Sri Lanka Institute of Information Technology

APPLICATION FRAMEWORKS

JAVASCRIPT, AND NOSQL

LECTURE 03

Faculty of Computing
Department of Software Engineering
Module Code: SE3040

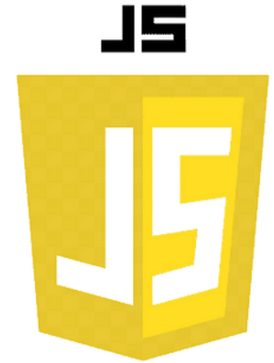
Agenda

1. JavaScript
2. NoSQL

JAVASCRIPT

- Introduction
- Classes, objects
- How ‘this’ acts
- Function closure
- Callbacks and promises

01. JAVASCRIPT



Code is executed line-by-line without a separate compilation step.

- JavaScript is an interpreted programming language.
- It programs run using a single thread. Though there are ways to create new threads, it is considered as Single threaded language.
- It is considered asynchronous because it follows a Non-Blocking I/O (NIO) model.

allow the program to continue executing other tasks while waiting for a long-running operation to complete.

- JavaScript is dynamically typed.

simply assign a value, and the type is inferred automatically. eg - let number = 10

- JavaScript support OOP as well as functional programming (Multi-paradigm).

objects can inherit properties and methods from other objects rather than requiring formal classes.

Functions can be assigned to variables, passed as arguments, and returned from other functions.

- JavaScript has an eventing system which manages it's asynchronous operations.

constructor function - use "new" keyword

"class" keyword

object literals

I. CLASSES AND OBJECTS

- A constructor function is used with the ‘new’ key keyword when creating an object.

Before ES6, JavaScript didn't have the class keyword, but we could create objects using constructor functions

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
```

```
const john = new Person("John", 25);
```

- When function is used with ‘new’ keyword that function acts as a Class.
- Recently JavaScript introduced ‘class’ keyword, but it is not yet adopted by all JavaScript engines (ES6 Classes)

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

- Another way of creating an object is using object literals (‘{}’).

```
const person = {
    name: "Alice",
    age: 28,
};

person.greet();
```

- JavaScript supports static methods and variables.

```
class MathUtil {
    static add(a, b) {
        return a + b;
    }
}

console.log(MathUtil.add(5, 3));
```

Static methods and properties belong to the class itself, not to instances.

Feature	Static	Instance (Non-Static)
Belongs to	The **class**	Each **object**
Access	`ClassName.member`	`object.member`
Memory	One copy (shared)	Separate copy per object
Use Case	Counters, utilities	Object-specific data/actions

II. “THIS” IN JAVASCRIPT

Unlikely other languages in JavaScript ‘this’ keyword acts differently.

- Inside an object ‘this’ refers to object itself.

```
const person = {
    name: "Alice",
    greet: function () {
        console.log(`Hello, my name is ${this.name}.`);
    }
};
```

- In global context ‘this’ refers to global object (in browser it is the window object). This behavior will get changed in strict mode.

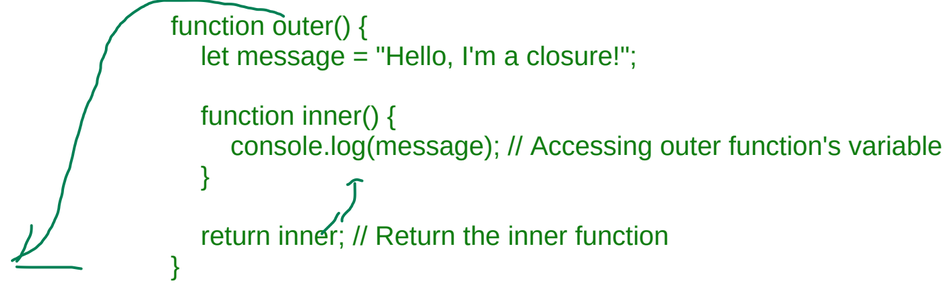
```
console.log(this); // open window
```

- If a function which is using ‘this’ keyword is being passed to another object then ‘this’ will refers to that object, but not to the original object where function was declared at first place.

