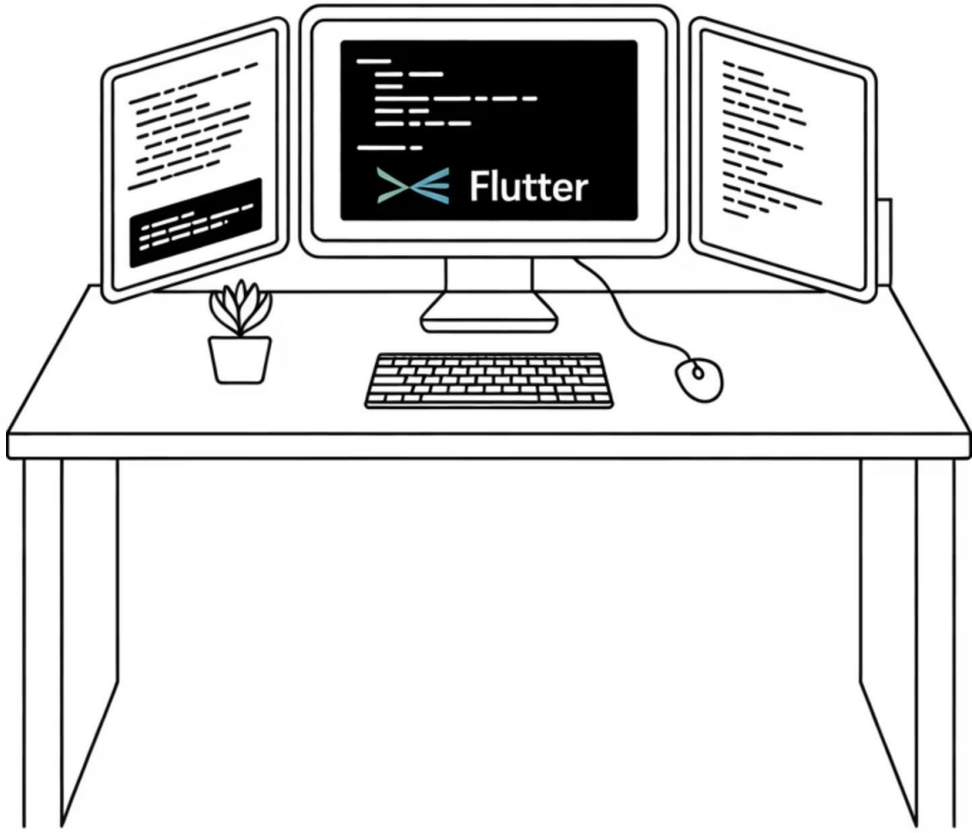
Welcome Everyone!

Class Topics:

- **What is Flutter?**
- **Why do companies choose Flutter?**
- **What is Computer**
- **History of Computer**
- **How a computer works**
- **What is RAM?**
- **What is ROM?**
- এগুলা জেনে কী হবে?

Why Start With Computers?

Before diving into Flutter development, we need to understand the foundation - how computers actually work. This knowledge will make you a better developer and help you avoid common pitfalls.

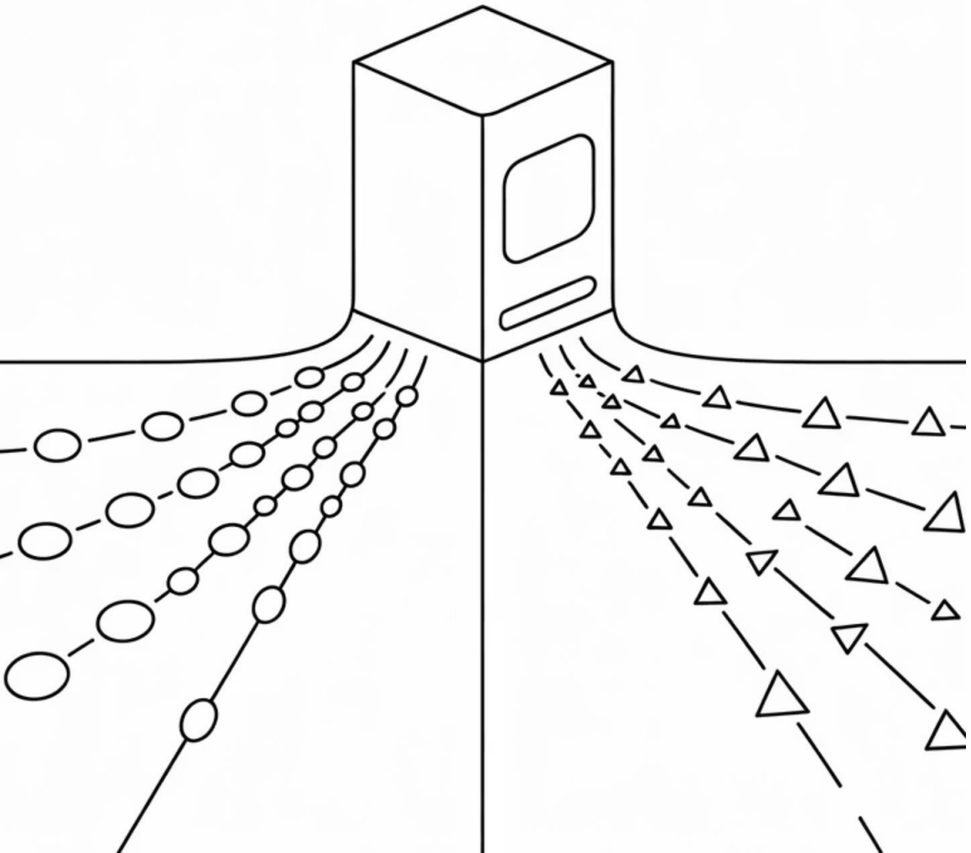


Imagine trying to build a skyscraper without knowing what bricks are. 💡

Same with Flutter – before coding apps, you must know how computers think.

Flutter runs on computers, mobiles, devices – let's peek inside their brains first.

Story: "When I first learned Flutter, I ignored basics. Later, one bug took me 3 days to fix because I didn't understand memory properly. Don't repeat my mistake!"



What is a Computer? (Simple Definition)

1. Accepts input
2. Processes
3. Produces output
4. Stores it

A computer = *A smart servant* (takes orders, gives results).

Not "magic", just fast calculations + storage.

Interactive: Ask students: "If your brain is a computer, what's your RAM? What's your ROM?"

A Quick History of Computers

1st Generation: Big, heavy, vacuum tubes. — 1

2 — 2nd Gen: Transistors made computers smaller.

3rd Gen: Integrated circuits (ICs) – faster, cheaper. — 3

4 — 4th Gen: Microprocessors – personal computers.

5th Gen: Today – AI, smartphones, Flutter apps! — 5

Story: "My dad had a PC with floppy disks. That thing took 5 minutes to start. Now, I open Flutter and my emulator runs in seconds. That's progress!"

Why History Matters for Flutter Devs?

Shows how hardware limitations shaped software.

Explains why we care about performance in apps.

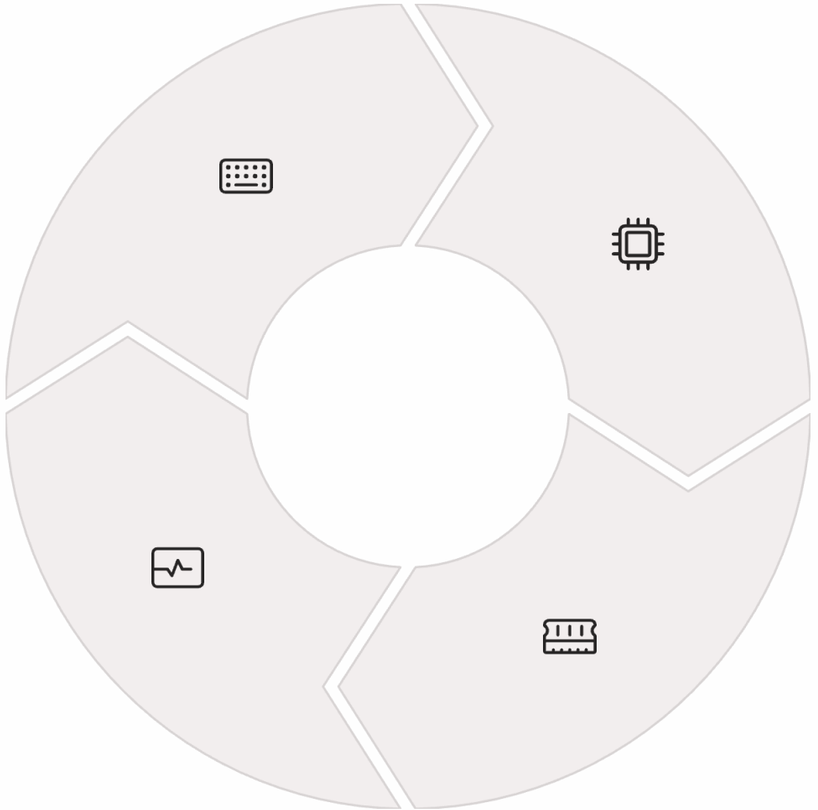
Flutter apps run on small devices → optimization is key.

Example: Old computers had 64KB memory. Today's phones have 8GB RAM. That's why Flutter apps can look smooth like native apps.

How a Computer Works (The Cycle)

Input: Keyboard, touch, voice.

Process: CPU (the brain).



Output: Screen, speakers.

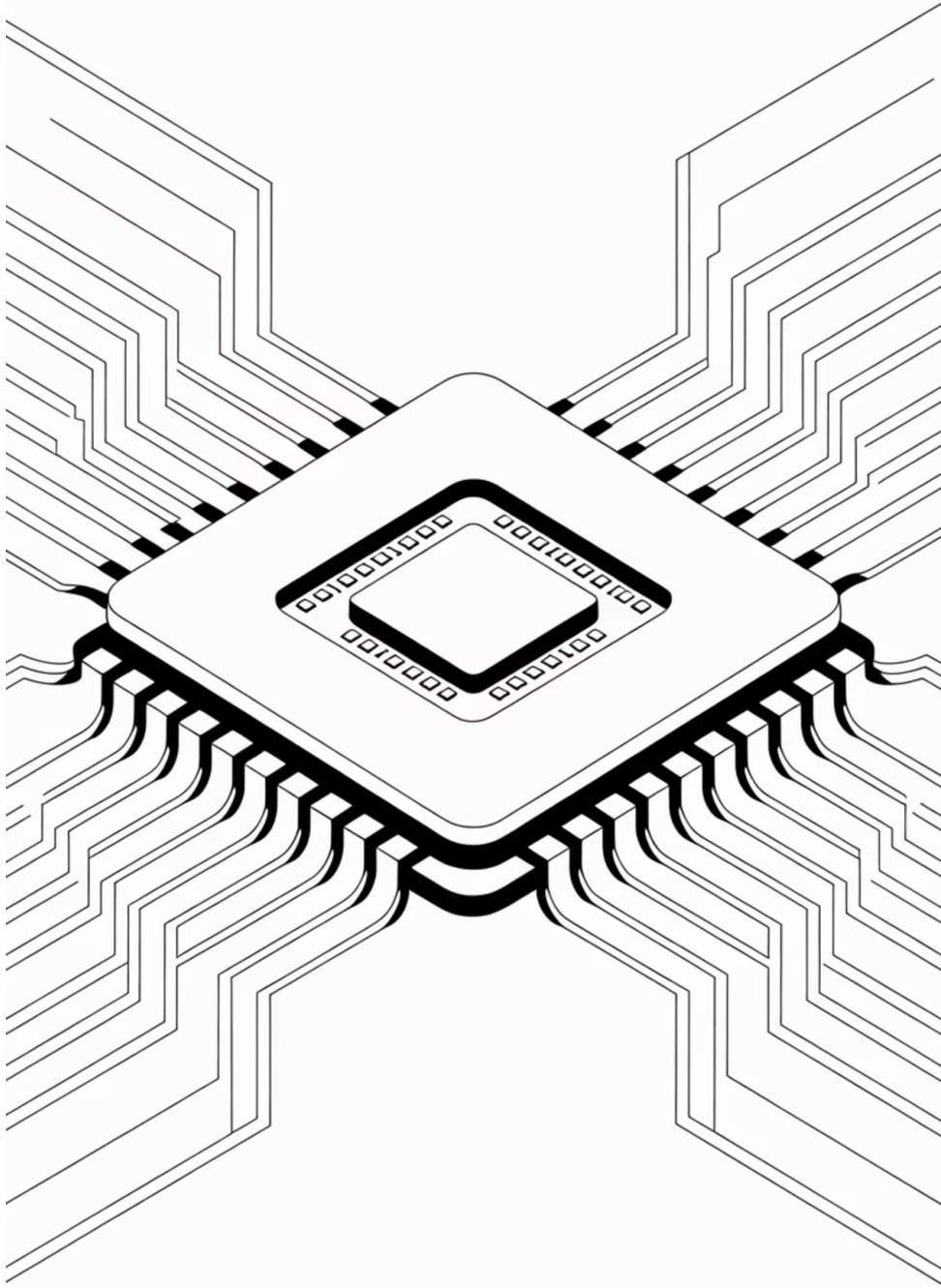
Memory: RAM/ROM.

Interactive: Draw the cycle on board. Ask students: "When you tap a button in Flutter, which part of the cycle is triggered?"

CPU – The Brain of Computer

- Executes instructions line by line.
- Works with RAM closely.
- Speed measured in GHz (billions of steps per second).

Story: "One of my first apps ran super slow. I blamed Flutter. Later I realized, my old laptop's CPU was just too weak. Hardware matters!"



RAM – The Short-Term Memory

Temporary memory, clears after shutdown.

Stores data while app is running.

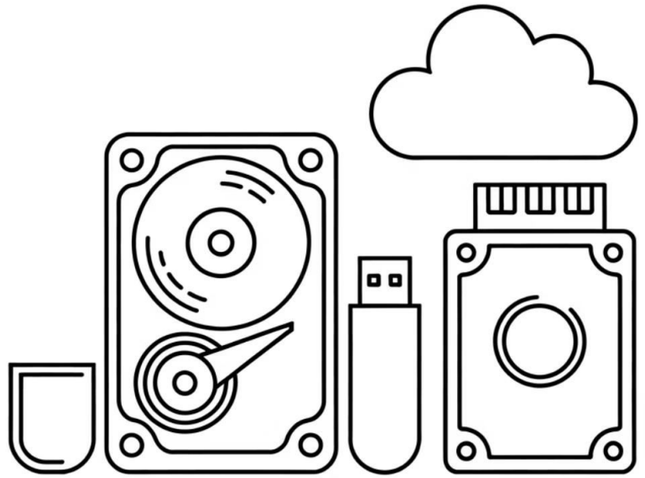
More RAM = more smooth multitasking.

