CYDEO

JavaScript Programming Day03

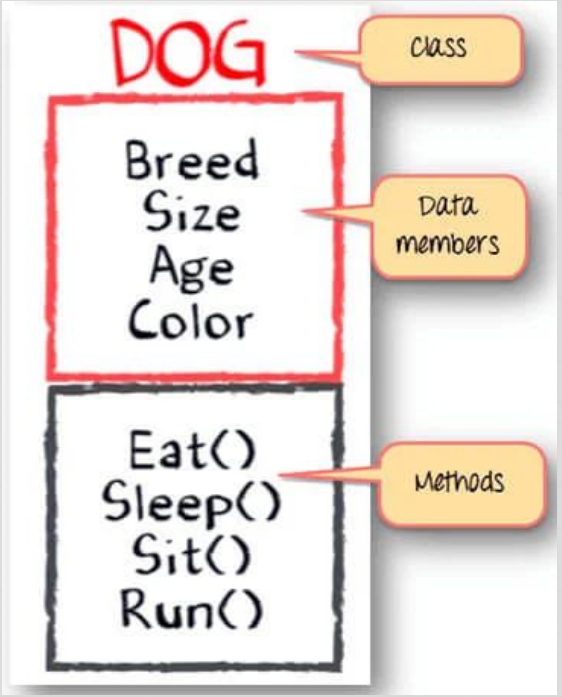
Content

- Class & Objects
- Inheritance
- Exceptions/Errors
- Promises
- Async & Await
- Introduction to Playwright

What is A Class?

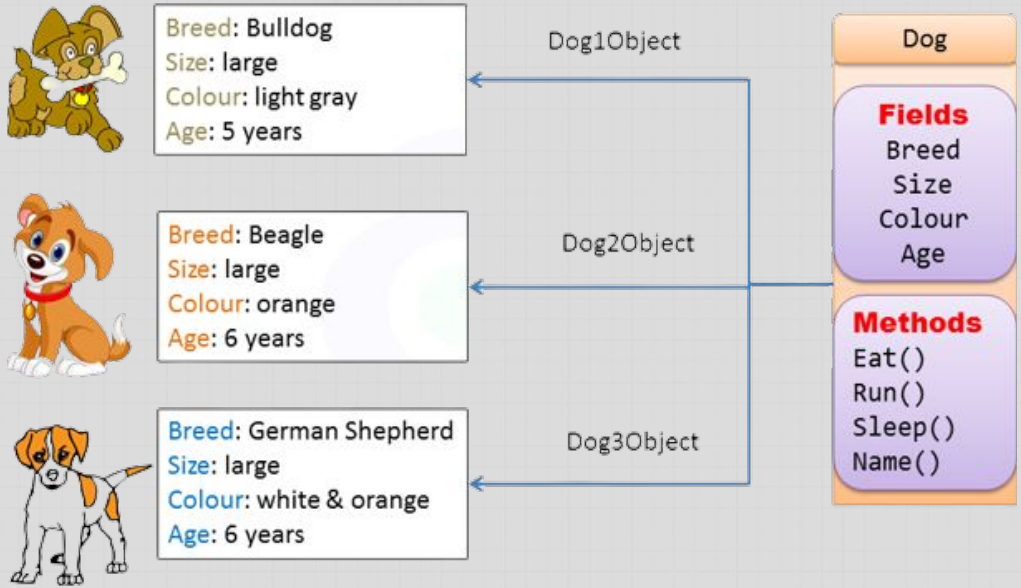
- Where objects came from
- A blueprint or set of instructions to build

  a specific type of Object
- No memory allocated for a class



What is An Object?

- An instance of a class
- Multiple objects can be created from a class
- Each object has its own memory
- The data stored in an object are called fields



Writing A Custom Class

Class Name	Dog
Fields (Attributes)	name breed size age color ...
Methods (Actions)	eat() drink() play() ...

keyword

Class Name

```
class Dog {

  constructor(name, breed, age, color) {
    this.name = name;
    this.breed = breed;
    this.age = age;
    this.color = color;
  }

  eat() {
    console.log(`${this.name} is eating.`);
  }

  drink() {
    console.log(`${this.name} is drinking.`);
  }

  play() {
    console.log(`${this.name} is playing.`);
  }

}
```

