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

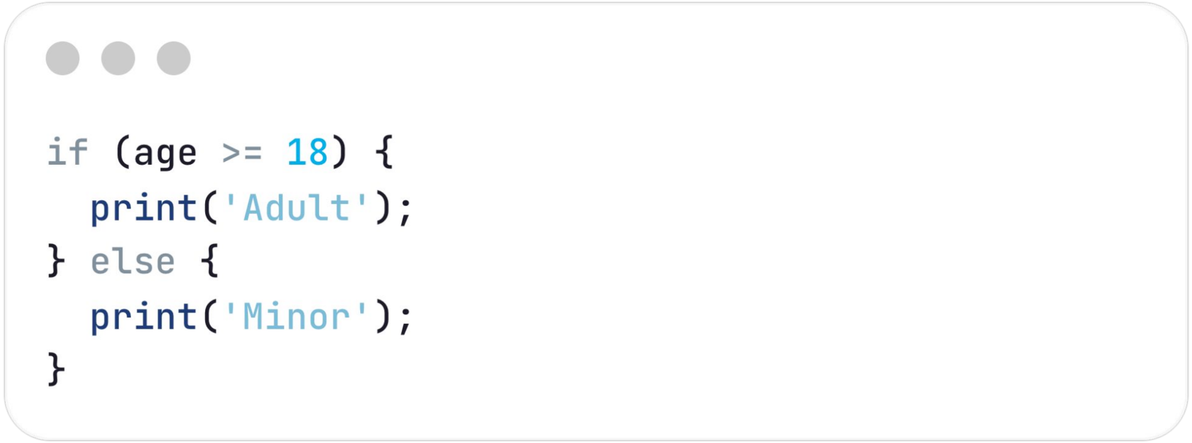
- Control flow allows decision making in programs.
- Automates repetitive tasks.
- Makes code dynamic and responsive instead of running linearly.
- Without control flow, all programs would behave the same way every time.

