesis ( ).
* We can pass arbitrary data to functions using parameters.
* When the function is called with arguments, the given values are copied to local variables. Then the function uses them.
* A parameter is the variable listed inside the parentheses in the function declaration (it’s a declaration time term)
* An argument is the value that is passed to the function when it is called (it’s a call time term).
* We declare functions listing their parameters, then call them passing arguments.
* If a function is called, but an argument is not provided, then the corresponding value becomes undefined.
* We can specify the so-called “default” (to use if omitted) value for a parameter in the function declaration, using =:. e.g function showMessage(from, text = “no text given”)

**Functions Returns**

* A function can return a value back into the calling code as the result.
* The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).
* There may be many occurrences of return in a single function. For instance, an if..else block
* It is possible to use return without a value. That causes the function to exit immediately.
* If a function does not return a value, it is the same as if it returns undefined:
* If we want the returned multiple results, then  wrap them in parenthesis

**Functions Tips**

* Functions are actions. So their name is usually a verb. e.g **getUser**, **calculatePercent**, **addInterest**, etc
* A function should do exactly what is suggested by its name, no more.
* Functions should be short and do exactly one thing. If that thing is big, maybe it’s worth it to split the function into a few smaller functions
* Two independent actions usually deserve two functions, even if they are usually called together
* A function is similar to a comment. That is, it should be self descriptive from it’s name and usage

**Function Expression**

* There is another syntax for creating a function that is called a **Function Expression**. It allows us to create a new function in the middle of any expression.
* The syntax that we used before is called a Function Declaration.

**Callback functions**

* Functions are like values, and they can be assigned, as well as passed around just like variables
* In practice, such functions are quite useful
* The arguments showFullName and showUserName of displayUser are called callback functions or just callbacks.
* The idea is that we pass a function and expect it to be “called back” later if necessary. In our case, showFullName becomes the callback for display type “full”, and showUserName as opposite.

**Arrow Functions**

* There’s another very simple and concise syntax for creating functions, that’s often better than Function Expressions.
* It’s called “arrow functions”, because it looks like this
* Arrow functions can be used in the same way as Function Expressions.
* Arrow functions may appear unfamiliar and not very readable at first, but that quickly changes as the eyes get used to the structure.
* They are very convenient for simple one-line actions

**3.1.2.9** [**Scopes and Closures**](https://thealtschool.com/topic/scopes-and-closures/)

**Scopes & Closures**

* Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript. The two types of scope are local and global:
  + Global variables are those declared outside of a block
  + Local variables are those declared inside of a block
* A **closure** is a function that remembers its outer variables and can access them. In JS  all functions are naturally closures
* All functions and Objects have access to a Global scope.
* If a variable is declared inside a code block {…}, it’s only visible inside that block.

**3.1.3 Putting it All Together**

**3.1.3.1** [**Week 1 Project**](https://thealtschool.com/topic/week-1-project/)

**Putting it all together**

* In this last section, we’ll consolidate all you have learnt this week, with some mini projects
  + Project 1: Create a webpage that validates a user and returns user details
    - Accept details using prompts
    - Use an object to store user details
    - Validate user details
      * Username must be less than ten
      * Password must be greater than six
      * User must confirms password
    - If username and password do not match, alert the user
    - Display user details

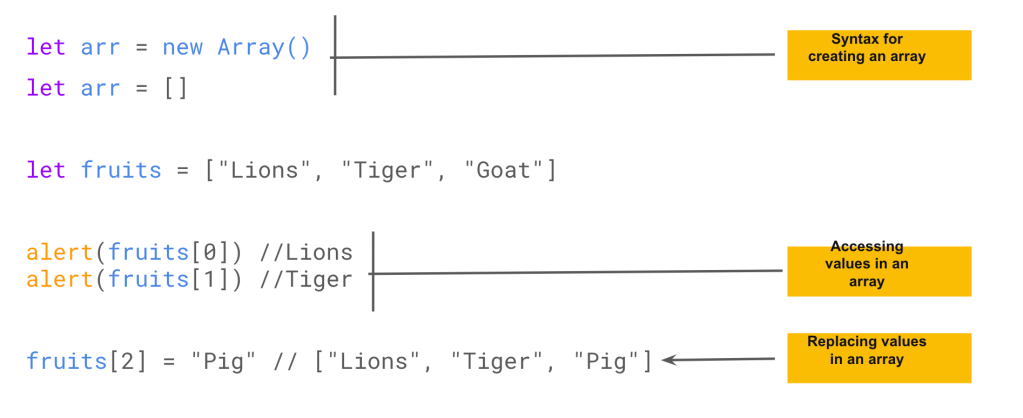
**3.1.3.2** [**Week 1 Project 2**](https://thealtschool.com/topic/week-1-project-2/)

**3.1.3.3** [**Week 1 Project 3**](https://thealtschool.com/topic/week-1-project-3/)

**Month 3: Week 2**

**3.2.1** [**Arrays**](https://thealtschool.com/topic/arrays/)

* Arrays are used to store ordered collections.
* There are two syntaxes for creating an empty array:

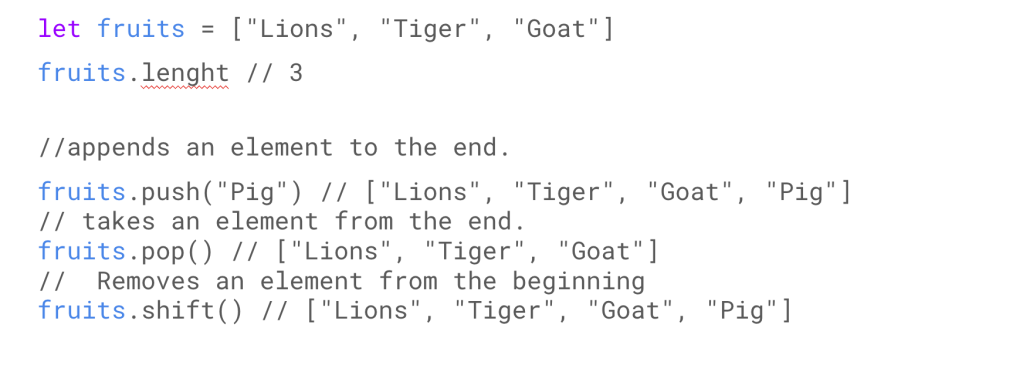


**3.2.1.1 Array**

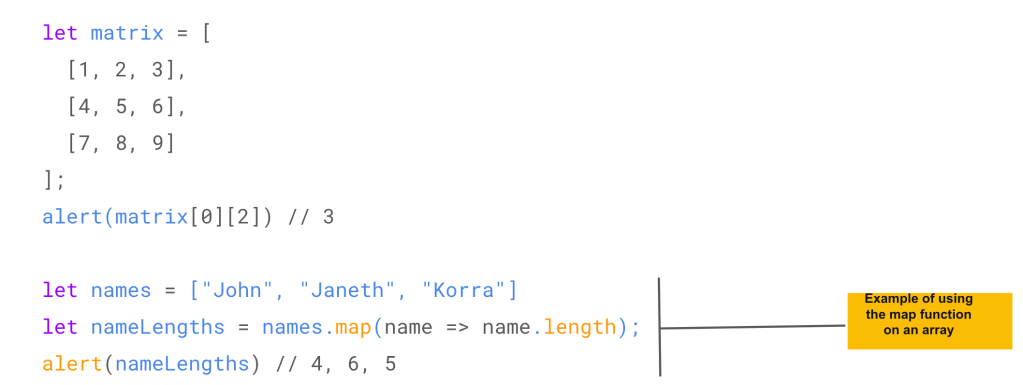
* Almost all the time, the second syntax is used. We can supply initial elements in the brackets:
* Array elements are numbered, starting with zero. We can get an element by its number in square brackets:
* An array can store elements of any type. Objects, functions, etc

**3.2.1.1 Array Method**

* Array methods are functions we can call on an array
* There are lots of methods built into Arrays. See full list [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)
* You can loop over arrays with **for, for…of, forEach**



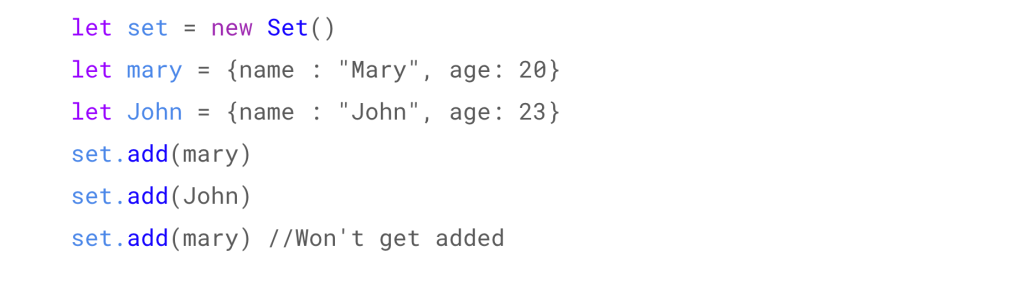
* The length **property** automatically updates when we modify the array. To be precise, it is actually not the count of values in the array, but the greatest numeric index plus one.
* Arrays can have items that are also arrays. We can use it for multidimensional arrays, for example to store matrices:



* Arrays can have items that are also arrays. We can use it for multidimensional arrays, for example to store matrices:
* Other popular array methods are map, slice, and splice

**3.2.2** [**Set**](https://thealtschool.com/topic/set/)

* A Set is a special type collection – “set of values” (without keys), where each value may occur only once.
* Set has the following methods and properties:
  + set.add(value) – adds a new value and returns the set
  + set.delete(value) – removes the value.
  + set.clear() – removes everything from the set.
  + set.has(value) – returns true if the value exists, false otherwise.
  + set.size – returns the set element count.

