Objectives

- Object-Oriented Programming
- Error handling

Object-Oriented Programming

Introduction

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of "objects." These objects can contain data in the form of fields (often called properties) and code in the form of procedures (often called methods). The goal of OOP is to make programs more modular, reusable, and easier to maintain. JavaScript, while initially a prototype-based language, has evolved to support class-based OOP, allowing developers to implement complex systems more intuitively.

Core Principles of OOP:

- **Encapsulation**: Combines data and the functions that operate on that data within a single unit or class. This keeps data safe from outside interference and misuse.
- **Abstraction**: Allows programmers to hide all but the relevant data about an object to reduce complexity and increase efficiency.
- **Inheritance**: Facilitates the creation of new classes based on existing ones. It promotes code reusability by allowing shared behaviors to be defined once and inherited by child classes.
- **Polymorphism**: Enables different classes to be treated through the same interface, often by overriding methods or implementing the same methods in different ways.

Objects

Objects in JavaScript are collections of key-value pairs where the keys are strings (or Symbols) and the values can be any data type, including functions. Objects are the foundation of OOP in JavaScript.

Object Literal Example:

```
const person = {
  name: "Alice",
  age: 30,
  greet() {
```

```
    console.log(`Hi, I'm ${this.name}`);
  }
};
```

In the example above, `person` is an object with properties `name` and `age`, and a method `greet()`. JavaScript makes it easy to define and manipulate objects using simple syntax. Properties can be accessed with dot notation (`person.name`) or bracket notation (`person['name']`). New properties can be added dynamically, and existing ones can be updated or deleted.

Objects can also be nested, and they often serve as the building blocks for more complex data structures.

Methods and Attributes

In JavaScript, objects are collections of related data and functionality. They consist of **attributes** (also called properties) and **methods**.
Attributesdefine the object's characteristics and store its data. For example, a `car` object might have attributes like `make`, `model`, and `year`. These properties hold specific values that describe the object's state.

Methods are functions attached to objects that define their behavior. Using the same `car` example, methods like `startEngine()` or `drive()` would represent actions the car can perform. Methods often use the object's attributes to perform their tasks.

Constructor Functions

Constructor functions are special functions we use to create and initialize objects. They serve as blueprints for generating multiple objects with similar properties and methods.
When we want to create a new object using a constructor function:

1. We always use the `new` keyword when calling the function
2. Inside the function, `this` refers to the new object we're creating
3. We use `this` to assign properties and methods to our object

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet=function () {
  console.log(`Hello, I'm ${this.name}`);
};
```

```
}

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

Prototypes

When we create a constructor function, JavaScript automatically provides it with a `.prototype` property. This prototype object serves as a template for all instances created using the `new` keyword - each new object's internal `[[Prototype]]` gets linked to this shared prototype. This connection enables us to efficiently add new methods or properties to all instances at once by simply adding them to the constructor's prototype. For example, if we add a method to `Person.prototype`, all existing and future Person instances immediately gain access to it through the prototype chain

```
function Car(brand, model, year) {
  this.brand = brand;
  this.model = model;
  this.year = year;
}

Car.prototype.getInfo = function() {
  return `${this.brand} ${this.model} (${this.year})`;
};

const car1 = new Car("Toyota", "Corolla", 2020);
const car2 = new Car("Honda", "Civic", 2022);

console.log(car1.getInfo()); // Toyota Corolla (2020)
console.log(car2.getInfo());
```

Classes

Classes, introduced in ES6 (2015), give us a cleaner way to create objects and handle inheritance in JavaScript.
A class serves as a blueprint for creating objects, letting us define both properties (the data each object will store) and methods (the actions objects can perform) in one organized structure. This approach replaces the older, more cumbersome combination of constructor functions and prototype assignments, giving us code that's easier to write, read, and maintain.

