Dart

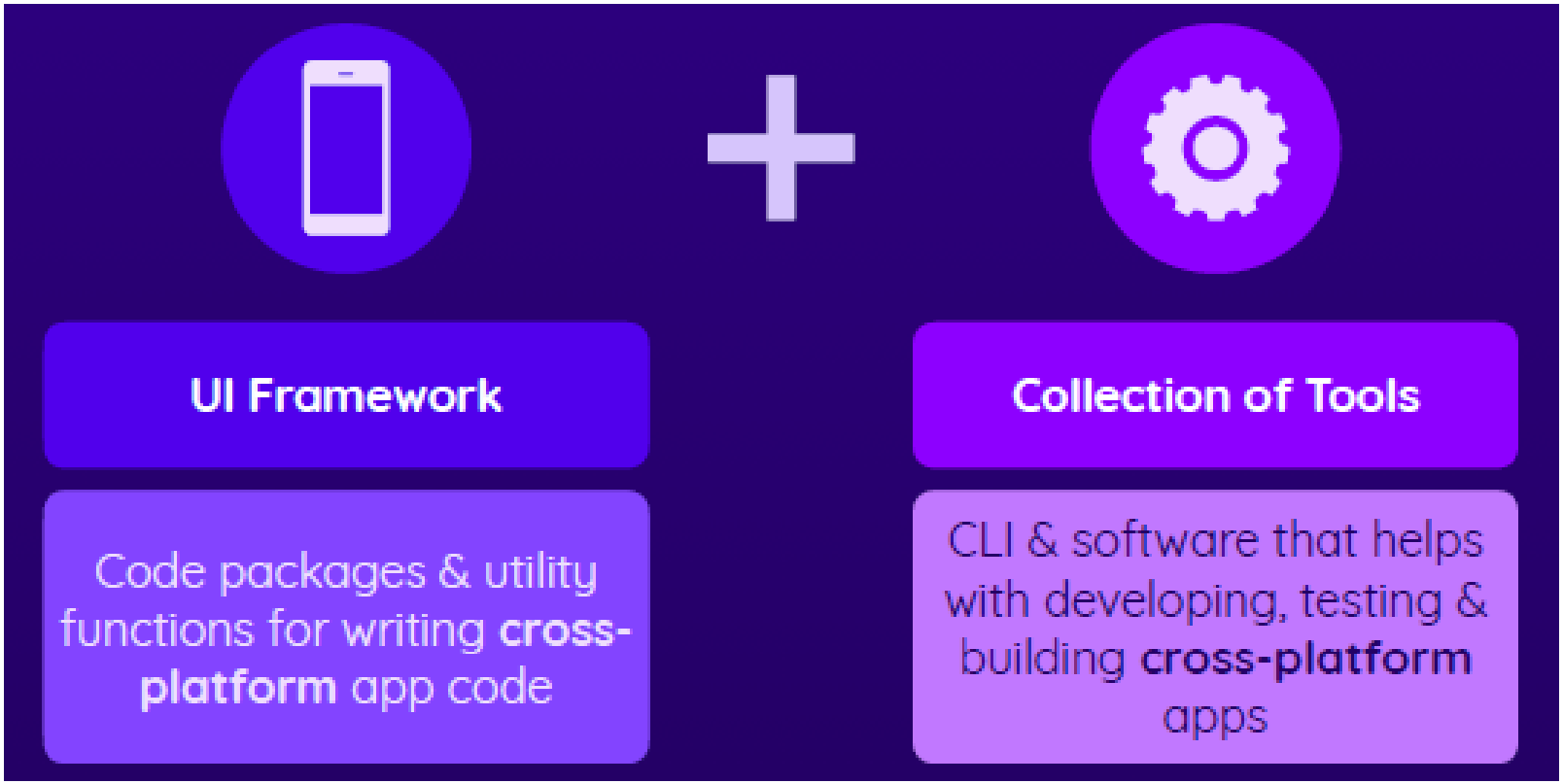
https://dart.dev

Table of Contents

Some of the slides are based on Flutter Complete Course content

Introduction to Flutter and Dart

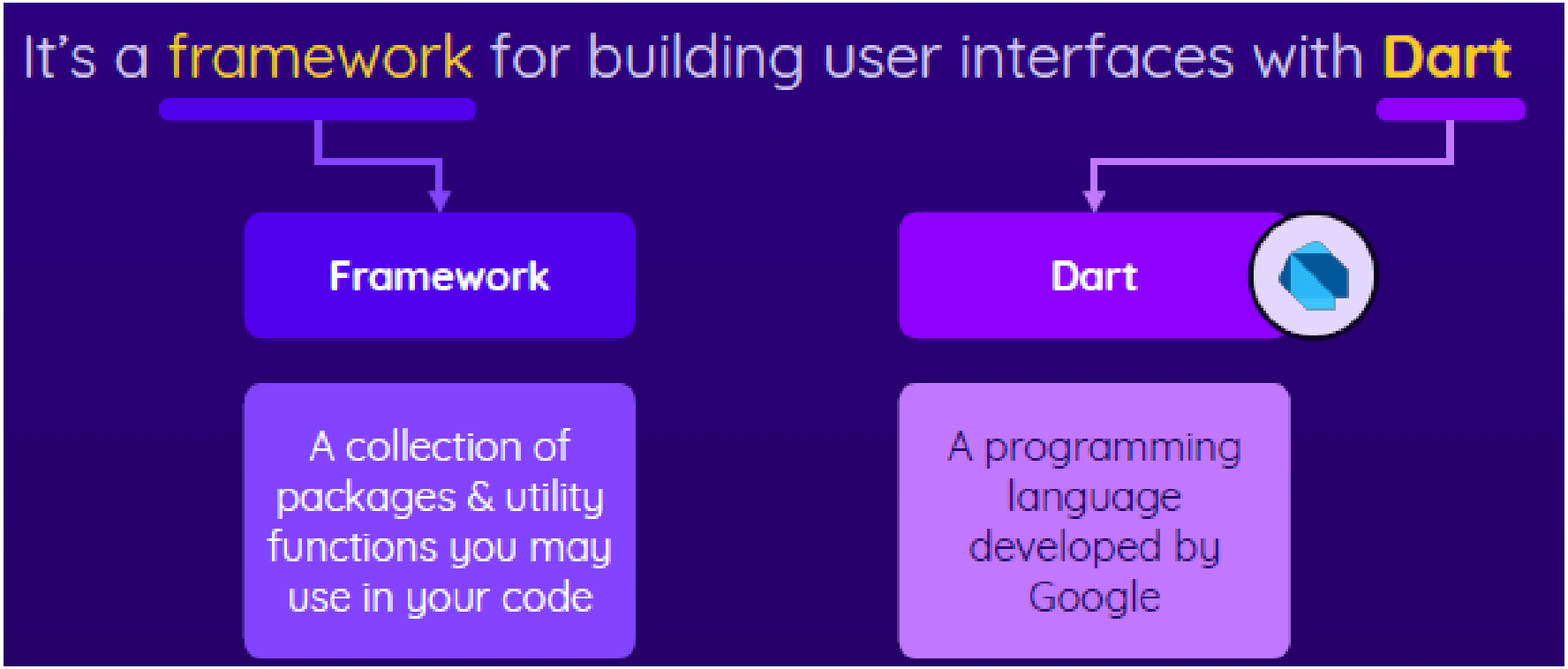
What is Flutter?



