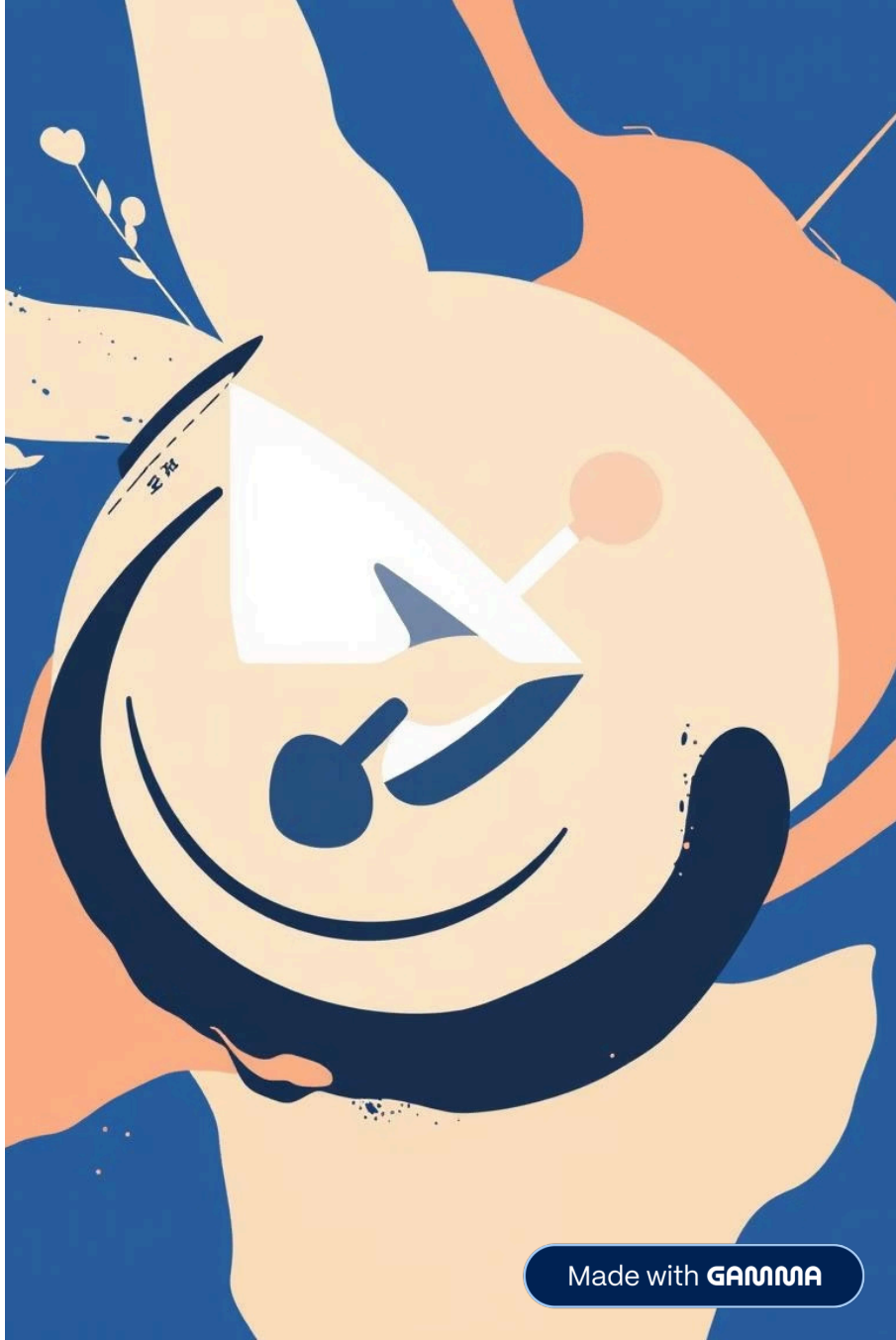
Lec 3 :OOP



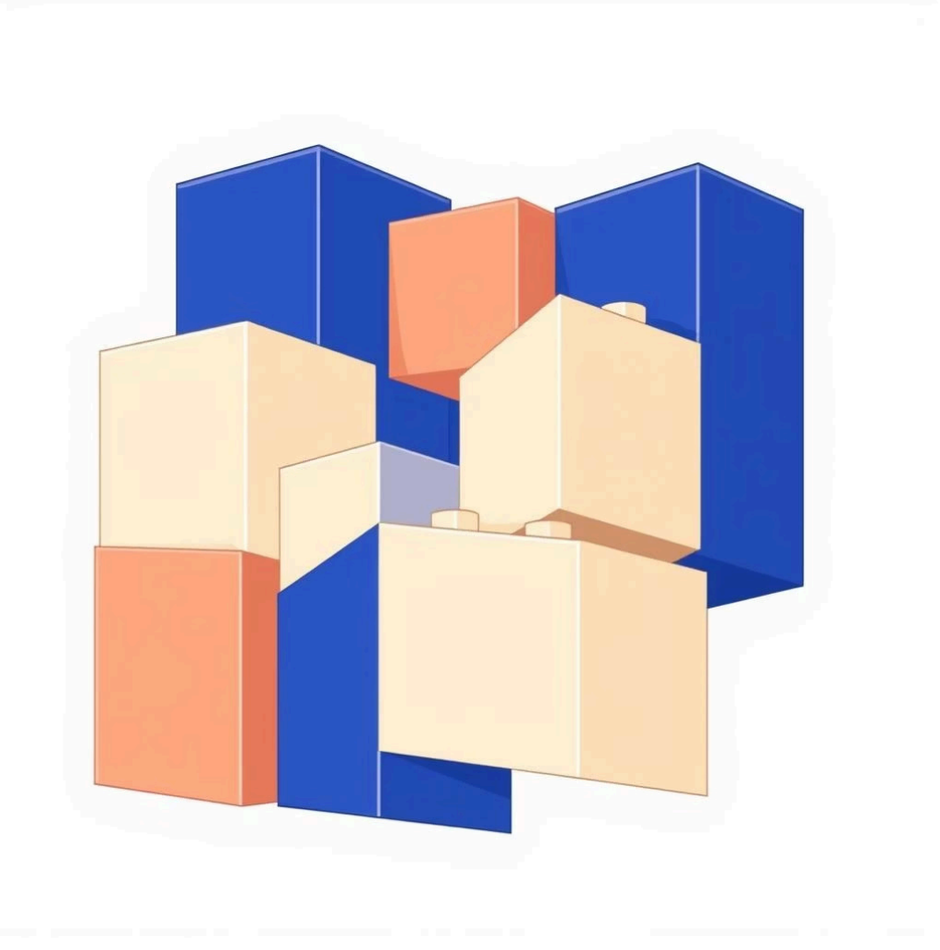
Procedural vs. Object-Oriented Programming

Procedural Programming

- Focuses on step-by-step instructions and procedures.
- Data and functions are separate.
- Less flexible for complex systems.

Object-Oriented Programming

- Organizes code around "objects" (data + behavior).
- Data and functions are bundled together.
- Highly flexible and adaptable for complex projects.



The Pillars of OOP: Key Concepts

Let's explore the core concepts that define Object-Oriented Programming.

