Computing and Software Engineering

GET211

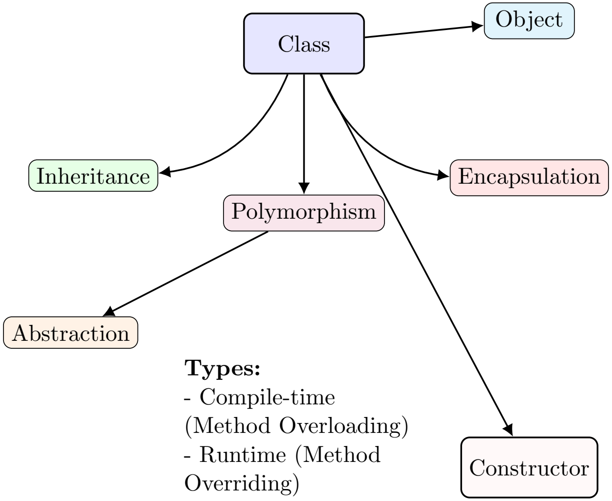
September 23, 2025

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects rather than functions or logic. It enables code modularity, reusability, and scalability.

- OOP is a programming paradigm based on the concept of **objects**.
- An object is an instance of a **class**.
- Classes define the structure and behavior of objects, including their **properties** and **methods**.
- Key concepts: **Encapsulation**, **Inheritance**, **Polymorphism**, **Abstraction**.
- By focusing on objects and their interactions, OOP simplifies complex systems and promotes best practices in software engineering.

Main Concepts in OOP



Class

Object

Inheritance

Polymorphism

Encapsulation

Abstraction

Constructor

Object Characteristics:
- State (Attributes)
- Behavior (Methods)
- Unique Identity

Types:
- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

Access Modifiers:
- Public
- Private
- Protected
- Package Private

Abstraction Mechanisms:
- Abstract Classes
- Interfaces
- Pure Virtual Functions

Types:
- Compile-time (Method Overloading)
- Runtime (Method Overriding)

Constructor Types:
- Default Constructor
- Parameterized Constructor
- Copy Constructor

Concepts of OOP

- **Class**: A blueprint or template that defines attributes and behaviors. A **class** is a blueprint for creating objects.
- **Object**: An instance of a class with specific values for its attributes. An **object** is an instance of a class.

Class

- A **class** is a blueprint or template for creating objects.
- It defines:
    - **Attributes**: The data or properties of the objects.
    - **Methods**: The functions or behaviors of the objects.
- A class encapsulates both data and functionality.
- **Constructor**: A special method to initialize new objects with the same name as the class.

Object

- An **object** is an instance of a class.
- It represents a specific entity with attributes (data) and behaviors (methods) as defined in the class.
- Multiple objects can be created from the same class, each with unique attributes.

Attributes

- **Attributes** (properties) are the variables that hold the data for an object.
- Attributes represent the state or properties of an object.
- Attributes are defined in the class and assigned specific values in objects.

Methods

- **Methods** are functions defined inside a class that operate on the attributes of an object.
- They define the behavior or actions an object can perform.
- Methods often manipulate or retrieve object attributes.

Relationships Between Classes, Objects, Attributes, and Methods

- **Class**:
  - A blueprint that defines attributes and methods.
- **Object**:
  - An instance of a class with unique attribute values.
- **Attributes**:
  - Data that defines the state or properties of an object.
- **Methods**:
  - Functions that define the behavior of an object and interact with its attributes.

Encapsulation

Encapsulation is the principle of bundling data (attributes) and methods that operate on the data into a single unit or class.

- Encapsulation is the bundling of data and methods into a single class.
- It restricts access to an object's internal state, exposing only necessary operations.
- This improves security and reduces complexity.

Access to the data is restricted using access modifiers (‘private‘, ‘protected‘, ‘public‘), ensuring that object data is only accessed or modified in controlled ways.