Flutter Relevance: When you open Android Emulator + VS Code + Flutter project → RAM usage spikes. That's why low-RAM PCs lag badly.

ROM – The Long-Term Memory

- Permanent storage (doesn't vanish after restart).
- Stores OS, system files.
- Without ROM → no booting.

Flutter Relevance: When you install Flutter SDK, Android Studio, Emulator → All stored in ROM (your hard drive).



Why RAM & ROM Matter in Flutter

Flutter apps use RAM during hot reloads.

Flutter SDK + Emulator live in ROM.

Knowing this saves you from silly issues.

Example: Students often say "My emulator is not running!" → 90% of time, low RAM or storage issue.

Wrap-Up & Takeaway

- Computer = smart servant (input → process → output).
- History shows why performance matters.
- CPU, RAM, ROM = the real stage where Flutter performs.
- Learn basics → Build apps without fear.

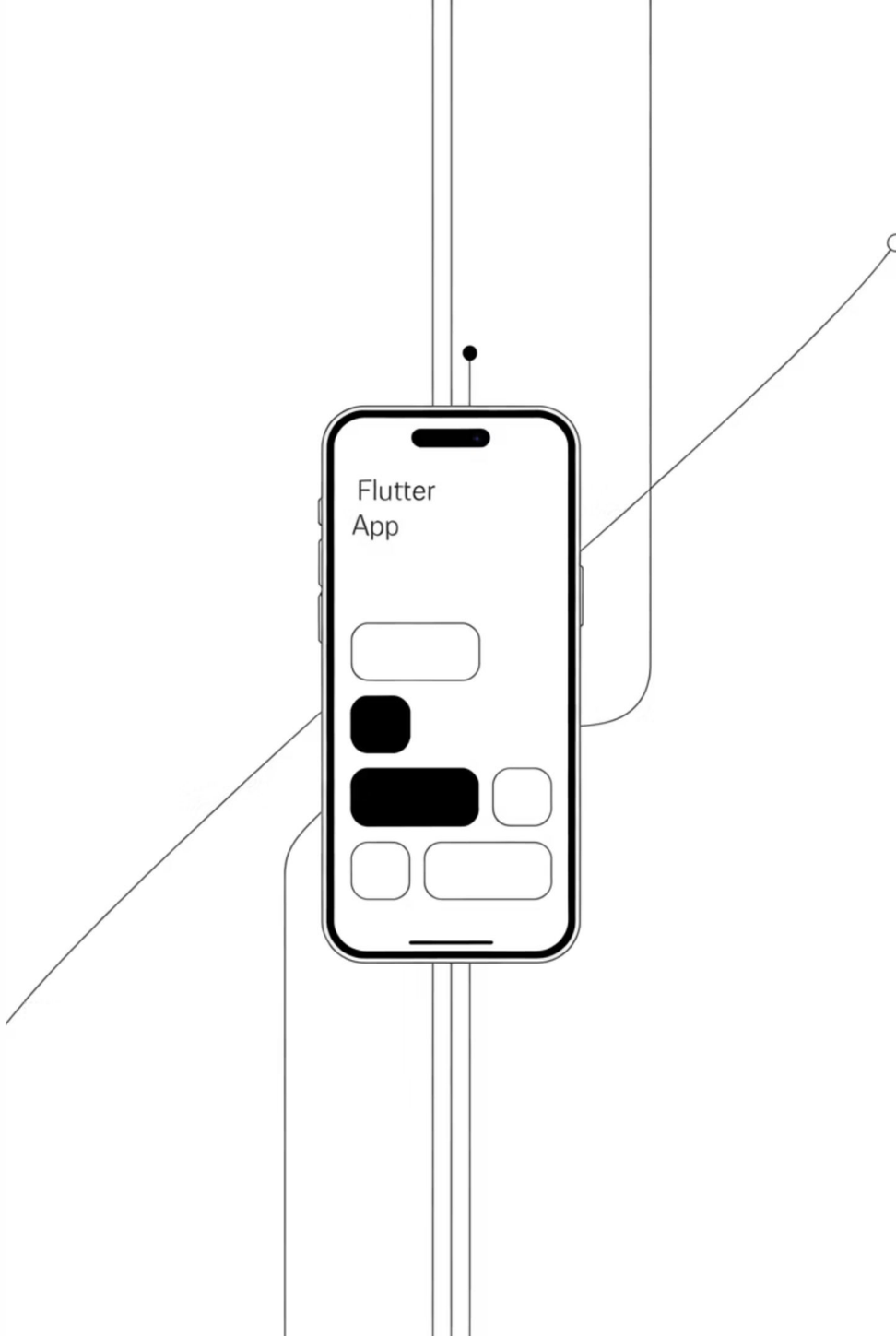
Interactive Ending: Ask: "Next time your emulator crashes, what will you check first: RAM or ROM?"

Class Topics:

- **Basic Networking**
- **Basic Operating System**
- **How a mobile device works**
- এগুলো জেনে কী হবে? কেনো শিখছি?

Why Networking, OS & Mobiles?

Understanding the foundation that powers your Flutter applications



Why Networking, OS & Mobiles?

- You want to build Flutter apps → Where will they run? On mobiles!
- Who controls mobiles? Operating Systems!
- How do apps talk to each other? Networking!

> Story: "When I made my first Flutter chat app, it worked perfectly… on my phone only. Didn't send messages to others. Why? I didn't understand networking basics."

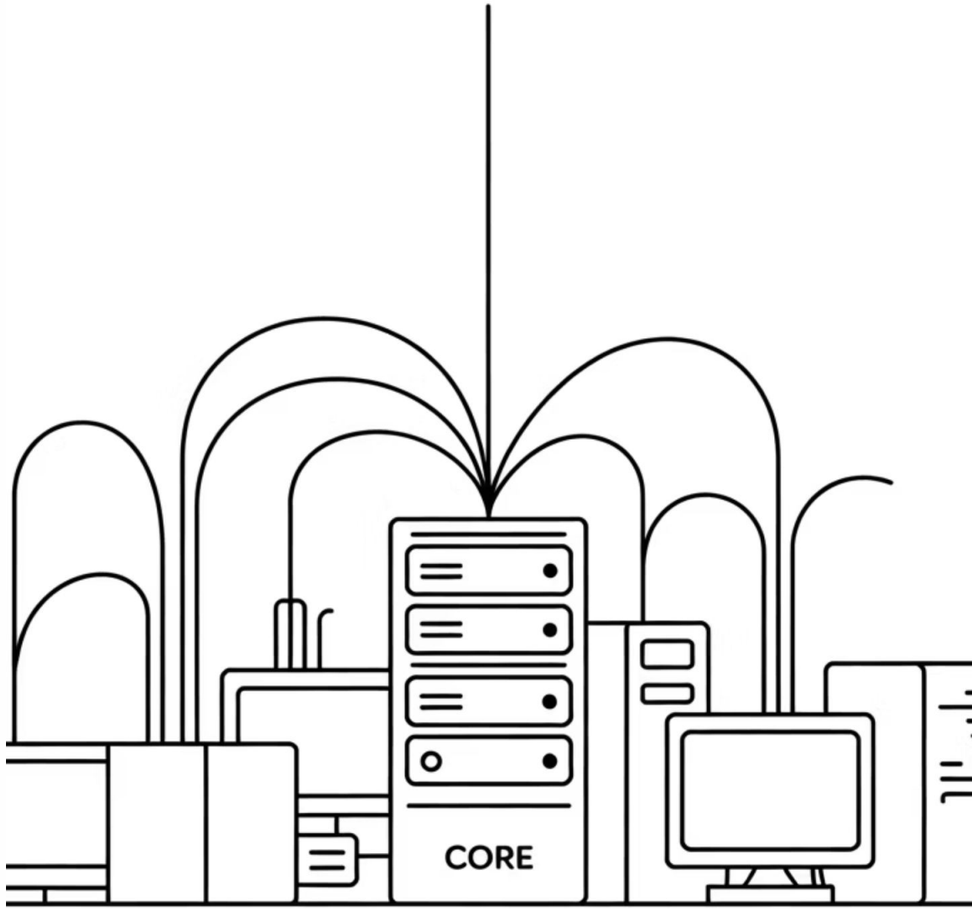
What is Networking (Simple)

Networking = devices talking to each other

Like humans need language, computers need protocols

Example: WhatsApp → You send text, it travels through network → Shows up on friend's phone

Interactive: Ask: "When you press LIKE on Facebook, where does that info go first?"



Networking Basics for Flutter Devs

IP Address

IP address = unique identity (like your house number)

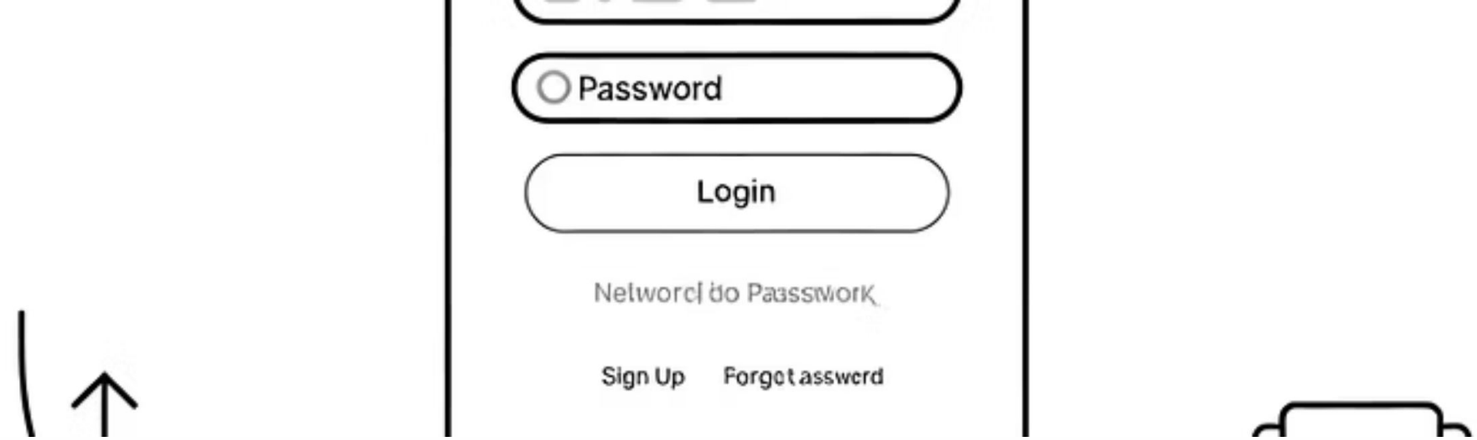
Protocols

Protocols (HTTP, HTTPS) = rules of talking

API

API = the waiter who brings food from kitchen to your table

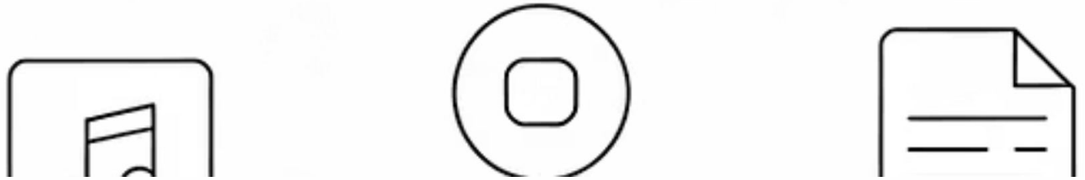
Flutter Relevance: When you call an API in Flutter (http.get), you're literally doing networking.



Real Example – Flutter & Networking

1	2	3
Login Screen	**Server Processing**	**Without Networking**
Flutter sends data to server	Server checks, replies with success/failure	app = offline calculator

> Story: "I once built a weather app in Flutter. It showed only *Dhaka weather*. Why? I hardcoded it. Didn't fetch data from network API. Students laughed at me."



What is an Operating System (OS)?

OS = the manager of your device

Controls hardware, apps, memory

Examples: Android, iOS, Windows

Interactive: Ask: "What's running your Flutter emulator – hardware or OS?"

OS for Flutter Developers

How Flutter Apps Work

- Flutter apps don't run directly on hardware
- They talk with OS → OS controls device

Platform Differences

- iOS has strict rules (App Store)
- Android = more flexible (Google Play)

Flutter Relevance: You must test your Flutter app on both OS because same code may behave differently.

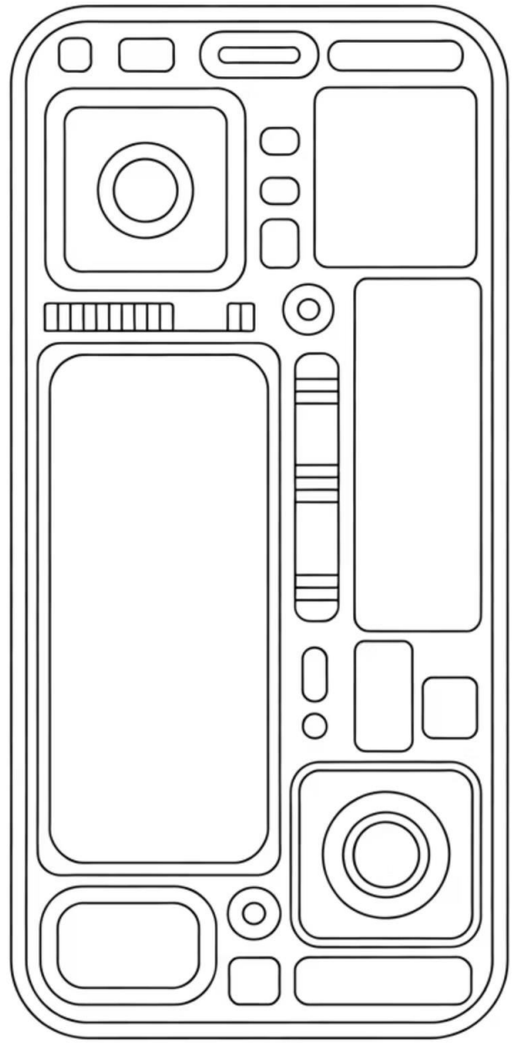
Mobile Device – The Playground for Flutter

Mobile = a small computer in your pocket

Has CPU, RAM, ROM, Battery, Sensors

Works with OS + Network to run apps

Example: Camera app → needs camera hardware + OS permission. Flutter app → same.