The Constructor Method

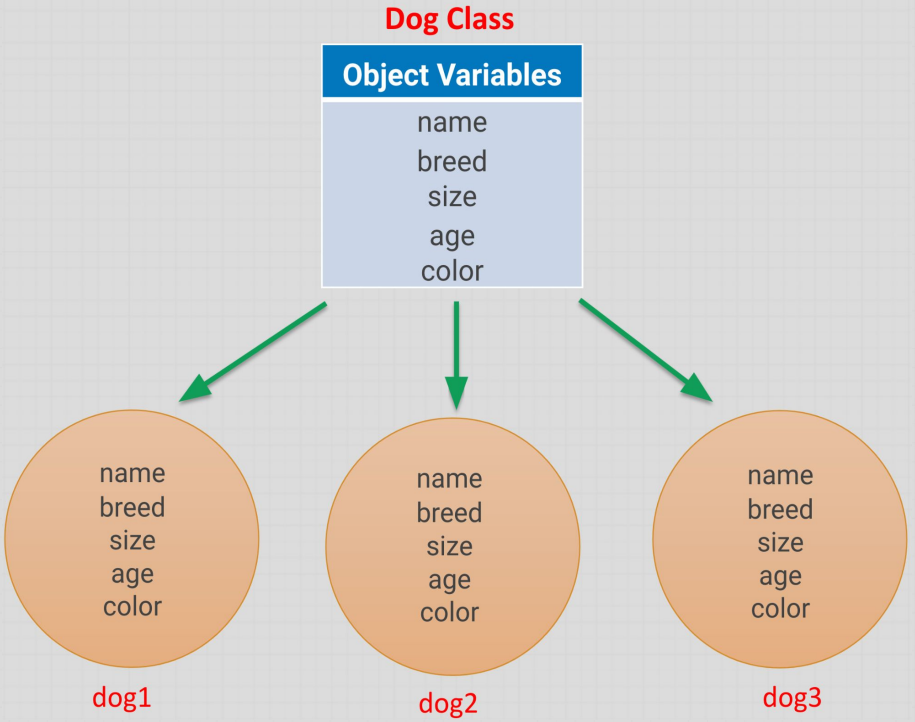
- Built-in constructor method used for defining & initializing the attributes
- Belongs to the object and each object has its own memory
- Gets executed when an object is created from the class

```
class Dog {

  constructor(name, breed, age, color) {
    this.name = name;
    this.breed = breed;
    this.age = age;
    this.color = color;
  }

}
```

Object Variables

- Belongs to the object, each object has a different copy of the instance variable
- Used by the objects to store their data members

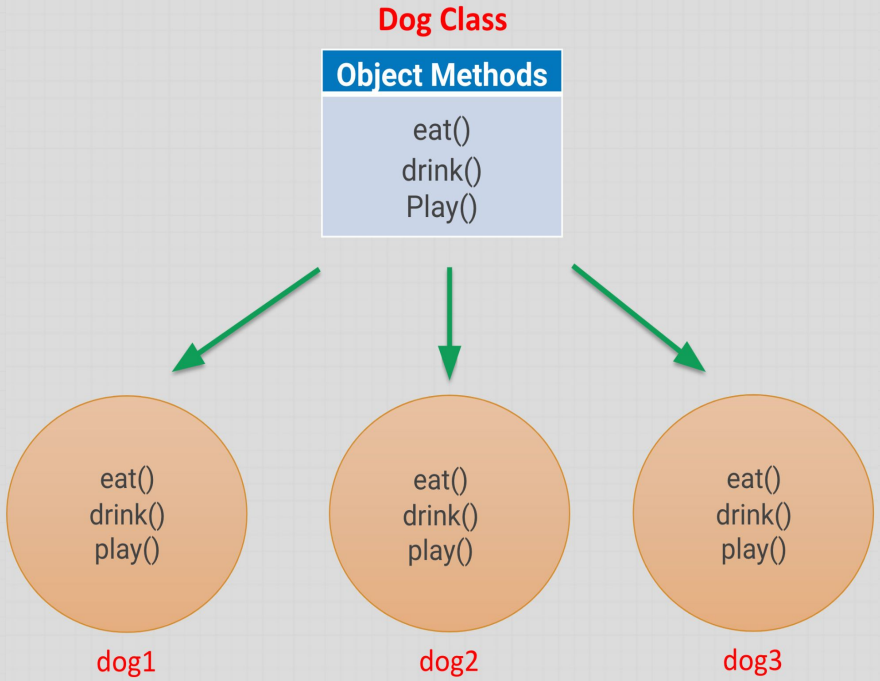
```
constructor(name, breed, age, color) {
  this.name = name;
  this.breed = breed;
  this.age = age;
  this.color = color;
}
```



