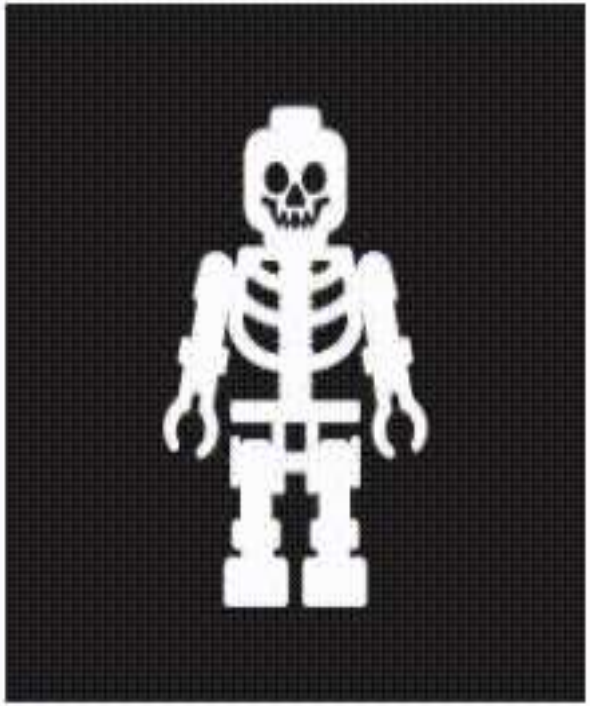
JAVASCRIPT

(JS)

HTML + CSS + JavaScript = DYNAMIC Web Pages

HTML
structure



CSS
presentation/appearance

JavaScript
dynamism/action

Syllabus :

- 2.1 Introduction to Java Script
- 2.2 Data types
- 2.3 Functions
- 2.4 Arrays
- 2.5 Objects
- 2.6 Regular expressions
- 2.7 Prototypal Inheritance
- 2.8 Understanding callbacks
- 2.9 Promise
- 2.10 Closure
- 2.11 XHR request response
- 2.12 Asynchronous Task in JS

2.1 INTRODUCTION

- **SCRIPTING LANGUAGES:**
  - Do not require compilation
  - Directly Interpreted/Executed by another program at Run Time
  - Designed to add interactivity to static HTML pages.
- Types:
  - Client –Side:
    - Executed at client-side by the user's web browser
    - Examples: JavaScript, Action Script, VB Script
  - Server-Side :
    - Executed at Web Server to generate Dynamic web pages.
    - Examples: PHP, Perl, ASP.net

JAVASCRIPT(JS):

- "Client-side Scripting Language"
- Designed to **add Interactivity/ dynamic nature** to the HTML pages
- **Interpreted language** – executed without any compilation
- **embedded directly into HTML Pages.**
- **It is an Object-based Scripting Language.**
- **All Java script statements end with a semicolon**
- **Java script ignores white space**
- **Java script is Case sensitive language**
- **Open and cross-platform.**

- JavaScript was invented by **Brendan Eich** in 1995, and became an **ECMA standard** in 1997.
- **ECMAScript is the official name of the language**.
- ECMAScript versions have been abbreviated to **ES1, ES2, ES3, ES5, and ES6.**
- Since 2016 new versions are named by year (ECMAScript 2016 / 2017 / 2018).

- **JavaScript engines:**
  - A JavaScript engine is a software component (interpreter) that executes JavaScript code.
- **Notable engines:**
  - **V8 from Google** is the most used JavaScript engine.
  - **SpiderMonkey** is developed by Mozilla for use in Firefox
  - **JavaScriptCore** is Apple's engine for its Safari browser.
  - **Chakra** is the engine of the Internet Explorer browser.

- **<script> </script>** - used to insert JS code into a HTML page

Attributes	Values
language	javascript
type	text/javascript
src	URL of the external JS file

- **Syntax:**

```
<script  language="javascript">
        JavaScript code
</script>
```

- **Example:**

- **Advantages/features of JavaScript:**
  - Used to Validate Form input data
    - Examples: Login form – username & password
  - Can add Dynamic text into an HTML page
  - Can react to Events:
    - Displaying message when user clicks on a Button
  - Gives HTML designers a Programming tool :
    - Simple syntax
    - Example: testing whether a number read from user is prime or not, even or odd etc.,
  - Can read and write HTML elements/Tags.
  - Security
    - Cannot read from/write to the local hard drive and the system
  - No need of any special plugins to use JS

JAVA

1. Object Oriented Programming Language
2. Platform Independent
3. both compiled and interpreted
4. used to create server side applications and standalone programming
5. Strongly typed language
6. developed by sun Microsystems
7. programs can be standalone

JAVASCRIPT

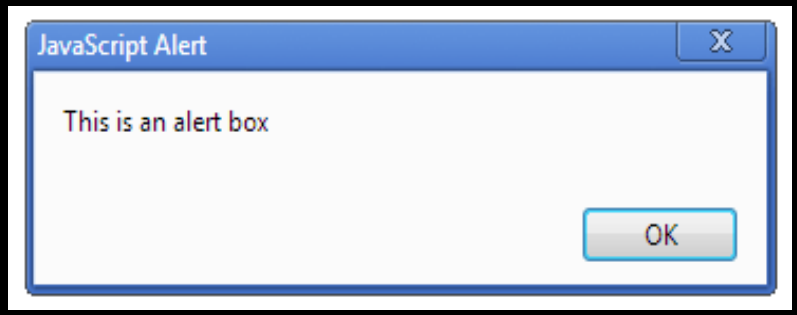
1. Object based Scripting Language
2. Browser Dependant
3. Interpreted at runtime
4. used to make the web pages more interactive
5. not strongly typed(Loosely Typed)
6. developed by Netscape
7. must be placed inside an HTML document to function

JavaScript Output

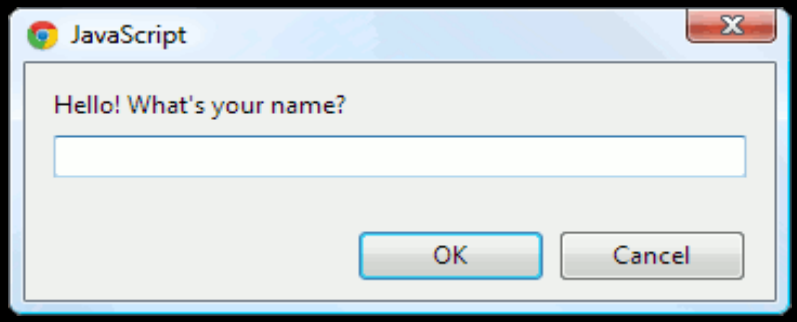
- JavaScript can "display" data in different ways:
  - Writing into the **HTML output** using **document.write()**
  - Writing into an **HTML element**, using **innerHTML**
  - Writing into an **alert box**, using **window.alert()**
  - Writing into the **browser console**, using **console.log()**
- **window.print()** method in the browser to print the content of the current window.
- In order to access HTML element in Javascript – **document.getElementById()**

POP-UP BOXES

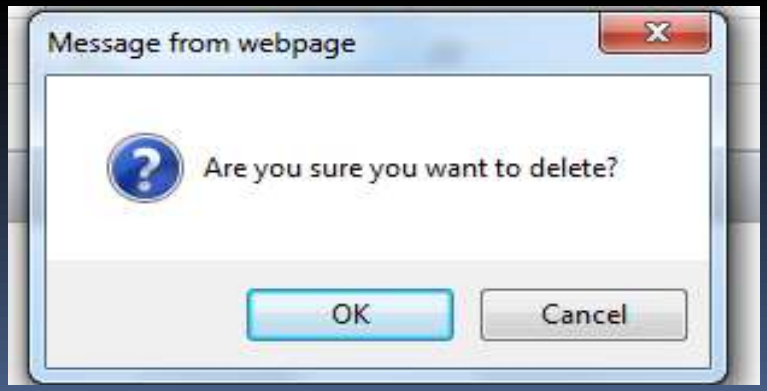
- **Alert box:**
  - Alert("message");



- **Prompt box:**
  - Prompt ("some text");
  - Prompt(" some text", "value");
  - Return type – String/null



- **Confirm box:**
  - Confirm ("some text");
  - Return type - boolean



Variables in JavaScript:

- Variables are **containers for storing data** (values).
- There are 3 ways to declare/create a JavaScript variable:
  - Using **var**
  - Using **let**
  - Using **const**
- **Naming Conventions:**
  - **can contain letters, digits, underscores, and $ signs.**
  - **must begin with a letter, $ or underscore**
  - **can't use spaces in names**
  - **case sensitive**
  - **can't use a reserved word as a variable name**

Types of Scope:

- Scope determines the **accessibility (visibility)** of variables.
- JavaScript has 3 types of scope:
  - **Block scope**
  - **Function scope**
  - **Global scope**

1. Block Scope:

- ES6 introduced two important new JavaScript keywords: **let** and **const**.
- These two keywords provide **Block Scope** in JavaScript.
- Variables declared inside a { } block cannot be accessed from outside the block.

