

JavaScript

Introduction:

- JavaScript is a dynamic computer programming language.
- It is lightweight and most commonly used as a part of web pages.
- Javascript implementations allow client-side script to interact with the user and make dynamic pages.
- It is an interpreted programming language with object-oriented capabilities.

- European Computer Manufacturers Association (ECMAScript) or (ES) is a standard for scripting languages like JavaScript, ActionScript and Jscript.

Advantages:

1. Less server interaction
2. Immediate Feedback to the visitors
3. Increased interactivity
4. Rich Interfaces

Introduction: ES5

- Data Types:
- JavaScript allows you to work with three primitive data types –
- **Numbers**, eg. 123, 120.50 etc.
- **Strings** of text e.g. "This text string" etc.
- **Boolean** e.g. true or false.
- JavaScript also defines two trivial data types, **null** and **undefined**, each of which defines only a single value.
- In addition to these primitive data types, JavaScript supports a composite data type known as **object**

Syntax:

- The console is a panel that displays important messages, like errors, for developers.
- Much of the work the computer does with our code is invisible to us by default.
- If we want to see things appear on our screen, we can print, or log, to our console directly.
- In JavaScript, the console keyword refers to an object, a collection of data and actions, that we can use in our code.
- Keywords are words that are built into the JavaScript language, so the computer will recognize them and treats them specially.
- One action, or method, that is built into the console object is the `.log()` method.
- When we write `console.log()` what we put inside the parentheses will get printed, or logged, to the console.

ES6 Syntax:

- A JavaScript program can be composed of –
- **Variables** – Represents a named memory block that can store values for the program.
- **Literals** – Represents constant/fixed values.
- **Operators** – Symbols that define how the operands will be processed.
- **Keywords** – Words that have a special meaning in the context of a language.
- **Modules** – Represents code blocks that can be reused across different programs/scripts.
- **Comments** – Used to improve code readability. These are ignored by the JavaScript engine.

- **Identifiers** – These are the names given to elements in a program like variables, functions, etc.
- The rules for identifiers are –
 - Identifiers can include both, characters and digits.
 - The identifier cannot begin with a digit.
 - Identifiers cannot include special symbols except for underscore (_) or a dollar sign (\$).
 - Identifiers cannot be keywords.
 - They must be unique.
 - Identifiers are case sensitive.
 - Identifiers cannot contain spaces.

White Spaces and Line Breaks:

- ES6 ignores spaces, tabs, and newlines that appear in programs.
- JavaScript is case-sensitive.
- Each line of instruction is called a **statement**.
- Semicolons are optional in JavaScript.
- `Console.log("Hi")`

Variables:

- acts as a container for values in a program.
- Variable names are called **identifiers**.
- A variable must be declared before it is used.
- ES5 syntax used the **var** keyword to achieve the same

JavaScript and Dynamic Typing:

- JavaScript is an un-typed language.
- This means that a JavaScript variable can hold a value of any data type.
- You don't have to tell JavaScript during variable declaration what type of value the variable will hold.
- The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically.

JavaScript Variable scope:

- Global Variables:
 - it can be defined anywhere in your JavaScript code.
- Local Variables:
 - visible only within a function where it is defined.
 - Function parameters are always local to that function.
- E.g.
- ES6 defines a new variable scope- The Block Scope

Var, let and const

- Var variables can be update and redeclared within its scope.
- Let variables can be updated but not redeclared
- Const variables can neither be updated nor redeclared.
- Variables declared with var are in the function scope.
- Variables declared as let are in the block scope
- Variables declared as const are in the block scope.

- If we try to declare a let variable twice within the same block it will throw an error.
- But let variable can be used in different block level scopes without any syntax error.

The Let and Block Scope:

- The block scope restricts a variable's access to the block in which it is declared.
- The **var** keyword assigns a function scope to the variable.