How a Mobile Device Works

Input

Touch, mic, sensors

Processing

CPU + GPU

Output

Screen, speaker, vibration

Memory

RAM for running, ROM for storing

Interactive: Ask: *"When you swipe Instagram feed, which part of mobile is working hardest?"* (GPU + Network).

Mobile & Flutter Connection

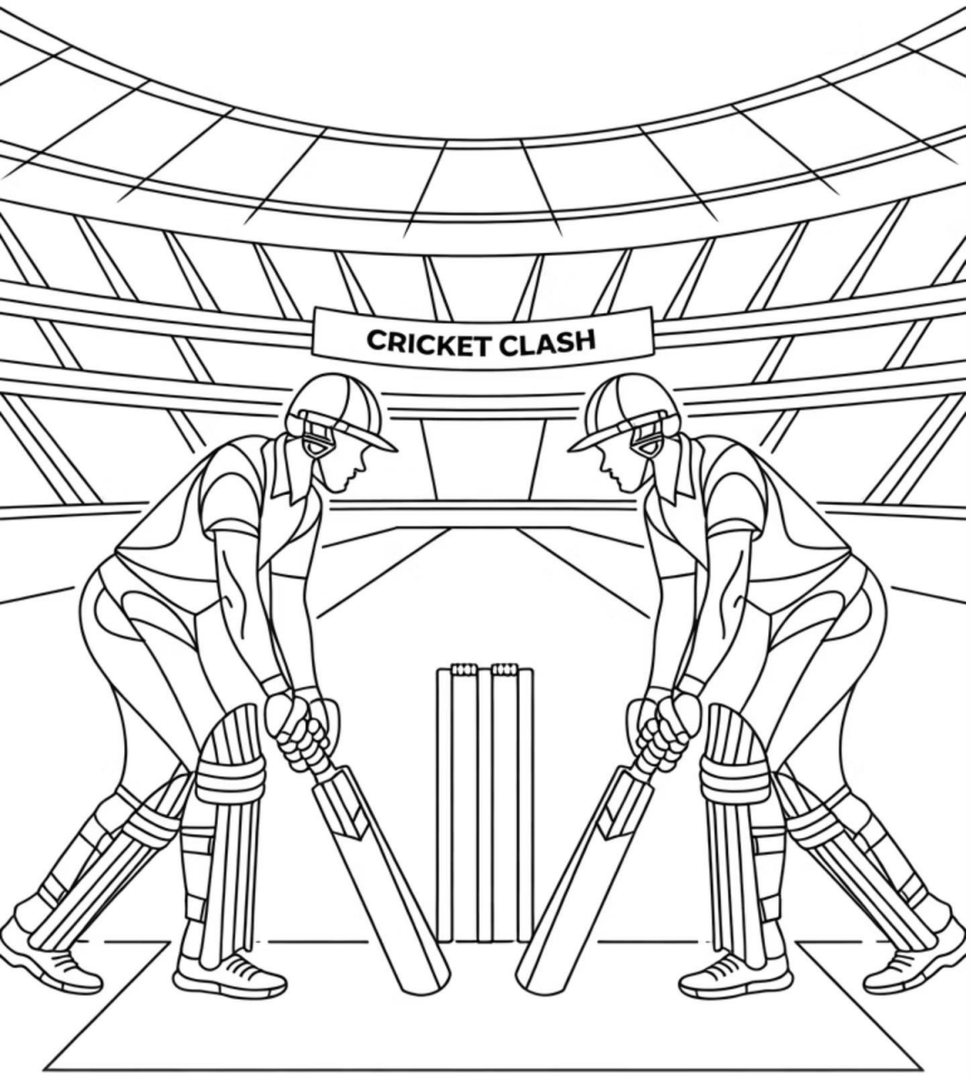
Cross-Platform Power

Flutter is cross-platform: One code → Multiple devices.

But devices have different RAM, OS, battery limits.

That's why developers need to optimize apps.

> Story: "I built an animation-heavy Flutter app. Worked great on my phone. Crashed on my friend's 3GB RAM phone. Lesson: test on low-end mobiles too!"



Android vs iOS War

Think of Android & iOS as two rival cricket teams. Both play the same game → but different styles. As Flutter devs, we must please BOTH audiences.

Story: "My first Flutter app looked perfect on Android… but on iPhone, it looked like an alien invasion. That's when I learned about design languages."

Android OS Basics

- Developed by Google.
- Open source, customizable.
- Runs on thousands of device models.

Flutter Relevance: Your Android Flutter app must run on low-end & high-end phones (Samsung to Symphony).

iOS OS Basics

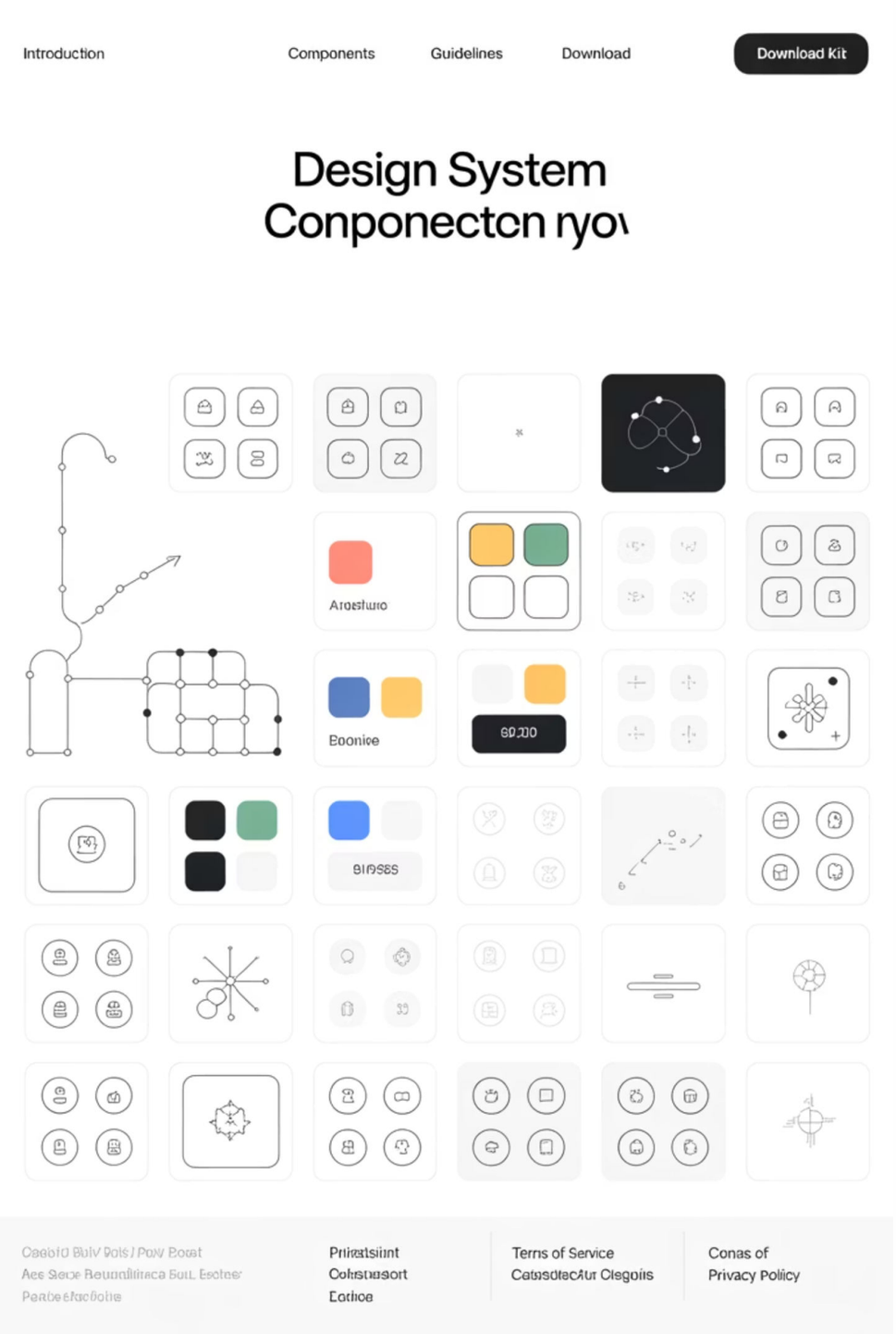
- Developed by Apple.
- Closed, highly controlled.
- Limited devices → iPhones, iPads only.

Flutter Relevance: Your iOS Flutter app must follow Apple's strict design & App Store rules.

Why OS Matters for Flutter Devs

- Flutter writes one codebase.
- But Android & iOS handle UI & performance differently.
- Example: Permissions (camera, storage) → Different flows.

Interactive: Ask: "Who here uses Android? Who uses iPhone? Which feels smoother to you?"



Design Language – What is It?

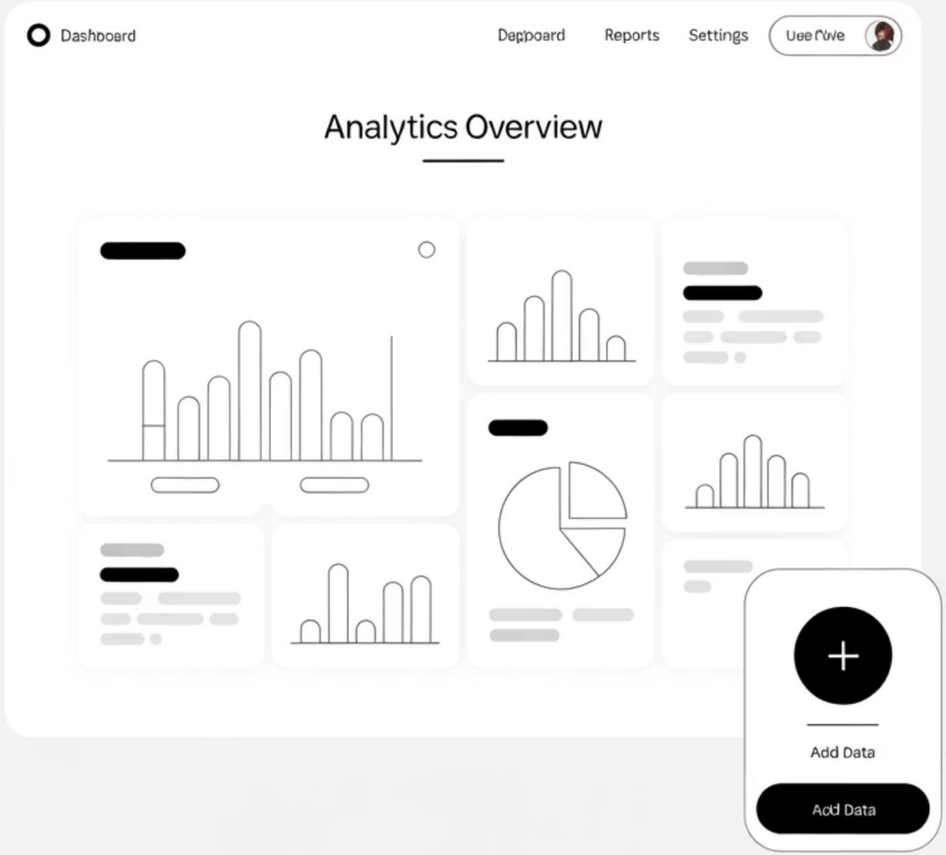
- A set of rules that define how apps "look & feel."
- Colors, buttons, animations, icons, spacing.
- Without design language → apps feel messy & inconsistent.

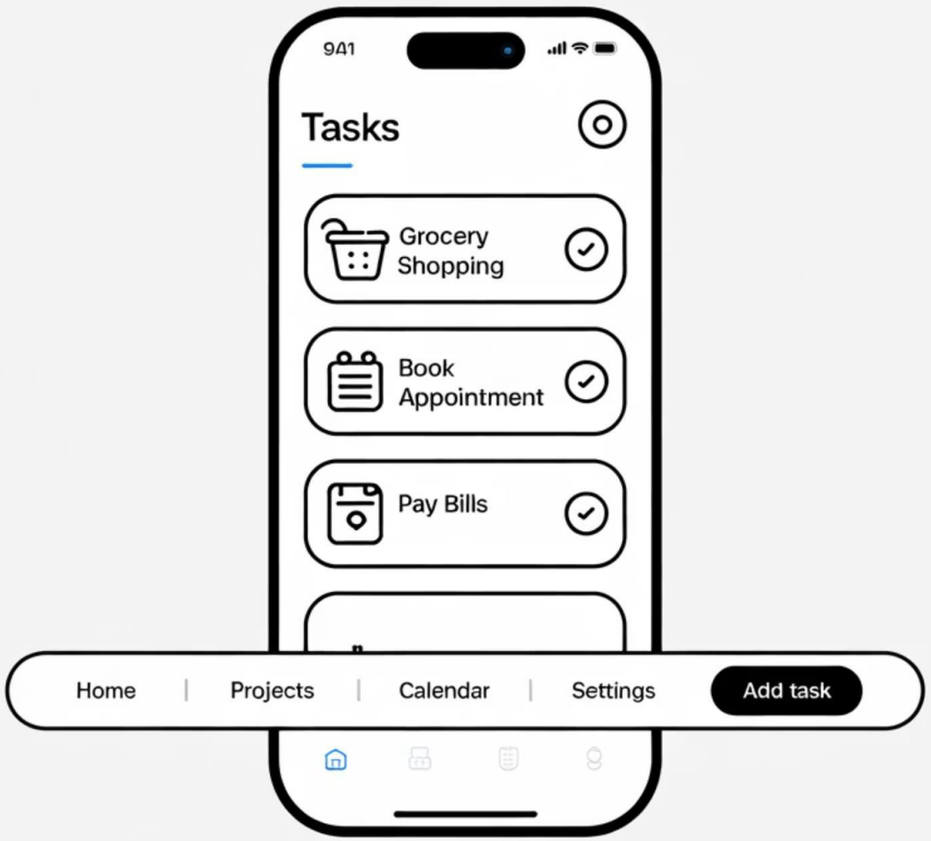
Example: Imagine a Facebook app where buttons look different on Android vs iOS → confusing!

Material Design (Google's Language)

- Born from Android.
- Clean, bold, flat look.
- Floating Action Buttons, Snackbars, Cards.

Flutter Relevance: Most Flutter widgets (Scaffold, AppBar, FAB) are built using Material design.