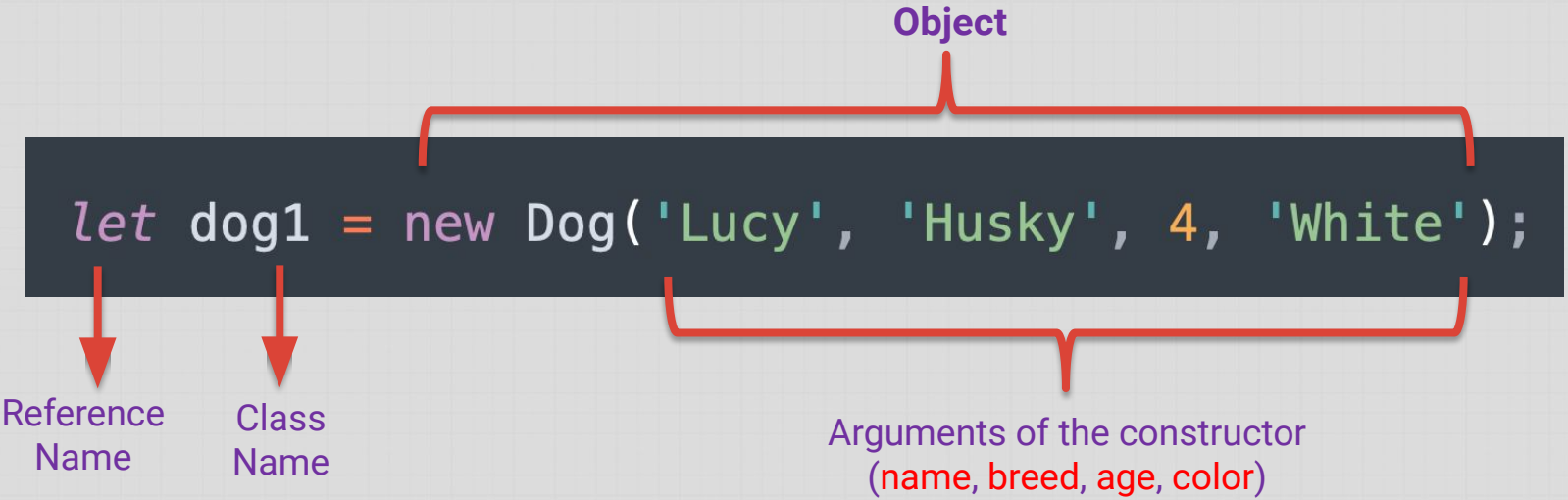
Object Methods

- Objects can share the methods created within the class
- Methods can be called through the object once it's instantiated

```
eat() {
  console.log(`${this.name} is eating.`);
}

drink() {
  console.log(`${this.name} is drinking.`);
}

play() {
  console.log(`${this.name} is playing.`);
}
```



Creating an Object



Accessing an object's data and methods

- An Object's members refer to its data fields and methods. After the object is created its data can be accessed and its methods can be invoked using the dot operator (.)

```
let dog1 = new Dog('Lucy', 'Husky', 4, 'White');

console.log(dog1.name); // Lucy
console.log(dog1.breed); // Husky
console.log(dog1.color); // White

dog1.eat(); // Output: Lucy is eating.
dog1.drink(); // Output: Lucy is drinking.
dog1.play(); // Output: Lucy is playing.
```

Class vs Object

Class	Object
Class is a collection of similar objects	Object is an instance of a class
Class is conceptual (is a template)	Object is real
No memory is allocated for a class	Each object has its own memory
Class can exist without any objects	Objects can not exist without a class

Inheritance

Inheritance

- Used for creating Is A relationship among the classes
- Allows one class to inherit the variables and methods from another class

Child Inherits qualities from parent

Inheritance

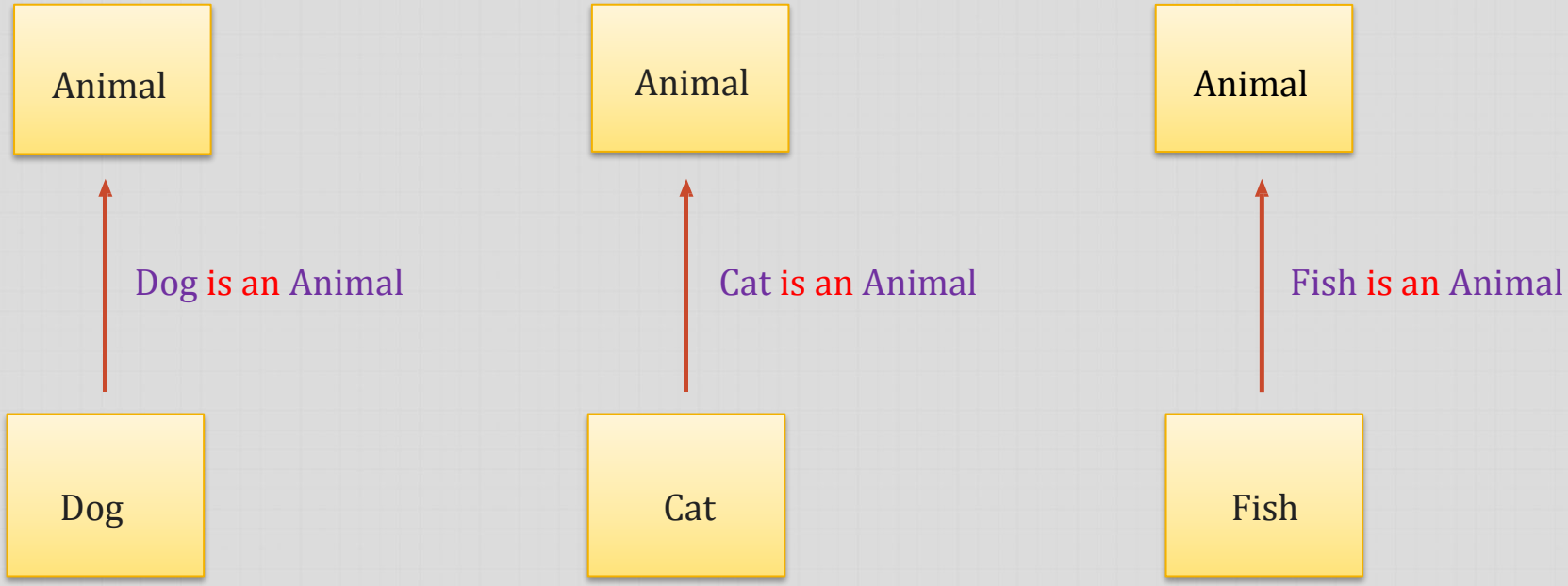
name
breed
size
weight
eat()
move()

bark()

meow()
scratch()

swim()