Function scope

- Variable having Function-scope means, variable will only be available to use inside the function it declared,
- will not be accessible outside of function,
- and will give Reference Error if we try to access.

```
function myFun() {  
    var a=10;  
    console.log(a);  
}
```

```
myFun();  
console.log(a)
```


Block scope:

- Block means a pair of curly brackets,
- a block can be anything that contains an opening and closing curly bracket.
- Variable having Block-scope will only be available to use inside the block it declared,
- will not be accessible outside the block,
- and will give Reference Error if we try to access.

```
<script>  
    if(true) {  
        let a = 10;  
        console.log(a);  
    }  
    console.log(a);  
  
    </script>
```

The const:

- The **const** declaration creates a read-only reference to a value.
- Constants are block-scoped, much like variables defined using the let statement.
- The value of a constant cannot change through re-assignment, and it can't be re-declared.
- The following rules hold true for a variable declared using the **const** keyword –
 - Constants cannot be reassigned a value.
 - A constant cannot be re-declared.
 - A constant requires an initializer. This means constants must be initialized during its declaration.
 - The value assigned to a **const** variable is mutable.

JavaScript “use strict”

- Use strict;
- Defines that javascript should be executed in strict mode.
- The “use strict” directive was new in ES5
- It is not a statement but a literal, ignored by earlier version of Javascript.
- Its purpose is to indicate that the code should be executed in strict mode.
- With strict mode we can not use undeclared variables.
- Strict mode is declared by adding "use strict"; to the beginning of a script or a function.
- Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode)

JavaScript “use strict”

- The strict mode in JavaScript does **not** allow following things:
- Use of undefined variables
- Use of reserved keywords as variable or function name
- Duplicate properties of an object
- Duplicate parameters of function

Hoisting:

- Prior to ES6, the **var** keyword was used to declare a variable in JavaScript.
- Variables declared using **var** do not support block level scope.
- This means if a variable is declared in a loop or **if block** it can be accessed outside the loop or the **if block**.
- This is because the variables declared using the **var** keyword support hoisting.
- Hoisting is JavaScript's default behavior of moving declarations to the top.
- a variable can be used before it has been declared.

- **Variable hoisting** allows the use of a variable in a JavaScript program, even before it is declared.
- Such variables will be initialized to **undefined** by default.
- JavaScript runtime will scan for variable declarations and put them to the top of the function or script.
- Variables declared with **var** keyword get hoisted to the top.

Operator:

- An **expression** is a special kind of statement that evaluates to a value. Every expression is composed of –
- **Operands** – Represents the data.
- **Operator** – Defines how the operands will be processed to produce a value.
- JavaScript supports the following types of operators –
 - Arithmetic operators
 - Logical operators
 - Relational operators
 - Bitwise operators
 - Assignment operators
 - Ternary/conditional operators
 - String operators
 - Type operators
 - The void operator

Arithmetic Operators

- Addition
- Subtraction
- Multiplication
- Division
- Modulus
- Increment
- Decrement

Test ? expr1 : expr2

Conditional Operator:

- This operator is used to represent a conditional expression.
- The conditional operator is also sometimes referred to as the ternary operator.
- Where,
- **Test** – Refers to the conditional expression
- **expr1** – Value returned if the condition is true
- **expr2** – Value returned if the condition is false

String Operators : Concatenation operator (+)

- The + operator when applied to strings appends the second string to the first.
- The concatenation operation doesn't add a space between the strings.
- Multiple strings can be concatenated in a single statement.

Typeof Operator

- It is a unary operator.
- This operator returns the data type of the operand.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"

Spread Operator

- **ES6** provides a new operator called the **spread operator**.
- The spread operator is represented by three dots “...” .
- The spread operator converts an array into individual array elements.
- The spread operator can be used to copy one array into another.
- It can also be used to concatenate two or more arrays.

Decision Making:

Loops:

- Certain instructions require repeated execution.
- Loops are an ideal way to do the same.
- A loop represents a set of instructions that must be repeated.
- Definite Loop:
 - A loop whose number of iterations are definite/fixed is termed as a definite loop.
 - The 'for loop' is an implementation of a definite loop.

```
for (variablename in object) { statement or block to execute }
```

Loops: for...in loop

- The for...in loop is used to loop through an object's properties.

Functions:

- **Functions** are the building blocks of readable, maintainable, and reusable code.
- Functions are defined using the function keyword.
- To force execution of the function, it must be called.
- This is called as function invocation

Classification of function:

1. Returning

2. Parametrized

- Functions may also return the value along with control, back to the caller.
- Such functions are called as returning functions.
- A returning function must end with a return statement.
- A function can return at the most one value. In other words, there can be only one return statement per function.
- The return statement should be the last statement in the function.