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

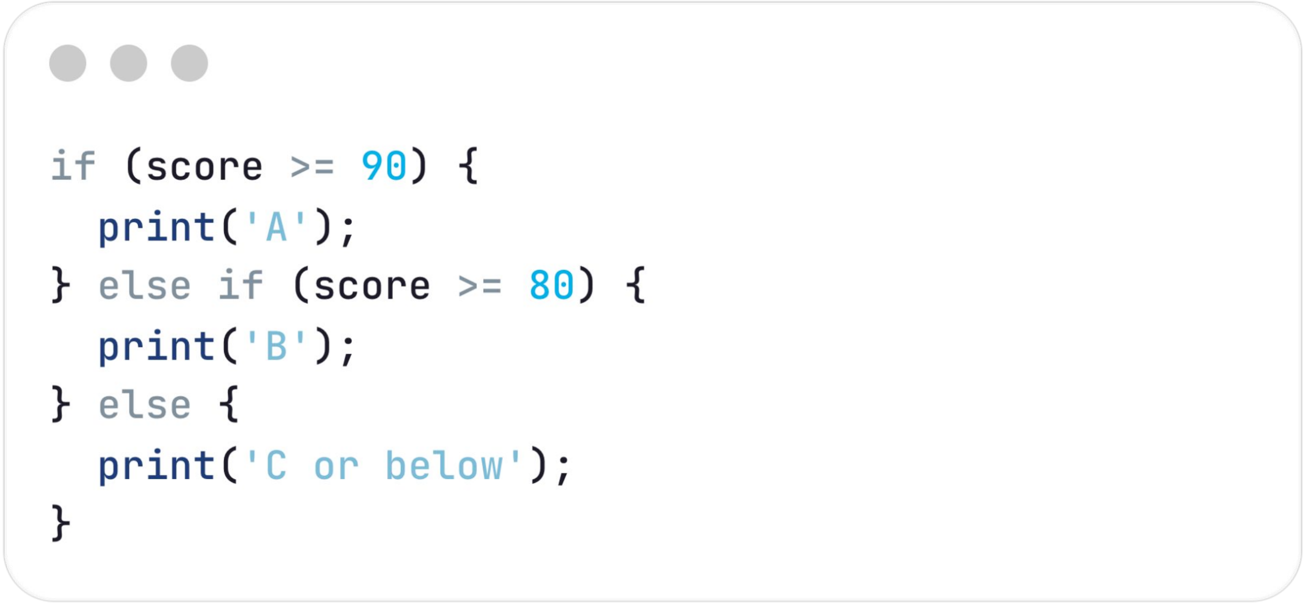
Used to execute code based on a condition



```
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)

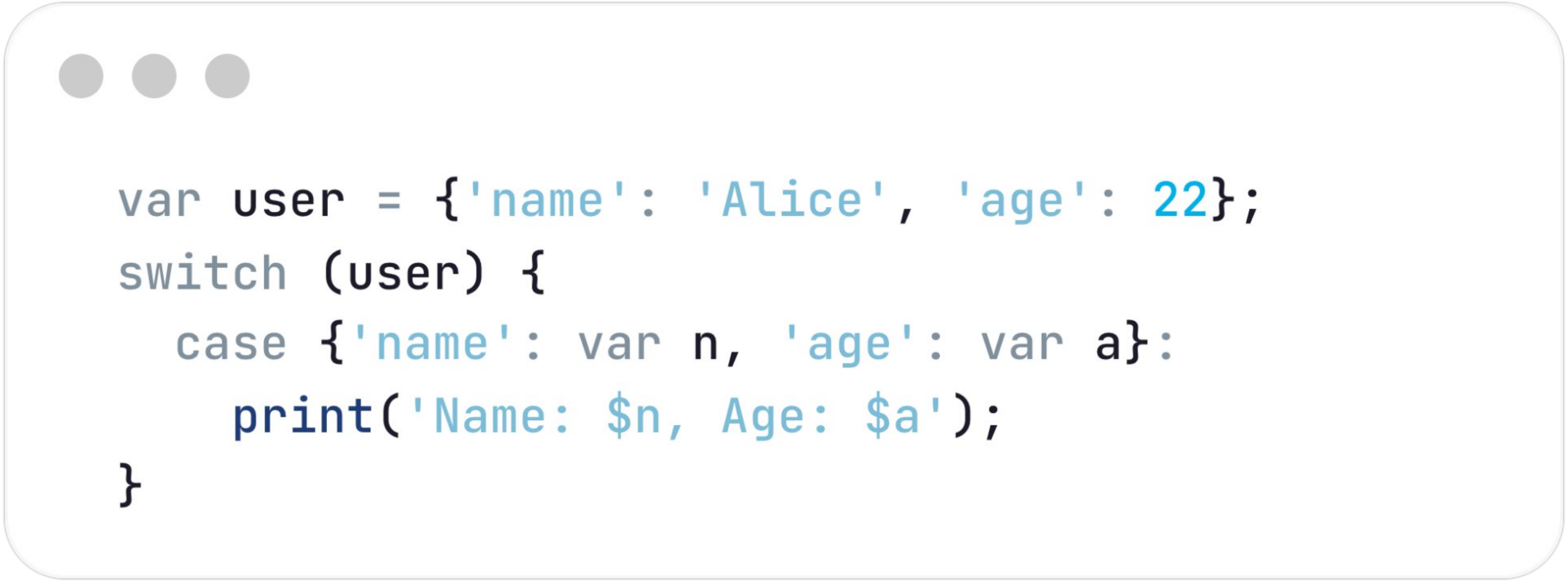
Modern way to handle conditions and extract values

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

What is Pattern Matching?

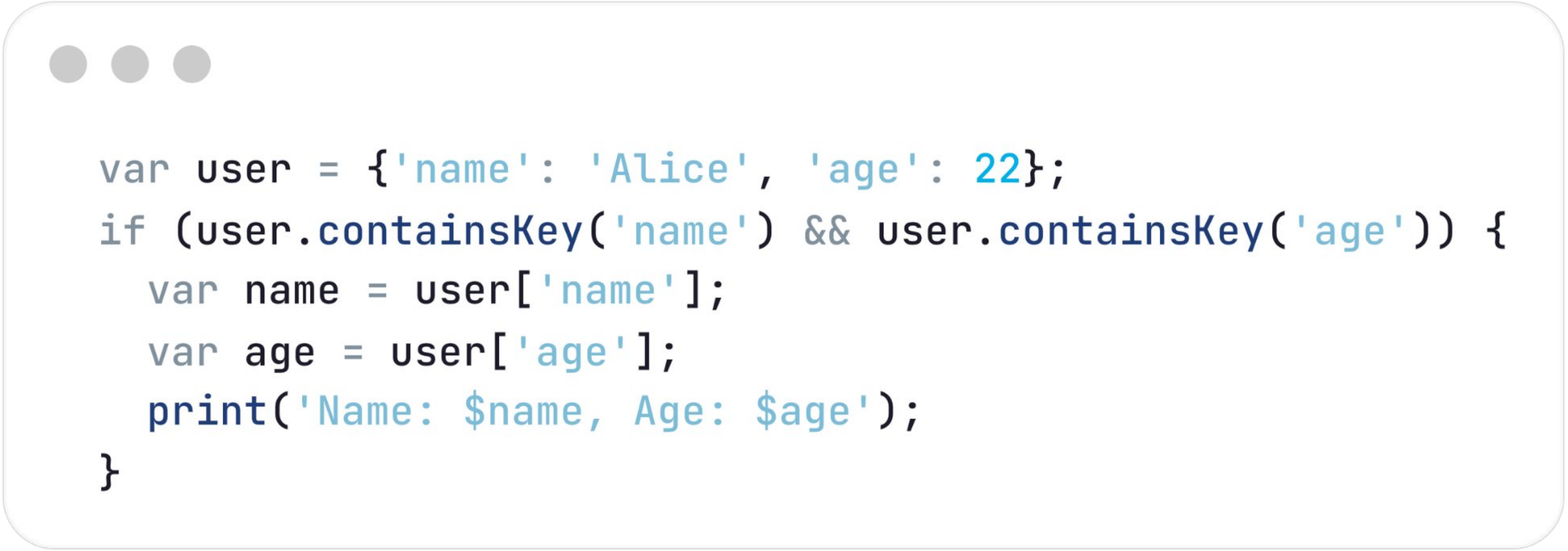
- **Definition**:
  Pattern matching is a way to check a value against a pattern and extract data from it in a clean and readable way.
- Key Idea:
  - Instead of writing long if-else chains or switch statements, you match a structure directly.
  - The pattern itself declares what to check and what to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

Pattern Matching Way



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

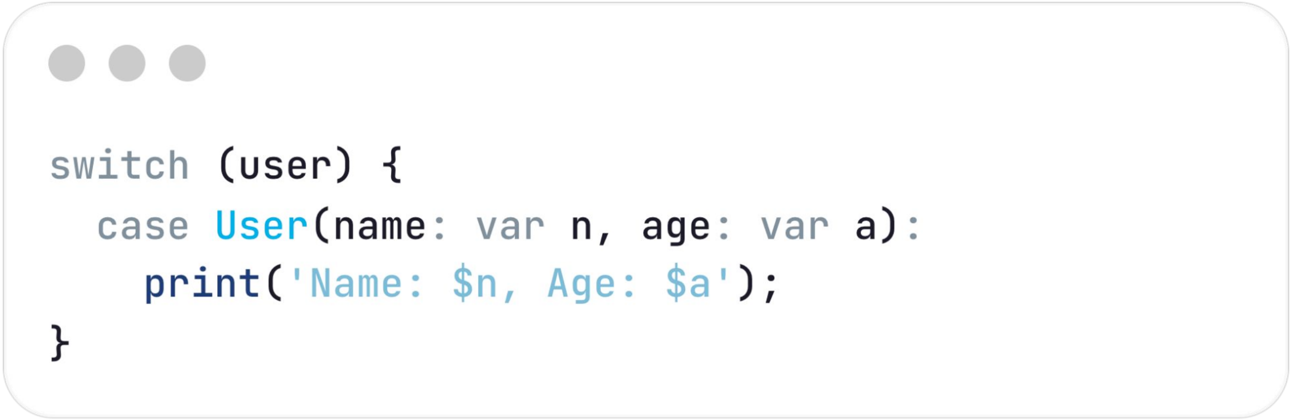
Regular Way (If-else)