Inheritance

Inheritance allows one class (child or subclass) to inherit the properties and methods of another class (parent or superclass).

- Inheritance allows a class to inherit attributes and methods from another class.
- The child class can extend or modify the functionality of the parent class.
- This promotes code reuse and allows for the creation of more specialized versions of general classes.

Overriding methods: A subclass can override methods of the parent class to provide its own specific implementation.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

- Polymorphism allows methods to be used in different forms.
- Methods can behave differently depending on the object they are called on.
- It can manifest in method overriding (dynamic polymorphism) or method overloading (static polymorphism).
- Polymorphism simplifies code maintenance and extends the flexibility of the codebase.

Abstraction

Abstraction is the practice of hiding the complex implementation details of an object and exposing only the necessary parts of the object.

- Abstraction hides the complex implementation details of a class and exposes only necessary functionality.
- It simplifies the interaction with objects by hiding internal workings.
- This is achieved through abstract classes or interfaces, which define method signatures without implementation.

Advantages of OOP

- **Modularity**: Code is organized into independent objects.
- **Reusability**: Code can be reused through inheritance and composition.
- **Maintainability**: Changes can be made to one part of a system without affecting others.
- **Scalability**: Easier to scale systems by adding new objects or modifying existing ones.

Challenges

- **Complexity**: While OOP provides powerful tools, it can also lead to overly complex designs if not used appropriately.
- **Performance Overhead**: Dynamic features like polymorphism or reflection can introduce runtime overhead.
- **Learning Curve**: Mastering OOP, especially concepts like design patterns and generics, can take time and experience.

OOP in MATLAB

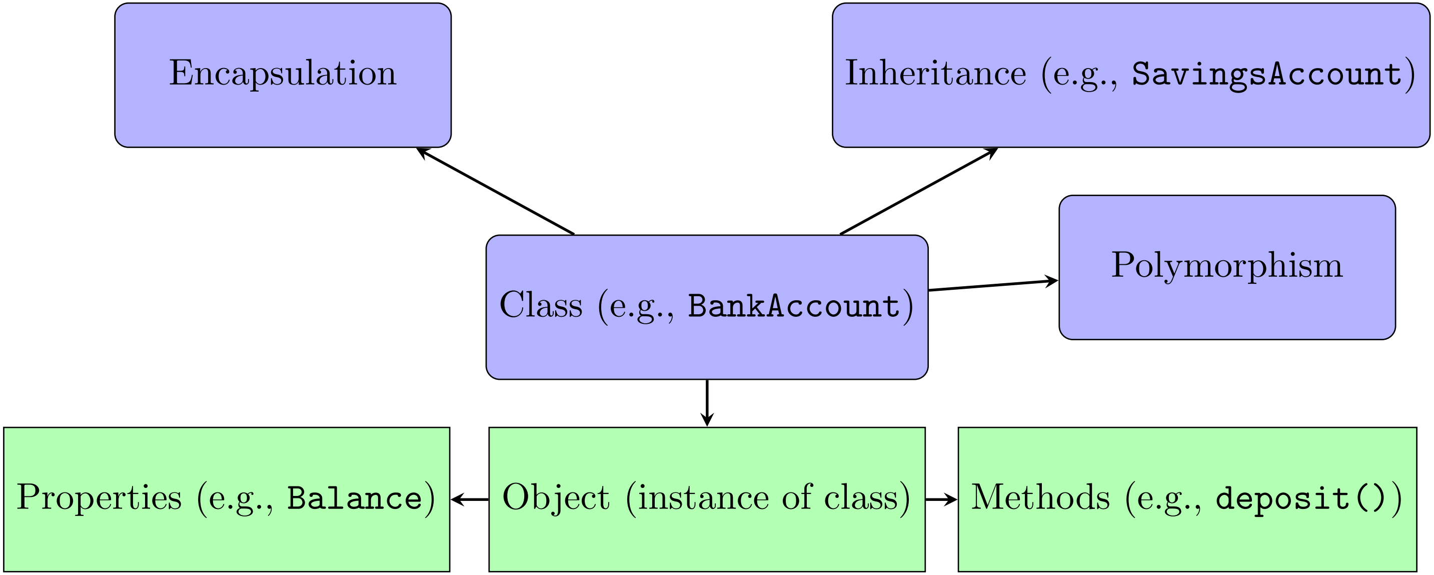
- MATLAB supports object-oriented programming (OOP) from version 2008a and onwards.
- OOP in MATLAB enables code reuse, modularity, and encapsulation.
- Key concepts: **Classes**, **Objects**, **Methods**, **Inheritance**, **Encapsulation**.
- MATLAB's OOP is class-based, meaning you define classes and create objects from these classes.