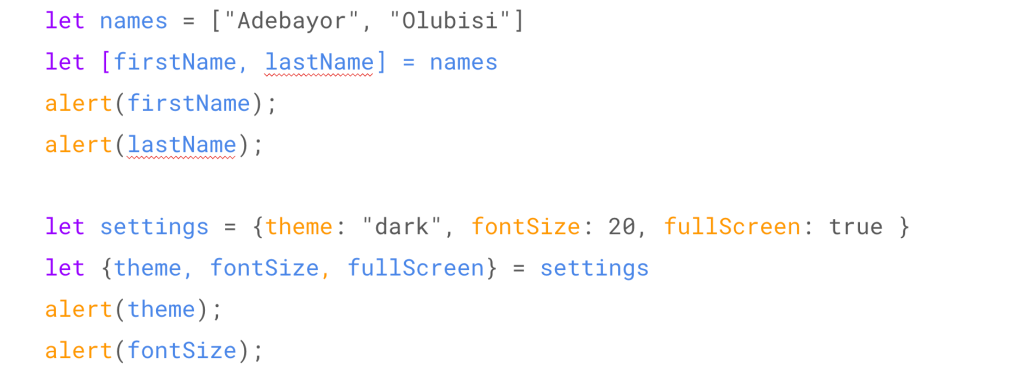


**Set**

* The main feature is that repeated calls of set.add(value) with the same value don’t do anything. That’s the reason why each value appears in a Set only once.
* For example, we have visitors coming, and we’d like to remember everyone. But repeated visits should not lead to duplicates. A visitor must be “counted” only once. Set is just the right thing for that:
* We can loop over a set either with for..of or using forEach

**3.2.3** [**Destructuring Assignment**](https://thealtschool.com/topic/destructuring-assignment/)

* Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes that’s more convenient.
* Destructuring also works great with complex functions that have a lot of parameters, default values, and so on. Soon we’ll see that.



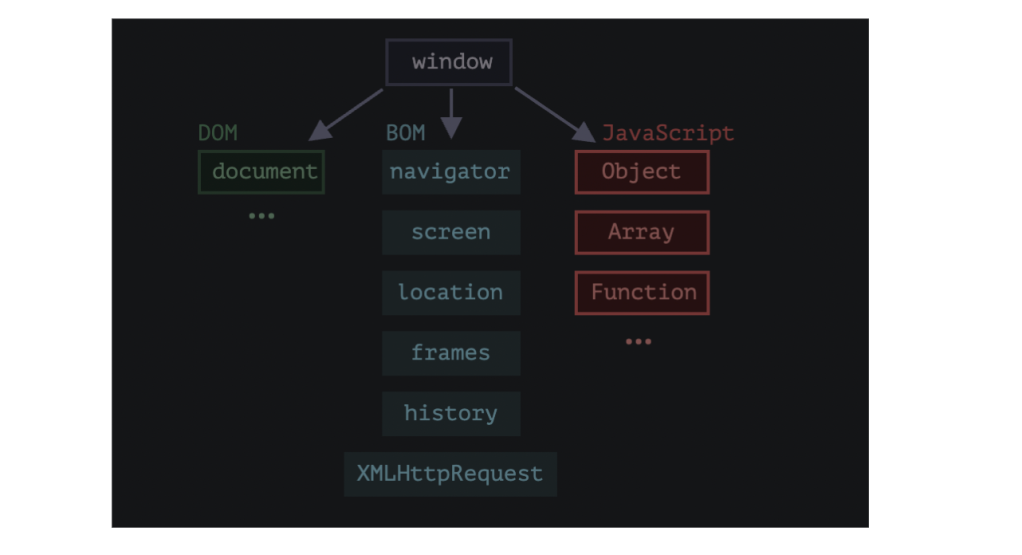
**Destructuring assignment**

* It’s called “destructuring assignment,” because it “destructurizes” by copying items into variables. But the array itself is not modified. let firstName = arr[0];
* Unwanted elements of the array can also be thrown away via an extra comma: let [firstName, , title] = [“Julius”, “Caesar”, “Consul”, “of the Roman Republic”]; In the code above, the second element of the array is skipped, the third one is assigned to title, and the rest of the array items is also skipped (as there are no variables for them).
* We can use it with any iterable, not only arrays: let [a, b, c] = “abc”; // [“a”, “b”, “c”]
* The order of destructuring in objects does not matter
* If we have a complex object with many properties, we can extract only what we need.
* If an object or an array contain other nested objects and arrays, we can use more complex left-side patterns to extract deeper portions.

**3.2.4** [**Dom & Dom Manipulation**](https://thealtschool.com/topic/dom-dom-manipulation/)

**Browser specification**

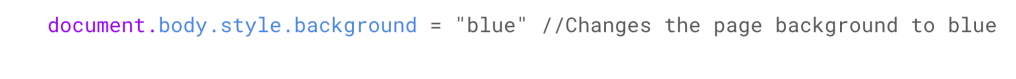
* Web browsers give a means to control web pages.
* There’s a “root” object called window. It has two roles:
  + First, it is a global object for JavaScript code, as described in the chapter Global object.
  + Second, it represents the “browser window” and provides methods to control it.



