OOP Foundations

Class	Class & Objects
Topcis	- Create & Use Class & Object - Constructors (Default & Named) - `this` Keyword Usage

Class Content: Classes and Objects

1. Lesson Objectives

By the end of this class, you will be able to:

- **Define** what a Class and an Object are using an analogy.
- **Create** a new Class with properties (fields) and behaviors (methods).
- **Instantiate** (create) an Object from a Class.
- **Use** an Object's properties and methods.
- **Explain** the role of a Constructor.
- **Implement** a default (parameterized) Constructor.
- **Use** `this` keyword to resolve ambiguity.
- **Implement** a Named Constructor for flexible object creation.

---

2. Core Concepts (Lecture & Demo)

We will use a single, evolving example throughout this lesson: a `Car`.

Part 1: The Blueprint and the Product (Class & Object)

- **What is a Class?**
  - A **blueprint**, template, or a set of instructions for creating something.
  - It defines a *type* of thing by grouping related data (properties) and functions (methods).

- **Analogy:** The architectural blueprint for a "Car." It defines that a car *must have* wheels, a color, and an engine, and that it *can* `start()`, `stop()`, and `honk()`.
- **Example (Code):**

```
// Part 1: The Blueprint (Class)
class Car {
  // Properties (fields)
  // Nullable types (?) are used since they aren't initialized
  // in a constructor in this first example.
  String? make;
  String? model;
  int? year;
  String? color;

  // Methods (behaviors)
  void startEngine() {
    print("Engine started!");
  }

  void drive() {
    // Use string interpolation ($)
    print("Driving the $make $model");
  }
}
```

- **What is an Object?**
  - An **instance** of a Class. It's the *actual thing* you build from the blueprint.
  - You can have many objects (instances) from one class, each with its own data.
  - **Analogy:** Your specific red 2023 Toyota Camry is an **object**. Your neighbor's blue 2021 Honda Civic is *another* **object** made from the *same* `Car` blueprint.

- **Example (Code)**

```
// The 'new' keyword is optional in Dart.
var myCar = Car(); // 'myCar' is an object
var neighborsCar = Car(); // 'neighborsCar' is a separate object
```

Part 2: Using Your Objects

- Once you have an object, you use **dot notation** (`.`) to access its properties and call its methods.
- **Example (Code):**

```
// Part 1: The Blueprint (Class)
class Car {
  // Properties (fields)
  // Nullable types (?) are used since they aren't initialized
  // in a constructor in this first example.
  String? make;
  String? model;
  int? year;
  String? color;

  // Methods (behaviors)
  void startEngine() {
    print("Engine started!");
  }

  void drive() {
    // Use string interpolation ($)
    print("Driving the $make $model");
  }
}

// Using Objects
void main() {
```

```
  // The 'new' keyword is optional in Dart.
  var myCar = Car(); // 'myCar' is an object
  var neighborsCar = Car(); // 'neighborsCar' is a separate object

  // Set properties on 'myCar'
  myCar.make = "Toyota";
  myCar.model = "Camry";
  myCar.year = 2023;
  myCar.color = "Red";

  // Set properties on 'neighborsCar'
  neighborsCar.make = "Honda";
  neighborsCar.model = "Civic";
  neighborsCar.year = 2021;
  neighborsCar.color = "Blue";

  // Call methods on 'myCar'
  myCar.startEngine(); // Output: Engine started!
  myCar.drive(); // Output: Driving the Toyota Camry

  // Call methods on 'neighborsCar'
  neighborsCar.drive(); // Output: Driving the Honda Civic
}
```

- **Problem:** Setting each property one by one is tedious and error-prone. What if we want to ensure *every* `Car` has a `make` and `model` as soon as it's created?

Part 3: The Factory Worker (Constructors)

- A **Constructor** is a special method that is called *automatically* when you create a new object (when you use the `new` keyword).
- Its job is to "construct" the object, typically by initializing its properties.
- **Rules:**
  1. It *must* have the exact same name as the Class.

2. It has *no return type* (not even `void`).

- **The Default (Parameterized) Constructor**
  - This is the most common type of constructor. You pass arguments to it, and it assigns those arguments to the object's properties.
  - **Example (Code):**

```
// Part 3: Constructor
class Car {
  // Properties are now non-nullable because the constructor
  // guarantees they will be initialized.
  String make;
  String model;
  int year;

  // The Parameterized Constructor (using Dart's 'this' sugar)
  // This automatically assigns the parameters to the properties
  // with the same name.
  Car(this.make, this.model, this.year);

  void drive() {
    // 'this' is optional here as there is no ambiguity.
    print("Driving the $make $model");
  }
}
```

Part 4: "This" Object (`this` Keyword)

- The Problem: In the constructor above, we have a name conflict.