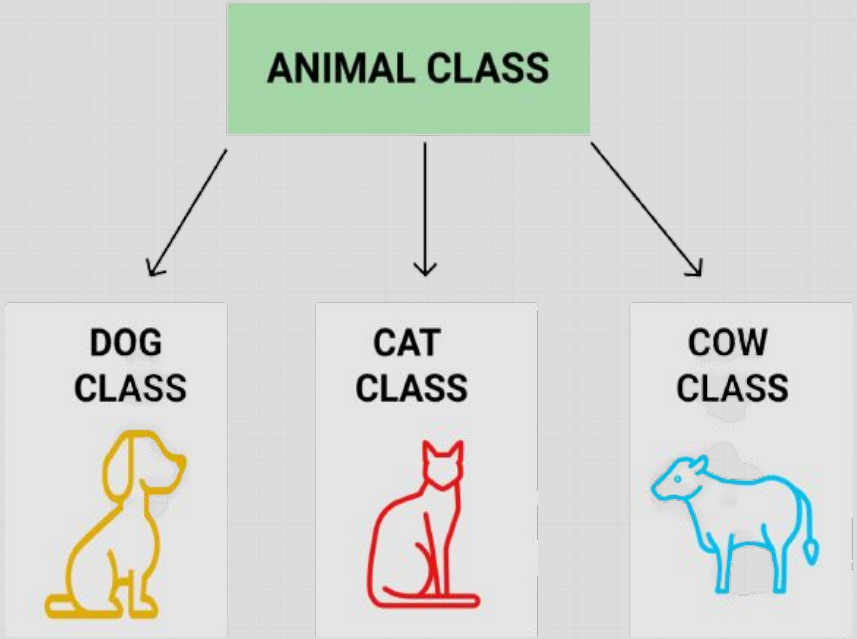
Inheritance



The animal is called SUPER class and the other classes are called SUB class

Inheritance

- A class can inherit from one parent class by specifying the parent class name after the extends keyword.
- The constructor and private members can not be inherited from parent to child




```
class Animal {

  constructor(name) {
    this.name = name;
  }

}

class Dog extends Animal {

  constructor(name) {
    super(name);
  }

  bark(){
    console.log(`${this.name} is barking.`);
  }

}
```

Super keyword

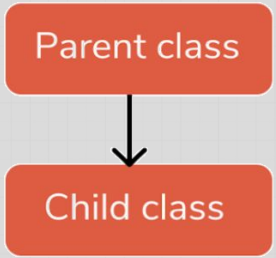
- The super keyword refers to the superclass (Parent).
- We can use super() to call a superclass's constructor

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

class Employee extends Person {

  constructor(name, age, jobTitle) {
    super(name, age);
    this.jobTitle = jobTitle;
  }