Class

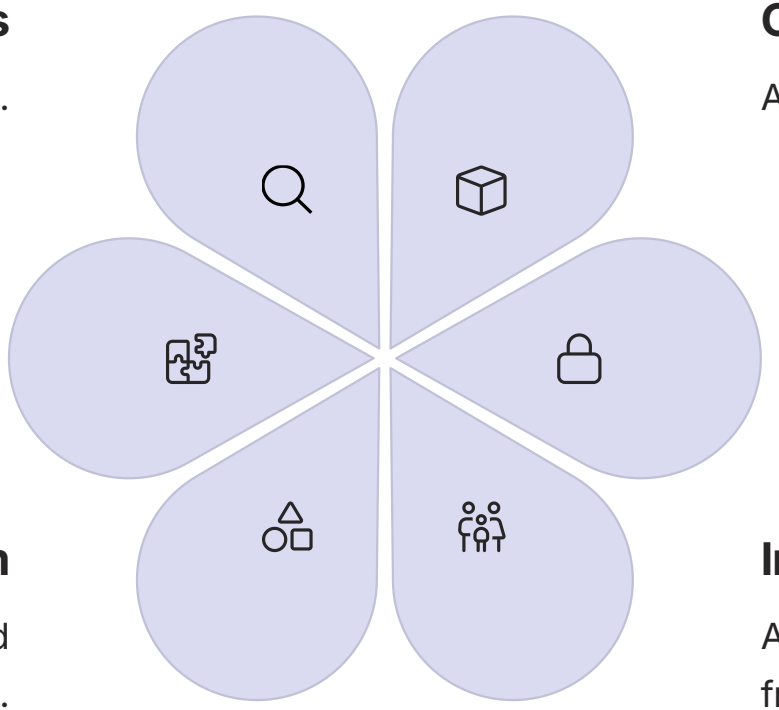
A blueprint for creating objects.

Abstraction

Hiding complex implementation details and showing only essential features.

Polymorphism

Objects of different classes can be treated as objects of a common type.



Object

An instance of a class.

Encapsulation

Bundling data and methods that operate on the data within a single unit.

Inheritance

A class can inherit properties and methods from another class.

Class & Object: The Building Blocks

Class Definition

a template defines the properties(fields_method) .

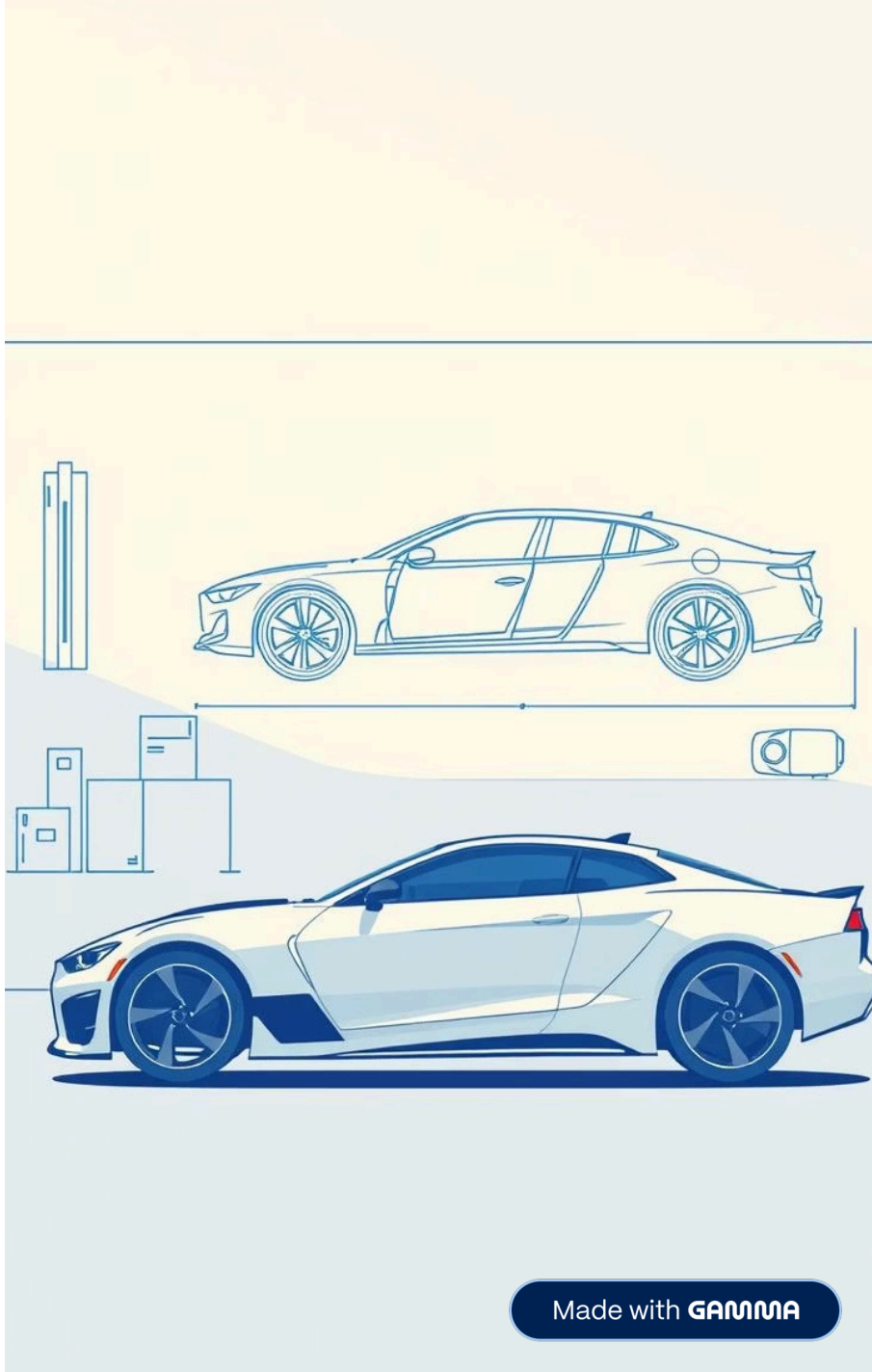
```
class Car {
  String brand;
  String model;
  int year;

  void displayInfo() {
    print('$brand $model ($year)');
  }
}
```