```
const obj1 = {
    name: "Object 1",
    print: function () {
        console.log(this.name);
    }
};

const obj2 = {
    name: "Object 2"
};

obj2.print = obj1.print; // Assign obj1's method to obj2
obj2.print(); // Output: Object 2
```

- This behavior is very noticeable in callback and closures.

III. CLOSURE



- JavaScript closure is a function which returns another function. In other words, a closure gives you access to an outer function's scope from an inner function.
- It is used to encapsulate variables into a function and restrict access to it from the outside.
- Closure help create private variables that can only be accessed through specific methods.

IV. CALLBACK AND PROMISES

Asynchronous (async) programming means:
Long tasks (like API calls, file reading) run in the background.
other code continues running without waiting. When the background task finishes, its result is handled.

- JavaScript uses asynchronous programming, and it is single threaded.
- Callbacks and Promises are two ways to handle asynchronous executi

call back hell (pyramid of dom)

CALLBACK:

- A callback is a function passed as an argument to another function and is executed later, usually after an asynchronous operation completes.

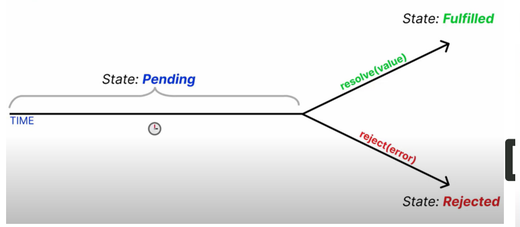
callback can be use few task can not use multiple task

- Nested callbacks passed into sequence of async tasks is referred to a ‘callback hell’.

```
// Nested callbacks (Callback Hell)
step1(() => {
  step2(() => {
    step3(() => {
      console.log("All steps completed!");
    });
  });
});
```

The code becomes deeply nested, making it hard to read and maintain.

```
function greet(name, callback) {
  console.log("Hello, " + name);
  callback();
}

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("Alice", sayGoodbye);
// Output:
// Hello, Alice
// Goodbye!
```



promisess can be define 2 ways -

Method 1 - Using the Promise constructor:

```
const myPromise = new Promise((resolve, reject) => {
  // Your asynchronous operation code here
  // If the operation is successful, call resolve(value)
  // If the operation fails, call reject(reason)
});
```

Method 2 - Defining a function that returns a Promise:

```
function myAsyncOperation() {
  return new Promise((resolve, reject) => {
    // Your asynchronous operation code here
    // If the operation is successful, call resolve(value)
    // If the operation fails, call reject(reason)
  });
}
```

Example of a Promise:

```
// Create a new Promise that resolves after a certain time
function myPromise() {
  return new Promise(function(resolve, reject) {
    setTimeout(() => {
      const randomNumber = Math.random();
      if (randomNumber < 0.5) {
        resolve("Success!"); // Resolve the Promise
      } else {
        reject("Error!"); // Reject the Promise
      }
    }, 2000);
  });
}
```

.then() method for Values

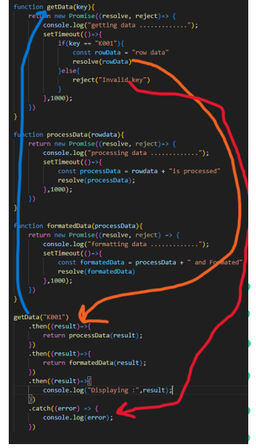
```
myPromise()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
    // Outputs "Error!" if the Promise was rejected
  });
```

Example of a Promise:

```
// Create a new Promise that resolves after a certain time
function myPromise() {
  return new Promise(function(resolve, reject) {
    setTimeout(() => {
      const randomNumber = Math.random();
      if (randomNumber < 0.5) {
        resolve("Success!"); // Resolve the Promise
      } else {
        reject("Error!"); // Reject the Promise
      }
    }, 2000);
  });
}
```

.catch() method for Errors