For instance, here we use it as a global object: alert(window.innerHeight); // inner window height

**DOM**

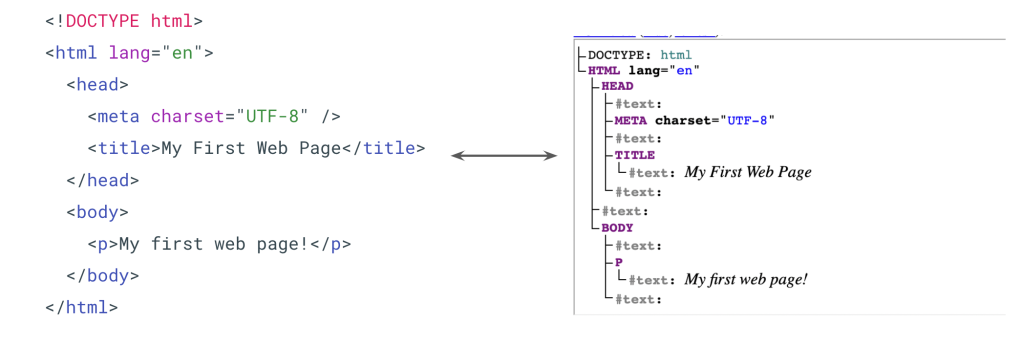
* Document Object Model, or DOM for short, represents all web page content as objects that can be modified.
* The document object is the main “entry point” to the page. We can change or create anything on the page using it.
* Properties and methods are described in the specification: [DOM Living Standard](https://dom.spec.whatwg.org/).
* An HTML/XML document is represented inside the browser as the DOM tree.



**3.2.5** [**Browser Specification**](https://thealtschool.com/topic/browser-specification/)

**3.2.6** [**DOM Tree**](https://thealtschool.com/topic/dom-tree/)

* All HTML tag in a page is an object. Nested tags are “children” of the enclosing one. The text inside a tag is an object as well.
* The Dom tree is a large object representing all page tags. You can see the Dom tree of page by inspecting the page in the developer console

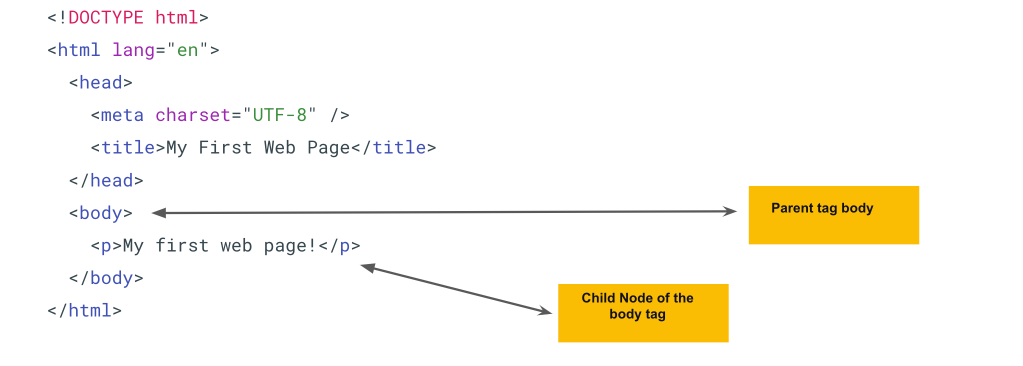


**DOM Tree**

* The backbone of an HTML document is tags.
* For example, document.body is the object representing the <body> tag. Run this code :**document.body.style.background = “red”**
* DOM nodes have properties and methods that allow us to travel between them, modify them, move around the page, and more. We’ll get down to them in the next chapters.

**3.2.7** [**DOM Navigation & Searching**](https://thealtschool.com/topic/dom-navigation-searching/)

* The DOM allows us to do anything with elements and their contents, but first we need to reach the corresponding DOM object.
* All operations on the DOM start with the document object. That’s the main “entry point” to DOM. From it we can access any node.



**DOM Navigation**

* The topmost tree nodes are available directly as document properties: <html> = document.documentElement
* <body> = document.body
* Child nodes (or children) – elements that are direct children. In other words, they are nested exactly in the given one. For instance, <head> and <body> are children of <html> element.
* Descendants – all elements that are nested in the given one, including children, their children and so on.
* The childNodes collection lists all child nodes, including text nodes. Show example of listing child nodes

**Searching**

* There are additional searching methods for that. These are:
  + **document.getElementById** or just id
  + **querySelectorAll**: returns all elements inside elem matching the given CSS selector.
  + **querySelector**: The call to elem.querySelector(css) returns the first element for the given CSS selector
  + **elem.getElementsByClassName**:  returns elements that have the given CSS class.
  + **elem.getElementsByTagName:** looks for elements with the given tag and returns the collection of them. The tag parameter can also be a star “\*” for “any tags”.
* The id must be unique. There can be only one element in the document with the given id. If there are multiple elements with the same id, then the behavior of methods that use it is unpredictable
* Only document.getElementById, not anyElem.getElementById
* This method querySelectorAll  is indeed powerful, because any CSS selector can be used.
* The call to elem.querySelector(css) returns the first element for the given CSS selector. In other words, the result is the same as elem.querySelectorAll(css)[0], but the latter is looking for all elements and picking one, while elem.querySelector just looks for one. So it’s faster and also shorter to write.