- Parameters are a mechanism to pass values to functions.
- Parameters form a part of the function's signature.
- The parameter values are passed to the function during its invocation.
- Unless explicitly specified, the number of values passed to a function must match the number of parameters defined.

Default Function Parameter

- In ES6, a function allows the parameters to be initialized with default values, if no values are passed to it or it is undefined.
- Rest Parameters:
 - Rest parameters are similar to variable arguments in Java.
 - Rest parameters doesn't restrict the number of values that you can pass to a function.
 - The values passed must all be of the same type.
 - To declare a rest parameter, the parameter name is prefixed with three periods, known as the spread operator.

Anonymous Function

- Functions that are not bound to an identifier (function name) are called as anonymous functions.
- These functions are dynamically declared at runtime.
- Anonymous functions can accept inputs and return outputs, just as standard functions do.
- An anonymous function is usually not accessible after its initial creation.
- Variables can be assigned an anonymous function.
- Such an expression is called a **function expression**.

Function Constructor:

- We can define function dynamically using Function() constructor along with the new operator.
- Recursion:
- Recursion is a technique for iterating over an operation by having a function call itself repeatedly until it arrives at a result.
- Recursion is best applied when you need to call the same function repeatedly with different parameters from within a loop.

Lambda Function/ Arrow Function

- refers to anonymous functions in programming.
- Lambda functions are a concise mechanism to represent anonymous functions.
- These functions are also called as Arrow functions.
- There are 3 parts to a Lambda function –
 - Parameters – A function may optionally have parameters.
 - The fat arrow notation/lambda notation (\Rightarrow): It is also called as the goes to operator.
 - Statements – Represents the function's instruction set.

- Arrow functions remove the need to type out the keyword function every time you need to create a function.
- Instead, you first include the parameters inside the () and then add an arrow => that points to the function body surrounded in { }.


```
( [param1, parma2,...param n] )=>statement;
```

- Lambda Expression:
- It is an anonymous function expression that points to a single line of code.
- It is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

ZERO PARAMETERS

```
const functionName = () => {};
```

ONE PARAMETER

```
const functionName = paramOne => {};
```

TWO OR MORE PARAMETERS

```
const functionName = (paramOne, paramTwo) => {};
```

SINGLE-LINE BLOCK

```
const sumNumbers = number => number + number;
```

MULTI-LINE BLOCK

```
const sumNumbers = number => {  
  const sum = number + number;  
  return sum; } — RETURN STATEMENT  
};
```

Iterator:

- JavaScript iterators were introduced in ES6
- They are used to loop over a sequence of values.
- An iterator implements a `next()` function, that returns an object in the form of `{ value, done }`
- where `value` is the next value in the iteration sequence
- and `done` is a boolean determining if the sequence has already been consumed.
- `For`
- `For...in`
- `For ...of`
- `forEach(callback())`

```
let ranks = ['A', 'B', 'C'];
```

```
for (let i = 0; i < ranks.length; i++) {  
  console.log(ranks[i]);  
}
```

The for loop issues

- The for loop uses the variable `i` to track the index of the `ranks` array.
- The value of `i` increments each time the loop executes as long as the value of `i` is less than the number of elements in the `ranks` array.
- The code complexity grows when we nest a loop inside another loop.
- Keeping track of multiple variables inside the loops is error-prone.
- ES6 introduced a new loop construct called `for...of` to eliminate the standard loop's complexity
- and avoid the errors caused by keeping track of loop indexes.

The for loop issues

- To iterate over the elements of the ranks array, use the following for...of construct:

```
for(let rank of ranks) {  
    console.log(rank);  
}
```

- For...in loop allows you to iterate over all property keys of an object.
- forEach():
- The forEach() method calls a function once for each element in an array in order
- The forEach() method passes a callback function for each element of an array together.

Callback Function

- 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.

