🧩 Section 3: Object-Oriented Programming (OOP)

🎯 Objectives

By the end of this section, students will be able to:

- Define and use **classes** and **objects**.
- Implement **constructors** and understand `const` vs normal constructors.
- Apply **inheritance** and use the `super` keyword.
- Combine functionality using **mixins**.
- Define and use **abstract classes**.

🔷 1. Classes and Objects

🧠 Concepts

- A **class** defines a blueprint for objects (data + behavior).
- **Objects** are instances of classes.
- Class members can be fields (variables) or methods (functions).

💡 Example

```
class Car {
  String brand;
  int year;

  void start() {
    print('$brand is starting...');
  }
}

void main() {
  Car myCar = Car();
  myCar.brand = 'Toyota';
  myCar.year = 2022;
```

```
  myCar.start();
  print('Brand: ${myCar.brand}, Year: ${myCar.year}');
}
```

Explanation:

- `Car` defines two fields and one method.
- You create an object using `Car()`.
- Use `.` to access fields and methods.

🧩 Exercise 3.1 — Simple Class

Create a class `Person` with:

- `name` (String)
- `age` (int)
- A method `introduce()` that prints "Hi, I'm `<name>` and I'm `<age>` years old."

Then, create a `Person` object and call `introduce()`.

🔹 2. Constructors

🧠 Concepts

- Constructors initialize object fields when creating an object.
- You can use **default**, **named**, or **const** constructors.

💡 Example

```
class Student {
  String name;
  int grade;

  // Default constructor
  Student(this.name, this.grade);

  // Named constructor
  Student.guest() {
    name = 'Guest Student';
    grade = 0;
  }
```

```
  void info() => print('Student: $name, Grade: $grade');
}

void main() {
  var s1 = Student('Alice', 10);
  var s2 = Student.guest();

  s1.info();
  s2.info();
}
```

Explanation:

- `this.name` and `this.grade` initialize fields directly.
- Named constructors provide flexibility (like factory presets).

🧩 Exercise 3.2 — Constructors Practice

Create a class `Book` with:

- Fields: `title`, `author`, and `price`.
- A constructor that initializes all fields.
- A named constructor `Book.free()` that sets price to 0.

In `main()`, create two `Book` objects and print their info.

🔷 3. Inheritance

🧠 Concepts

- A class can **inherit** from another using `extends`.
- Subclasses get parent properties and methods.
- `super` is used to call parent constructors or methods.

💡 Example

```
class Animal {
  String name;
  Animal(this.name);
```

```
  void makeSound() => print('$name makes a sound');
}

class Dog extends Animal {
  Dog(String name) : super(name);

  void bark() => print('$name barks loudly!');
}

void main() {
  var dog = Dog('Buddy');
  dog.makeSound();
  dog.bark();
}
```

Explanation:

- `Dog` inherits from `Animal`.
- `super(name)` calls the parent constructor.
- The subclass can extend or override behavior.

🧩 Exercise 3.3 — Inheritance Practice

Create:

- A base class `Shape` with a method `area()` (prints "Calculating area...").
- A subclass `Circle` with a field `radius` and an overridden `area()` method that prints `3.14 * radius * radius`.

In `main()`, create a `Circle` object and call `area()`.

🔹 4. Mixins

🧠 Concepts

- **Mixins** allow code reuse from multiple classes.
- Defined using `mixin` keyword.
- A class uses them via `with`.

💡 Example

```
mixin CanFly {
  void fly() => print('Flying high!');
}

mixin CanSwim {
  void swim() => print('Swimming fast!');
}

class Bird with CanFly {}
class Duck with CanFly, CanSwim {}

void main() {
  var bird = Bird();
  var duck = Duck();

  bird.fly();
  duck.fly();
  duck.swim();
}
```

Explanation:

- Mixins are like “traits” or “capabilities.”
- A class can combine several mixins separated by commas.

🧩 Exercise 3.4 — Mixins Practice

Create:

- mixin CanRun with method run().
- mixin CanJump with method jump().
- A class `Athlete` using both.

In `main()`, create an `Athlete` object and call both methods.

🔹 5. Abstract Classes

🧠 Concepts

- **Abstract classes** can’t be instantiated directly.
- Used to define interfaces or base templates.
- Subclasses must implement abstract methods.

💡 Example

```
abstract class Vehicle {
  void move(); // abstract method
}

class Car extends Vehicle {
  @override
  void move() => print('Car is moving');
}

class Bike extends Vehicle {
  @override
  void move() => print('Bike is moving');
}

void main() {
  Vehicle v1 = Car();
  Vehicle v2 = Bike();

  v1.move();
  v2.move();
}
```

Explanation:

- `abstract` marks a class that can't be created directly.
- Child classes must implement its abstract methods.

🧩 Exercise 3.5 — Abstract Class Practice

Create:

- An abstract class `Employee` with method `work()`.
- Two subclasses `Teacher` and `Developer` that implement `work()` differently.

In `main()`, create both and call `work()` on each.

🏁 Summary

In this section, you learned how to:

✅ Define and instantiate **classes**
✅ Use **constructors** (default, named, const)
✅ Implement **inheritance** and `super`
✅ Reuse functionality using **mixins**
✅ Create **abstract classes** and enforce contracts