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound`);
  }
}

const dog = new Animal("Buddy");
dog.speak();
```

Inheritance

Inheritance allows a class to inherit properties and methods from another class. In JavaScript, this is done using the `extends` keyword and `super()` function.

```
class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks`);
  }
}

const myDog = new Dog("Rex");
myDog.speak(); // Rex barks
```

The `Dog` class inherits from the `Animal` class. The `super()` call in the constructor allows the child class to access the parent class's constructor and initialize its properties. Inheritance is useful for representing hierarchical relationships between classes.

Encapsulation

Encapsulation is the practice of hiding the internal state and requiring all interaction to be performed through an object’s methods. In JavaScript, you can use closures or private fields (with `#`) to encapsulate data.

```
class BankAccount {
  #balance = 0;
```

```
  deposit(amount) {
    this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount();
account.deposit(100);
console.log(account.getBalance()); // 100
```

Using `#` before a property name makes it private, meaning it cannot be accessed outside the class. This ensures that sensitive data is only modified in controlled ways.

Static Methods and Properties

Static methods belong to the class itself rather than to any instance. They are useful for utility functions or class-wide behaviors.

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

console.log(MathHelper.add(5, 3)); // 8
```

You cannot call a static method on an instance; it must be called on the class. Static properties work similarly and are shared across all instances.

Polymorphism

Polymorphism means having many forms. In JavaScript, this is often achieved by method overriding, where a subclass defines its own version of a method from the superclass.

```
class Shape {
  draw() {
    console.log("Drawing a shape");
  }
```

```
}

class Circle extends Shape {
  draw() {
    console.log("Drawing a circle");
  }
}

const s1 = new Shape();
const s2 = new Circle();
s1.draw(); // Drawing a shape
s2.draw(); // Drawing a circle
```

Even though both objects implement the `draw` method, the version that's executed depends on the actual object's class. This allows different types to be used interchangeably if they follow the same interface.

Getters and Setters

When working with object properties in JavaScript, we often encounter scenarios where we need more control over how values are assigned and retrieved. For instance, we might want to validate a value before storing it, transform data upon retrieval, or restrict access to certain sensitive properties. However, with direct property assignment like `obj.property = value`, we lose the ability to enforce such logic. This is where *getters* and *setters* become useful.

Setters:

Setters are special methods that run when a value is assigned to a property. They allow us to validate or transform the incoming data before actually saving it. In JavaScript, setters are defined using the `set` keyword. Here’s a simple example:

```
const user = {
  firstName: '',
  set name(value) {
    if (value.length < 3) {
      console.log('Name is too short.');
    } else {
      this.firstName = value;
    }
  }
};
```

```
user.name = 'Al';    // Output: Name is too short.
user.name = 'Alice'; // Valid assignment
```

Getters

Getters, on the other hand, are methods that execute when a property is read. They enable us to return a modified or formatted version of the stored value. Getters are defined using the `get` keyword. Consider this example:

```
const user = {
  firstName: 'Alice',
  get name() {
    return this.firstName.toUpperCase();
  }
};

console.log(user.name); // Output: ALICE
```

Error handling

Introduction

When writing scripts and code, errors are inevitable. These errors can be **syntax errors**, such as misspelling a function or variable name, or **logical errors**, like mistakes in `if` conditions or loops that lead to unexpected behavior. Such errors can cause our program to fail, requiring fixes before the script can run properly.
Another category is **runtime errors**, which occur during script execution. These can be caused by invalid user input or unpredictable conditions. While we can't always prevent runtime errors, JavaScript provides tools to handle them gracefully.

Types of Errors

- **Syntax Errors:** These occur when we make mistakes in writing our script, such as misspelling a function or variable name, or forgetting parentheses or curly braces.
- **Runtime Errors:** These errors occur while our program is running and cause it to stop. They are often caused by user input, such as entering invalid values.
- **Logic Errors:** These errors happen when the code does not work as intended due to incorrect formatting or logic in conditions. They don't crash the program but cause it to behave in unexpected ways.