Example:

```
{
        let  x = 3;
}
// x can NOT be used here
```

- **Variables declared with the var keyword can NOT have block scope.**

2. Function Scope:

- Each function creates a new scope.
- **Variables defined inside a function are not accessible (visible) from outside the function.**
- Variables declared with **var, let and const** are quite similar when declared inside a function. They **all have Function Scope**

Example:

```
function myFun () {
  let car = "Volvo";   // Function Scope
}
```

3. Global Scope:

- Variables declared **Globally, outside any function** have **Global Scope**.
- **Global variables can be accessed from anywhere in a JavaScript program.**
- Variables declared with **var, let and const** are quite similar when declared outside a block. They **all have Global Scope**

Example:

```
var x = 2;       // Global scope
function  myfun1()
{
     // x can be accessed here
}
```

- **Local Variables:**
  - Variables **declared within a JavaScript function**, become **LOCAL** to the function.
  - Local variables have **Function Scope**
  - They **can only be accessed from within the function**.
  - Local variables are **created when a function starts, and deleted when the function is completed**.
  - Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

- Global Variables:
  - A variable **declared outside a function**, becomes **GLOBAL**.
  - A global variable **has Global Scope**:
  - **All scripts and functions on a web page can access it.**
- **The Lifetime of JavaScript Variables:**
  - The lifetime of a JavaScript variable **starts when it is declared.**
  - **Function (local) variables are deleted when the function is completed.**
  - **In a web browser, global variables are deleted when you close the browser window (or tab).**

Javascript ‘var’:

- Declare/create a JavaScript variable with the **var** keyword.
- Variables declared with the var keyword **can NOT have block scope**.
- **If you re-declare a JavaScript variable using var, it will not lose its value.**
- Variables defined with var are **hoisted** to the top and **can be initialized at any time**.
  - In JavaScript, **a variable can be used before it has been declared.**
  - **Hoisting - default behavior of moving declarations to the top of the current scope.**

JavaScript ‘let’

- The let keyword was introduced in ES6
- Variables defined with let have **Block Scope**.
- Variables defined with let **cannot be Redeclared.**
- Variables defined with let are **also hoisted to the top of the block, but not initialized**.
- Variables defined with let **must be Declared before use.**
- **Using a let variable before it is declared will result in a ReferenceError.**

JavaScript 'Const'

- The const keyword was introduced in ES6
- Variables defined with const **have Block Scope**.
- const variables **cannot be Redeclared.**
- const variable **cannot be reassigned.**
- **const variables must be assigned a value when they are declared.**
- It **does not define a constant value**. It **defines a constant reference to a value**.
- Use const when you declare:
  - **A new Array**
  - **A new Object**
  - **A new Function**
  - **A new RegExp**

Differences between var and let and const

- **Scope:**
  - **let and const provide Block Scope** in JavaScript.
    - declared inside a { } block cannot be accessed from outside the block
  - Variables declared with the **var keyword can NOT have block scope.**
    - declared inside a { } block can be accessed from outside the block.

- **Redeclaring:**
  - Redeclaring a JavaScript variable with **var is allowed** anywhere in a program.
  - With **let**, **redeclaring a variable in the same block is NOT allowed**
  - const variables **cannot be Redeclared and cannot be reassigned.**

- **Hoisting:**
  - Variables defined with **var** are hoisted to the top and can be initialized at any time.
    - **can use the variable before it is declared.**
  - Variables defined with **let** are also hoisted to the top of the block, but **not initialized**.
    - **Cannot use the variable before it is declared.**
  - Just like let, **const** declarations are hoisted to the top but are not initialized.

2.2 JAVASCRIPT DATA TYPES

- JavaScript is a ***loosely typed*** and ***dynamic*** language.
- JavaScript has **dynamic types**.
  - same variable can be used to hold different data types

PRIMITIVE DATATYPES:

- 7 types:
  - **1. number :**
    - can be written **with, or without decimals** (integer or floating point).
    - can also be **+Infinity, -Infinity, and NaN** (not a number)
    - can only represent numbers less than **$(2^{53} - 1)$** and more than **$-(2^{53} - 1)$**
  - 2. **bigInt**:
    - can represent whole numbers bigger than **$2^{53} - 1$**
    - created by **appending n** to the end of an integer literal or by calling the function **BigInt()**
  - **3. String:** set of characters placed within quotes **' '** or **" "**

4. **Boolean:** true/false

5. **null:** no value exist

6. **undefined:**

- a data type whose variable is not initialized
- variable is **declared** but the **value is not assigned**, then the value of that variable will be undefined.

7. **symbol:**

- Symbols are **immutable** (cannot be changed) and are **unique**
- use the Symbol() function to create a Symbol

NON PRIMITIVE/REFERENCE Types:

- Object
- Array
- **Function**
- RegEx

2.3 JAVASCRIPT FUNCTIONS

- A JavaScript function is a block of code designed to perform a particular task.
- Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).
- **Function declaration:**
  - Defined with the **function** keyword, followed by a **name**, followed by a **list of parameters** to the function, enclosed in parentheses and separated by commas.
  - Syntax:

```
function name(parameter1, parameter2, parameter3) {
  // JavaScript statements that define the function
}
```

- **Supports Reusability:** Define the code once, and use it many times.
- **Function Invocation/call:**
  - When an event occurs (when a user clicks a button)
  - When it is invoked (called) from JavaScript code
  - Automatically (self invoked)
- **Functions can return a value using return statement.**
- When JavaScript reaches a return statement, the function will stop executing. The return value is "returned" back to the "caller".
- JavaScript functions **can be called before they are declared (Function hoisting).**
  - Hoisting applies to variable declarations and to function declarations.

Passing parameters:

- Function **parameters** are the **names** listed in the function definition.
- **Parameter Rules:**
  - do not specify data types for parameters
  - do not perform type checking on the passed arguments.
  - do not check the number of arguments received.
- **Default Parameters:**
  - If a function is called with **missing arguments** (less than declared), the missing values are **set to 'undefined'**.
  - Es6 allows **default parameter values** in the function declaration.
  - **Default function parameters "allow parameters to be initialized with default values if no value or undefined is passed."**

- Rest parameters
- The rest parameter syntax **allows a function to accept an indefinite number of arguments as an array.**
- When **...** is at the end of function parameter, it is the rest parameter. It stores **n number of parameters as an array**.
- **Syntax:**

```
function f(a, b, ...ar)
{
        //code
}
```

The ‘Arguments’ Object

- JavaScript functions have a **built-in object** called the **arguments object**.
- **arguments** is an **Array-like object accessible inside functions** that **contains the values of the arguments passed to that function**.
- Syntax:

```
arguments[i]   // i is the index number
```

- **Anonymous functions:**
  - An anonymous function is a ***function without a name.***
  - Functions stored in variables do not need function names. They are always **invoked (called) using the variable name**.
  - Example:

```
let show = function () {
        console.log('Anonymous function');
};
show();
```

Arrow functions/fat arrows

- ES6 introduced **arrow functions**
- **Also knows as fat arrow functions**
- Arrow functions **allows a short syntax for writing function expressions**.
- Arrow functions are always anonymous and **provides a shorthand for declaring anonymous functions**
- **don't need** the **function keyword**, the **return keyword**, and the **curly brackets**.

```
var f = function(x, y) {
        return x * y;
}
```

Using Arrow functions:

- `const f = (x, y) => x * y;`

- **Syntax:**
  - with no parameter
    - () => expression;
  - with one parameter
    - parameter => expression;
  - with multiple parameters
    - (param1, param2, ….. paramN) => expression;
  - with single statement - explicit return is not required.
  - with multiline statements - explicit return is needed
    - (param1, param2,….,paramN) => {

      statements;

      return expression;

      }
  - Rest parameters and default parameters were supported
    - (param1, param2, …rest) => expression

- Arrow functions are not hoisted. They must be defined before they are used.