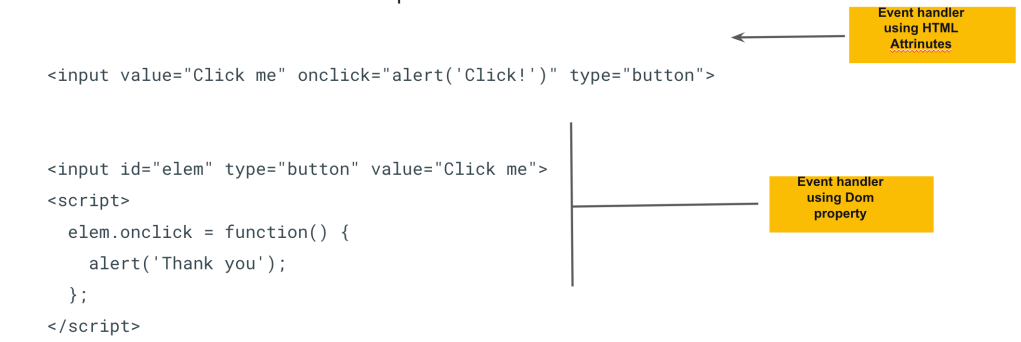
**3.2.8** [**DOM Modification**](https://thealtschool.com/topic/dom-modification/)

* DOM modification is the key to creating “live” pages.
* To create DOM nodes, there are two methods:
  + document.createElement(tag): Creates a new element node with the given tag:
  + document.createTextNode(text): Creates a new text node with the given text:
  + You can insert into the DOM using :   document.body.append(div). There are other variants like prepend, before, after, replaceWith. See
* To remove a node, there’s a method node.remove().
* The id must be unique. There can be only one element in the document with the given id. If there are multiple elements with the same id, then the behavior of methods that use it is unpredictable
* Only document.getElementById, not anyElem.getElementById
* This method querySelectorAll  is indeed powerful, because any CSS selector can be used.
* The call to elem.querySelector(css) returns the first element for the given CSS selector. In other words, the result is the same as elem.querySelectorAll(css)[0], but the latter is looking for all elements and picking one, while elem.querySelector just looks for one. So it’s faster and also shorter to write.

**3.2.9** [**Browser Object Model**](https://thealtschool.com/topic/browser-object-model/)

**3.2.10** [**Dom Events & Events Handlers**](https://thealtschool.com/topic/dom-events/)

* An event is a signal that something has happened. All DOM nodes generate such signals (but events are not limited to DOM).
* The following events can be tracked in the DOM:
  + Mouse events: E.g click, mouseover, mouseup
  + Keyboard events: E.g keydown, keyup
  + Form element events: E.g submit, focus
  + Document events: E.g DOMContentLoaded
  + CSS events: E.g transitions
* Almost all the time, the second syntax is used. We can supply initial elements in the brackets:
* Array elements are numbered, starting with zero. We can get an element by its number in square brackets:
* An array can store elements of any type. Objects, functions, etc
* To react on events we can assign a handler – a function that runs in case of an event.
* We can set event handlers using:
  + HTML Attributes: A handler can be set in HTML with an attribute named on<event>.
  + DOM property: We can assign a handler using a DOM property**on<event>.**
  + Methods event listeners. Explained in next slide

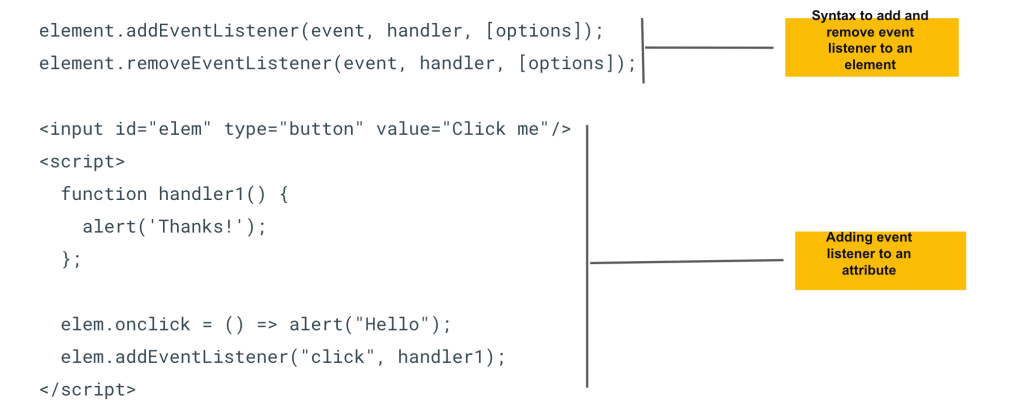


Event Handlers

* Handlers are a way to run JavaScript code in case of user actions. There are several ways to assign a handler. Let’s see them, starting from the simplest one.
* For instance, to assign a click handler for an input, we can use onclick, like here: On mouse click, the code inside onclick runs.
* Please note that inside onclick we use single quotes, because the attribute itself is in double quotes. If we forget that the code is inside the attribute and use double quotes inside, like this: onclick=”alert(“Click!”)”, then it won’t work right.
* An HTML-attribute is not a convenient place to write a lot of code, so we’d better create a JavaScript function and call it there.
* As we know, HTML attribute names are not case-sensitive, so ONCLICK works as well as onClick and onCLICK… But usually attributes are lowercased: onclick.
* If the handler is assigned using an HTML-attribute then the browser reads it, creates a new function from the attribute content and writes it to the DOM property.