Cupertino Design (Apple's Language)

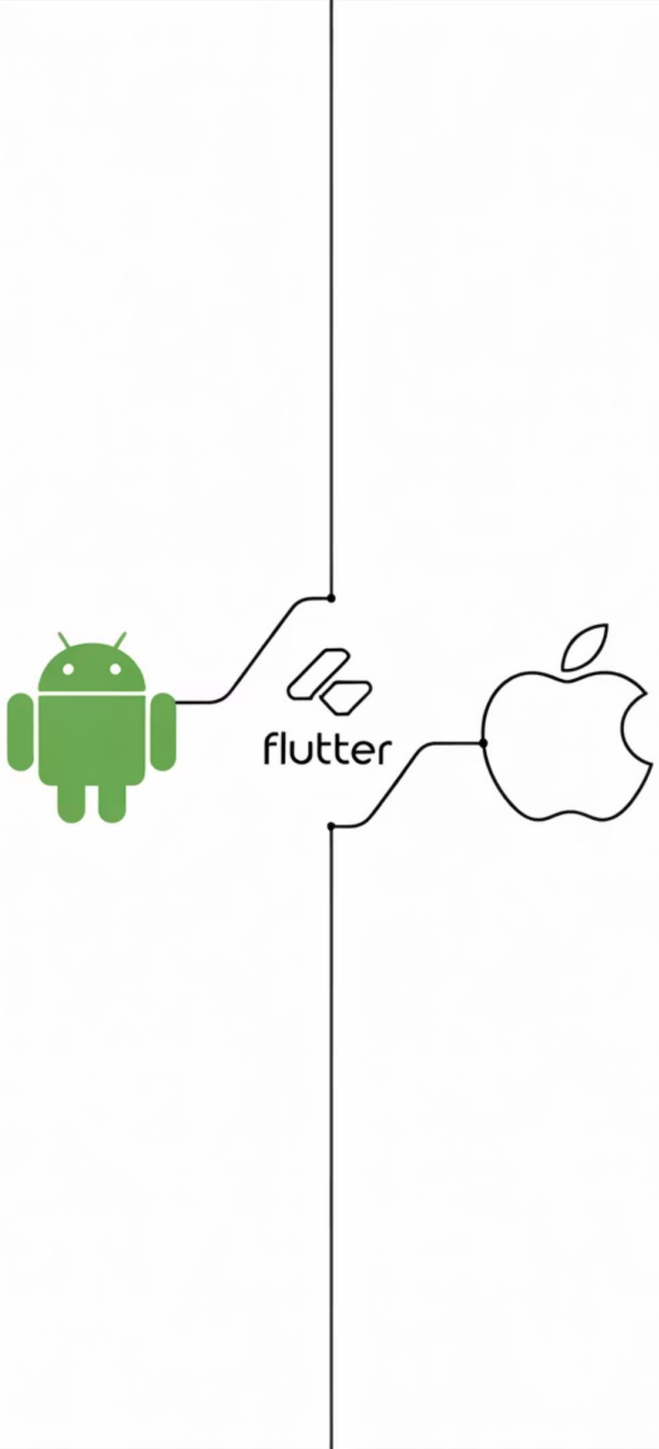
- Born from iOS.
- Smooth, elegant, "glass-like" design.
- Bottom tabs, sliding navigation, switches.

Flutter Relevance: Flutter provides Cupertino widgets (CupertinoButton, CupertinoSwitch, etc.) to make iOS users feel at home.

Material vs Cupertino (Side by Side)

Feature	Material (Android)	Cupertino (iOS)
Buttons	Bold, filled, flat	Rounded, smooth, elegant
Navigation	Drawer, AppBar	Bottom tab bar, gestures
Animation	Fast, snappy	Smooth, elastic

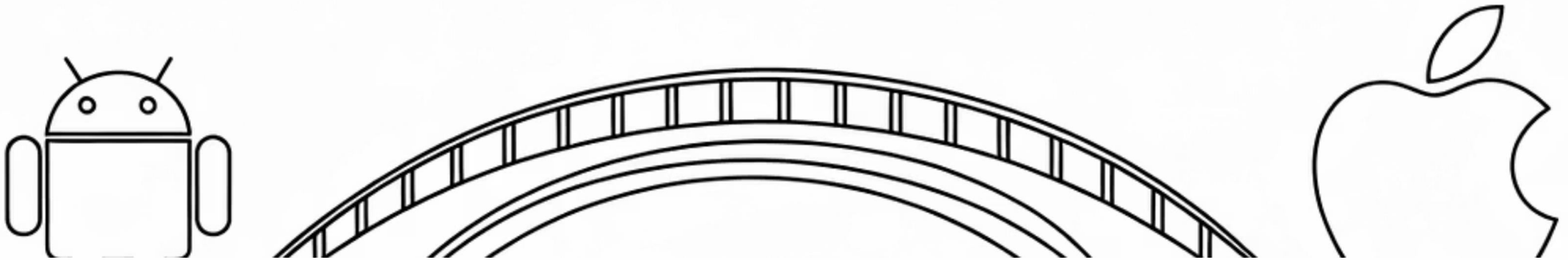
Story: "I once used Material design on an iOS app. My iPhone-using friend said: 'Bro, this feels like a cheap Android copy.' Never again!"



Flutter's Superpower

- With Flutter, you can mix & match.
- MaterialApp → Android look.
- CupertinoApp → iOS look.
- Hybrid? Use both in same app!

Interactive: Ask: "If you build a banking app, would you keep Android bold design or iOS elegant look?"



Wrap-Up & Takeaway

- Android OS → open, customizable, wide audience.
- iOS OS → controlled, premium, strict rules.
- Material Design = Android's style.
- Cupertino = iOS's style.
- As Flutter devs → You're the bridge. One code, two worlds.

Ending Question: *"Next time you design a button in Flutter, will you choose Material or Cupertino?"* 😉

Students Will Clearly See

- Why OS matters (different ecosystems)
- Why design languages matter (user expectations)
- How Flutter handles both with ease

👉 **Students will clearly see:**

- **Why OS matters (different ecosystems).**
- **Why design languages matter (user expectations).**
- **How Flutter handles both with ease.**

Sets: Unique Elements

Ensuring uniqueness of elements
Topics: union, intersection, difference

What is set?

In computer science, a set is defined as a data structure that stores a collection of distinct elements.

Dart has built in support for set. A set in Dart is an unordered collection of unique elements.

```
1  var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine'};
```

Types of Set

There are two types of sets:

1. Unordered Set
2. Ordered Set

Dart doesn’t support ordered set.

Declaring Set

There are two ways to declare set:

```
// Method 1
var variable_name = <variable_type>{};

// Method 2
Set <variable_type> variable_name = {};
```

Set Declaration Styles

```dart
// Method 1: Using 'var'
var fruits = <String>{};        // Empty Set of Strings
fruits.add('Apple');
fruits.add('Banana');
fruits.add('Apple');            // Duplicate ignored

print(fruits); // {Apple, Banana}

// Method 2: Using 'Set<Type>'
Set<String> usernames = {};     // Empty Set of Strings
usernames.add('alice');
usernames.add('bob');
usernames.add('alice');         // Duplicate ignored

print(usernames); // {alice, bob}
```

Set Operations in Dart

- Dart Set supports common mathematical operations:
  - **union()** → combine sets
  - **intersection()** → common elements
  - **difference()** → elements in one but not the other
- Useful for filtering, merging, or comparing data.

Set Operation: Union

Definition: Combines all unique elements from both sets.

```
var a = {'apple', 'banana'};
var b = {'banana', 'orange'};
print(a.union(b)); // {apple, banana, orange}
```

Merges both sets, removing duplicates.

Set Operation: Intersection

Definition: Returns elements common to both sets.

```
var a = {'apple', 'banana'};
var b = {'banana', 'orange'};
print(a.intersection(b)); // {banana}
```

Keeps only matching elements.

Set Operation: Difference

Definition: Returns elements in one set but not in the other.

```
var a = {'apple', 'banana'};
var b = {'banana', 'orange'};
print(a.difference(b)); // {apple}
```

Filters out elements present in the second set.

Real-World Use Cases of Sets

- Useful for working with unique data
- Helps in filtering, comparison, and analytics
- Common in apps, websites, and backend systems

Real-World Use Cases of Union – Merging Data

- Combine users from multiple platforms (e.g., app + web)a
- Merge tags or categories from different sources
- Gather unique IDs from multiple lists

Real-World Use Cases of Intersection – Finding Common Items

- Find mutual friends between users
- Identify common interests or skills
- Detect shared access between roles or teams

Real-World Use Cases of Difference – Filtering Out Data

- Find users exclusive to one platform
- Remove already processed items from a new batch
- Identify missing permissions or roles

Closing

- Set operations make data handling simpler and faster
- Ideal for unique data, comparisons, and analytics
- Think of:
  - Union → combine
  - Intersection → common
  - Difference → exclude

Maps: Key-Value Pairs

Storing data as key-value associations
Topics: Nested Maps | Methods: keys, values, entries, forEach | Collection if/spread operators

What are Maps?

In programming, **Maps** (also known as dictionaries, hash maps, or associative arrays depending on the language) are **collections that store data in key–value pairs**.

Each **key** is unique and maps to a **specific value**.

- Think of it like a real-world **dictionary**: the *word* is the key, and the *definition* is the value.
- You use the key to find or update the corresponding value quickly.

```
In a map, each element is a key-value pair. Each key within a pair is associated with a value, and both keys and values can be any type of object. Each key can occur only once, although the same value can be associated with multiple different keys. Dart support for maps is provided by map literals and the Map type.
```

Why Maps Were Needed?

- Early programs relied on **arrays and lists** → could only use integer indexes.
- Real-world data often needs lookup by **names**, **IDs**, or **labels**, not just numbers.
- Linear search through lists was **too slow** for large datasets.
- Need arose for:
    - Fast lookup using flexible keys
    - Better memory usage for sparse data
    - Cleaner abstraction for key-value relationships.

      Motivation: Efficient *key* → *value* retrieval beyond arrays.

Key Characteristics of Map

- Each key is unique.
- Values can be duplicated.
- Fast lookup and insertion.
- Keys can be of many types (e.g., strings, numbers, objects — depending on the language).

```
var gifts = {
  // Key:    Value
  'first': 'partridge',
  'second': 'turtledoves',
  'fifth': 'golden rings',
};

var nobleGases = {2: 'helium', 10: 'neon', 18: 'argon'};
```

Example of Map

```dart
void main() {
  // Creating a Map
  var user = {
    'name': 'Alice',
    'age': 25,
    'country': 'Bangladesh'
  };

  // Accessing values
  print(user['name']);      // Output: Alice

  // Adding or updating values
  user['age'] = 26;

  // Adding a new key-value pair
  user['email'] = 'alice@example.com';

  print(user);
}
```

Common Map Operations — Overview

- Maps let us store and retrieve values efficiently using **keys**.
- Common operations make Maps flexible for everyday use:
  - Lookup
  - Add or Update
  - Check Existence
  - Remove
  - Iterate Keys/Values

Core Operations

- `map[key]` → **Get** value by key
  Example: `user['name'] → "Alice"`
- `map[key] = value` → **Add or update** a key-value pair
  Example: `user['age'] = 26`
- `map.containsKey(key)` → **Check** if a key exists
  Example: `user.containsKey('email')` → `true / false`

Managing and Iterating Data

- `map.remove(key)` → **Remove** a key-value pair
  Example: `user.remove('email')`
- `map.keys` → Access all keys
  Example: `['name', 'age']`
- `map.values` → Access all values
  Example: `['Alice', 26]`
- `map.entries` → Access all entries (key–value pairs)
  Example: `(name: Alice, age: 26)`
- `map.forEach` → Loop through key-value pairs
  Example: `user.forEach((key, value) => ...);`

Nested Map

A **"nested map"** refers to a data structure where t**he value associated with a key in a map is itself another map**. This creates a **hierarchical** or **multi-level** organization of data.

```
var users = {
  'user1': {
    'name': 'Alice',
    'age': 26,
  },
  'user2': {
    'name': 'Bob',
    'age': 30,
  },
};

print(users['user1']?['name']); //Output: Alice
```

Collection If & Spread Operators - 1/2

Collection If

- Allows **conditional elements** inside collections (List, Set, Map).
- Useful for cleaner code without manual if-else wrapping.

```
var isAdmin = true;
var users = [
  'Alice',
  if (isAdmin) 'Bob', // Added only if condition is true
];

print(users); // ['Alice', 'Bob']
```

Collection If & Spread Operators - 2/2

Spread Operator (... and ...?)

- Allows inserting multiple elements from another collection.
- ...? handles null collections safely.

```
var base = {'name': 'Alice'};
var extra = {'age': 26};

var user = {
  ...base,
  ...?extra,
};

print(user); // {name: Alice, age: 26}
```

Real World Use Cases of Maps

- **User Profiles / Settings**
  - Store user info like name, age, preferences, roles, etc.
  - Example: `{ 'name': 'Alice', 'theme': 'dark' }`
- **Configuration & Environment Variables**
  - Key-value pairs for system or app settings.
  - Example: `{ 'API_URL': '...', 'MODE': 'production' }`
- **JSON / API Response Handling**
  - Maps are perfect for parsing structured JSON data.
  - Example: API → `Map<String, dynamic>` in Dart.

Regular Dot vs Cascade Operator

Dot (.) = "Do this, then work with the result"
Cascade (..) = "Do this to the same object, then do more"

Problem Statement

```dart
void main() {
  List<String> cities = ['New York', 'London', 'Tokyo', 'Paris', 'Sydney'];

  print('\nCombining where and map:');
  var result1 = cities
    ..where((city) => city.length > 5) // filter
    ..map((city) => city.toUpperCase())
    ..toList();

  print('  $result1'); //   [New York, London, Tokyo, Paris, Sydney]

  var result2 = cities
      .where((city) => city.length > 5) // filter
      .map((city) => city.toUpperCase())
      .toList();

  print('  $result2'); // [NEW YORK, LONDON, SYDNEY]
}
```

Regular Dot Operator (.) - 1/2

What it does:

- **Returns the result** of each method call
- Enables **method chaining** on different objects
- Each method operates on the **return value** of the previous method

Regular Dot Operator (.) - 2/2

Flow:

- `cities.where()` → Returns filtered Iterable
- `.map()` operates on the Iterable (not cities)
- `.toList()` operates on the mapped Iterable

```
List<String> cities = ['New York', 'London', 'Tokyo', 'Paris', 'Sydney'];

var result = cities
    .where((city) => city.length > 5)  // Returns: Iterable
    .map((city) => city.toUpperCase()) // Returns: Iterable
    .toList();                         // Returns: List

print(result); // Output: [NEW YORK, SYDNEY]
```

Cascade Operator (..) - 1/2

What it does:

- **Returns the original object**, not the method result
- Allows multiple operations on the **same object**
- Useful for object configuration/initialization

Cascade Operator (..) - 2/2

```
class Button {
  String? text;
  String? color;
  Function? onClick;
}

// Using cascade operator
var button = Button()
  ..text = 'Click me'
  ..color = 'Blue'
  ..onClick = () => print('Clicked!');

// Equivalent to:
var button = Button();
button.text = 'Click me';
button.color = 'Blue';
button.onClick = () => print('Clicked!');
```

What Happens with Wrong Usage? - 1/2

Using Cascade Where Regular Dot is Needed:

```dart
List<String> cities = ['New York', 'London', 'Tokyo', 'Paris', 'Sydney'];

var result = cities
    ..where((city) => city.length > 5)  // Called on cities
    ..map((city) => city.toUpperCase()) // Called on cities
    ..toList();                         // Called on cities

print(result);
// Output: [New York, London, Tokyo, Paris, Sydney]
// The ORIGINAL list! Transformations were ignored!
```

What Happens with Wrong Usage? - 2/2

Why it fails:

1. `cities..where()` → executes but **returns cities**
2. `cities..map()` → executes but **returns cities**
3. `cities..toList()` → tries to call on List (may error)
4. result = original cities list (unchanged!)