Object Instantiation

An object is an instance of a class. You create objects from classes, and each object will have its own set of data.

```
void main() {
 Car myCar = Car(); // Creating an object
 myCar.brand = 'Toyota';
 myCar.model = 'Camry';
 myCar.year = 2023;
 myCar.displayInfo(); // Output: Toyota Camry (2023)
}
```



Constructors and the `this` Keyword

Constructors are special methods used to initialize objects. The `this` keyword refers to the current instance of the class.

1. **Default Constructor**

   Automatically provided if no other constructor is defined.

2. **Named Constructor**

   Allows for multiple constructors with different purposes.

3. **Optional Parameters**

   Allows for parameters that are not required during object creation.

4. **Required Parameters**

   Ensures certain parameters are always provided when creating an object.

Let's put this into practice with our `Car` class!

Task

Create a `Student` class with the following properties:

- `name` (required)
- `age` (required)
- `grade` (optional, default `"Not Assigned"`)
- `city` (optional, default `"Unknown"`)

Add a method `displayInfo()` that prints the student's information.

Encapsulation: Protecting Your Data

Encapsulation is about bundling the data (attributes) and methods .

"Keep data safe and control access to it." ====»»by validation

Data Hiding

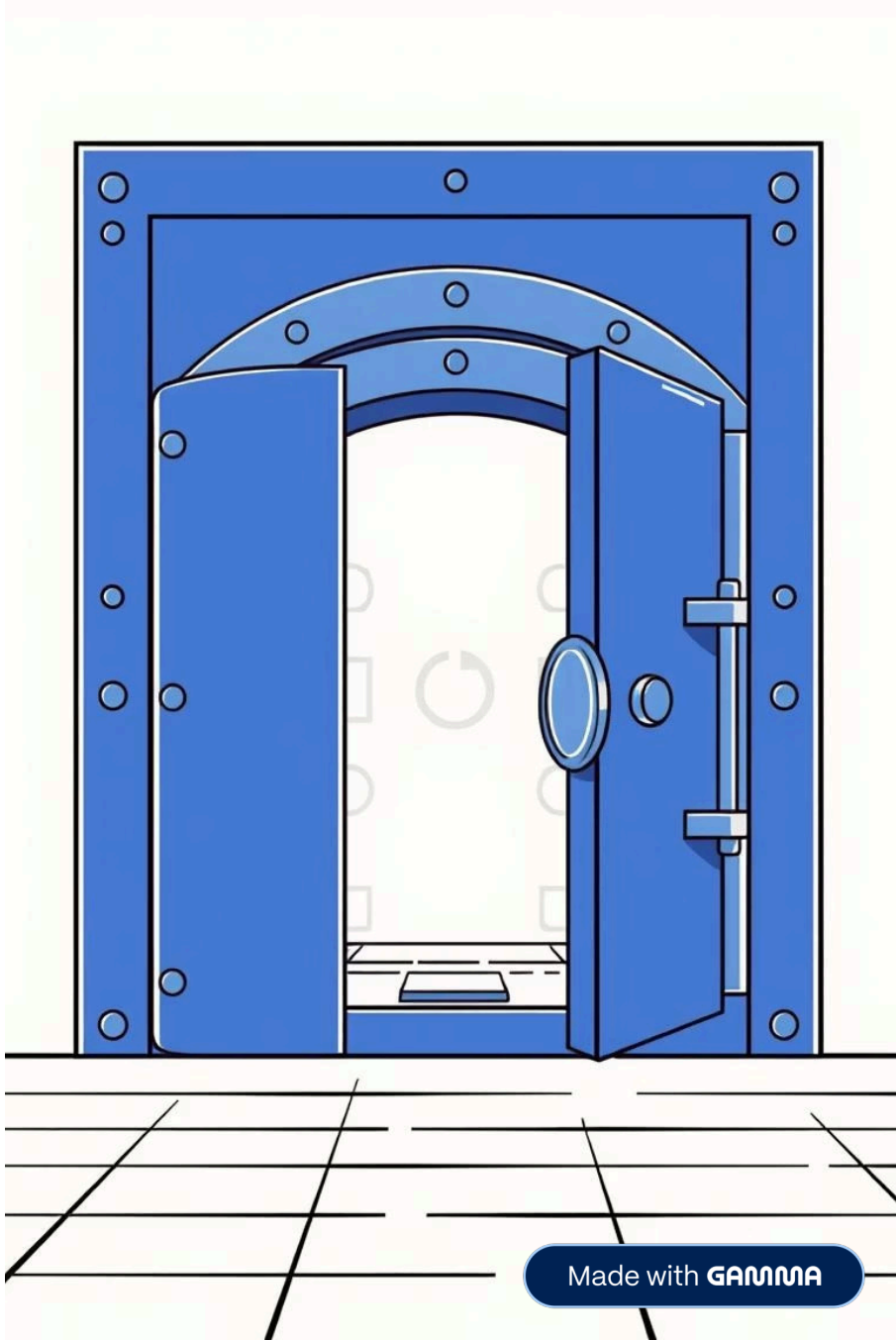
Private attributes (prefixed with _ in Dart) prevent external modification.

Getters

Provide controlled access to read private data.

Setters

Provide controlled access to modify private data, often with validation.



```
class User {
  String _name = "karim";
int _age = 18;
//getter
String get getName => _name;
int get getAge => _age;
//setter
set setName(String name) {
if (name.isNotEmpty) {
_name = name;
    } else {
print('name cant be empty');
    }
  }
set setAge(int age) {
if (age > 0) {
_age = age;
    } else {
print('age cant be negative');
    }
  }
}
```

Task

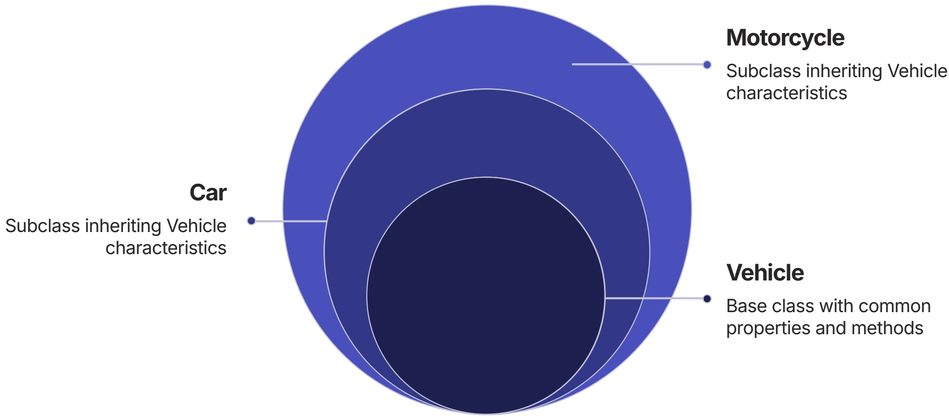
1. Bank Account

- Create a class `BankAccount` with:
  - A private property `_balance`.
  - A method `deposit(amount)` → adds money only if `amount > 0`.
  - A method `withdraw(amount)` → subtracts money only if there is enough balance.
  - A getter to return the current balance.
- **Task:** Try depositing a negative value or withdrawing more than the balance.