}
```

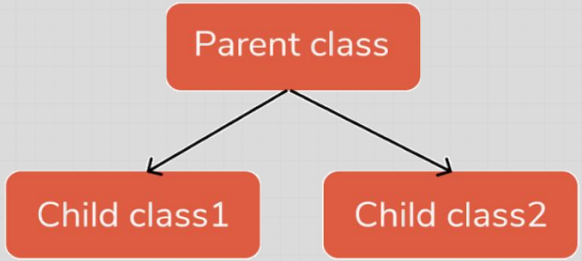
Accesses the parent class members

Types of Inheritance

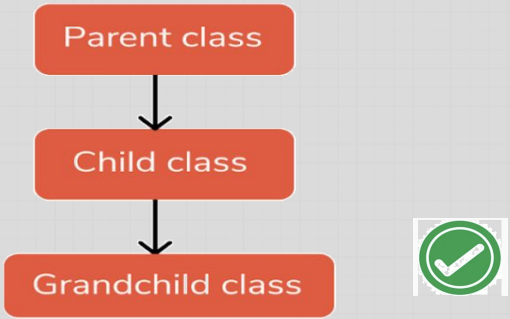
Single Inheritance



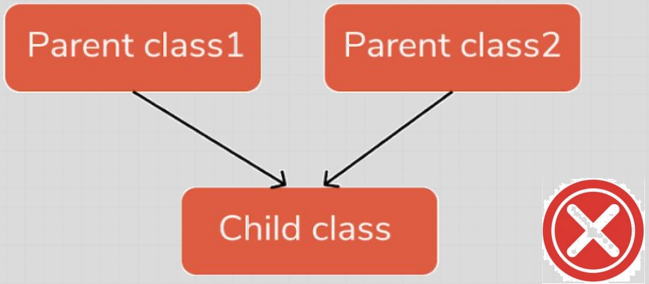
Hierarchical Inheritance



Multilevel Inheritance



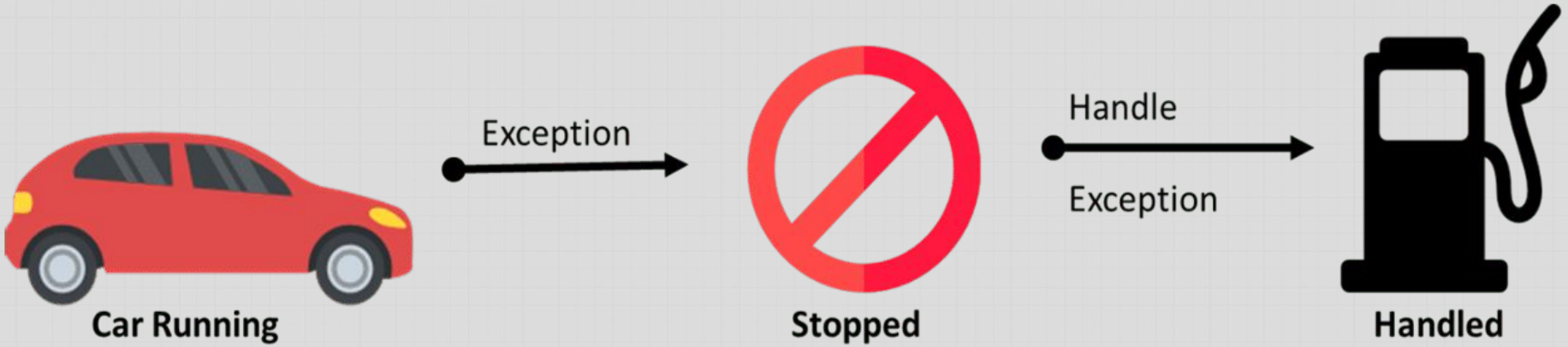
Multiple Inheritance



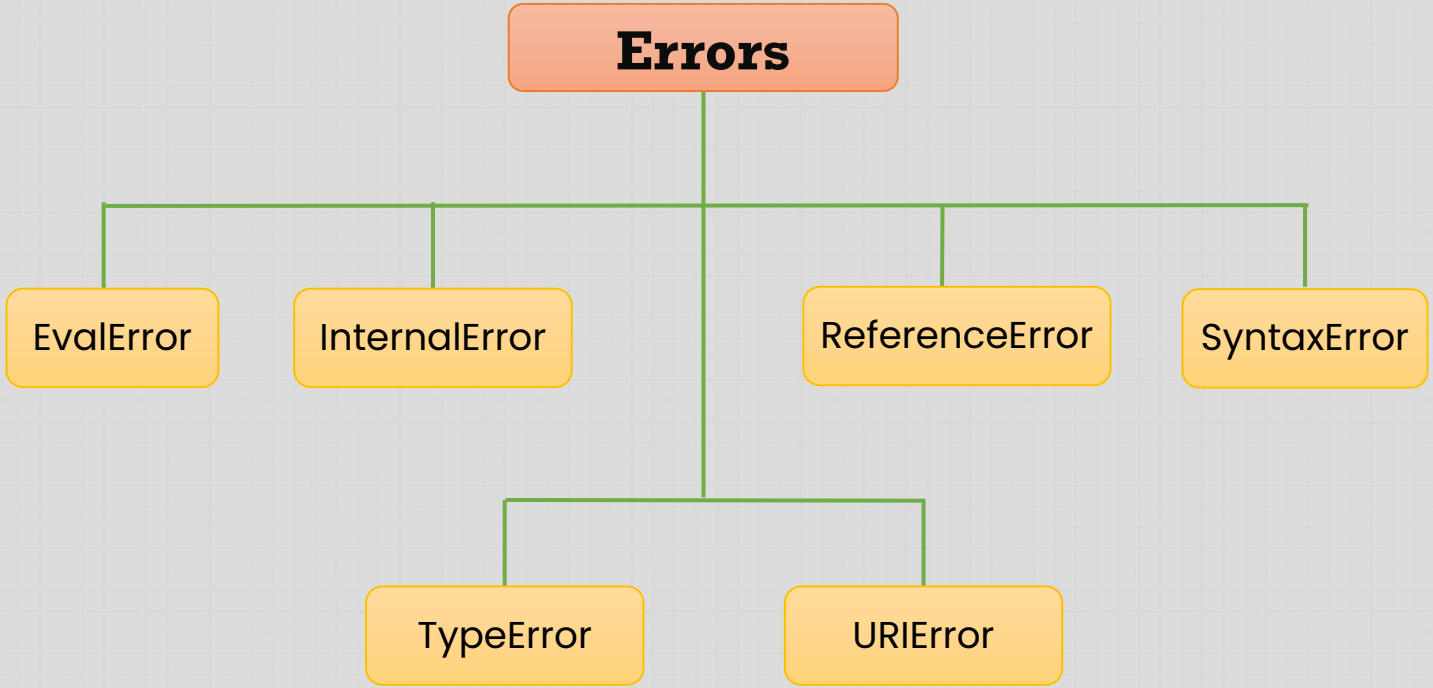
Exceptions/Errors

Exceptions/Errors

- An unwanted or unexpected event (Something went wrong)
- Exceptions/error are runtime errors that disrupt the normal flow of code execution



Exceptions/Errors Hierarchy



Try & Catch Blocks

- Exceptions are handled using try-catch blocks.