Quick Reference & Best Practices

Use Regular Dot (.) when:

- Chaining transformations on different return values
- Working with streams, iterables, futures
- Each step produces a new object to work with

Example: `list.where().map().toList()`

Use Cascade Operator (..) when:

- Setting multiple properties on the same object
- Calling multiple methods on the same object
- Building/configuring objects

Example: `object..prop1 = x..prop2 = y..method()`

Control Flow

If-else statements | Nested if-else conditions | Switch-case basic usage | Pattern Matching in Dart 3

What is Control Flow?

In programming, control flow refers to the order in which individual statements, instructions, or function calls are executed in a program.

By default, most programs run from top to bottom, but control flow lets you change or control this order based on conditions, loops, or function calls.

Types of Control Flow

1. Sequential Flow
   The program runs line by line in the order it's written.
2. Conditional Flow (Decision Making)
   The program chooses a path based on a condition.
3. Looping Flow (Repeating Actions)
   The program repeats a block of code multiple times.
4. Branching / Jump Flow (Changing Flow Explicitly)
   Using statements like break, continue, return to jump or skip parts of the program.
5. Function Calls / Invocations
   The flow moves to a function, executes it, then comes back to where it was called.

Importance of Control Flow

- Control flow allows decision making in programs.
- Automates repetitive tasks.
- Makes code dynamic and responsive instead of running linearly.
- Without control flow, all programs would behave the same way every time.

Control Flow with Conditions in Dart

- Control flow allows your program to make decisions
- Common ways to handle conditional logic:
  - if-else statements
  - Nested if-else
  - switch-case
  - Pattern Matching (Dart 3+)

If-Else Statements

Used to execute code based on a condition

```
if (age >= 18) {
  print('Adult');
} else {
  print('Minor');
}
```

Nested If-Else

Used when there are multiple conditions

```
if (score >= 90) {
  print('A');
} else if (score >= 80) {
  print('B');
} else {
  print('C or below');
}
```

Switch-Case Statements

Cleaner alternative to multiple if-else

```
switch (day) {
  case 'Mon':
    print('Start of the week');
    break;
  case 'Fri':
    print('Almost weekend');
    break;
  default:
    print('Regular day');
}
```

Pattern Matching (Dart 3)

Modern way to handle conditions and extract values

```
switch (user) {
  case {'role': 'admin', 'active': true}:
    print('Welcome Admin');
  case {'role': 'user'}:
    print('Welcome User');
  default:
    print('Unknown role');
}
```

What is Pattern Matching?

- **Definition**:
  Pattern matching is a way to check a value against a pattern and extract data from it in a clean and readable way.
- Key Idea:
  - Instead of writing long if-else chains or switch statements, you match a structure directly.
  - The pattern itself declares what to check and what to extract.

Pattern Matching Way

```
var user = {'name': 'Alice', 'age': 22};
switch (user) {
  case {'name': var n, 'age': var a}:
    print('Name: $n, Age: $a');
}
```

Regular Way (If-else)

```
var user = {'name': 'Alice', 'age': 22};
if (user.containsKey('name') && user.containsKey('age')) {
  var name = user['name'];
  var age = user['age'];
  print('Name: $name, Age: $age');
}
```

What is Destructuring?

- Destructuring means breaking down an object or data structure into its parts.
- When a pattern matches, it can automatically bind the matched values to variables.
- Makes it easy to access nested values directly.

```
switch (user) {
  case User(name: var n, age: var a):
    print('Name: $n, Age: $a');
}
```

What are Guards?

- A guard adds an extra condition to a pattern match.
- Useful when structure matches, but value needs to satisfy additional rules.
- Keeps logic clean without nested if blocks.

```
switch (user) {
  case User(age: var a) when a >= 18:
    print('Adult');
  case User(age: var a):
    print('Minor');
}
```

Control Flow

For loop | While loop | Do-while loop

Introduction to Loops

A loop is a programming construct that allows repeating a block of code multiple times.

Useful for:

- Repeating a task
- Iterating through lists or data
- Automating repetitive operations

Common types of loops:

- for loop
- while loop
- do-while loop

For Loop – Definition

Used when the number of iterations is **known or predictable**.

Syntax:

```
for (initialization; condition; increment) {
  // code block
}
```

Key points:

- Initialization runs once at the beginning
- Condition is checked before each iteration
- Increment executes after each iteration

For Loop – Example

```
for (int i = 1; i <= 5; i++) {
  print('Iteration $i');
}

/// Output
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

- i starts at 1
- Loop runs while i <= 5
- i increments by 1 after each iteration

While Loop – Definition

Used when the number of iterations is not fixed and depends on a condition.

Syntax:

```
while (condition) {
  // code block
}
```

Key points:

- Condition is checked before each iteration
- If the condition is false initially, the loop body may not execute at all

While Loop – Example

```
int i = 1;
while (i <= 5) {
  print('Iteration $i');
  i++;
}
/// Output
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

- The condition is evaluated at the start of each iteration
- Increment happens inside the loop

Do-While Loop – Definition

Used when the loop must execute at least once, even if the condition is false.

Syntax:

```
do {
  // code block
} while (condition);
```

Key points:

- Loop body runs first
- Condition is checked after each iteration

Do-While Loop – Example

```
int i = 1;
do {
  print('Iteration $i');
  i++;
} while (i <= 5);
/// Output
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

- Code runs once before checking the condition
- If the condition is false, it will not repeat

Real World Use Cases

- **for loop**: Iterating over a list of products, printing numbers, pagination.
- **while loop**: Waiting for user input, monitoring a sensor value.
- **do-while loop**: Showing a menu at least once, password verification loop.

Key Takeaways

- Choose the loop type based on when you want the condition to be checked.
- Ensure the condition will eventually become false to avoid infinite loops.
- Loops make code more efficient and cleaner by avoiding repetition.

Control Flow

For-in loop for collections | Break and Continue usage | Nested loops (multiplication table)

Iterating Through Collections with for-in

Cleaner and more readable than a traditional for loop. Ideal when you don’t need the index of each element.

Syntax:

```
for (var element in collection) {
  // use element
}
```

Key points:

- The for-in loop is used to iterate over iterable collections like List, Set, or Map.values.
- It automatically goes through each element in sequence.

Controlling Loop Flow with break and continue

- **`break`** → Immediately exits the loop.
- **`continue`** → Skips to the next iteration of the loop.

Example:

```
void main() {
  for (var i = 1; i <= 5; i++) {
    if (i == 3) continue; // skip number 3
    if (i == 5) break; // stop the loop
    print(i);
  }
} // Output: 1 2 4
```

Using Nested Loops to Generate a Table - 1/2

- A nested loop is a loop inside another loop.
- Commonly used for tables or combinations (e.g., multiplication table).

Example:

```
void main() {
  for (var i = 1; i <= 3; i++) {
    for (var j = 1; j <= 3; j++) {
      print('$i × $j = ${i * j}');
    }
    print('---');
  }
}
```

Using Nested Loops to Generate a Table - 2/2

- The outer loop controls rows, and the inner loop controls columns.
- Useful for working with grids, charts, and matrices.

Example:

```
1 × 1 = 1
1 × 2 = 2
1 × 3 = 3
---
2 × 1 = 2
2 × 2 = 4
2 × 3 = 6
---
3 × 1 = 3
3 × 2 = 6
3 × 3 = 9
---
```

Functions

Function declaration and invocation | Return types in functions | Arrow functions (=>)

What is Function in Programming?

- A function is a block of code designed to perform a specific task.
- It can take inputs (parameters), process them, and return an output.
- Functions help to break down complex problems into smaller, reusable parts.
- They are also known as methods, procedures, or subroutines in some languages.

```
function greet() {
  print("Hello, World!");
}
```

Why Are Functions Needed?

- **Reusability:** Write once, use multiple times.
- **Organization:** Keeps code modular and easier to read.
- **Debugging:** Errors are easier to find in smaller code blocks.
- **Maintainability:** Updating logic in one place updates all usages.
- **Team Collaboration:** Different team members can work on separate functions.

Example:
Instead of repeating the same calculation multiple times, define a single function and reuse it.

What is a Function in Dart?

- In Dart, a function is an object that represents an operation or behavior.
- You can store a function in a variable, pass it as a parameter, or return it from another function.
- Functions are defined using the void or return type, followed by a name and parameter list.

Example:

```dart
void greetUser(String name) {
  print('Hello, $name!');
}
```

Types of Functions in Dart

- **Built-in Functions:** Provided by Dart (e.g., `print()`, `main()`).
- **User-defined Functions:** Written by developers to perform specific tasks.
- **Anonymous Functions (Lambdas):** Functions without a name.
- **Arrow Functions:** Short-hand syntax for single-expression functions.
- **Higher-Order Functions:** Take functions as parameters or return functions.

Function Declaration and Invocation

Declaration: Writing a function.
Invocation (Call): Using the function to execute it.

Example:

```
// Declaration
void sayHello() {
  print('Hello!');
}

// Invocation
sayHello();
```

Return Types in Functions

- A function can **return a value** using the `return` keyword.
- The return type must match the declared type.
- If nothing is returned, use `void`.

Example:

```
int add(int a, int b) {
  return a + b;
}

void showMessage() {
  print('Task completed');
}
```

Arrow Functions (=>)

- Used for **short, single-line** functions.
- Syntax: `returnType functionName(parameters) => expression;`
- Automatically returns the expression result.

Example:

```dart
int add(int a, int b) => a + b;

void greet(String name) => print('Hi $name!');
```

Summary

- Functions organize code into **reusable blocks**.
- Dart supports **multiple types**: user-defined, anonymous, arrow, and more.
- **Return types** define output, while **arrow syntax** simplifies short functions.
- Functions make programs **modular**, **readable**, and **efficient**.

Functions

Positional parameters | Named parameters (required, default values) | Optional parameters

What Are Parameters?

- **Parameters** are **placeholders** used in a function to receive values when it's called.
- The values you pass during function call are called **arguments**.
- Parameters allow functions to **work with dynamic data** instead of fixed values.

```
void greet(String name) {
  print('Hello, $name!');
}

greet('Momshad'); // Output: Hello, Momshad!
```

Positional Parameters

- These are **parameters passed** in a **specific order**.
- The **order and number** of arguments must match during function call.
- Most common type of parameters in Dart.

```dart
void printInfo(String name, int age) {
  print('Name: $name, Age: $age');
}

printInfo('Alice', 25); // ✅ Correct
printInfo(25, 'Alice'); // ❌ Wrong order
```

Named Parameters

- You **label parameters by name** when calling the function.
- Makes code **more readable** and **order-independent**.
- Defined using curly braces `{}` in the function definition.

```dart
void registerUser({String? name, int? age}) {
  print('Name: $name, Age: $age');
}

registerUser(name: 'Bob', age: 22);
registerUser(age: 22, name: 'Bob'); // Order doesn't matter
```

Required Named Parameters

- By default, named parameters are **optional** and **nullable**.
- To make them **mandatory**, use the `required` keyword.
- Helps prevent missing values at compile-time.

```dart
void login({required String username, required String password}) {
  print('Welcome, $username');
}

login(username: 'admin', password: '1234'); // ✅
login(username: 'admin'); // ❌ Missing required parameter
```

Default Values in Named Parameters

- You can assign **default values** to parameters.
- If the caller doesn’t pass a value, the default one is used.

```dart
void greet({String name = 'Guest'}) {
  print('Hello, $name!');
}

greet(); // Output: Hello, Guest!
greet(name: 'Momshad'); // Output: Hello, Momshad!
```

Optional Positional Parameters

- Wrapped in **square brackets** `[]`.
- Can be skipped, but **must come after required positional parameters**.
- Useful for flexible function calls.

```dart
void showDetails(String name, [int? age]) {
  print('Name: $name, Age: ${age ?? "Unknown"}');
}

showDetails('Alice'); // Age skipped
showDetails('Alice', 25); // Age provided
```

Advanced Functions

Anonymous functions | Higher-order functions (passing functions as arguments) | Recursive functions | Scope → local vs global variables

Anonymous Functions 1/3

- Also called **Lambda** or **Inline** functions.
- A function **without a name** — defined where it’s used.
- Often used for **short, one-time tasks**.
- Can be stored in variables or passed as arguments.

```
(parameters) {
  // body
}
```

Anonymous Functions 2/3

```
void main() {
  var greet = (String name) {
    print('Hello, $name!');
  };

  greet('Momshad');
}
```

Anonymous Functions 3/3

Anonymous functions are commonly used with collection methods like `.forEach()`.

```
void main() {
  var numbers = [1, 2, 3];

  numbers.forEach((num) {
    print('Number: $num');
  });
}
```

Higher-Order Functions 1/3

- A Higher-Order Function (HOF) is a function that:
  - Takes another function as a parameter, OR
  - Returns a function as a result.
- Helps with code reuse, callbacks, and functional programming.

Higher-Order Functions 2/3

```dart
void main() {
  executeTask(printMessage);
}

void executeTask(Function task) {
  task();
}

void printMessage() {
  print('Task executed successfully!');
}
```

Higher-Order Function with Anonymous Function 3/3

You can also pass anonymous functions directly.