2.4 JAVASCRIPT ARRAYS

- **used to store multiple values in a single variable.**
- It is a common practice to **declare arrays** with the **const** keyword.
- **"An array is a special variable, which can hold more than one value at a time".**
- Arrays are a **special type of objects**.
- The **typeof operator** in JavaScript **returns "object"** for arrays.
- **Arrays with named indexes** are called **associative arrays** (or hashes).
- **JavaScript** does not support **arrays with named indexes.**
- In JavaScript, **arrays always use numbered indexes**.

Creating Arrays:

- 1. Syntax:

  const *array_name* = [*item1*, *item2*, ...];

- 2. Syntax:

  const array_name = [];

  array_name[0] = value;

  array_name[1] = value;

  3. Syntax: using new keyword

  const array_name = new Array(item1, item2, item3);

- **Accessing Array Elements:**
  - can access an array element by referring to the **index number**
  - Syntax:

    Array_name[index_number]
- **Loop over an Array:**
  - The forEach() method calls a function once for each element in an array, in order.
  - Syntax:
    - ***array*.forEach(*function(value, index, array), thisValue*)**

Common operations

- **length** - returns the number of array elements.
- **Push()** – Adds a new element to the end of an Array and returns the new array length.
- **Pop()** – Removes and returns the last element from an array.
- **Unshift()** - Add an item to the beginning of an Array and returns the new array length.
- **Shift()** - Removes and returns the first element from an array.
- **indexOf()** - Search the array for an element and returns its position. returns -1 if not found
- **concat()** - creates a new array by merging (concatenating) existing arrays.
- **slice()** - slices out a piece of an array into a new array.
- **sort()** - sorts an array alphabetically

2.5 OBJECTS

- **JavaScript objects** are containers for named values called **properties**.
- The properties of an object define the characteristics of the object.
  - **Example: CUP** -> color, a design, weight, a material it is made of are the properties of a cup.
- **Each property is written as key-value pair (name-value).**
- **Key-value pairs are separated by a comma.**
- The **key** of a property can be a **string**
- The **value** of a property can be any valid JavaScript value (a string, a number, an array, and even a function)

- objects are generally declared using keyword const and were surrounded by curly braces { }.
- Objects can also have methods.
- Methods are actions that can be performed on objects.
  - Example:
    - CAR - > properties: name, model, weight, color
      methods: start(), stop(), brake(), drive()
- An object method is an object property containing a function definition.

Example:

```
const book = {
  name          : "HTML5 Black Book",
  author        :  "kogent",
  price         : 595.99,
  available     : true
};
```

- **Accessing an object:**

  Objects can be accessed in two ways:

  1. **Dot (.) Notation:**
     - **Syntax:** Objectname.Keyname
     - **Example:** Accessing name in book object.
       - **book.name**
  2. **Bracket ([ ]) Notation:**
     - **Syntax:** Objectname["Keyname"]
     - **Example:** Accessing name in book object.
       - **book["name"]**

‘this’ keyword:

- **this keyword** generally **refers to the current object**.
- In an object **method**, this refers to the "**owner**" of the method.
- In a **function**, this refers to the **global object**.
- In an **event**, this refers to the **element that received the event**.

Creating a JavaScript Object

- There are different ways to create new objects:
  1. **Create a single object, using an object literal.**
  2. **Create a single object, with the keyword new.**
  3. **Define an object constructor, and then create objects of the constructed type.**
  4. **Create an object using Object.create().**

1. Using an Object Literal

- Using an object literal, you both define and create an object in one statement.
- An object literal is **a list of name:value pairs inside curly braces {}.**
- **Syntax:**
  - const object = { property1:value1, property2:value2..... propertyN:valueN };

2. Using the JavaScript Keyword 'new'

- Syntax:
  - const objectname = **new Object();**

3. Using ‘object constructor’

- Sometimes we need a "**blueprint**" for creating many objects of the same "type".
- The way to create an "object type", is to use an **object constructor function**.
- a constructor function is **used to create multiple similar objects.**
- The **name of a constructor function starts with a capital letter**
- To create an object from a constructor function or to make a call to constructor function, use the **new** keyword.

4. using Object.create()

- The **Object.create()** method **creates a new object, using an existing object as the prototype** of the newly created object.
- **returns a new object** with the specified prototype object and properties.

- **Looping an Object:**
  - Object properties can be accessed through for-in loop.
  - ***Syntax:***

```
for (var in object) {
        code block to be executed
}
```

- **Modifying object values:**
  - Modification means adding new value or replacing new value.
  - Values of an objects can be modified in two ways:

1. Dot (.) Notation:

   Syntax: Objectname.Keyname="value"

2. Bracket ([ ]) Notation:

   Syntax: Objectname["Keyname"]="value"

- **Delete object values:**
  - By using ***delete*** keyword properties of JSON object can be deleted.
  - **Syntax:**
    - delete objectname.keyname;

2.6 REGULAR EXPRESSIONS

- A regular expression is a sequence of characters that forms a search pattern.
- The search pattern can be used for text search and text replace operations.
- A regular expression can be a single character, or a more complicated pattern.
- In JavaScript, regular expressions are **objects**.
- JavaScript provides the built-in **RegExp** type that allows you to work with regular expressions effectively.
- Can be **used to validate form fields**

Creating a regular expression

- 2 ways:
  - **Using a regular expression literal.**
  - **Using the constructor of the RegExp object.**

1. Using a regular expression literal

- which consists of a pattern enclosed between slashes
- Syntax:
  - */pattern/modifiers*;

Modifier	Description
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

Regular Expression Patterns

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character NOT between the brackets
[0-9]	Find any character between the brackets (any digit)
[^0-9]	Find any character NOT between the brackets (any non-digit)
(x|y)	Find any of the alternatives specified

Metacharacters

- characters with a special meaning.

\w	Find a word character
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character

Quantifiers

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero or more occurrences of n
n?	Matches any string that contains zero or one occurrences of n
n{X}	Matches any string that contains a sequence of X n's
n{X,Y}	Matches any string that contains a sequence of X to Y n's
n{X,}	Matches any string that contains a sequence of at least X n's
n$	Matches any string with n at the end of it
^n	Matches any string with n at the beginning of it
?=n	Matches any string that is followed by a specific string n
?!n	Matches any string that is not followed by a specific string n

- **String methods**:
  - **search()** - uses an expression to search for a match, and returns the position of the match.
  - **replace()** - method returns a modified string where the pattern is replaced.