```dart
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?

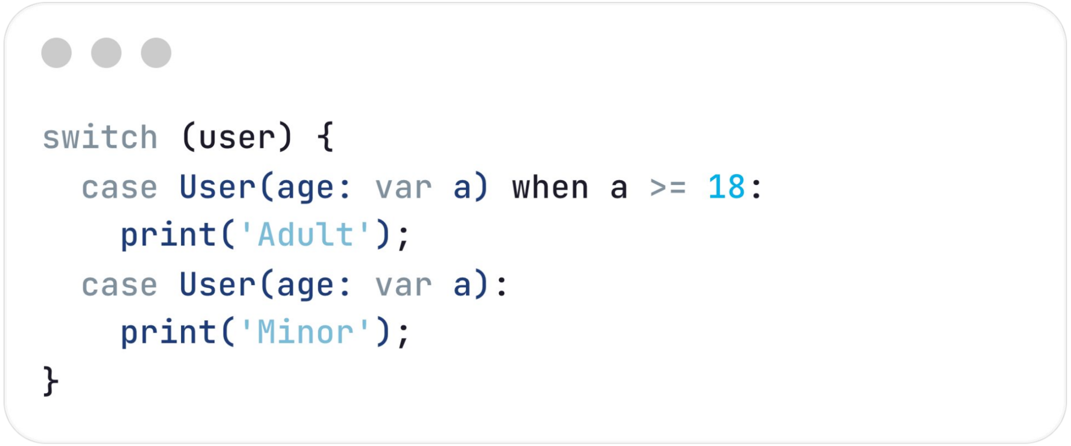
- Destructuring means breaking down an object or data structure into its parts.
- When a pattern matches, it can automatically bind the matched values to variables.
- Makes it easy to access nested values directly.



```
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');
}
```