Syntax:

```
Arr.forEach(callback(current value [, index [, array]]), thisArgs)
```

Parameters:

- The forEach method passes a callback function for each element of an array together with the following parameters:
- Current Value (required) - The value of the current array element
- Index (optional) - The current element's index number
- Array (optional) - The array object to which the current element belongs

Iterator

- Iterator is an object which allows us to access a collection of objects one at a time.
- The following built-in types are by default iterable –
 - String
 - Array
 - Map
 - Set
- An object is considered **iterable**, if the object implements a function whose key is **[Symbol.iterator]** and returns an iterator.
- A for...of loop can be used to iterate a collection.


```
<script>
let marks = [10,20,30]
//check iterable using for..of
for(let m of marks){
console.log(m);
}
</script>
```

```
<script>
let marks = [10,20,30]
let iter = marks[Symbol.iterator]();
console.log(iter.next())
console.log(iter.next())
console.log(iter.next())
console.log(iter.next())
</script>
```

```
{value: 10, done: false}  
{value: 20, done: false}  
{value: 30, done: false}  
{value: undefined, done: true}
```

Generator:

- Prior to ES6, functions in JavaScript followed a run-to completion model.
- ES6 introduces functions known as Generator which can stop midway and then continue from where it stopped.
- A generator prefixes the function name with an asterisk * character and contains one or more **yield** statements.
- The **yield** keyword returns an iterator object.

```
function * generator_name()  
{  
yield value1  
...  
yield valueN  
}
```

```
<script>
//define generator function
function * getMarks(){
  console.log("Step 1")
  yield 10
  console.log("Step 2")
  yield 20
  console.log("Step 3")
  yield 30
  console.log("End of function")
}
//return an iterator object
let markIter = getMarks()
```

```
//invoke statements until first yield
console.log(markIter.next())

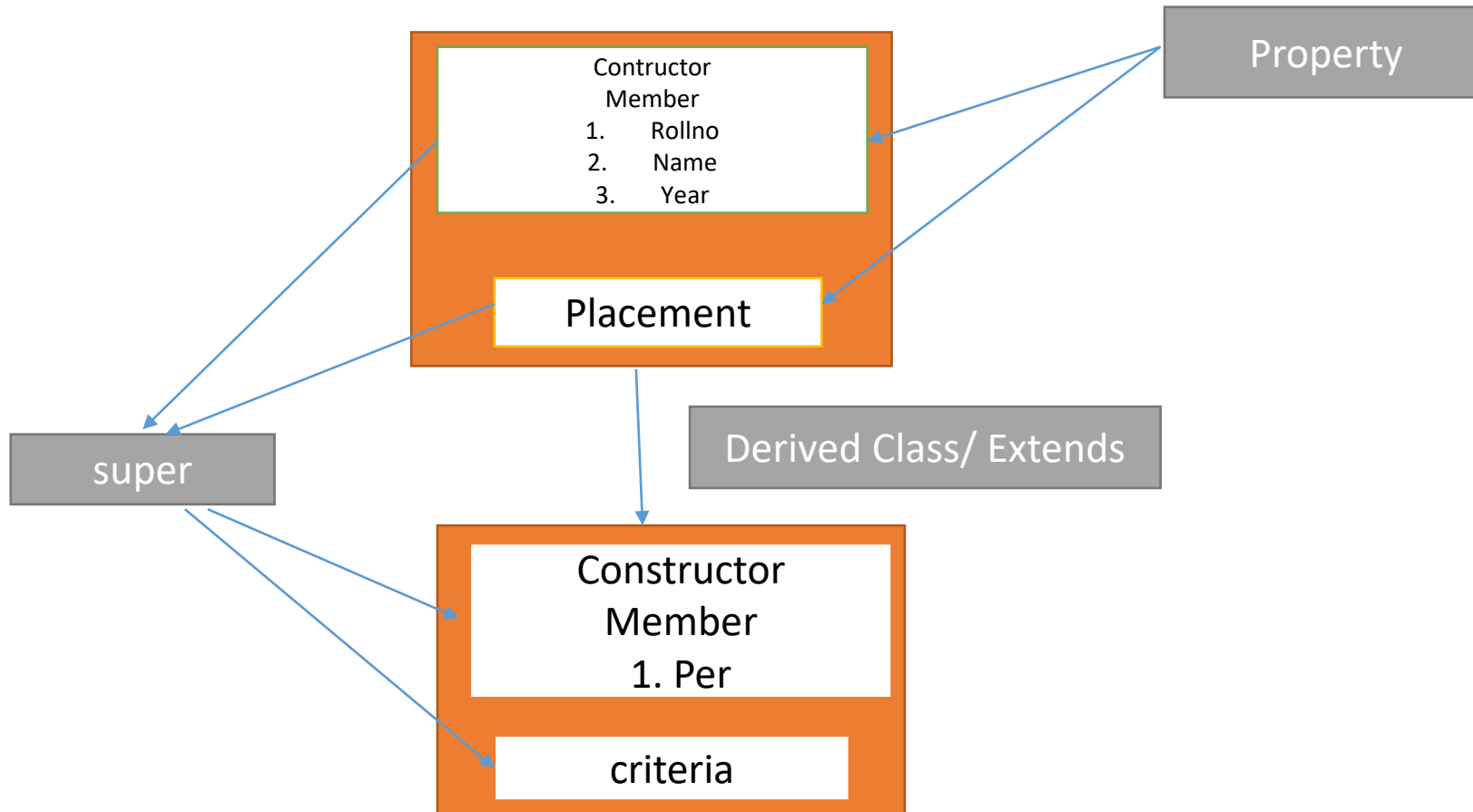
//resume execution after the last yield until
second yield expression
console.log(markIter.next())

//resume execution after last yield until third
yield expression
console.log(markIter.next())

console.log(markIter.next())
// iteration is completed;no value is returned
</script>
```