  String make; (the class property)

  Car(String make, ...) (the constructor parameter)

  If we just write make = make;, the code is confused. It thinks we mean "set the parameter make equal to itself."

- **The Solution:** `this`

- The `this` keyword is a reference to the **current object instance** (the specific object being created).
- `this.make` means "the `make` property belonging to *this object*."
- `make` (by itself) means "the `make` parameter that was just passed into the method."

- **Example (Code):**

```
// Part 4: Constructor and 'this' Keyword
class Car {
  // Properties are now non-nullable because the constructor
  // guarantees they will be initialized.
  String make;
  String model;
  int year;

  // The Parameterized Constructor (using Dart's 'this' sugar)
  // This automatically assigns the parameters to the properties
  // with the same name.
  Car(this.make, this.model, this.year);

  void drive() {
    // 'this' is optional here as there is no ambiguity.
    print("Driving the $make $model");
  }
}

void main() {
  // Now, creation is clean and efficient.
  var myCar = Car("Toyota", "Camry", 2023);
  var neighborsCar = Car("Honda", "Civic", 2021);

  myCar.drive(); // Output: Driving the Toyota Camry
  neighborsCar.drive(); // Output: Driving the Honda Civic
}
```

Part 5: Multiple Ways to Build (Named Constructors)

- **The Problem:** What if you want different ways to create a `Car`?
  - Maybe one way is with `make`, `model`, and `year`.
  - Maybe another way is just for a brand-new car, where the `year` is always the current year.
  - Maybe you want to create a `Car` from data in a database (e.g., a JSON object).
- **The Solution: Named Constructors**
  - These let you create "factory" methods that act as constructors but have descriptive names.
  - The syntax is `ClassName.constructorName()`.
- **Example (Code):**

```
// Part 5: Named Constructors
class Car {
  String make;
  String model;
  int year;

  // 1. The main (default) constructor
  Car(this.make, this.model, this.year);

  // 2. A Named Constructor for just 'make' and 'model'
  // It calls the main constructor using an initializer list (with ': this(...)')
  Car.newModel(String make, String model)
      : this(make, model, 2025); // Assumes 2025 is the current year

  // 3. A Named Constructor for a classic car
  Car.classic(String make, String model)
      : this(make, model, 1970);
```

```
  void drive() {
    print("Driving the $year $make");
  }
}

void main() {
  // Now we have flexible creation options:
  var standardCar = Car("Toyota", "Camry", 2023);
  var newTruck = Car.newModel("Ford", "F-150"); // Year is 2025
  var classicCar = Car.classic("Chevrolet", "Chevelle"); // Year is 1970

  standardCar.drive(); // Output: Driving the 2023 Toyota
  newTruck.drive(); // Output: Driving the 2025 Ford
  classicCar.drive(); // Output: Driving the 1970 Chevrolet
}
```

---

3. Class Exercise (15 minutes)

Goal: Practice everything we just learned.

1. **Create a Class:**
   - Define a new class named `Student`.
2. **Add Properties:**
   - `String name`
   - `String studentID`
   - `double gpa`
3. **Add a Constructor:**
   - Create a parameterized constructor that takes `name`, `studentID`, and `gpa` as arguments.
   - Use the `this` keyword to assign the arguments to the properties.
4. **Add a Method:**
   - Create a method called `printStudentInfo()` that has no return type (`void`).

- Inside this method, print out the student's name, ID, and GPA in a clean format.

5. **Add a Named Constructor:**
   - Create a named constructor called `Student.freshman()` that only takes a `name` and `studentID`.
   - This constructor should call the main constructor, setting the `gpa` to `0.0` by default.
6. **Instantiate and Use:**
   - In your `main` method (or equivalent), create two `Student` objects:
     - One "star student" using the main constructor (e.g., "Jane Doe", "S123", 3.9).
     - One "freshman" using the `Student.freshman()` constructor (e.g., "John Smith", "S124").
   - Call the `printStudentInfo()` method on both objects and check your output.

---

4. Key Takeaways & Review

- **Class:** Blueprint. (e.g., `Car`)
- **Object:** Instance, the real thing. (e.g., `myCar`)
- **Constructor:** Special method to initialize an object when it's created (`new`).
- `this`: Refers to the **current object instance**. Used to solve name conflicts (e.g., `this.name = name`).
- **Named Constructor:** A way to create multiple, descriptively-named constructors (e.g., `Car.newModel()`).