```
myPromise()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
    // Outputs "Error!" if the Promise was rejected
  });
```



PROMISES:

Promises make the code more readable by avoiding nested callbacks.

- Promise is an object that represents the eventual completion (or failure) of an asynchronous operation. Promise have properties to deal with async operations synchronously.
- Promise object was introduced to solve callback hell problem.
- Promise object has a set of properties, methods and mechanism of chaining to handle complex async tasks nicely.

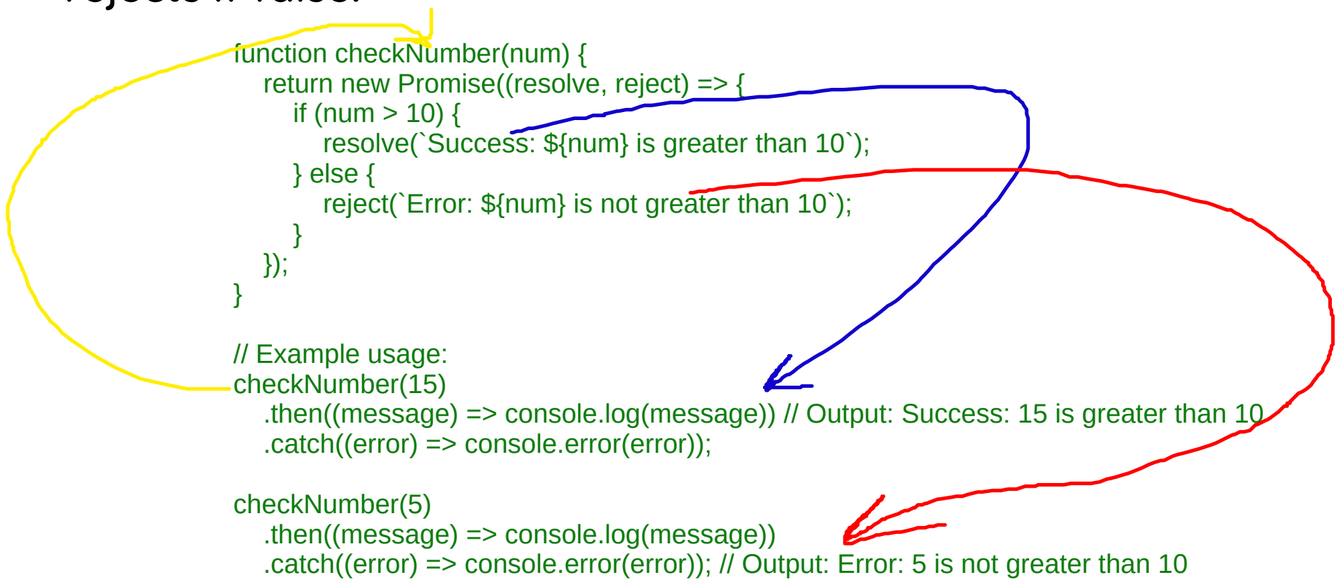
ACTIVITY

Objective:

Learn to work with JavaScript Promises to handle asynchronous operations.

Tasks:

1. Write a function that returns a Promise which resolves if a condition (e.g., a number comparison) is true, and rejects if false.

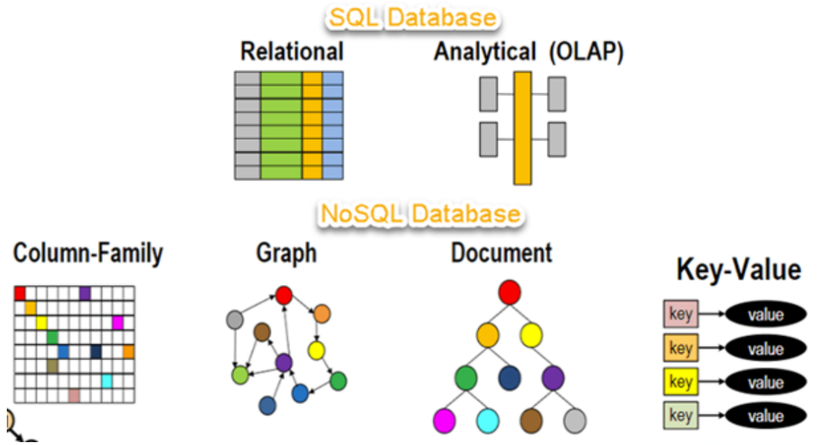


```
function checkNumber(num) {
    return new Promise((resolve, reject) => {
        if (num > 10) {
            resolve(`Success: ${num} is greater than 10`);
        } else {
            reject(`Error: ${num} is not greater than 10`);
        }
    });
}

// Example usage:
checkNumber(15)
    .then((message) => console.log(message)) // Output: Success: 15 is greater than 10
    .catch((error) => console.error(error));