```
void main() {
  performAction(() {
    print('Performing an inline action!');
  });
}

void performAction(Function action) {
  action();
}
```

Recursive Functions 1/2

- A recursive function calls itself.
- Commonly used for problems like: Factorials, Fibonacci series, Tree traversal

```
int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}

void main() {
  print('Factorial of 5 is ${factorial(5)}');
} // Output: Factorial of 5 is 120
```

Recursive Functions 2/2

Fibonacci Sequence

```
int fibonacci(int n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

void main() {
  for (var i = 0; i < 6; i++) {
    print(fibonacci(i));
  }
} // Output: 0, 1, 1, 2, 3, 5
```

Scope in Dart

- Scope defines where a variable can be accessed.
- Two main types:
  - Global scope – Declared outside any function/class.
  - Local scope – Declared inside a function or block.

```dart
String globalVar = 'I am global';

void main() {
  String localVar = 'I am local';
  print(globalVar); // ✅ Accessible
  print(localVar);  // ✅ Accessible
}

void anotherFunc() {
  print(globalVar); // ✅ Accessible
  // print(localVar); ❌ Error: not in scope
}
```

Variable Shadowing

- A local variable can hide (shadow) a global variable with the same name.
- The nearest scope wins.

```
String message = 'Global Message';

void main() {
  String message = 'Local Message';
  print(message); // Output: Local Message
}
```

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()`).

Inheritance

Class	Inheritance
Topcis	- Inheritance with `extends` - `super` keyword for parent class access - Method overriding

1. Lesson Objectives

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

- **Explain** the concept of inheritance and its "is-a" relationship.
- **Use** the `extends` keyword to create a subclass (child class).
- **Understand** the difference between a superclass (parent) and a subclass (child).
- **Call** the parent class constructor using the `super` keyword.
- **Override** parent class methods using the `@override` annotation.
- **Access** parent class methods from a child class using `super`.

---

2. Core Concepts

The central theme of this lesson is **code reusability**. We want to avoid duplicating code, and inheritance is a primary way to achieve this.

Analogy: Think of a general blueprint for an "Animal." This blueprint defines that all animals have an `age` and can `eat()`. Instead of creating separate, new blueprints for "Dog" and "Cat" from scratch, we can just *extend* the "Animal" blueprint. The "Dog" blueprint *inherits* everything from "Animal" and adds a specific `bark()` method.

A `Dog` **"is-a"** `Animal`. This "is-a" relationship is the test for when to use inheritance.

Part 1: The `extends` Keyword (Creating the "is-a" link)

- **Inheritance** allows one class (the **subclass** or **child class**) to inherit the properties and methods of another class (the **superclass** or **parent class**).
- We use the `extends` keyword in Dart.
- **Example (Parent Class):**

```
// The Superclass (Parent)
class Animal {
  String name;

  Animal(this.name); // Constructor

  void eat() {
    print('$name is eating.');
  }

  void makeSound() {
    print('Animal makes a sound.');
  }
}
```

- **Example (Child Class):**

```
// The Subclass (Child)
class Dog extends Animal {
  // 'Dog' now has 'name', 'eat()', and 'makeSound()' automatically!

  // We'll fix this constructor error in the next step.
}

// In our main function:
var myDog = Dog('Buddy'); // Error: 'Animal' has a constructor, so 'Dog' m
ust call it.
myDog.eat(); // This *would* work if we could create the object.
```

Part 2: The `super` Keyword (Calling the Parent Constructor)

- **The Problem:** The `Animal` class needs a `name` to be created. When we create a `Dog`, we are *also* creating an `Animal` inside it. We must tell Dart how to initialize that parent `Animal` part.
- **The Solution:** We use `super()` in the child's constructor initializer list to pass the required data up to the parent's constructor.
- **Example (Fixed):**

```
class Dog extends Animal {

  // This constructor takes a 'name'
  // and passes it 'up' to the Animal constructor using 'super(name)'.
  Dog(String name) : super(name);

}

// In our main function:
var myDog = Dog('Buddy');
myDog.eat(); // Output: Buddy is eating.
```

- Now, `myDog` is an object that is both a `Dog` and an `Animal`. It has access to all public methods from `Animal`.

Part 3: Method Overriding (Changing Parent Behavior)

- **The Problem:** `myDog.makeSound()` would print "Animal makes a sound." This isn't specific enough. We want a `Dog` to "Woof."
- **The Solution:** We **override** the `makeSound` method. This means we provide a new implementation for a method that already exists in the parent.
- In Dart, we **must** use the `@override` annotation. This is a safety check—it tells the compiler "I am *intentionally* replacing a parent method." If you misspell the method, the compiler will give you an error.
- **Example (Code):**

```
class Dog extends Animal {

  Dog(String name) : super(name);

  // We are providing a new version of 'makeSound'
  @override
  void makeSound() {
    print('Woof! Woof!');
  }
}

class Cat extends Animal {

  Cat(String name) : super(name);

  @override
  void makeSound() {
    print('Meow.');
  }
}

// In our main function:
var myDog = Dog('Buddy');
var myCat = Cat('Whiskers');

myDog.makeSound(); // Output: Woof! Woof!
myCat.makeSound(); // Output: Meow.
```

Part 4: Accessing the Parent Method (Using `super.` somewhere else)

- **The Problem:** What if we don't want to *replace* the parent's method, but just *add to it*?
- **Example:** Let's say `Animal.eat()` handles the logic for hunger. We want the `Dog` to do that, *and also* wag its tail.

- **The Solution:** We can use `super.methodName()` to call the parent's version of the method from *inside* the child's overridden method.
- **Example (Code):**

```
class Dog extends Animal {

  Dog(String name) : super(name);

  @override
  void makeSound() {
    print('Woof! Woof!');
  }

  // Let's override eat()
  @override
  void eat() {
    super.eat(); // This calls the 'eat()' method from the Animal class
    print('$name wags its tail.');
  }
}

// In our main function:
var myDog = Dog('Buddy');
myDog.eat();
// Output:
// Buddy is eating.
// Buddy wags its tail.
```

---

3. In-Class Exercise (15 minutes)

Goal: Build a simple hierarchy for `Vehicle` and `Car`.

1. **Create the Superclass (Parent):**
   - Create a class named `Vehicle`.
   - Give it two properties: `String make` and `String model`.

- Create a constructor that initializes these properties: `Vehicle(this.make, this.model)`.
- Create a method: `void drive()` that prints "Driving the $make $model."

2. **Create the Subclass (Child):**
   - Create a class named `Car` that `extends Vehicle`.
   - Create a constructor for `Car` that takes `make` and `model`.
   - Use `super()` to pass these values to the `Vehicle` constructor.
3. **Test It:**
   - In `main()`, create a `Car` object: `var myCar = Car('Toyota', 'Corolla');`
   - Call `myCar.drive()`. It should print "Driving the Toyota Corolla."
4. **Override a Method:**
   - In the `Car` class, add a new property: `int numberOfDoors = 4`.
   - `@override` the `drive()` method.
   - Make the new `drive()` method call the *parent's* version first (using `super.drive()`).
   - After that, make it print "This car has $numberOfDoors doors."
5. **Final Test:**
   - Run `myCar.drive()` again. The output should now be:

     ```
     Driving the Toyota Corolla.
     This car has 4 doors.
     ```

---

4. Key Takeaways & Review

- **Inheritance:** Creates an **"is-a"** relationship (a `Car` *is a* `Vehicle`).
- **`extends`:** The keyword used to create a subclass.
- **`super()`:** Used in the child's **constructor** to call the parent's constructor.
- **`@override`:** An annotation that *must* be used when replacing a parent's method. It's a vital safety check.
- **`super.method()`:** Used *inside* a method to call the parent's version of that method.

- **Main Benefit: Code Reusability**. We write common code once in the parent and reuse it in all child classes.

Abstraction and Polymorphism

≡ Topcis	- Abstract classes and abstract methods, - Implementing Interfaces, - Polymorphism basics, - Real-world modeling with classes (Car, Dog, BankAccount)

Class Content: Abstraction & Polymorphism

1. Lesson Objectives

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

- **Define** Abstraction: Hiding complex implementation details and showing only functionality.
- **Create** `abstract` classes and methods to define templates for subclasses.
- **Understand** the difference between `extends` (Inheritance) and `implements` (Interfaces).
- **Implement** an Interface to force a class to adhere to a specific "contract."
- **Apply** Polymorphism to treat different objects (e.g., `Car`, `Boat`) as a single type (e.g., `Vehicle`).
- **Model** a real-world scenario using a `BankAccount` system.

---

2. Core Concepts (Lecture & Demo)

Part 1: Abstraction (Separating the “what” from the “how”)

Abstraction allows you to hide complex implementation details and expose only the necessary features of an object. We achieve this in two ways:

A. Abstract Classes (extends)

An abstract class serves as a base hierarchy. It represents a conceptual "is-a" relationship.

- **Cannot be instantiated:** You cannot write `new Employee()`.
- **Partial Implementation:** Can contain both abstract methods (no body) and concrete methods (with body).
- **State:** Can maintain internal state (fields/variables) that children inherit.
- **Constructor:** Can have constructors to initialize that state.
- **Use Case:** When creating a family of related classes that share common logic but need specific implementations for certain behaviors.

```
// Abstract Base Class
abstract class Employee {
  String name;
  String id;

  Employee(this.name, this.id);

  // Concrete Method: Shared logic for all employees
  void clockIn() {
    print('$name ($id) clocked in at ${DateTime.now()}');
  }

  // Abstract Method: Specific logic per role (must be overridden)
  double calculateSalary();
}

class Developer extends Employee {
  double hourlyRate;
  int hoursWorked;

  Developer(String name, String id, this.hourlyRate, this.hoursWorked)
      : super(name, id);

  @override
  double calculateSalary() {
    return hourlyRate * hoursWorked;
```

```
  }
}
```

B. Interfaces (implements)

Unlike Java or C#, Dart does not have an interface keyword. Every class is implicitly an interface.

- **Contract Enforcement:** When you use `implements`, you must override *every* public field and method.
- **No Inheritance:** You do not inherit code or logic from the parent; you only inherit the "shape" (signatures).
- **Multiple Implementation:** A class can implement multiple interfaces.
- **Use Case:** When unrelated classes need to share a capability (a **"can-do"** relationship).

From Dart 3.0, `interface` keyword was introduced. To achieve a pure interface, you need to combine it with `abstract` and use `abstract interface` before `class`

```
// Acts as an Interface
class Logger {
  void log(String message) {
    print('Default logging: $message');
  }
}

class Database {
  void connect() {}
}

// Implementation
// 'implements' forces us to redefine log(), ignoring the code in Logger
class FileLogger implements Logger {
```

```
  @override
  void log(String message) {
    print('Writing to file: $message');
  }
}

// Multiple Interfaces
class SecureService implements Logger, Database {
  @override
  void log(String message) { /* ... */ }

  @override
  void connect() { /* ... */ }
}
```

Part 2: Polymorphism

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

A. Compile-time Polymorphism (Static Binding)

In many languages, this is achieved via Method Overloading (same method name, different parameters).

- **Crucial Note:** Dart **does not** support traditional Method Overloading.
- **The Dart Way:** Dart achieves compile-time flexibility through **Optional and Named parameters**.

```
class Printer {
  // Dart style "Overloading"
  void printData(String data, {bool isBold = false, String? prefix}) {
    String output = data;
    if (prefix != null) output = "$prefix $output";
    if (isBold) output = "**$output**";

    print(output);
```

```
  }
}

void main() {
  var p = Printer();
  p.printData("Hello");                // Form 1
  p.printData("Hello", isBold: true);   // Form 2
  p.printData("Hello", prefix: ">>>");  // Form 3
}
```

B. Runtime Polymorphism (Dynamic Binding)

This is achieved via Method Overriding. The runtime environment determines which method to call based on the actual object type, not the variable type.

- **Requires:** Inheritance (`extends`) or Implementation (`implements`).
- **Mechanism:** The parent reference holds a child object.
- **Use Case:** Handling a collection of different objects uniformly.

```
abstract class Shape {
  void draw();
}

class Circle extends Shape {
  @override
  void draw() => print("Drawing Circle");
}

class Square extends Shape {
  @override
  void draw() => print("Drawing Square");
}

void renderShapes(List<Shape> shapes) {
  for (var shape in shapes) {
    // Runtime Polymorphism:
```

```
    // The runtime checks if 'shape' is actually a Circle or Square
    // and calls the correct draw() method.
    shape.draw();
  }
}
```

3. In-Class Exercise (Combined)

Goal: Model a Banking System using Abstraction and Polymorphism.

The Scenario:

We need a system for a bank. All bank accounts have a balance and can deposit(). However, withdraw() works differently depending on the account type:

1. **Savings Account:** Cannot withdraw if the balance goes below 0.
2. **Checking Account:** Can withdraw below 0, but charges a fee (Overdraft).

Instructions:

1. **Create an Abstract Class `BankAccount`:**
   - Property: `double balance`.
   - Constructor: Initialize `balance`.
   - Concrete Method: `void deposit(double amount)` (increases balance).
   - Abstract Method: `void withdraw(double amount)` (no body).
2. **Create `SavingsAccount` (extends BankAccount):**
   - Override `withdraw`: Check if `balance >= amount`. If yes, subtract. If no, print "Insufficient funds".
3. **Create `CheckingAccount` (extends BankAccount):**
   - Override `withdraw`: Subtract the amount. If the resulting balance is negative, print "Overdraft fee applied" and subtract an extra $10.
4. **Polymorphism Test:**
   - In `main()`, create a `List<BankAccount>`.

- Add one `SavingsAccount` (start with $100) and one `CheckingAccount` (start with $100).
- Loop through the list and try to `withdraw(150)` from each.
- Print the final balance of each.

---

4. Summary Comparison

Feature	Abstract Class	Interface (Dart Implementation)
Keyword	`abstract class A` / `extends A`	`class A` / `interface class A` / `abstract interface class A` / `implements A`
Logic Sharing	**Yes**. Child inherits actual code.	**No**. Child must rewrite all logic.
State (Fields)	Can have variables/state.	Fields are treated as getters/setters you must override.
Multiplicity	Single Inheritance only.	Multiple Interfaces allowed.
Best For	"Is-A" relationship (Dog is an Animal).	"Can-Do" relationship (Dog implements Swimmable).

---

5. Key Takeaways

- **Abstract Class:** A partial blueprint. Cannot be created, only extended.
- **Abstract Method:** A rule. Subclasses *must* implement it.
- **Interface (`implements`):** A contract. You must build everything yourself, inheriting nothing.
- **Polymorphism:** Treating a `Dog` as an `Animal`. It allows for flexible lists and functions that accept generic types but run specific code.

Encapsulation

Module: OOP Advanced

Class: Encapsulation

Topics: Getters & Setters | Private Fields in Dart

1. Lesson Objectives

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

- Define **Encapsulation** and explain its importance in object-oriented design.
- Implement **private fields** in Dart using the underscore (`_`) syntax.
- Create **Getters and Setters** to control access to class properties.
- Apply validation logic within setters to protect object integrity.

2. Core Concepts: Encapsulation with Private Fields

Encapsulation is the practice of bundling data (variables) and methods (functions) that operate on that data into a single unit (class), while restricting direct access to some of an object's components.

Why is Encapsulation Important?

Encapsulation is often referred to as the "shield" of your code. It is crucial for three main reasons:

1. **Protection:** It prevents external code from accidentally corrupting the object's internal state (e.g., setting an `age` variable to -5).
2. **Flexibility (Maintainability):** You can change the internal implementation logic later (like renaming a private variable or changing a data type) without breaking the external code that uses your class, provided the public methods/getters remain the same.

3. **Simplification:** It hides complex implementation details, exposing only what is necessary for the user of the class to know.

Real-World Analogies

1. **A Car Dashboard:**
    - **Public Interface:** The steering wheel, gas pedal, and brake. You use these simple controls to drive.
    - **Private Implementation:** The fuel injection system, piston firing order, and combustion engine.
    - **Encapsulation:** You don't need to interact directly with the engine pistons to drive, and the dashboard prevents you from accidentally disconnecting the fuel line while driving.
2. **A Bank ATM:**
    - **Public Interface:** The keypad and screen.
    - **Private Implementation:** The internal cash vault and counting mechanism.
    - **Encapsulation:** You can request a withdrawal (using a public method), but you cannot physically reach inside the machine to change your balance or grab cash directly.

How Dart Handles Privacy

Unlike Java or C#, Dart does not have keywords like `private`, `public`, or `protected`.

- **Public:** By default, everything is public.
- **Private:** To make a member private to its **library** (file), prefix the name with an underscore (`_`).

Getters and Setters

We use special methods called **Getters** (`get`) and **Setters** (`set`) to read and write private fields. This allows us to add logic (like validation) before modifying data.

Lecture Code Example