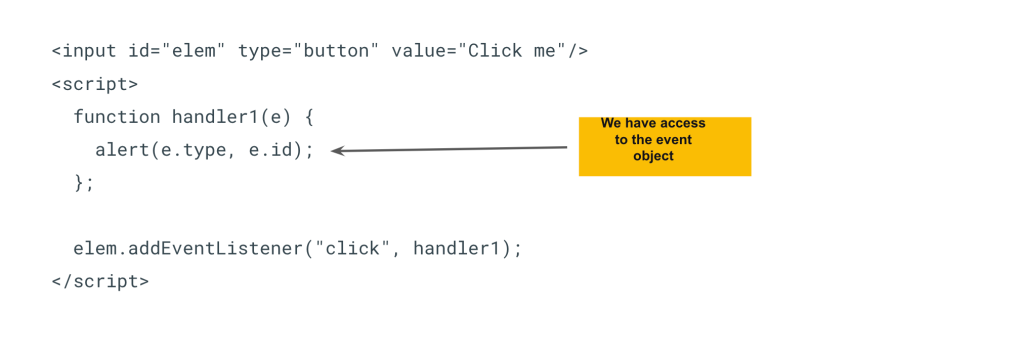
**3.2.11** [**Event Listeners, Event Object**](https://thealtschool.com/topic/event-listeners/)

* Listeners can be attached to Dom elements, and will fire if the event happens
* You can use **addEventListener**to assign multiple event handlers to an element
* **removeEventListener**can be used to remove an event listener from an element



**Event Listeners**

* Using event handlers, you cannot attach more than one handler to an attribute. The fundamental problem of the aforementioned ways to assign handlers – we can’t assign multiple handlers to one event.
* Developers of web standards understood that long ago and suggested an alternative way of managing handlers using special methods addEventListener and removeEventListener. They are free of such a problem.
* When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler.
* Some properties of the event object are:  event.id, event.type, event.name etc. See full list [here](https://developer.mozilla.org/en-US/docs/Web/API/Event)

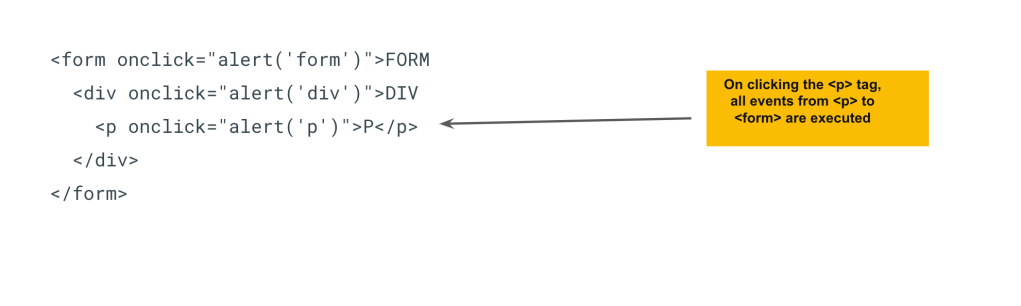


**Event Object**

* To properly handle an event we’d want to know more about what’s happened. Not just a “click” or a “keydown”, but what were the pointer coordinates? Which key was pressed? And so on.
* When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler.

**3.2.11** [**UI Events**](https://thealtschool.com/topic/ui-events/)

* When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.
* Bubbling is convenient. Don’t stop it without a real need: obvious and architecturally well thought out.
* To stop bubbling, use the method **event.stopPropagation()**.
* Almost all events bubble. One exception is the focus event.



**Event Bubbling**

* To properly handle an event we’d want to know more about what’s happened. Not just a “click” or a “keydown”, but what were the pointer coordinates? Which key was pressed? And so on.
* When an event happens, the browser creates an event object, puts details into it and parses it as an argument to the handler.

**Month 3: Week 3**

**3.3.1 Promises, Async/Await**

**3.3.1.1** [**Intro to Week 3**](https://thealtschool.com/topic/intro-to-week-3/)

**3.3.1.2** [**Callbacks**](https://thealtschool.com/topic/callbacks/)

* A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
* This execution may be immediate as in a synchronous callback, or it might happen at a later point in time as in an asynchronous callback.
* What if the script loading fails? Our callback should be able to react on that.
* Multiple asynchronous call back can result in the callback hell.

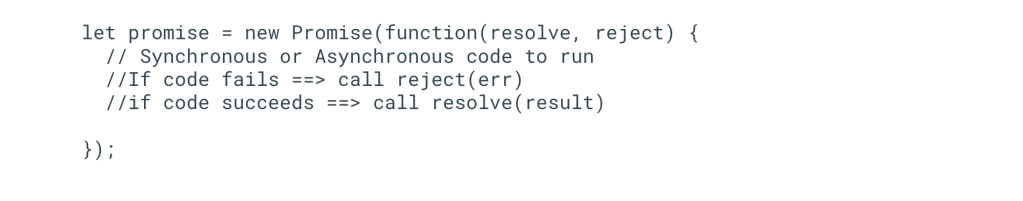


**Callbacks**

* Many functions are provided by JavaScript host environments that allow you to schedule asynchronous actions. In other words, actions that we initiate now, but they finish later. For instance, one such function is the **setTimeout** function.
* Luckily, there are other ways to avoid such pyramids. One of the best ways is to use “promises”, described in the next slide.