Classes and Inheritance:

- **Classes:** In OO programming, a class is blueprint for creating objects, providing initial values for state and implementations of behavior
- **Constructor:** The constructor is called on an object after it has been created and is a good place to put initialization code.
- **Property:** is a special sort of class member, intermediate in functionality between a field and a method
- **Object:** Each object is an instance of a particular class or subclass with the class own methods or procedures and data variables.
- **Extends:** For inheritance
- **Super:** to access member or property and method from parent class
- **Instance:** can pass member to derived class or child class
- **Static:** can not pass member to derived class or child class



What is DOM?

DOCUMENT OBJECT MODEL: STRUCTURED
REPRESENTATION OF HTML DOCUMENTS.
ALLOWS JAVASCRIPT TO ACCESS HTML
ELEMENTS AND STYLES TO MANIPULATE
THEM.

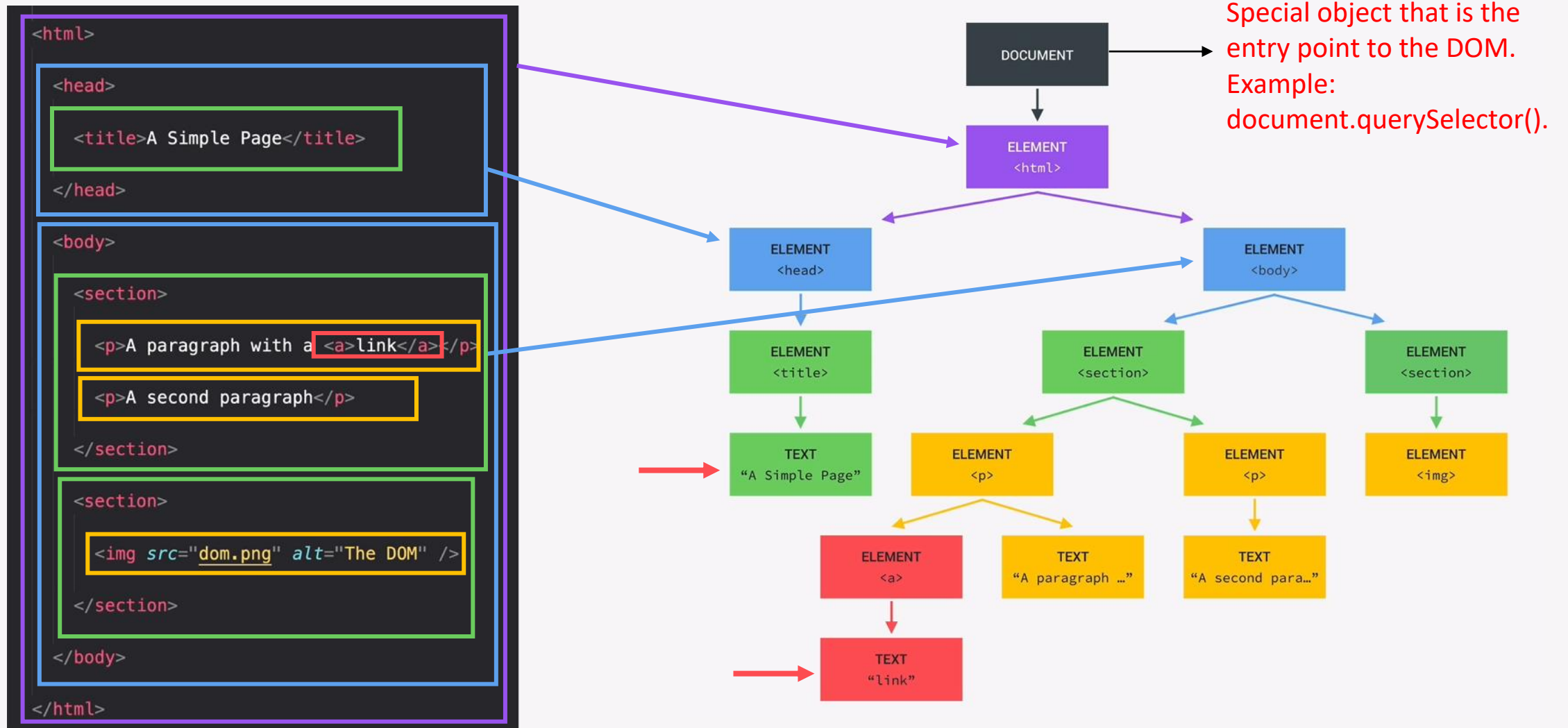


Change text, HTML
attributes, and even
CSS styles.

DOM Manipulation:

- The Document Object Model (DOM) is an application programming interface (API) for manipulating HTML and XML documents.
- The DOM represents a document as a tree of nodes.
- It provides API that allows you to add, remove, and modify parts of the document effectively.
- In DOM tree, the document is the root node.
- The root node has one child which is the <html> element.
- The <html> element is called document element.
- Each document can have only one document element.
- In an HTML document, the document element is the <html>.
- Each markup can be represented by a node in the tree.

THE DOM TREE STRUCTURE