```
class Employee {
  // Private field: Accessible only within this file/library
  String _name;
  double _salary;

  // Constructor
  Employee(this._name, this._salary);

  // --- GETTERS (Read Access) ---

  // Getter for name
  String get name ⇒ _name;

  // Getter for formatted salary
  String get salaryInfo ⇒ "Salary: \$${_salary.toStringAsFixed(2)}";

  // --- SETTERS (Write Access with Validation) ---

  // Setter for salary
  set salary(double newSalary) {
    if (newSalary < 0) {
      print("Error: Salary cannot be negative.");
    } else {
      _salary = newSalary;
      print("Salary updated to $_salary");
    }
  }
}

void main() {
  var emp = Employee("Alice", 50000);

  // Accessing via Getter
  print(emp.salaryInfo); // Output: Salary: $50000.00
```

```
  // Accessing via Setter (Valid)
  emp.salary = 55000; // Output: Salary updated to 55000.0

  // Accessing via Setter (Invalid)
  emp.salary = -100; // Output: Error: Salary cannot be negative.

  // NOTE: emp._salary would result in a compile error if accessed
  // from a different file, enforcing encapsulation.
}
```

3. In-Class Exercise

Scenario:

You are building a logic system for a Thermostat. The temperature should be private to prevent accidental extreme values.

Instructions:

1. Create a class named `Thermostat`.
2. Define a private field `_temperature` (double).
3. Create a constructor to initialize it.
4. Create a **Getter** `celsius` that returns the temperature.
5. Create a **Setter** `celsius` that:
   - Allows setting the temperature only if it is between -30 and 50 degrees.
   - Prints "Warning: Temperature out of range" if the value is invalid.
6. *Bonus:* Create a getter `fahrenheit` that converts the internal celsius value to fahrenheit ($C \times 9/5 + 32$).

4. Key Takeaways

1. **Control:** Encapsulation allows you to control **how** variables are accessed or modified.

2. **Validation:** Setters provide a specific place to validate data *before* it saves to the object (e.g., preventing negative age or salary).
3. **Read-Only:** By providing a Getter but **no** Setter, you make a property read-only from the outside.
4. **Abstraction:** The internal representation (private fields) can change without affecting external code that uses the public getters/setters.

Static & Factories, Code Reuse

Module: OOP Advanced

Class: Static, Factories & Code Reuse

Topics: Static Members | Factory Constructors | Singletons | Mixins | Extensions

1. Static Variables & Methods

The "Blueprint" Concept

Static members belong to the class itself rather than to any specific object instance.

Core Concepts

- **Class-Level vs. Instance-Level:** You do not need to create an object (using `new`) to access static members.
- **Memory Efficiency:** There is only *one* copy of a static variable in memory, shared by all instances of that class.
- **Utility Methods:** Perfect for helper functions that don't need to access the state of a specific object (e.g., `Math.sqrt()`).
- **Access:** Accessed using the class name directly: `ClassName.variable`.

Code Example

```
class Constants {
  // Shared by all instances
  static double pi = 3.14159;

  static double calculateArea(double r) {
```

```
    return pi * r * r;
  }
}

void main() {
  // No instance needed to access 'pi'
  print(Constants.pi);

  // Calling a static method
  print(Constants.calculateArea(10));
}
```

2. Factory Constructors & Singletons

Controlling Instance Creation

Unlike a normal constructor, a `factory` constructor in Dart does not necessarily create a new instance of the class.

The `factory` Keyword

- **Return Existing:** Can return an instance from a cache.
- **Return Subclass:** Can return an instance of a derived class (polymorphism).
- **Singleton:** Essential for implementing the Singleton pattern.

The Singleton Pattern

This pattern ensures a class has only **one instance** and provides a global point of access to it.

Common Use Cases:

- Database Connections
- Configuration Managers
- Logging Services
- File Systems

Code Example: Singleton

```
class Database {
  // 1. Private static instance variable
  static final Database _instance = Database._internal();

  // 2. Private named constructor
  Database._internal();

  // 3. Factory constructor returns the same static instance
  factory Database() {
    return _instance;
  }

  void connect() {
    print("Connected to DB");
  }
}

void main() {
  var db1 = Database();
  var db2 = Database();

  // Both variables point to the exact same object in memory
  print(db1 == db2); // true
}
```

3. Code Reuse: Mixins

Composition over Inheritance

Mixins allow you to reuse a class's code in multiple class hierarchies without requiring a parent-child relationship.

Understanding Mixins

1. **No Inheritance:** Mixins solve the limitations of single inheritance.
2. **"Has-A" Ability:** Think of Mixins as abilities (e.g., `Flyable`, `Swimmable`) that you can layer onto any class.
3. **The `with` Keyword:** Used to apply a mixin to a class.

Code Example

```
mixin Flyable {
  void fly() => print("I'm flying!");
}

class Animal {}

// Composing behavior: Duck IS-A Animal, HAS-A Flyable ability
class Duck extends Animal with Flyable {
}

void main() {
  var donald = Duck();
  donald.fly(); // Output: I'm flying!
}
```

4. Dart Extensions

Enhancing Existing Libraries

Extensions allow you to add functionality to existing libraries that you don't own (like the `String`, `int`, or `List` classes) without creating a subclass.

Code Example

```
extension NumberParsing on String {
  int parseInt() {
    return int.parse(this);
  }
```

```
}

void main() {
  // Usage: The method appears as if it belongs to the String class
  print("42".parseInt());
}
```

5. Key Takeaways

- **Static:** Use for global constants and utility methods that don't require instance state.
- **Factory:** Use to control the instantiation process (caching, polymorphism).
- **Singleton:** A pattern to restrict a class to exactly one instance.
- **Mixins:** A way to reuse code across multiple class hierarchies (Capabilities).
- **Extensions:** Add new methods to existing types (even built-in ones) keeping code clean.
- **Design:** Prefer composition (Mixins) over deep inheritance trees.

OOP Advanced Features

Module: OOP Advanced

Class: Advanced Features

Topics: Operator Overloading | Copy Constructors | Class Composition

1. Operator Overloading

Customizing Object Behavior

By default, operators like `==` and methods like `toString()` have default behaviors defined by the base `Object` class. Overloading allows us to define how these operators work for our specific custom classes.

The `toString()` Method

- **Default Behavior:** Returns "Instance of 'ClassName'".
- **Overriding:** We override this to provide a meaningful string representation of the object, which is crucial for debugging and logging.

The `==` Operator (Equality)

- **Reference Equality (Default):** Two variables are equal only if they point to the *exact same location* in memory.
- **Value Equality (Override):** We override `==` to check if two distinct objects contain the same *data*.
- **The `hashCode` Rule:** If you override `==`, you **must** also override `hashCode`. Equal objects must have the same hash code.

Code Example

```
class Point {
  Point(this.x, this.y);

  final int x;
  final int y;

    // 1. Overriding toString for better print output
  @override
  String toString() ⇒ 'Point($x, $y)';

  // 2. Overriding == for Value Equality
  @override
  bool operator ==(Object other) {
    if (identical(this, other)) return true;

    return other is Point &&
      other.x == x &&
      other.y == y;
  }

  // 3. Must override hashCode if == is overridden
  @override
  int get hashCode ⇒ x.hashCode ^ y.hashCode;
}

void main() {
  var p1 = Point(1, 2);
  var p2 = Point(1, 2);

  print(p1); // Output: Point(1, 2) instead of Instance of 'Point'
  print(p1 == p2); // Output: true (Value Equality)
}
```

2. Copy Constructor Concept (copyWith)

Cloning Objects & Immutability

In modern Dart, instead of a traditional "copy constructor," we typically use a `copyWith` instance method. This is essential for **Immutable** objects (where fields are `final`). Since you cannot change the fields of an existing object, you create a *new* object based on the old one, changing only specific properties.

- **Immutability:** Fields are `final` and cannot be changed after the object is created.
- **The `copyWith` Pattern:** A method that takes optional named parameters for every field. If a parameter is provided, it uses the new value; otherwise, it falls back to the existing value (`this.value`).

Why is this crucial for Flutter State Management?

In Flutter (and libraries like Bloc, Riverpod, or Redux), state is often immutable to ensure performance and predictability.

1. **Efficient Rebuilds:** Flutter needs to know *when* to rebuild the UI. Comparing memory addresses (Identity) is instant, while checking every field of a large object (Deep Equality) is slow.
2. **The Trigger:** If you mutate an object in place (`user.age = 26`), the object reference remains the same. The framework looks at the old and new state, sees the same memory reference, and assumes **nothing changed**, resulting in the UI not updating.
3. **The Solution:** Using `copyWith` creates a completely **new instance** (new memory reference). The framework sees `oldState != newState` and immediately knows to trigger a rebuild.

Code Example

```
class User {
  // Constructor requires all fields
  User({required this.name, required this.age});

  final String name;
```

```
  final int age;

  // Idiomatic 'copyWith' pattern
  // Parameters are nullable to allow 'null' to mean "don't change this field"
  User copyWith({
    String? name,
    int? age,
  }) {
    return User(
      name: name ?? this.name,
      age: age ?? this.age,
    );
  }

  @override
  String toString() ⇒ 'User(name: $name, age: $age)';
}

void main() {
  var original = User(name: "Alice", age: 25);

  // Create a copy, changing ONLY the name
  var copy = original.copyWith(name: "Bob");

  // Create a copy, changing ONLY the age
  var olderCopy = original.copyWith(age: 26);

  print(original);   // User(name: Alice, age: 25)
  print(copy);       // User(name: Bob, age: 25)
  print(olderCopy);  // User(name: Alice, age: 26)

  // Proof of different instances (Crucial for Flutter)
  print(original == copy); // false
}
```

3. Class Composition

"Has-A" Relationship

Composition is the design principle where a class contains objects of other classes as member variables.

Composition vs. Inheritance

- **Inheritance (Is-A):** `Car` *is a* `Vehicle`. Good for hierarchy.
- **Composition (Has-A):** `Car` *has an* `Engine`. Good for building complex objects from smaller, reusable components.

Why prefer Composition?

- **Flexibility:** You can easily swap components (e.g., change the `Engine` type inside a `Car`) at runtime.
- **Loose Coupling:** Changes in the component class don't ripple through a hierarchy as strictly as inheritance.

Code Example

```
// Component 1
class Engine {
  Engine(this.type);

  String type;

  void start() => print("$type engine starting...");
}

// Component 2
class Tires {
  Tires(this.size);

  int size;
}
```

```
// Composite Class
class Car {
  Car(this.model, this.engine, this.tires);

  String model;
  // Composition: Car HAS-A Engine and HAS-A Tires
  Engine engine;
  Tires tires;

  void startCar() {
    print("Checking system for $model...");
    engine.start(); // Delegating behavior to the component
  }
}

void main() {
  var v8 = Engine("V8");
  var offRoadTires = Tires(22);

  // Constructing object via composition
  var myCar = Car("Monster Truck", v8, offRoadTires);

  myCar.startCar();
}
```

4. Key Takeaways

1. **Operator Overloading:** Gives your objects natural behavior (like `==` for equality) and readable logging (`toString`).
2. **Reference vs. Value:** Without overriding `==`, two identical objects are considered different because they live in different memory addresses.
3. **Cloning:** Use the `copyWith` pattern to safely duplicate data. This is the standard for state management in Flutter, allowing the framework to detect state

changes efficiently by comparing object references.

4. **Composition over Inheritance:** Build complex objects by combining smaller, isolated classes. It creates more flexible and maintainable code structures.

Error Handling

Topcis	**Try-catch-finally block | Throwing custom exceptions | Printing stack trace**

Module: Error Handling & Asynchronous Programming

Class: Error Handling

Topics: Try-Catch-Finally | Custom Exceptions | Stack Traces

1. Lesson Objectives

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

- Implement robust error handling using **try-catch-finally** blocks.
- Differentiate between standard Errors and Exceptions.
- Create and throw **Custom Exceptions** for specific application logic.
- Utilize **Stack Traces** to debug and locate the origin of errors effectively.

2. Core Concepts: Handling Errors

In Dart (and Flutter), proper error handling prevents your application from crashing unexpectedly and allows you to provide user-friendly feedback when things go wrong.

The Try-Catch-Finally Block

- **Try:** Wraps the code that *might* throw an exception (e.g., fetching data, parsing JSON).
- **Catch:** Catches the exception if one occurs, preventing a crash. You can catch specific types of exceptions or a general one.

- **Finally:** A block that *always* runs, regardless of whether an error occurred or not. It is typically used for cleanup (e.g., closing files, resetting loading states).

The Stack Trace

A **Stack Trace** is a snapshot of the call stack at the moment the error occurred. It tells you exactly *where* (which file and line number) the error happened and the sequence of function calls that led there.

Lecture Code Example