2. Using the constructor of the RegExp object

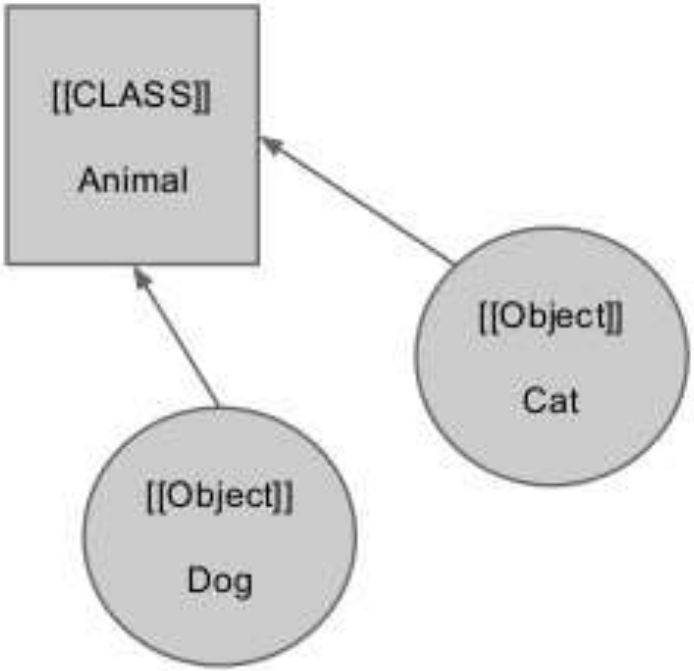
- In JavaScript, the RegExp object is a **regular expression object** with predefined properties and methods.
- **Syntax:**
  - **new RegExp(pattern);**

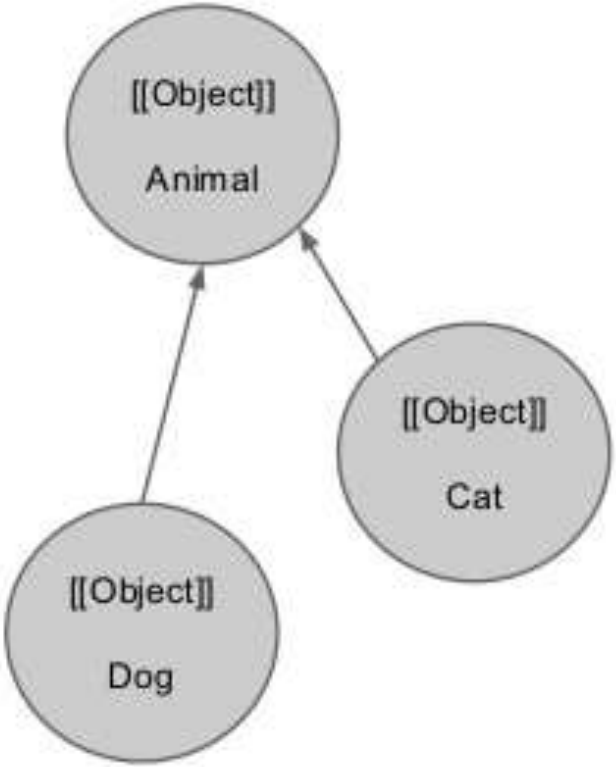
- **Methods:**
  - **test() -** Tests for a match in a string. Returns true or false.
  - **exec() -** Tests for a match in a string. Returns the first match

2.7 PROTOTYPAL INHERITANCE

- Javascript is a **prototyped- based** programming language.
- In prototype-based languages there are **no explicit classes**.
- JavaScript only has one construct: **objects**.
- **Prototype-based programming:** It is a style of object-oriented programming in which inheritance is performed via a process of reusing existing objects that serve as prototypes.
- JavaScript doesn't use **classical inheritance**. Instead, it uses **prototypal inheritance**.
- **Objects inherit directly from other objects** through a **prototype** property.

Classical vs Prototypal Inheritance

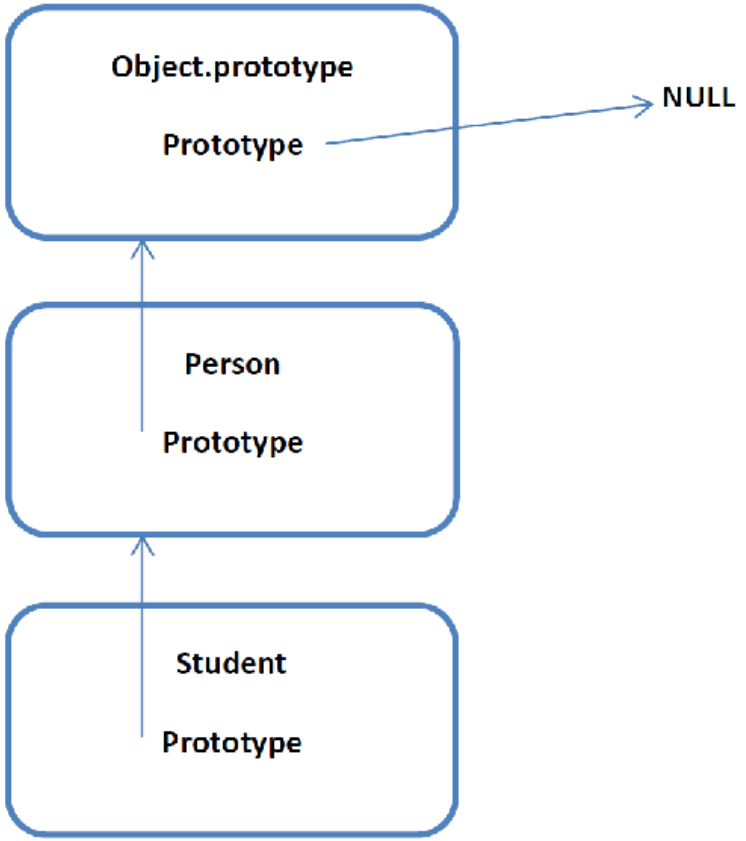




- **Prototypal Inheritance**: "It is a method by which an object can inherit the properties and methods of another object."
- In prototypal inheritance, an object "inherits" properties from another object via the prototype linkage.
- Every object with its methods and properties and contains an **internal and hidden property** known as **[[Prototype]]** -which is either null or references another object.
- **all objects in JavaScript are instances of "Object" i.e by default prototype property links to "Object"**

- One way to set and access **[[prototype]]** property is by using **__proto__** property.
  - **Syntax:**

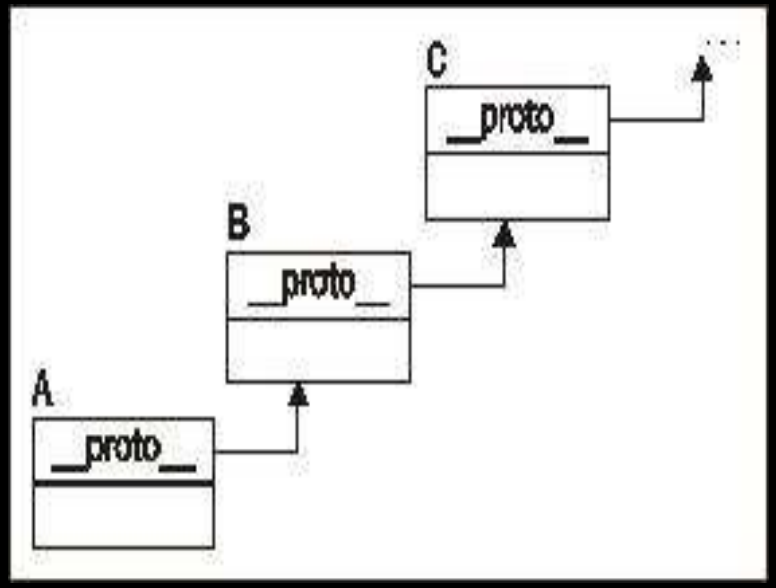
    ChildObject.__proto__ = ParentObject



- ES5 provided a standard way to work with prototypal inheritance by using the **Object.create()** method.
- The **Object.create()** method **creates a new object and uses an existing object as a prototype of the new object**:
  - **Syntax:**

```
Object.create(proto);
```

Prototype chaining



Example:

- an object **A** contains a number of properties and hidden **__proto__ property**, which points to another object, **B**.
- **B**'s **__proto__ property** points to **C**.
- This chain ends with the Object object, which is the highest-level parent, and every object inherits from it.
- When trying to access a property of an object A, the property will be searched in that object A then on the prototype of the object i.e B, the prototype of the prototype i.e C, and so on until either a property with a matching name is found or the end of the prototype chain is reached i.e null.