**3.3.1.3** [**Promise**](https://thealtschool.com/topic/promise/)

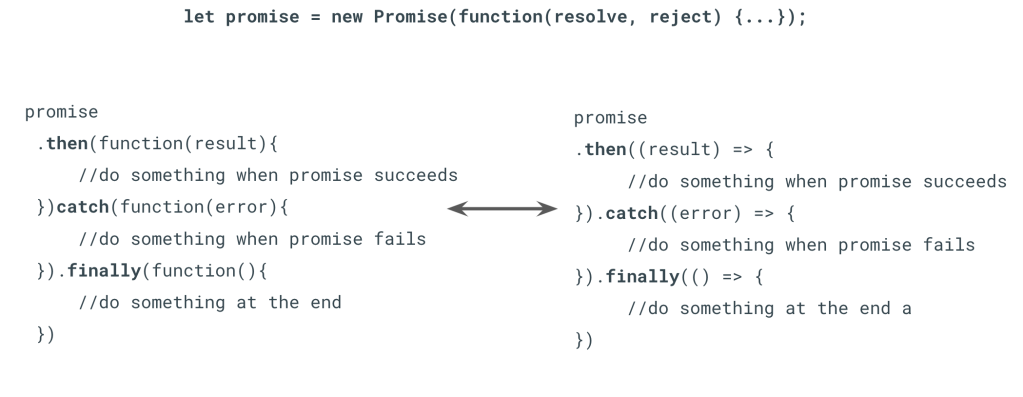
* The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
* A Promise object returns a **promise** to supply the resulting value at some point in the future. So we can wait for it to keep the promise
* A promise can be in any of these state:
  + **pending**: initial state, neither fulfilled nor rejected.
  + **fulfilled**: meaning that the operation was completed successfully.
  + **rejected**: meaning that the operation failed.



**Promise**

* A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action’s eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.
* The methods promise.then(), promise.catch(), and promise.finally() are used to associate further action with a promise that becomes settled.

Consuming functions can be registered (subscribed) using methods .then, .catch and .finally.

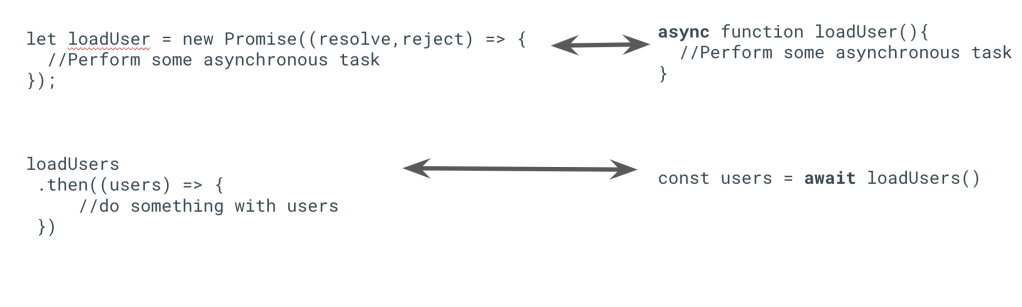


**Consuming a Promise**

* A Promise object serves as a link between the executor (the “producing code” or “singer”) and the consuming functions (the “fans”), which will receive the result or error. Consuming functions can be registered (subscribed) using methods .then, .catch and .finally.
* The first argument of .then is a function that runs when the promise is resolved, and receives the result.
* The call .finally(f) is similar to .then(f, f) in the sense that f always runs when the promise is settled: be it resolve or reject.
* A finally handler has no arguments. In finally we don’t know whether the promise is successful or not. That’s all right, as our task is usually to perform “general” finalizing procedures.

**3.3.1.3** [**Async/Await**](https://thealtschool.com/topic/async-await/)

* async/await are special syntax to work with promises in a more comfortable fashion.
* The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.
* The keyword await makes JavaScript wait until that promise settles and returns its result.



**async/await**

* **await** only works inside an async function. If we try to use await in a non-async function, there would be a syntax error:
* Show example of converting the chapter Promises chaining and rewrite it using async/await:
* If a promise resolves normally, then await promise returns the result. But in the case of a rejection, it throws the error, just as if there were a throw statement at that line. So handle errors with try..catch
* When we use async/await, we rarely need .then, because await handles the waiting for us. And we can use a regular try..catch instead of .catch. That’s usually (but not always) more convenient.
* But at the top level of the code, when we’re outside any async function, we’re syntactically unable to use await, so it’s a normal practice to add .then/catch to handle the final result or falling-through error, like in the line (\*) of the example above.

**3.3.2 OOP in JavaScript**

**3.3.2.1** [**Intro to OOP**](https://thealtschool.com/topic/oop/)

* Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects”, which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). (Wikipedia)
* This approach to programming is well-suited for programs that are large, complex and actively updated or maintained.
* OOP uses the concept of reusable templates (classes) which encapsulate actions (methods), and features (properties) in them.

**Structure of OOP**

* The structure, or building blocks, of object-oriented programming include the following:
  + **Classes:** These are user-defined data types that act as the blueprint for individual objects, attributes and methods.
  + **Objects:**These are instances of a class created with specifically defined data.
  + **Methods:**These are functions that are defined inside a class that describe the behaviors of an object.
  + **Attributes:**These are properties defined in the class template and represent the state of an object.
* Objects can correspond to real-world objects or an abstract entity. When class is defined initially, the description is the only object that is defined.
* Each method contained in class definitions starts with a reference to an instance object. Additionally, the subroutines contained in an object are called instance methods. Programmers use methods for reusability or keeping functionality encapsulated inside one object at a time.
* Objects will have data stored in the attributes field. Class attributes belong to the class itself.