Creating a Class

- Classes in MATLAB are defined using the `classdef` keyword.
- A class can contain **properties** (attributes) and **methods** (functions).
- Syntax:

```
classdef ClassName
    properties
        % Define properties here
    end
    methods
        % Define methods here
    end
end
```

OOP Example in MATLAB


Encapsulation
Inheritance (e.g., SavingsAccount)
Class (e.g., BankAccount)
Polymorphism
Properties (e.g., Balance)
Object (instance of class)
Methods (e.g., deposit())

Example

```matlab
classdef BankAccount
    properties
        Balance
    end
    methods
        function obj = BankAccount(initialBalance)
            obj.Balance = initialBalance; % Constructor
        end
        function obj = deposit(obj, amount)
            obj.Balance = obj.Balance + amount; % Deposit Method
        end
        function displayBalance(obj)
            disp(['Balance: ', num2str(obj.Balance)]);
        end
    end
end
```

Creating and Using Objects

- Objects are created from classes using the class constructor.
- Objects encapsulate data (properties) and behavior (methods).
- Example of creating and using an object:

```
obj = BankAccount(100); % Create object with initial balance of 100
obj = deposit(obj, 50);  % Deposit 50
obj.displayBalance();    % Display updated balance
```

- In the example, `obj` is an instance of the `BankAccount` class.
- Methods are called using dot notation.

Encapsulation

- Encapsulation is the concept of bundling data (properties) and methods (functions) into a single unit.
- MATLAB supports encapsulation through access control on properties and methods.
- Properties can be made `public`, `private`, or `protected`.

```matlab
classdef BankAccount
    properties (Access = private)
        Balance
    end
    methods
        function obj = BankAccount(initialBalance)
            obj.Balance = initialBalance;
        end
        function deposit(obj, amount)
            obj.Balance = obj.Balance + amount;
        end
        function balance = getBalance(obj)
            balance = obj.Balance;
        end
    end
end
```

Inheritance

- Inheritance allows a class (child class) to inherit properties and methods from another class (parent class).
- MATLAB supports single inheritance, where a child class can inherit from only one parent class.
- Inheritance is defined using the < symbol.

Inheritance

```
classdef SavingsAccount < BankAccount
    properties
        InterestRate
    end
    methods
        function obj = SavingsAccount(initialBalance,
           rate)
            obj = obj@BankAccount(initialBalance); %
               Call parent constructor
            obj.InterestRate = rate;
        end
        function obj = applyInterest(obj)
            obj.Balance = obj.Balance + obj.Balance *
               obj.InterestRate;
        end
    end
end
```

Polymorphism

- Polymorphism allows methods in a subclass to have the same name as those in the parent class but with different behaviors.
- This is achieved by overriding methods in the subclass.
- MATLAB supports method overriding and dynamic dispatching.

```
classdef PremiumAccount < BankAccount
    methods
        function displayBalance(obj)
            disp(['Premium Balance: ',
               num2str(obj.Balance)]);
        end
    end
end
```

- The `PremiumAccount` class overrides the `displayBalance` method to customize how the balance is displayed.
- When an object of type `PremiumAccount` calls the `displayBalance` method, the overridden version is executed.