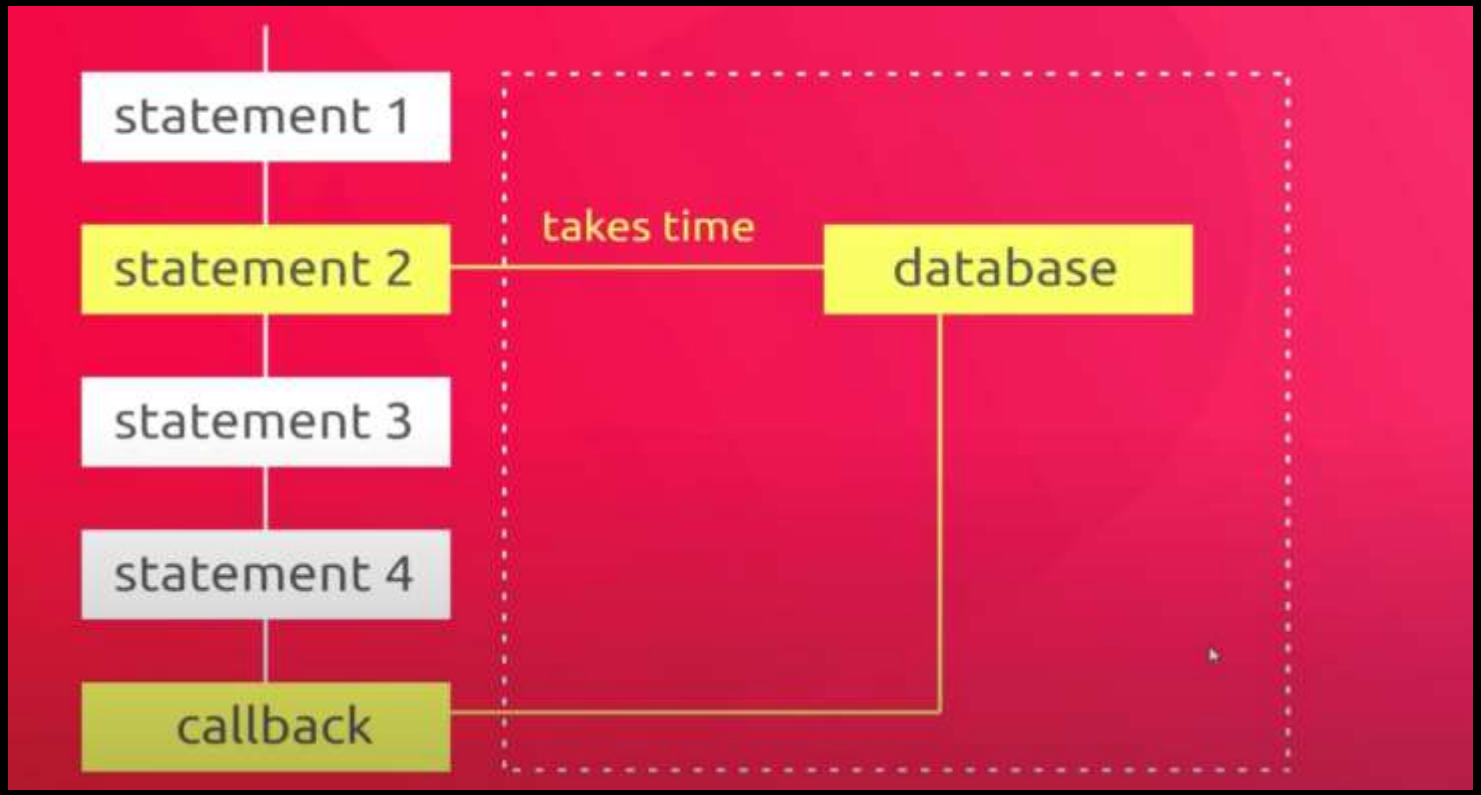
2.8 UNDERSTANDING CALLBACKS

- **First-class/higher-order functions** - **can be assigned to variables and passed around to other functions.**
- If we want to execute a function right after the return of some other function, then callbacks can be used.
- ***Definition:*** "**A callback is a function that is passed as an argument to another function**", which is then invoked inside the outer function to complete some kind of routine or action.
- Callbacks are most often used to develop an **asynchronous JavaScript code**.
- Also used in arrays, timer functions, promises, event handlers, etc.,

Asynchronous Javascript – using callbacks

- JavaScript is **synchronous by default** and is **single threaded.**
- Each statement in your code is executed one after the other, in sync.
- This means **each statement has to wait for the previous one to finish its executing**.
- File operations, database operations, fetching data from server, etc., often take more time to complete.
- Synchronous JS nature results in **blocking the main thread** until the current statement/operation is completed.
- **Asynchronous JavaScript** -> **non-blocking code, statements can be executed in parallel.**
- **Functions running in parallel with other functions are called asynchronous functions.**

- Asynchronous JavaScript code uses **callbacks**, **promises**, **and async/await**
- Callback functions used in Asynchronous JavaScript are called as **Asynchronous callbacks**.
- **Asynchronous callback** function executes when an asynchronous operation completes.

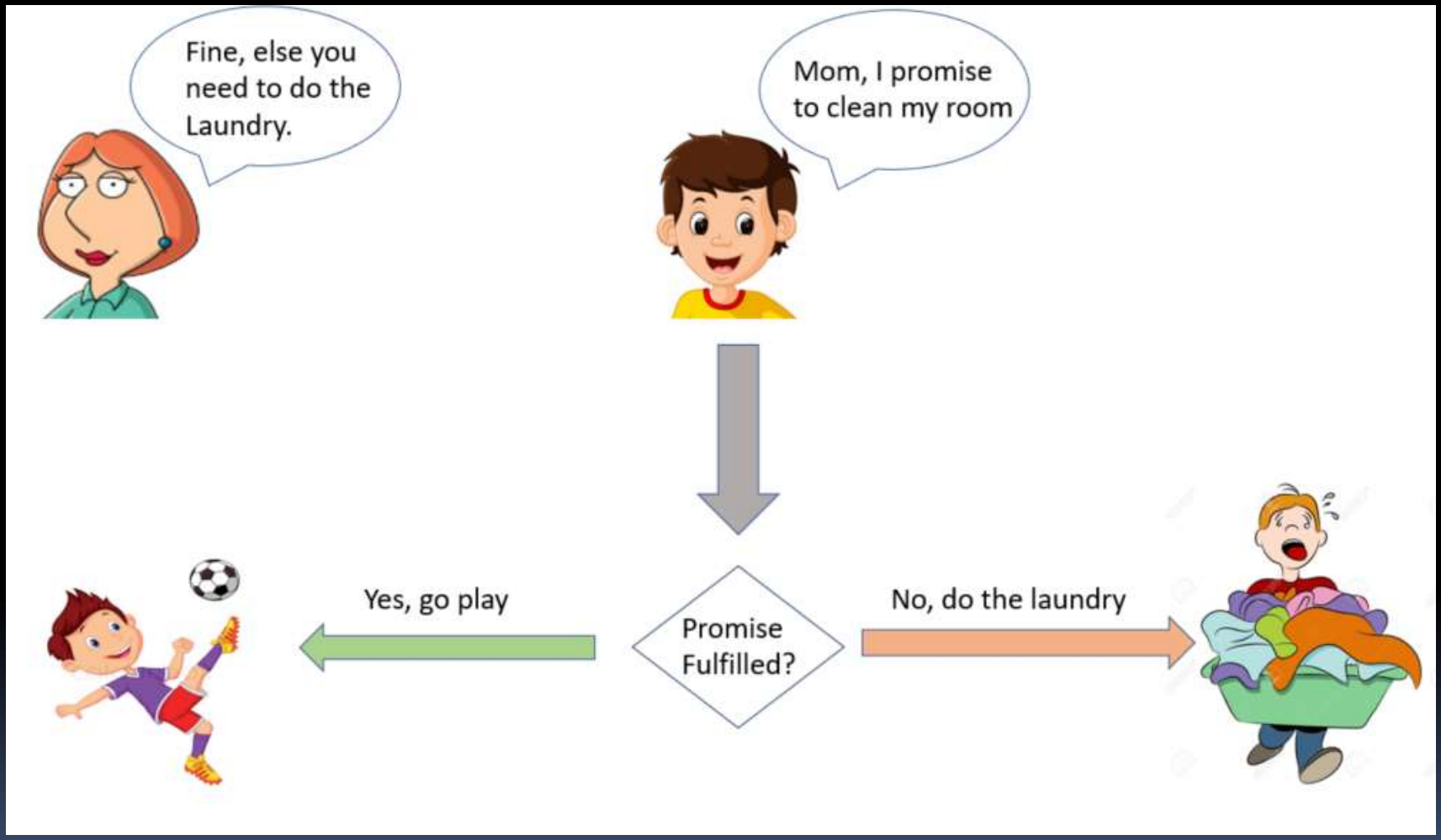


Statement 3 & 4 will be executed without waiting for Statement 2 to complete its operation. Once statement 2 has completed its operation callback will be executed