- **Arithmetic Errors:** These are similar to logic errors but specifically involve mathematical operations, such as dividing by zero, or improper use of operator precedence (e.g., `+`, `-`, `/`, `*`).
- **Resource Errors:** These errors occur when we exceed the resources available on our machine, such as running out of memory or creating an infinite loop.

JavaScript Errors

Error Type	Description	Example
`SyntaxError`	Invalid code structure	`function() {` (missing name)
`ReferenceError`	Accessing undefined variables	`console.log(notDefined)`
`TypeError`	Wrong data type usage	`null.toString()`
`RangeError`	Numeric value out of range	`new Array(-1)`
`Custom Errors`	Developer-defined exceptions	`throw new Error("Invalid input")`

Using The Console

The browser console is a powerful tool for detecting errors. It shows:

- The line where the error occurred.
- The type of error (e.g., `ReferenceError`, `TypeError`).
- While the console helps debug syntax and logical errors, **runtime errors** may depend on user input or application state, requiring a different approach.

Handling Runtime Errors with `try-catch`

We can handle runtime errors using the `try-catch` statement.

1. `try` **Block:**
   - Define the statements that may cause an error or crash the script inside the `try` block.
2. `catch` **Block:**
   - After the `try` block, create a `catch` block to handle any errors that occur.
   - A generic `catch` (without specifying an error type) will catch all errors.
   - Alternatively, we can catch specific types of errors by specifying them in the `catch` block.
3. `finally` **Block (Optional):**
   - At the end, we can add a `finally` block containing statements that will execute **regardless** of whether an error occurred.

```
const a = 4;
try {
  a = 5;
} catch (error) {
  console.log(error); // This will log the error object
} finally {
  console.log("done!");
}
```

Raising Custom Errors in JavaScript

In JavaScript, we can create and throw custom errors to enforce specific conditions. For example, we can define rules for valid function inputs and throw an error when invalid values are provided.

Steps to Create and Throw an Error:

1. **Create an Error Object:**
   - Use the `Error` constructor and pass a descriptive message.
   - Example: `const error = new Error("Invalid input: Only even numbers allowed");`
2. **Throw the Error:**
   - Use the `throw` keyword to raise the error when a condition is met.

```
let err = new Error("Only even numbers allowed.");
function divideByTwo(number) {
  if (number % 2 !== 0) {
    throw err;
  }
  return number / 2;
}

try {
  console.log(divideByTwo(5));
} catch (error) {
  console.log(error.message);
}
```

Tasks

Task 1

Define a `BankAccount` class with:

Properties

- `#accountNumber` (private, string) – A unique account identifier.
- `#ownerName` (private, string) – The account holder's name.
- `#balance` (private, number) – The current balance (default: `0`).
- `#transactions` (private, array) – Stores transaction history.

Methods

1. **`constructor(accountNumber, ownerName, initialBalance = 0)`**
   - Initializes the account with given details.
   - Throws an error if `initialBalance` is negative.
2. **`deposit(amount)`**
   - Adds `amount` to the balance.
   - Throws an error if `amount` is negative.
   - Logs the transaction in `#transactions`.
3. **`withdraw(amount)`**
   - Deducts `amount` from the balance.
   - Throws an error if:
     - `amount` is negative.
     - Insufficient funds (`balance < amount`).
   - Logs the transaction in `#transactions`.
4. **`getBalance()`**
   - Returns the current balance.
5. **`getStatement()`**
   - Prints all transactions in a formatted way.
6. **`static transfer(sourceAccount, targetAccount, amount)`**
   - A **static method** that transfers `amount` from `sourceAccount` to `targetAccount`.
   - Throws an error if the transfer fails.

Task 2

- Use `try-catch` blocks to handle:
  - Negative deposits/withdrawals.
  - Insufficient funds.

- Invalid transfers.