```
try {
    // Code that may throw an error
} catch (error) {
    // Code to handle the error
}
```

Finally Block

- An optional block that can be given after the last catch block
- Always executed after try & catch blocks whether an exception/error occurs or not

```
try {
    // Code that may throw an error
} catch (error) {
    // Code to handle the error
} finally {
    // Code that will always run
}
```

Throw Keyword

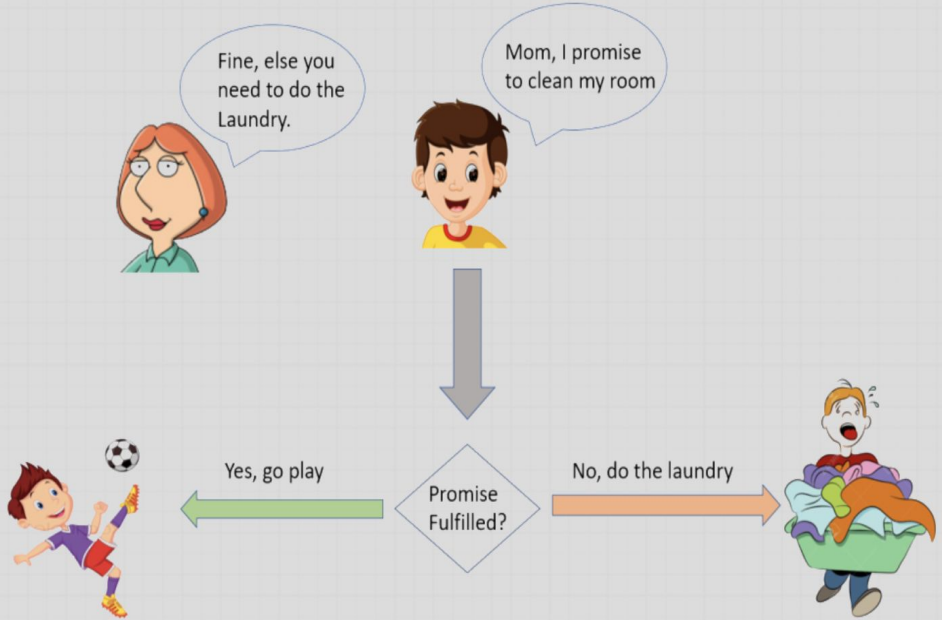
- Used for manually throwing an exception

```
throw new Error("Error Message");
```

Promises

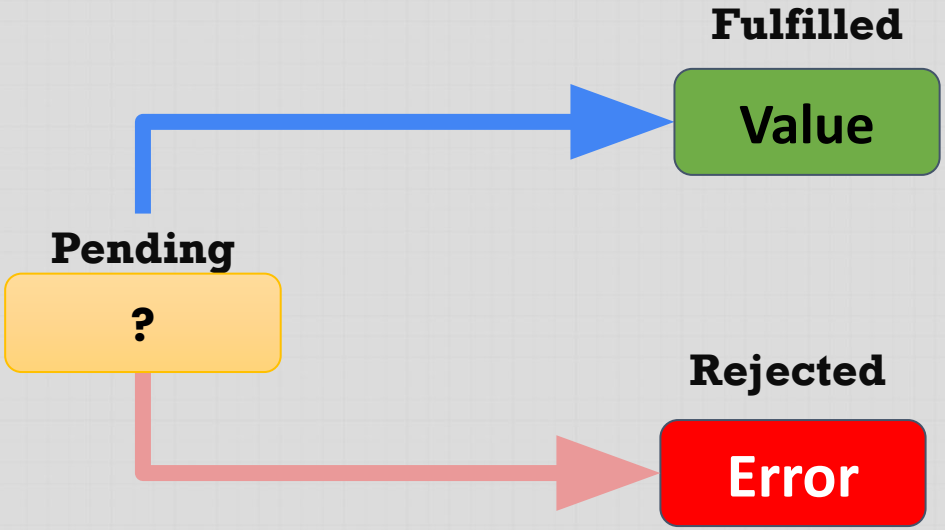
Promises

- Promises in JavaScript can Help handle async operations
- Promises are objects that represent the eventual completion (or failure) of an async operation and its resulting value
- Promises Wait for operations to finish



Promise States

- A Promise in JavaScript has three states:
  - Pending: Initial state, waiting
  - Fulfilled: Operation completed successfully
  - Rejected: Operation failed



Promise Creation