2.9 PROMISES

- When dealing with **multiple asynchronous operations** using **mulitple callbacks leads to unmanageable code**.
- **Promises** are **used to handle asynchronous operations** in JavaScript. They are easy to manage than callbacks
- **Promises can handle multiple asynchronous operations easily** and provide better error handling than callbacks.
- A promise is commonly defined as a "**proxy for a value not necessarily known when the promise is created**." - **may produce a value some time in the future** .
- A **promise** is an object that **represents the eventual result of an asynchronous operation**.

Example:



- **A promise consists of:**
  - **"Producing code" is code that can take some time**
  - **"Consuming code" is code that must wait for the result**
- A Promise is a **JavaScript object that links producing code and consuming code**.
- A JavaScript Promise object **contains both the producing code and calls to the consuming code**.
- A Promise is in one of these states:
  - ***pending***: initial state, neither fulfilled nor rejected.
  - ***fulfilled*: meaning that the operation was completed successfully.**
  - ***rejected*: meaning that the operation failed.**

- The Promise object supports **two properties**: **state** and **result**.
- cannot access the Promise properties **state** and **result**.

STATE	RESULT
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

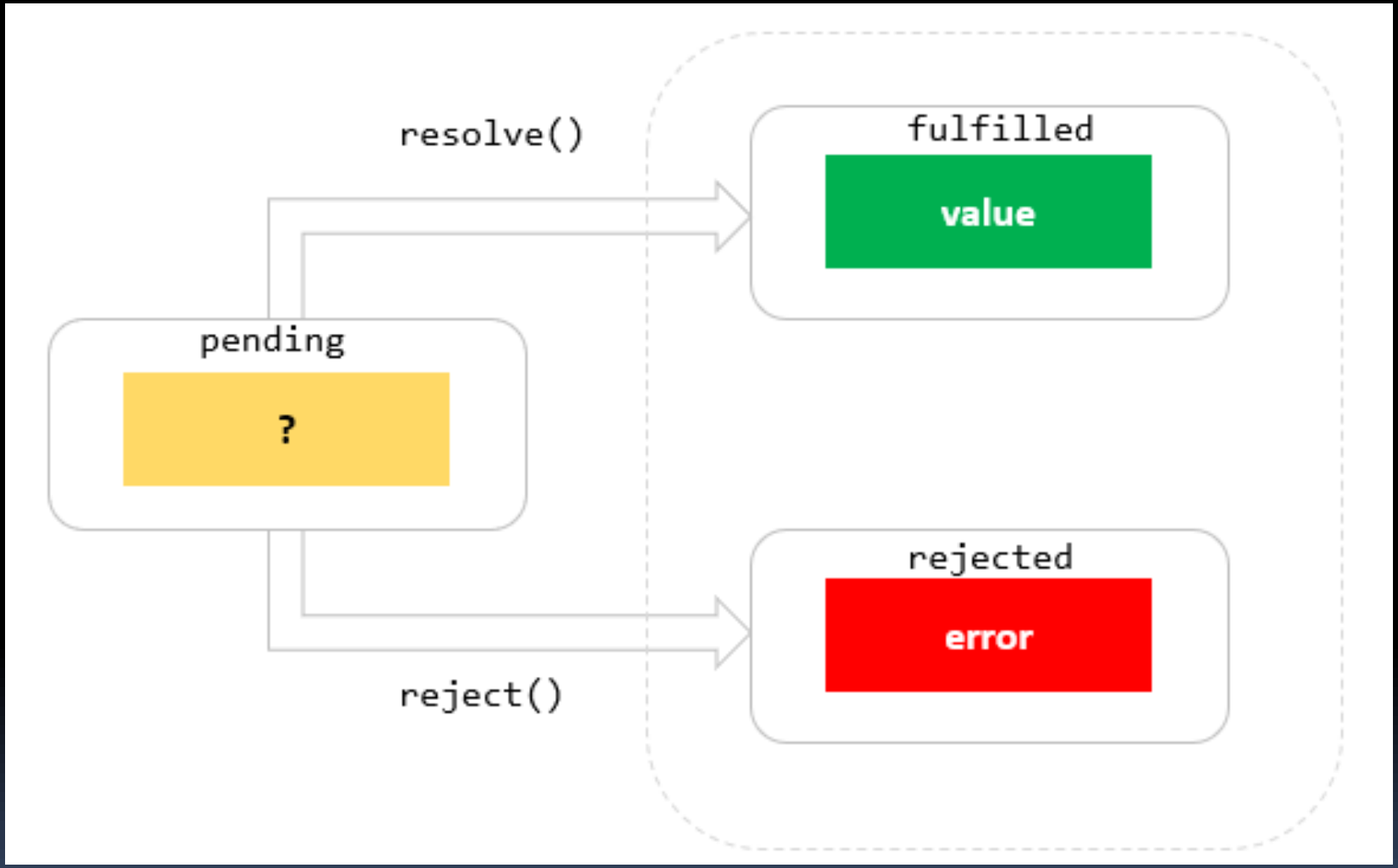
- **A pending promise can either be *fulfilled* with a value or *rejected* with a reason** (error).
- When either of these options happens, **the associated handlers (consuming code) queued up by a promise's *'then'* method are called**.
- **then()** -> **allows you to associate handlers with an asynchronous action's eventual success value or failure reason**.

Creating a promise

- **using Promise constructor:**

```
var myPromise = new Promise(function(resolve, reject)
{
// "Producing Code" (May take some time)

  resolve(value); // when successful
  reject(error);  // when error
});
```