**Principles of OOP**

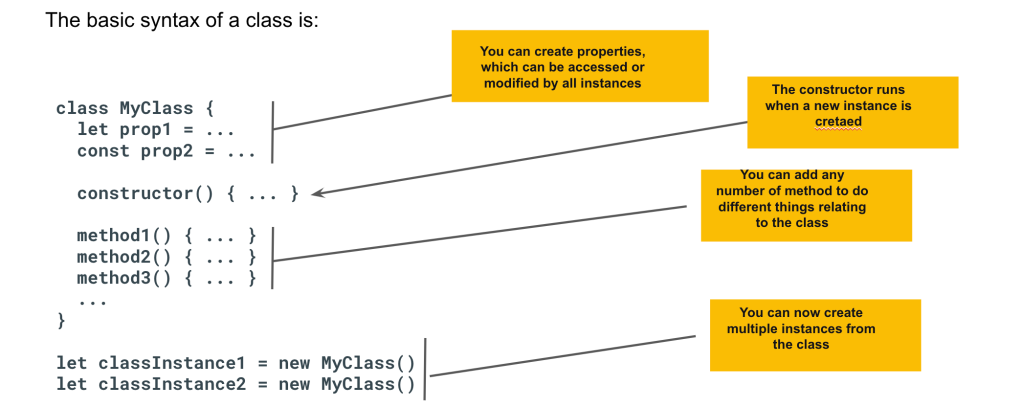
* Object-oriented programming is based on the following principles:
  + **Encapsulation:**Important information is contained inside an object and only select information is exposed
  + **Abstraction:**Reveal important features, hide unnecessary details
  + **Inheritance:** Classes can reuse, or extend code from other classes
* Encapsulation. This principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.
* Abstraction. Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers more easily make additional changes or additions over time.
* Inheritance. Classes can reuse code from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.

**Benefits of OOP**

* Benefits of OOP include:
  + **Modularity**. Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
  + **Reusability**. Code can be reused through inheritance.
  + **Maintainability**. Programs become easy to maintain and improve
  + **Security**. Using encapsulation and abstraction, complex code is hidden
* Modularity. Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
* Reusability. Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
* Productivity. Programmers can construct new programs quicker through the use of multiple libraries and reusable code.
* Easily upgradable and scalable. Programmers can implement system functionalities independently.
* Interface descriptions. Descriptions of external systems are simple, due to message passing techniques that are used for objects communication.
* Security. Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.
* Flexibility. Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

**3.3.2.2** [**Intro to OOP**](https://thealtschool.com/topic/oop/)

In JS you can create an object oriented program by using the **class** keyword

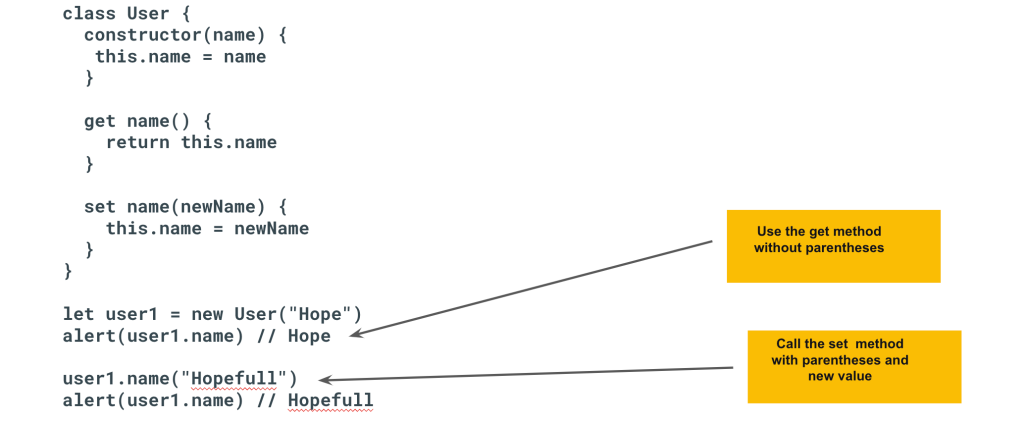


**OOP in JavaScript**

**3.3.2.3** [**Constructor**](https://thealtschool.com/topic/constructor/)

**3.3.2.3** [**Inheritance**](https://thealtschool.com/topic/inheritance/)

* Getters are methods in classes that return values. They are mostly used to return class properties
* Setters on the other hand, are use to update class properties.



**Getters and Setters**

**3.3.2.5** [**Live Coding Platforms**](https://thealtschool.com/topic/live-coding-platforms/)

* The following platforms allows you to write and run JavaScript without installing anything
  + [**CodeSandBox**](https://codesandbox.io/)
  + [CodePen](https://codepen.io/)
  + [**JSFiddle**](https://jsfiddle.net/)
  + [**Replit**](https://replit.com/)

**3.3.2.6** [**Wrapping Up**](https://thealtschool.com/topic/wrapping-up-2/)

**Month 3: Week 4**

**3.4.1 More CSS**

**3.4.1.1** [**Media Queries**](https://thealtschool.com/topic/media-queries/)

**Media queries** are useful when you want to