- There are three types of DOM supported by javascript.

1. Legacy DOM:

- This was the model used by early versions of JavaScript.
- This model provides read-only properties such as title, URL, etc.
- It also provides with lastModified information about the document as a whole.
- This model has a lot of methods that can be used to set and get the document property value.

Document Properties of Legacy DOM

- `alinkcolor`: this property defines color of activated links.
- `document.alinkcolor`
- `vlinkcolor`: this property defines color of visited links.
- `document.vlinkcolor`
- `linkcolor`: this property defines color of unvisited links.
- `document.linkcolor`
- `Title`: contents of title tag
- `document.title`
- `Fgcolor`: defines the default text color of the document
- `Document.fgcolor`
- `Bgcolor`: defines background color of the document

W3C DOM:

- With the object model, JavaScript gets all the power it needs to create dynamic HTML:
- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

Properties:

- Body:Contents of the tag.
- E.g. document.body
- Methods:
- **createAttribute(name_of_attr):** Returns a newly-created Attr node with the specified name.
- **createElement(tagname_of_new_ele):** creates and returns a new Element node with a specified tagname.
- **createTextNode(text):** Creates and returns a new Text node that contains the specified text.
- **getElementById(Id):** Returns the value from the document of the element with the mentioned Id.
- **getElementsByName(name):** Returns an array of nodes with the specified name from the document.
- **getElementsByTagName(tagname):** Returns an array of all element nodes in the document that have a specified tagname.