Using a promise:

```
// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

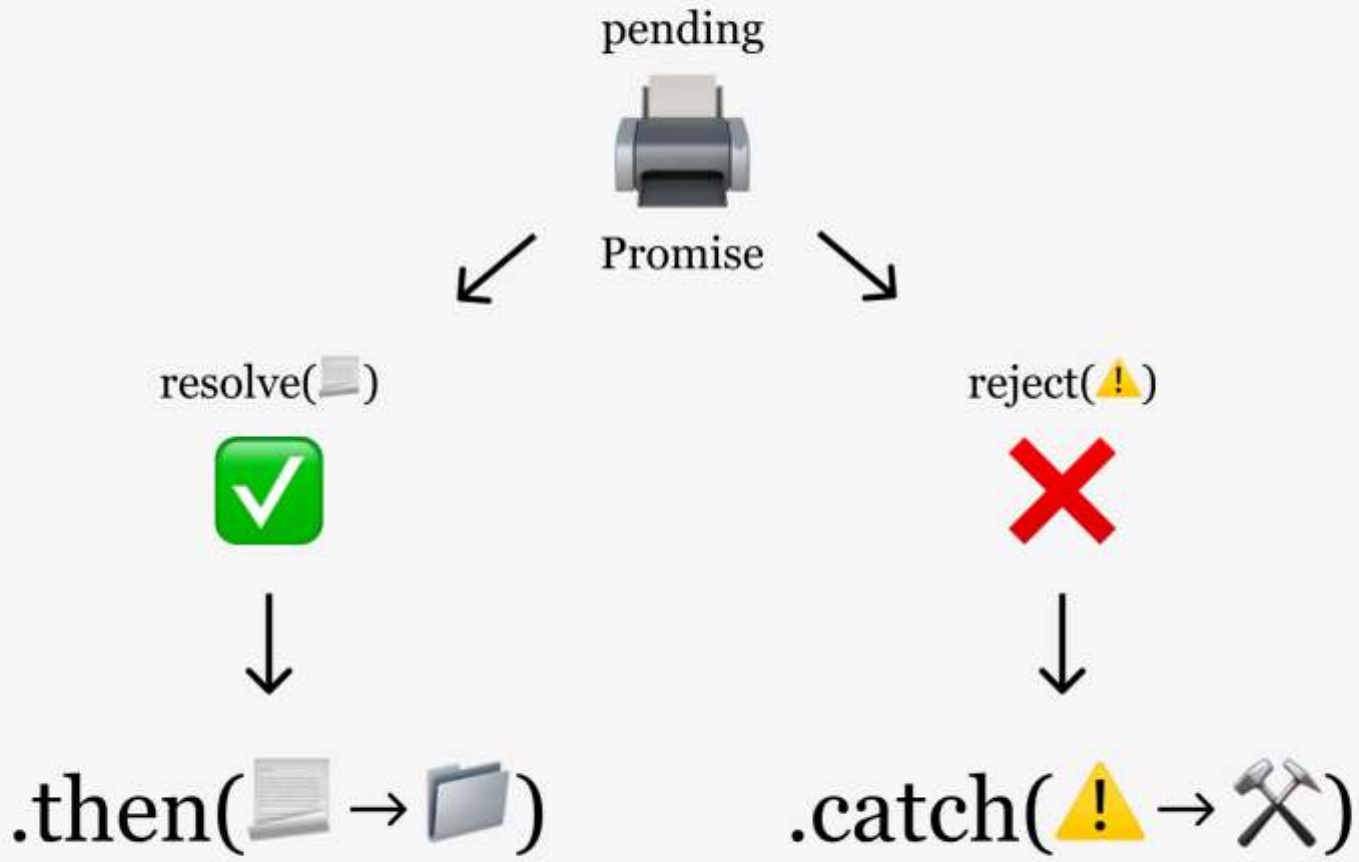
Promise.then() is invoked when a promise is either resolved or rejected and takes two callback functions as arguments:

- **First function is executed if promise is resolved and a result is received.**
- **Second function is executed if promise is rejected and an error is received or can handle error using catch()**
- Both are optional, so you can add a callback for success or failure only.

Catch():

- *catch()* is **invoked when a promise is either rejected or some error has occurred in execution**.
- *catch()* method takes **one function as parameter** which is used to handle errors or promise rejections.