Inheritance: Building on Existing Code

allows a new class (subclass or child class) to inherit properties and behaviors from an existing class (superclass or parent class)

code reusability



Practical Example: Vehicle Hierarchy in Dart

Let's demonstrate inheritance with a simple example in Dart, where `Car` and `Motorcycle` classes inherit from a base `Vehicle` class.

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

  Vehicle(this.brand, this.year);

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

  void stop() {
    print('$brand $year is stopping.');
  }
}
```

```
class Car extends Vehicle {
  int numberOfDoors;

  Car(String brand, int year, this.numberOfDoors) : super(brand, year);

  void honk() {
    print('Car horn: Beep! Beep!');
  }

  @override
  void start() {
    print('Car $brand $year is starting its engine.');
  }
}

class Motorcycle extends Vehicle {
  bool hasSidecar;

  Motorcycle(String brand, int year, this.hasSidecar) : super(brand, year);

  void wheelie() {
    print('Motorcycle $brand $year is doing a wheelie!');
  }
}
```

```
void main() {
  Car myCar = Car('Toyota', 2023, 4);
  myCar.start();
  myCar.honk();
  myCar.stop();

  print('');

  Motorcycle myMotorcycle = Motorcycle('Harley-Davidson', 2022, false);
  myMotorcycle.start();
  myMotorcycle.wheelie();
  myMotorcycle.stop();
}
```

Task

1. **Create a Base Class →** `Animal`
   - Properties: `name`, `age`
   - Method: `eat()` → prints:
     `Animal [name] is eating.`
2. **Create a** `Lion` **Class (inherits from** `Animal`**)**
   - Extra property: `isWild` (boolean)
   - Method: `roar()` → prints:
     `Lion [name] is roaring!`
3. **Create an** `Elephant` **Class (inherits from** `Animal`**)**
   - Extra property: `tuskLength` (double or int)
   - Method: `sprayWater()` → prints:
     `Elephant [name] is spraying water with its trunk!`

1. **In the** `main()` **function:**
   - Create one object from each class (`Lion`, `Elephant`).
   - Call `eat()` (inherited from `Animal`) for each.
   - Call their specific methods (`roar`, `sprayWater`, `speak`).

assignment

Library Management System

1. **Classes & Objects**
   - Create a `Book` class with attributes: `title`, `author`, `year`, `availableCopies`.
   - Create a `Student` class with attributes: `name`, `id`, `borrowedBooks`.
   - Each student can borrow multiple books.
2. **Encapsulation**
   - Make the attributes private (e.g., `_title`, `_author`).
   - Provide **getters** and **setters** with validation (e.g., availableCopies cannot be negative).
3. **Inheritance**
   - Create a `User` class (with common attributes: `name`, `id`).
   - `Student` should inherit from `User`.
   - `Librarian` should also inherit from `User` but with different permissions (e.g., adding/removing books).
4. **Constructors (Named / Positional / Optional)**
   - In the `Book` class:
     - Create a **named constructor** that initializes only `title` and `author` (with default values for the rest).
     - Create a **positional constructor** that initializes all attributes.
     - Use **optional parameters** for `year` (it can be provided or not).
5. **System Features (Methods):**
   - A student can `borrowBook()` → decrease the available copies and add the book to the student's borrowed list.
   - A student can `returnBook()` → increase the available copies and remove it from their list.
   - A librarian can `addBook()` or `removeBook()`.
   - Generate a **report** that prints:
     - All available books with the number of copies.
     - All students and the books they currently borrowed.