Example of Using Inheritance and Polymorphism

- Example of creating objects and demonstrating polymorphism:

```
obj1 = BankAccount(200);
obj2 = PremiumAccount(500);

obj1.displayBalance(); % Calls BankAccount's displayBalance
obj2.displayBalance(); % Calls PremiumAccount's overridden displayBalance
```

- Even though both `obj1` and `obj2` are objects of different types, the `displayBalance` method behaves differently based on the object's class.

Abstraction

Abstract Classes

Define a base class with abstract methods that must be implemented by subclasses.

Example:

```
classdef (Abstract) BankAccount
    methods (Abstract)
        dispBalance(obj, amount)
    end
end

classdef SavingsAccount < BankAccount
    properties
        Balance
    end

    methods
        function obj = BankAccount(initialBalance)
            obj.Balance = initialBalance;
        end

        function dispBalance(obj)
            fprintf('Current Balance: %.2f\n', obj.Balance);
        end
    end
end

account = SavingsAccount(500); % Initial balance: $500
account.dispBalance();         % Display balance
```

Exercise

Create a class to represent a Student, complete with properties and methods. Then, write a script to interact with the class.
Tasks
1. Define the Class
Create a class named Student with the following:
Properties:
Name (string): Name of the student.
Age (numeric): Age of the student. Grades (numeric array): List of grades. Methods: Constructor to initialize Name and Age. addGrade(obj, grade): Adds a new grade to the Grades array. calculateAverage(obj): Calculates and returns the average grade. displayInfo(obj): Displays the student's information (Name, Age, Average Grade).
2. Write a Script
Create a script to: Instantiate a Student object. Add grades using addGrade. Display the student's information.

Exercise

Create a class to represent a **Student**, complete with properties and methods. Then, write a script to interact with the class.
Tasks:

1. **Define the Class** Create a class named `Student` with the following:
   - **Properties:**
     - `Name` (*string*): Name of the student.
     - `Age` (*numeric*): Age of the student.
     - `Grades` (*numeric array*): List of grades.
   - **Methods:**
     - `Constructor`: Initializes `Name` and `Age`.
     - `addGrade(obj, grade)`: Adds a new grade to the `Grades` array.
     - `calculateAverage(obj)`: Calculates and returns the average grade.
     - `displayInfo(obj)`: Displays the student's information (`Name`, `Age`, Average Grade).
2. **Write a Script** Create a script to:
   - Instantiate a `Student` object.
   - Add grades using `addGrade`.
   - Display the student's information.

Exercise

Create a class hierarchy to represent different types of **Vehicles** using OOP principles. Then, write a script to interact with the classes.
Tasks:

1. **Define the Base Class**
   Create an abstract class named `Vehicle` with:
   - **Properties (Encapsulation):**
     - `Make` (*string*, `protected`): Manufacturer of the vehicle.
     - `Model` (*string*, `protected`): Model name.
     - `Speed` (*numeric*, `protected`): Current speed of the vehicle.
   - **Methods:**
     - Constructor to initialize `Make`, `Model`, and `Speed`.
     - Abstract method `accelerate(obj, increment)`: Increases the speed (implemented by subclasses).
     - Abstract method `brake(obj, decrement)`: Decreases the speed (implemented by subclasses).
     - `dispInfo(obj)`: Displays the vehicle's information (`Make`, `Model`, `Speed`).

Tasks (continued):

2. **Define Derived Classes (Inheritance and Polymorphism)**
    - Create a `Car` class:
        - Implements `accelerate` and `brake`.
        - Speed increases or decreases by a fixed amount, e.g., `increment/2` or `decrement/2`.
    - Create a `Bike` class:
        - Implements `accelerate` and `brake`.
        - Speed changes directly based on the increment or decrement.
3. **Write a Script**
    - Create objects of `Car` and `Bike`.
    - Perform the following:
        - Display their initial information.
        - Accelerate both vehicles using `accelerate`.
        - Apply brakes using `brake`.
        - Display their final information.