```
void main() {
  print("App Started");

  try {
    // 1. Risky code
    var result = 10 ~/ 0; // Integer division by zero throws an Exception
    print("Result: $result");

  } on IntegerDivisionByZeroException {
    // 2. Catching a specific exception
    print("Cannot divide by zero!");

  } catch (e, s) {
    // 3. Catching any other error
    // 'e' is the Exception object
    // 's' is the StackTrace object
    print("Unknown error: $e");
    print("Stack Trace:\n$s");

  } finally {
    // 4. Cleanup code
    print("Cleanup operations (closing streams/files)...");
  }

  print("App Continued..."); // App survives the error
```

```
}
```

3. Throwing Custom Exceptions

Sometimes standard exceptions (like `FormatException` or `TimeoutException`) aren't specific enough for your business logic. You can define your own.

How to Create a Custom Exception

Create a class that implements the built-in `Exception` interface.

Scenario

Imagine a user trying to withdraw money from a bank account. We need a specific error if they don't have enough funds.

Code Example

```
// 1. Define the Custom Exception
class InsufficientFundsException implements Exception {
  final String message;
  final double attemptedAmount;

  InsufficientFundsException(this.message, this.attemptedAmount);

  @override
  String toString() => "InsufficientFundsException: $message (Attempted: \$$at
temptedAmount)";
}

class BankAccount {
  double balance = 100.0;

  void withdraw(double amount) {
    if (amount > balance) {
      // 2. Throw the custom exception
```

```
      throw InsufficientFundsException("Not enough balance", amount);
    }
    balance -= amount;
    print("Withdrew \$$amount. Remaining: \$$balance");
  }
}

void main() {
  var account = BankAccount();

  try {
    account.withdraw(500.0); // This will fail
  } catch (e) {
    // 3. Handle the specific custom logic
    if (e is InsufficientFundsException) {
      print("Transaction failed: ${e.message}");
      // specific logic for this error, e.g., show a 'Top Up' dialog
    } else {
      print("General Error: $e");
    }
  }
}
```

4. Key Takeaways

1. **Don't Let Apps Crash:** Always wrap risky code (network calls, file I/O) in `try-catch` blocks to keep the app running.
2. **Be Specific:** Catch specific exceptions (using `on ExceptionName`) before catching general ones. This allows you to handle different errors differently (e.g., "No Internet" vs. "Invalid Password").
3. **The `finally` Clause:** Crucial for UI logic. For example, if you show a loading spinner in `try`, you *must* hide it in `finally` so it doesn't spin forever if an error occurs.

4. **Debug with Stack Traces:** When catching a generic error, always print the stack trace (`catch (e, s)`) during development to find the bug's origin quickly.

Asynchronous Programming

Class: Asynchronous Programming

Lesson Objectives

- Understand the non-blocking nature of Dart.
- Master the `Future` API and the `async`/`await` syntax.
- Handle errors effectively using `try-catch` and `.catchError`.
- Manage multiple asynchronous operations simultaneously.
- Grasp the basics of `Stream` for handling sequences of asynchronous events.

Topic 1: Futures, Async, and Await

I. High-level Overview

In Dart, operations that take time (like reading a file or downloading data) don't stop the rest of your code from running; instead, they return a "promise" that a result will be available later.

II. Definition

Future: An object representing a delayed computation. It has two states: Uncompleted (pending) and Completed (with a value or an error).

Async/Await: Syntactic sugar that allows you to write asynchronous code that looks and behaves like synchronous code, pausing execution only within the specific function until the Future completes.

III. Example

```
// Simulating a network request that takes 2 seconds
Future<String> fetchUserData() async {
```

```
  print('Fetching data...');

  // Pause this function for 2 seconds, but don't block the app
  await Future.delayed(Duration(seconds: 2));

  return 'User: John Doe';
}

void main() async {
  print('App Start');

  // Wait for the future to complete before moving to the print statement
  String user = await fetchUserData();

  print(user);
  print('App End');
}
```

IV. Real-world Simple Use Case

Login Screen: When a user clicks "Login," the app sends credentials to a server. The app shouldn't freeze while waiting for the server's response. Instead, it shows a loading spinner (awaiting the Future) and updates the UI once the server responds.

Topic 2: Future Chaining & Error Handling

I. High-level Overview

Sometimes you need to perform actions immediately after a task finishes without using `await`, or you need to handle failures gracefully to prevent the application from crashing.

II. Definition

Chaining (.then): A method to register a callback that triggers when a Future completes successfully.

Error Handling (.catchError / try-catch): Mechanisms to intercept exceptions thrown during asynchronous operations.

III. Example

```
Future<int> parseData(String data) async {
  if (data.isEmpty) throw Exception('Data is empty!');
  return data.length;
}

void main() {
  // Using .then and .catchError (Functional style)
  print('Starting operation...');

  parseData("")
      .then((length) ⇒ print('Length: $length'))
      .catchError((e) ⇒ print('Error caught: $e'))
      .whenComplete(() ⇒ print('Operation finished regardless of outcome.'));

  print('This prints BEFORE the error because we did not "await" above.');
}
```

IV. Real-world Simple Use Case

Analytics Logging: When a user performs an action, you might fire a network request to log this event (`analytics.logEvent()`). You don't need to pause the user's experience to wait for the log to succeed; you just fire it, chain a `.catchError` to log any failures silently to the console, and let the user continue.

Topic 3: Handling Multiple Tasks (Future.wait)

I. High-level Overview

This technique allows you to trigger multiple independent asynchronous tasks simultaneously and wait for all of them to finish before proceeding.

II. Definition

`Future.wait`**:** A static method that accepts a list of Futures and returns a new Future that completes with a list of results only when *all* provided Futures have completed.

III. Example

```
Future<String> fetchProfile() async ⇒ Future.delayed(Duration(seconds: 1), () ⇒ 'Profile Data');
Future<String> fetchSettings() async ⇒ Future.delayed(Duration(seconds: 2), () ⇒ 'Settings Data');

void main() async {
  print('Loading Dashboard...');
  final stopwatch = Stopwatch()..start();

  // Both run in parallel, effectively taking only as long as the slowest task (2s)
  final results = await Future.wait([fetchProfile(), fetchSettings()]);

  print('Loaded: ${results[0]} and ${results[1]}');
  print('Time elapsed: ${stopwatch.elapsed.inSeconds} seconds');
}
```

IV. Real-world Simple Use Case

App Initialization: When your app starts, you need to fetch the User Profile, App Configuration, and Notifications. Instead of fetching them one by one (taking 1s + 1s + 1s = 3s), you fetch them all at once (taking max(1s) = 1s), significantly reducing the loading screen time.

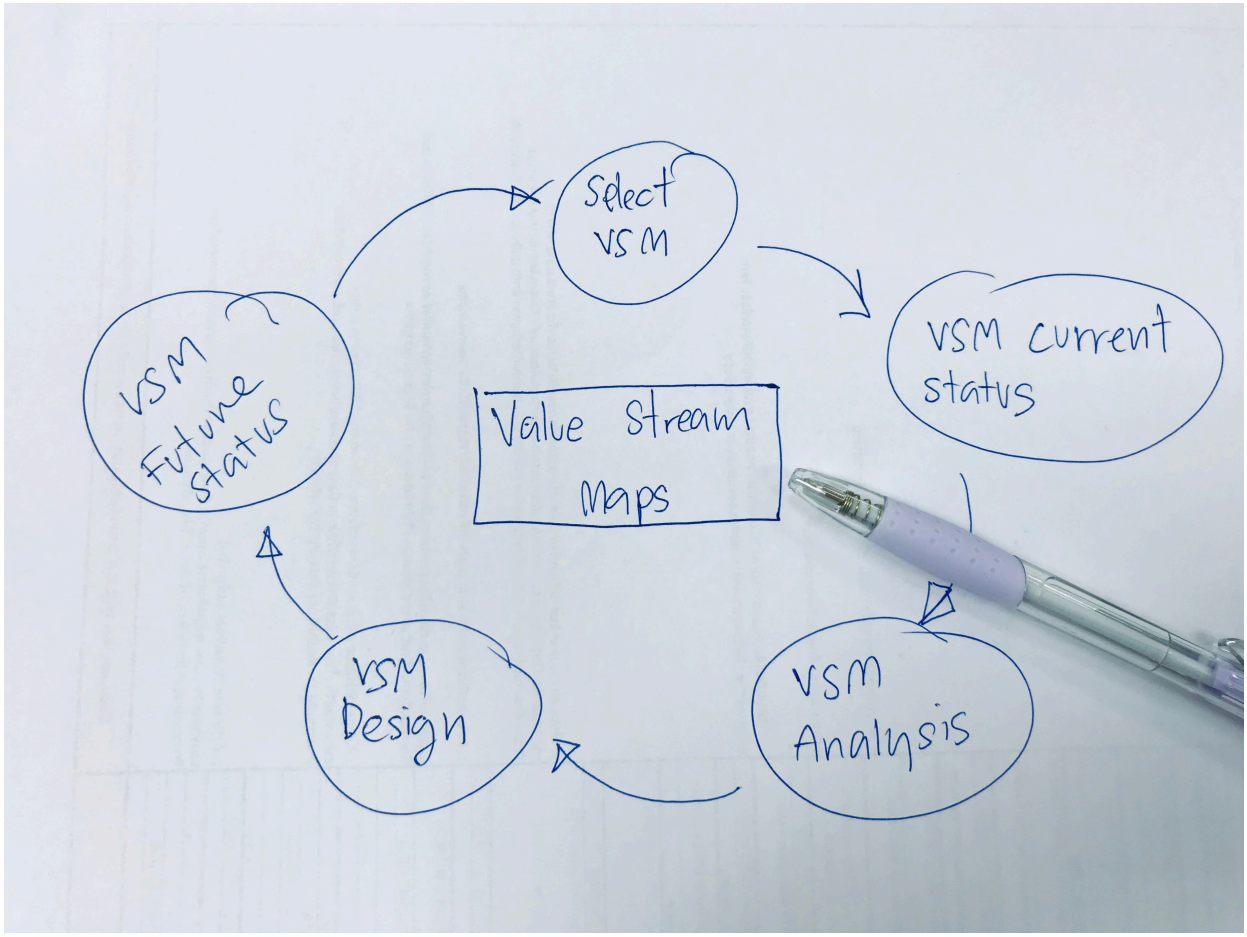
Topic 4: Streams Basics

I. High-level Overview

While a Future represents a *single* value delivered later, a Stream represents a *sequence* of values delivered over time.

II. Definition

Stream: A source of asynchronous data events. It is like an iterator that pushes data to you (the listener) whenever it arrives, rather than you asking for the next item.



III. Example

```
Stream<int> countSeconds(int max) async* {
  for (int i = 1; i <= max; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i; // 'yield' pushes data to the listener
  }
}

void main() {
```

```
  print('Timer started');

  // Listen to the stream
  countSeconds(3).listen(
    (time) ⇒ print('Tick: $time'),
    onDone: () ⇒ print('Timer finished!'),
  );
}
```

IV. Real-world Simple Use Case

Chat Application: In a chat app, you don't just receive one message. You open a connection (Stream) to the server, and every time a friend sends a message, the server pushes that new piece of data through the stream to your device to display immediately.

V. Problems for Learners

Problem 1: The Broken Fetch

Write a function fetchUsername that simulates a network delay of 2 seconds. Randomly, it should either return "DartMaster" or throw an exception "Server Error". Call this function in main using try-catch blocks to handle both scenarios.

Problem 2: The Race

Create three functions: runnerA, runnerB, and runnerC. Each waits for a random duration between 1 and 3 seconds and returns their name. Use Future.wait to start them all at once and print "Race Finished" only when all three are done.

Problem 3: The Countdown

Create a Stream that emits a number every second starting from 10 down to 1. When the stream emits 0, print "Blast off!". Listen to this stream in your main function.

VI. In-class Example: The Dashboard Simulator

This example simulates a real-world scenario where an application needs to fetch data from different services (User, Weather, News) with varying delays, handle potential errors, and utilize a continuous stream for a "Live Ticker."

```
import 'dart:async';
import 'dart:math';

// 1. Mock API Service
class ApiService {
  final Random _rng = Random();

  Future<String> getUserName() async {
    await Future.delayed(Duration(seconds: 1));
    return "Alice";
  }

  Future<int> getNotifications() async {
    await Future.delayed(Duration(seconds: 2));
    // Simulate random failure
    if (_rng.nextBool()) {
      throw Exception("Failed to fetch notifications");
    }
    return 5;
  }

  Future<String> getWeather() async {
    await Future.delayed(Duration(seconds: 3));
    return "Sunny, 25°C";
  }

  // Stream for a live stock ticker or news feed
  Stream<String> getLiveNewsTicker() async* {
    List<String> news = ["Market up", "Dart 3.0 released", "Flutter is awesom
e"];
    for (var item in news) {
      await Future.delayed(Duration(milliseconds: 500));
```

```
      yield item;
    }
  }
}

void main() async {
  final api = ApiService();
  print("--- Dashboard Loading ---");

  // 2. Handling Multiple Futures (some critical, some not)
  try {
    // We need User and Weather to show the main screen.
    // We run them in parallel.
    final criticalData = await Future.wait([
      api.getUserName(),
      api.getWeather()
    ]);

    print("Welcome back, ${criticalData[0]}!");
    print("Current Weather: ${criticalData[1]}");

  } catch (e) {
    print("Critical Error loading dashboard: $e");
    return; // Stop app if user/weather fails
  }

  // 3. Handling independent non-critical errors
  // We try to get notifications, but if it fails, we show 0 instead of crashing.
  int notifications = 0;
  try {
    notifications = await api.getNotifications();
  } catch (e) {
    print("Warning: Could not load notifications (Server error)");
  }
  print("You have $notifications unread notifications.");
```

```
  print("\n--- Connecting to Live Feed ---");

  // 4. Listening to a Stream
  // We 'await' the stream subscription to ensure we catch all events if needed,
  // or just let it run in background. Here we use 'await for' loop.
  await for (String newsItem in api.getLiveNewsTicker()) {
    print("BREAKING NEWS: $newsItem");
  }

  print("--- Dashboard Ready ---");
}
```

Summary

- **Futures** are for single async values (like a package arriving in the mail).
- **Async/Await** makes reading async code easier, avoiding "callback hell."
- **Future.wait** allows parallel execution, saving time.
- **Streams** are for continuous data flows (like a radio broadcast).
- Always handle errors (`try-catch` or `.catchError`) to prevent your app from crashing when the network or database fails.