checkNumber(5)
    .then((message) => console.log(message))
    .catch((error) => console.error(error)); // Output: Error: 5 is not greater than 10
```

NOSQL

Not Only SQL

- What?
- Why not SQL?
- Strengths and weaknesses
- MongoDB



WHAT?

- Database system
- Non relational (not using tables), Schema free.
- **Distributed**: Data spread across multiple machines
- **Horizontally scalable:** adding more nodes to a database cluster
- **Easy replication**: copying data across multiple servers
- **Eventually consistent**: prefer availability over strict consistency
- Open source (mostly).
- Lack support to full ACID (mostly)

WHY?

difficulty of handling different ways of organizing and representing data

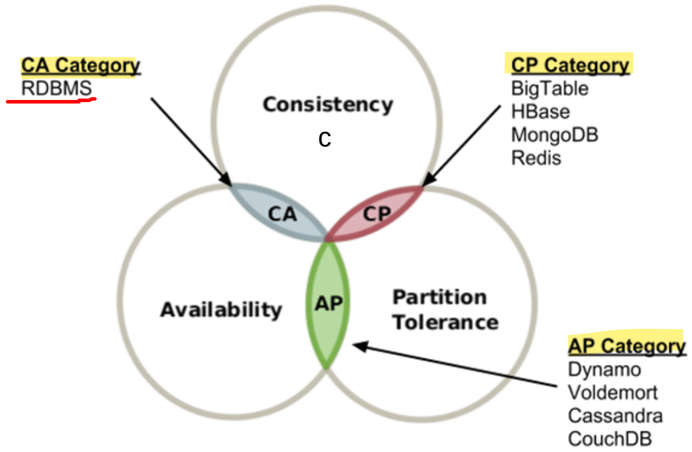
- Remove the burden of data structures mismatch between application in-memory (objects) and relational databases (table).
- Integrate databases using services. (ElasticSearch etc.)
- Relational databases not designed to run efficiently on clusters.
- Aggregate oriented databases (store related data together), are easier to manage inside clusters.
- But inter aggregate relationships are harder to manage.

CAP THEOREM

a distributed database system can only guarantee two out of the following three properties at a time:

It explains the trade-offs in distributed databases. It states that a distributed system can only guarantee two out of the three properties

- **Consistency** - All nodes see the same data at the same time.
- **Availability** - Every request gets a response (even if some nodes fail)
- **Partition tolerance** - The system continues working even if network failures occur



TYPES of NoSQL

- Key-Value - Stores data as key value pairs.
  - Ex: Redis, Riak, Memcached.
- Document - Stores data as documents (JSON, BSON, XML) in maps or collections.
  - Ex: MongoDB
- Column Family - Store data in column families as rows that have many columns associated with.
  - Ex: Cassandra
- Graph - Store entities(nodes) and relationships(edges) between them and represent it in a graph.
  - Ex: Neo4j

TYPES of NoSQL - Summary

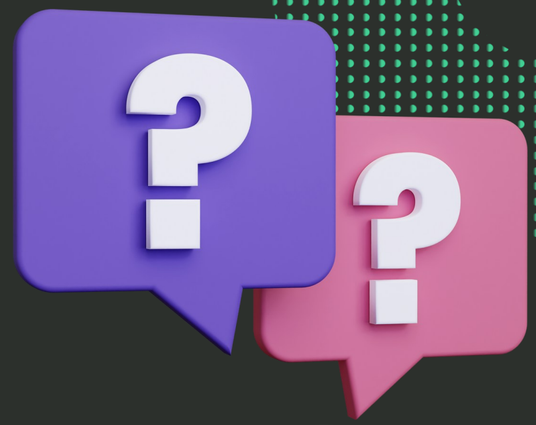
Feature	Document-Oriented	Key-Value Store	Column-Family Store	Graph Database
Data Structure	JSON, BSON Documents	Key-Value Pairs	Column-Family	Nodes & Relationships
Schema Flexibility	✅ Schema-free	✅ Schema-free	✅ Schema-free	✅ Schema-free
Best For	Complex, semi-structured data	Fast lookups, caching	Big data, analytics	Relationship-heavy data
Scalability	✅ High (Sharding & Replication)	✅ Very High	✅ Very High	❌ Limited
Querying	Queryable (like SQL)	Key-based (GET/SET)	Optimized for column queries	Graph queries
Joins Supported?	❌ No joins	❌ No joins	❌ No joins	✅ Supports joins
Example Databases	MongoDB, CouchDB, Firebase	Redis, DynamoDB, Riak	Cassandra, HBase, ScyllaDB	Neo4j, ArangoDB, JanusGraph

MongoDB

- NoSQL document database.
- Strong query capabilities with aggregations using JavaScript.
- Use SpiderMonkey JavaScript engine.
- High availability with replica sets.
- Reads and writes on primary by default.
- Eventually consistent on secondary instances.
- In built file storage called Grid File System.

MongoDB queries

- Insert
- Find
- Update
- Remove



THAT'S ALL FOLKS !

ANY QUESTIONS ?