```
let promise = new Promise((resolve, reject) => {
  // async operation

  if (success) {
    resolve('Success');
  } else {
    reject('Failure');
  }

});
```

- Resolve: Call if operation succeeds
- Reject: Call if operation fails

Handling Promises

- A Promises is handled by using then() and catch() methods of the promise objects
  - then(): Runs if promise is fulfilled
  - catch(): Runs if promise is rejected
  - finally(): Optional to call and always runs whether the promise is fulfilled or rejected

```
let promise = new Promise((resolve, reject) => {
  // async operation

  if (success) {
    resolve('Success');
  } else {
    reject('Failure');
  }

});
```

```
promise.then((result) => {
    console.log('Success:', result);
});
```

```
promise.catch((error) => {
    console.error('Error:', error);
});
```

Promise Example: Voting Eligibility Check

Creating a promise object

```
let checkVotingEligibility = (age) => {

    return new Promise((resolve, reject) => {

        if (age >= 18) {
            resolve("Eligible to vote");
        } else {
            reject("Not eligible to vote");
        }

    });

};
```

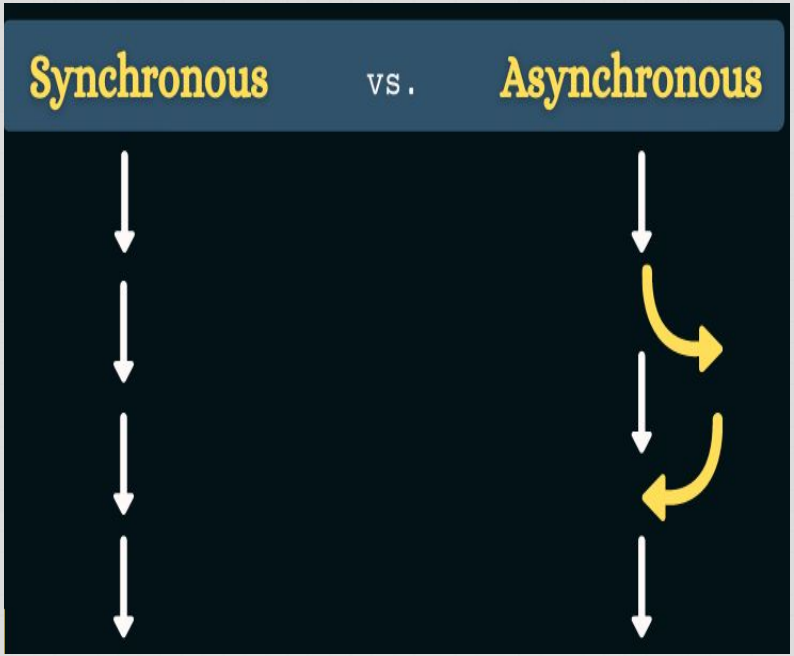
Handling the promise

```
checkVotingEligibility(20)
    .then((message) => {
        console.log(message);
    })
    .catch((error) => {
        console.error(error);
    })
    .finally(() => {
        console.log("Eligibility check completed.");
    });
```

Async & Await

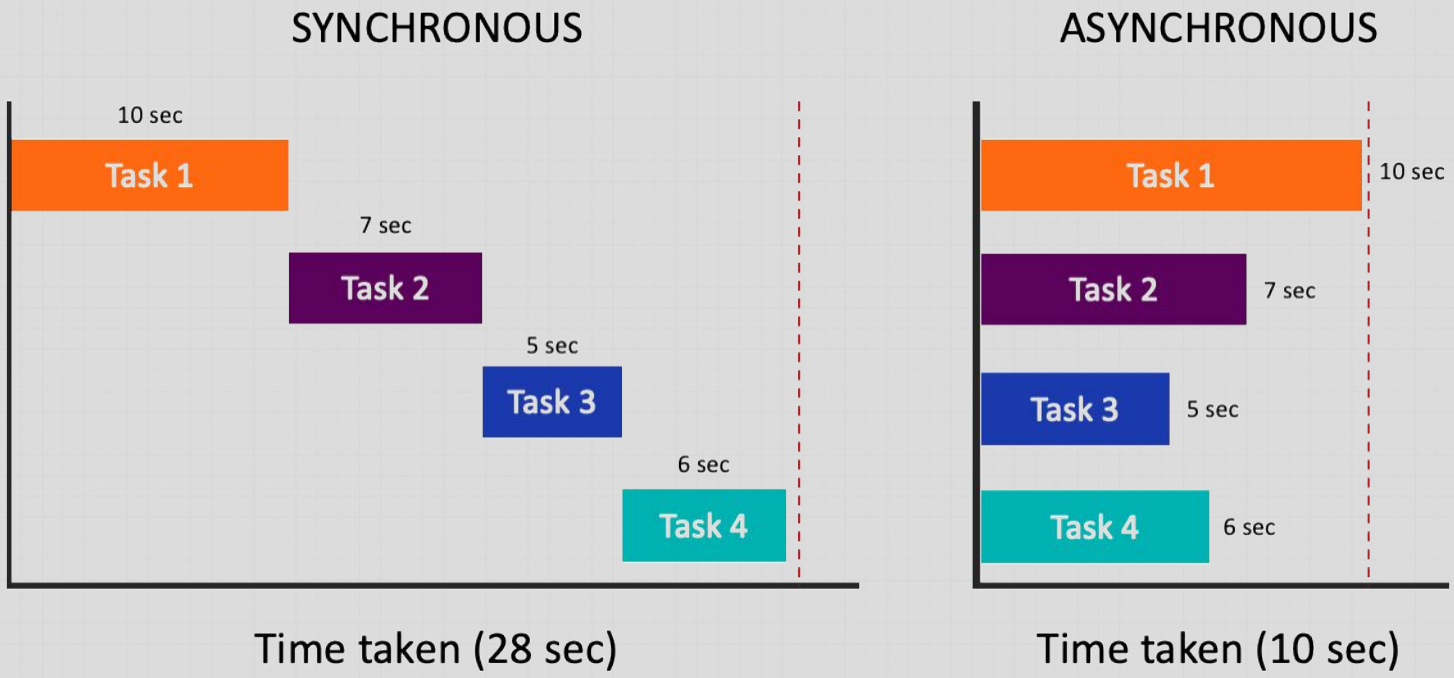
Asynchronous Functions

- Functions that operate asynchronously
- Allows the program to run other code while waiting for the time-consuming operations to complete
- Asynchronous functions don't block the execution of the rest of the code
- Asynchronous functions can be achieved using the async and await keywords



Benefits of Asynchronous Functions

- Improves performance by not blocking the execution of other code
- Provides a better user experience by Improving application responsiveness



Async keyword

- Used to declare functions as asynchronous
- The function returns a promise, even if it doesn't explicitly return one