- getElementById, innerHTML
- getElementById: To access elements and attributes id is set
- innerHTML: to access the content of an element

IE4 DOM

- This DOM was introduced in version 4 of Internet Explorer. Later versions expanded and went on to include features from W3C DOM.


```
var element = document.querySelector("< CSS selector >");
```

querySelector() and querySelectorAll()

- The querySelector() function takes an argument, and this argument is a string that represents the CSS selector for the element you wish to find.
- querySelector() returns the first element it finds - even if other elements exist that could get targeted by the selector.
- To return all the elements use querySelectorAll() method.

Promise:

```
let p =new Promise(function(resolve,reject)
```

- **p** is the promise object
- **resolve** is the function that should be called when the promise executes successfully
- **reject** is the function that should be called when the promise encounters an error

Example:

```
let p =new Promise(function(resolve,reject){
let workDone=true; // some time consuming work
if(workDone){
//invoke resolve function passed
resolve('success promise completed')
}
else{
reject('ERROR , work could not be completed')
}
})
```

```

<script>
function add_positivenos_async(n1, n2) {
let p = new Promise(function (resolve, reject) {
{
if (n1 >= 0 && n2 >= 0) {
//do some complex time consuming work
resolve(n1 + n2)
}
else
reject('NOT_Postive_Number_Passed')
})
return p;
}

```

o/p

end

Handling success 30

Handling error NOT_Postive_Number_Passed

```

add_positivenos_async(10, 20)
.then(successHandler) // if promise resolved
.catch(errorHandler); // if promise rejected
add_positivenos_async(-10, -20)
.then(successHandler) // if promise resolved
.catch(errorHandler); // if promise rejected

function errorHandler(err) {
console.log('Handling error', err)
}
function successHandler(result) {
console.log('Handling success', result)
}
console.log('end')
</script>

```

Promises Chaining

- **Promises chaining** can be used when we have a sequence of **asynchronous tasks** to be done one after another.
- Promises are chained when a promise depends on the result of another promise.

- **add_positivenos_async() function** adds two numbers asynchronously and rejects if negative values are passed.
- The result from the current asynchronous function call is passed as parameter to the subsequent function calls.
- each **then()** method has a return statement.

```
<script>
function add_positivenos_async(n1, n2) {
  let p = new Promise(function (resolve, reject) {
    if (n1 >= 0 && n2 >= 0) {
      //do some complex time consuming work
      resolve(n1 + n2)
    }
    else
      reject('NOT_Postive_Number_Passed')
  })
  return p
}
```

```
add_positivenos_async(10,20)
.then(function(result){
console.log("first result",result)
return
add_positivenos_async(result,result
)
}).then(function(result){
console.log("second result",result)
return
add_positivenos_async(result,result
)
}).then(function(result){
console.log("third result",result)
})
console.log('end')
</script>
```

Promise.all()

```
Promise.all(iterable);
```

- This method can be useful for aggregating the results of multiple promises.


```
<script>
function add_positivenos_async(n1, n2) {
let p = new Promise(function (resolve, reject)
{
if (n1 >= 0 && n2 >= 0) {
//do some complex time consuming work
resolve(n1 + n2)
}
else
reject('NOT_Positive_Number_Passed')
})
return p;
}
```

Promises

- Promises are a clean way to implement async programming in JavaScript (ES6 new feature).
- Prior to promises, Callbacks were used to implement async programming

Understanding Callback

- A function may be passed as a parameter to another function.
- This mechanism is termed as a Callback. A Callback would be helpful in events.

```
<script>
function notifyAll(fnSms, fnEmail)
{
  console.log('starting notification process');
  fnSms();
  fnEmail();
}
```

```
notifyAll(function()
{
  console.log("Sms send ..");
}, function()
{
  console.log("email send ..");
});
console.log("End of script");
</script>
```

- The function calls are synchronous.
- It means the UI thread would be waiting to complete the entire notification process.
- Synchronous calls become blocking calls.

Understanding Async Callback

- use the `setTimeout()` method of JavaScript.
- This method is async by default.
- The `setTimeout()` method takes two parameters:
 - A callback function
 - The number of seconds after which the method will be called

```
<script>
function notifyAll(fnSms, fnEmail)
{
  setTimeout(function()
  {
    console.log('starting notification process');
    fnSms();
    fnEmail();
  }, 2000);
}
```