```
.catch(function(error)
{
    //handle error
})
```



- ***chaining promises*** - asynchronous operations happening one after another, used when each next operation depends on the value from the previous asynchronous operation.

2.10 CLOSURES

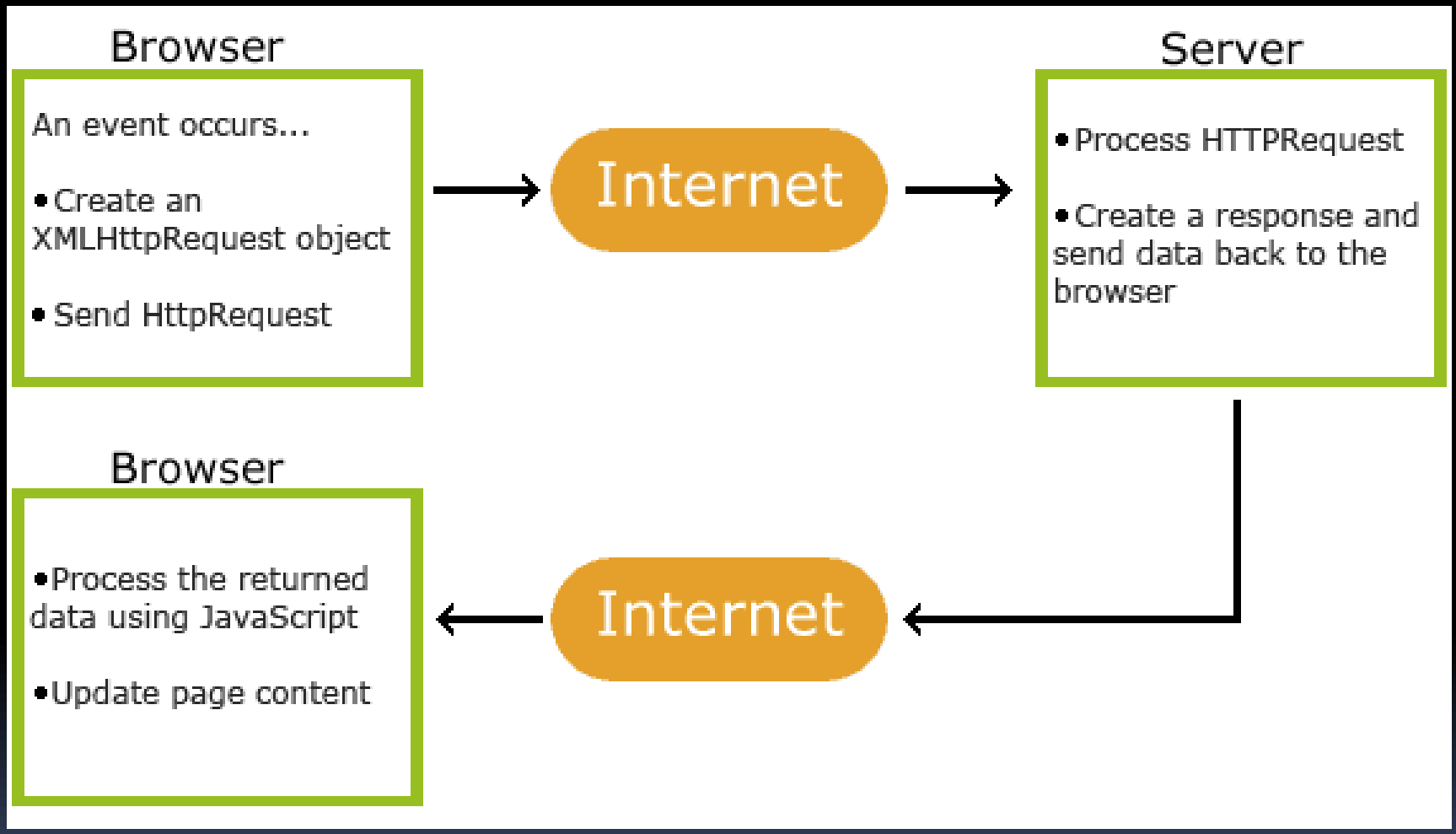
- Functions in JavaScript form **closures.**
- In JavaScript, **closures are created every time a function is created**, at function creation time.
- **Nested functions**:
  - **A function can have one or more inner functions -> Function declared inside another function.**
  - **Inner/nested function can access variables and parameters of an outer function.**
  - outer function cannot access variables defined inside inner functions.
  - **Lexical scope is the ability for a function scope to access variables from the parent scope.**

- A *closure* is the **combination of a function** and the **lexical environment** within which that function was declared.
- This **environment consists of any variables** that were in-scope at the time the closure was created.
- Closure gives you **access to an outer function's scope from an inner function**.
- A closure is a **function having access to the parent scope, even after the parent function has closed**.
- *Closures* ***'remembers' the environment in which it was created***.

2.11 XMLHttpRequest (XHR)

- **XMLHttpRequest** (**XHR**) is an **object whose methods transfer data between a web browser and a web server**.
- The XMLHttpRequest object **allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes**.
- XHR object is a core of **AJAX**.
- This **helps you to retrieve data from a URL without having to do a full page refresh**.
- This means that it is **possible to update parts of a web page, without reloading the whole page**.
- This object is **provided by the browser's JavaScript environment**.
- All modern browsers (Chrome, Firefox, Edge (and IE7+), Safari, Opera) have a built-in XMLHttpRequest object.

- The XMLHttpRequest object is **a developers dream**, because you can:
  - **Update a web page without reloading the page**
  - **Request data from a server - after the page has loaded**
  - **Receive data from a server - after the page has loaded**
  - **Send data to a server - in the background**



Creating XMLHttpRequest object

- The XMLHttpRequest object can be used to exchange data with a web server behind the scenes.
- **Syntax for creating an XMLHttpRequest object:**
  - *variable* = new XMLHttpRequest();
- **Example**
  - var xhttp = new XMLHttpRequest();

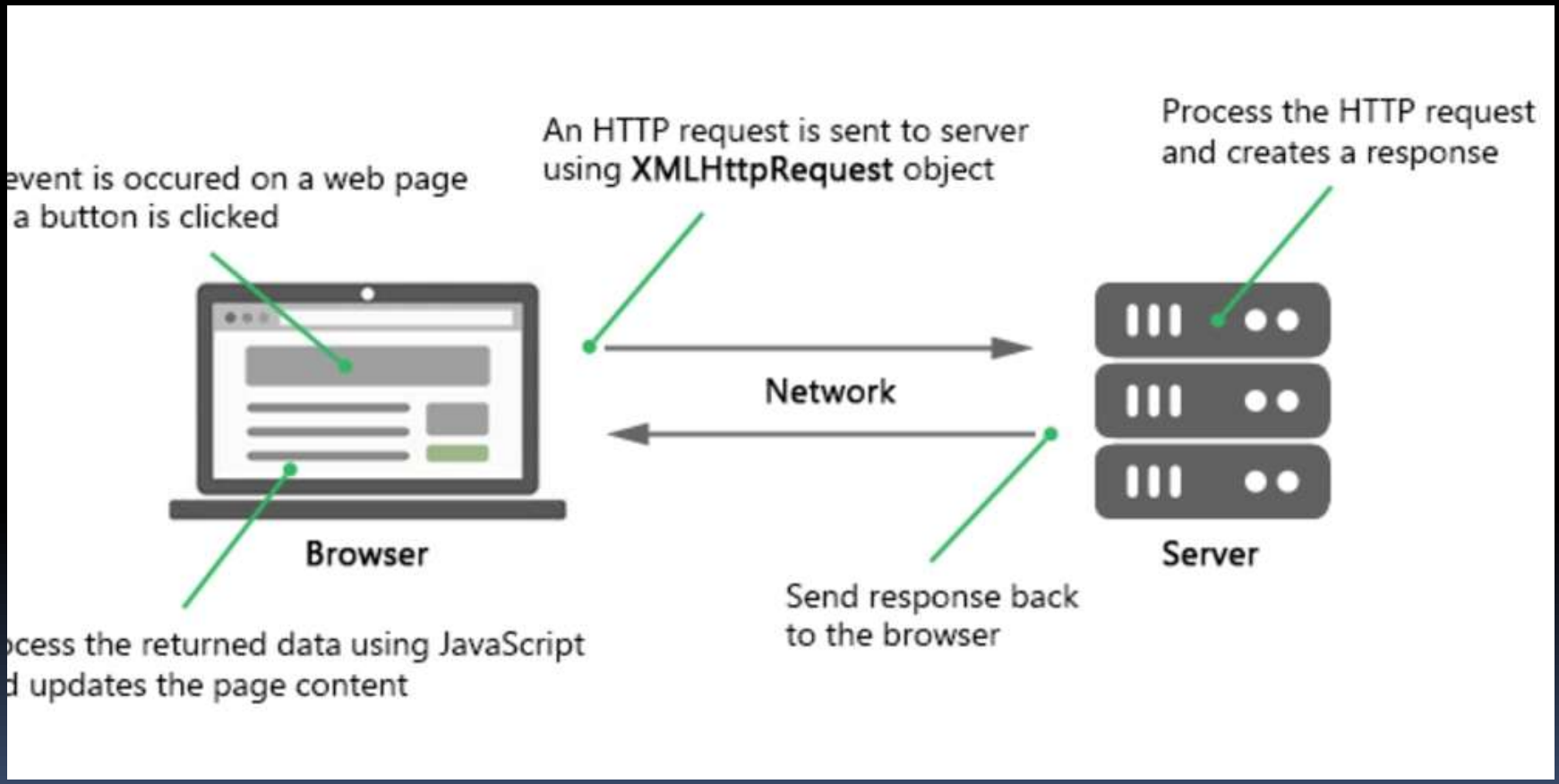
XMLHttpRequest object Properties:

Property	Description
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request **4: request finished and response is ready**
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number of a request **200: "OK"** 403: "Forbidden" 404: "Not Found"
statusText	Returns the status-text (e.g. "OK" or "Not Found")

XMLHttpRequest object Methods:

Method	Description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
open(method, url, async)	• Specifies the request. • XMLHttpRequest object must be initialized through the open. • This method must be invoked prior to the actual sending of a request **method:** the request type GET or POST **url:** the file location **async:** true (asynchronous) or false (synchronous)
send()	Sends the request to the server Used for GET requests
send(string)	Sends the request to the server. Used for POST requests
abort()	Cancels the current request

Example:



2.12 Asynchronous Tasks in JS

- JavaScript is **synchronous by default** and is **single threaded.**
- Each statement in your code is executed one after the other, in sync.
- This means **each statement has to wait for the previous one to finish its executing**.
- File operations, database operations, fetching data from server, etc., often take more time to complete.
- Synchronous JS nature results in **blocking the main thread** until the current statement/operation is completed.
- **Asynchronous JavaScript** -> **non-blocking code, statements can be executed in parallel.**
- **Functions running in parallel with other functions are called asynchronous functions.**

- Asynchronous JavaScript code uses **callbacks**, **promises**, **and async/await**
- Many Web API features now use **asynchronous code** to access or fetch some kind of resource from an external device, such as fetching a file from the network, accessing a database and returning data from it

async/await:

- **async and await make promises easier to write** and are used to makes our code Asynchronous.
- **An async function is a "function declared with the async keyword".**
- Async functions will **always return a** promise. Other values are wrapped in a resolved promise automatically.
- **async** makes a function **return a Promise**.
- Syntax:

```
async function name([param[, param[, ...param]]])
{
    statements
}
```

- Async functions can contain **zero or more await** expressions.
- **await makes a function wait for a Promise**
- The await keyword **can only be used inside an async function**.
- **await** can be put in front of any async promise-based function to **pause your code on that line until the promise fulfills**, then return the resulting value.
- **Syntax:**
  - let value = **await** promise;

EVENT HANDLING

- JavaScript is an "Event Driven System"
- EVENT:
  - any change that the user makes to the state of the browser
  - An HTML event can be something the browser does, or something a user does.
- Types:
  - Window Events:
    - A page loads or unloads
    - Focus is being moved to or away from a window or frame
    - After a period of time has elapsed
  - User Events:
    - occur when the user interacts with elements in the page using mouse or a keyboard.
    - button was clicked, Form is submitted, mouse is moved

EVENT HANDLERS:

- Event handlers are "JavaScript functions" which you associate with an HTML element as part of its definition in the HTML source code.
- Syntax: <element attributes eventAttribute="handler">

Attribute	Description
Onblur	The input focus is moved from the object
Onchange	The value of a field in a form has been changes by the user by entering or deleting data
Onclick	Invoked when the user clicked on the object.
Ondblclick	Invoked when the user clicked twice on the object.
Onfocus	Input focus is given to an element
Onkeydown	Invoked when a key was pressed over an element.
Onkeypress	Invoked when a key was pressed over an element then released.
Onkeyup	Invoked when a key was released over an element.
Onload	When a page is loaded by the browser

Onmousedown	The cursor moved over the object and mouse/pointing device was pressed down.
Onmousemove	The cursor moved while hovering over an object.
Onmouseout	The cursor moved off the object
onmouseover	The cursor moved over the object (i.e. user hovers the mouse over the object).
Onmouseup	The mouse/pointing device was released after being pressed down.
Onmove	A window is moved, maximized or restored either by the user or by the script
Onresize	A window is resized by the user or by the script
onmousewheel	Invoked when the mouse wheel is being rotated.
Onreset	When a form is reset
Onselect	Invoked when some or all of the contents of an object is selected. For example, the user selected some text within a text field.
Onsubmit	User submitted a form.
Onunload	User leaves the Page