```
async function findElement(locator) {

    let element = new Promise((resolve, reject) => {
        if (locator === "valid-locator") {
            resolve('Element found');
        } else {
            reject('Element not found');
        }
    });

    return element;
}
```

```
async function clickElement(locator) {

    findElement(locator)
        .then((foundMessage) => {
            console.log(foundMessage);
            console.log('Clicking the element');
        })
        .catch((errorMessage) => {
            console.error(errorMessage);
            console.log('Unable to click the element');
        });

}
```

Await keyword

- Used inside an async function to wait for a promise
- Pauses the function until the promise is resolved or rejected

```
async function clickElement(locator) {

    findElement(locator)
        .then((foundMessage) => {
            console.log(foundMessage);
            console.log('Clicking the element');
        })
        .catch((errorMessage) => {
            console.error(errorMessage);
            console.log('Unable to click the element');
        });

}
```

```
async function runTest() {
    await clickElement('valid-locator');
    await clickElement('invalid-locator');
}

runTest();
```

Example Explanation

```
async function findElement(locator) {
    let element = new Promise((resolve, reject) => {
        if (locator === "valid-locator") {
            resolve('Element found');
        } else {
            reject('Element not found');
        }
    });
    return element;
}

async function clickElement(locator) {
    findElement(locator)
        .then((foundMessage) => {
            console.log(foundMessage);
            console.log('Clicking the element');
        })
        .catch((errorMessage) => {
            console.error(errorMessage);
            console.log('Unable to click the element');
        });
}

async function runTest() {
    await clickElement('valid-locator');
    await clickElement('invalid-locator');
}

runTest();
```

- findElement function:
  - Uses async to return a promise
  - Resolves if the locator is valid
  - Rejects if the locator is invalid
- clickElement function:
  - Uses await to wait for findElement
  - Logs message based on promise result
- runTest function:
  - Uses await to wait for both clickElement

Introduction to Playwright

What is Playwright?

- An automation tool for automating web browsers
- Developed by Microsoft
- Provides support for multiple programming languages
- Supports chromium, firefox, and webKit browsers

Playwright

Why Use Playwright?

- Reliable and fast browser automation
- Cross-browser testing
- Auto-waiting functionality
- Easier setup and maintenance

Setting up Playwright project

- **Steps:**
  1. Create a new folder for VS Code project
  2. Open the folder in VS Code
  3. Open VS Code terminal
  4. Give the npm init playwright@latest command
  5. When asked Okay to proceed? type y
  6. Select JavaScript for the language
  7. Select the default folder for end-to-end tests by simply pressing the Enter key
  8. Select false for adding GitHub Actions workflow by simply pressing the Enter key
  9. Select true for installing playwright browsers by simply pressing the Enter key
  10. Wait for the installations to be completed
  11. Open the terminal and give the npx playwright test command (By default it runs the tests in headless mode
  12. Open the terminal and and give the npx playwright test --headed command

Recommended Naming Conventions

- For folder names: lowercase
- For test files: lowerCamelCase or Kebab-case
  - lowerCamelCase: loginTest.spec.js, loginTest.test.js
  - Kebab-case: login-test.spec.js, login-test.test.js
- For page object files: UpperCamelCase
  - UpperCamelCase: LoginPage.js