- Flutter uses **Dart** programming language to build **natively compiled** apps for **multiple platforms** from a **single codebase**

Source: Flutter Complete Course slides

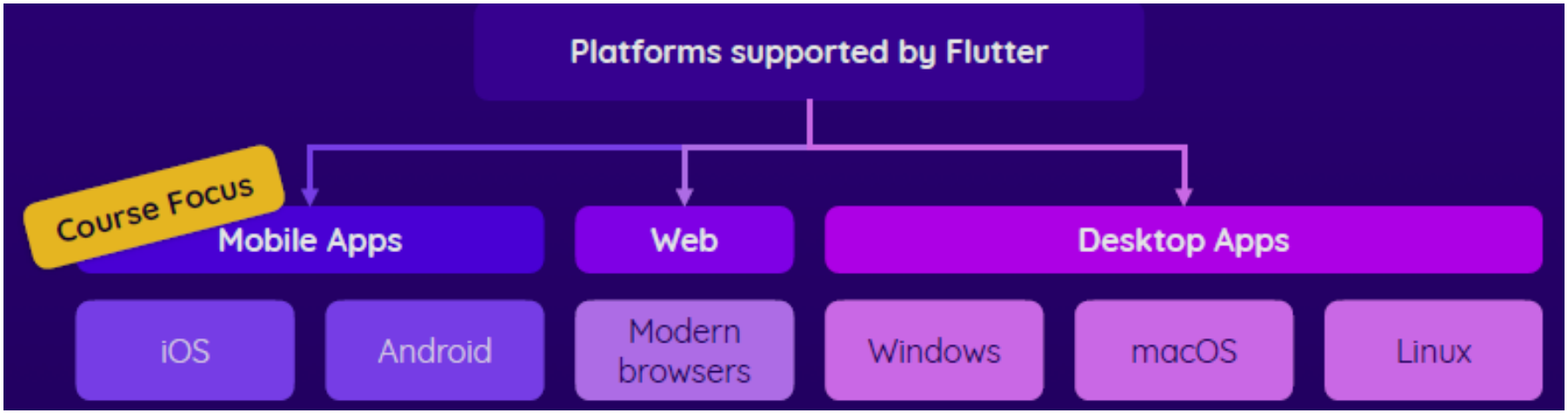
Flutter Is Not A Programming Language!



Source: Flutter Complete Course slides

One Codebase, Multiple Apps

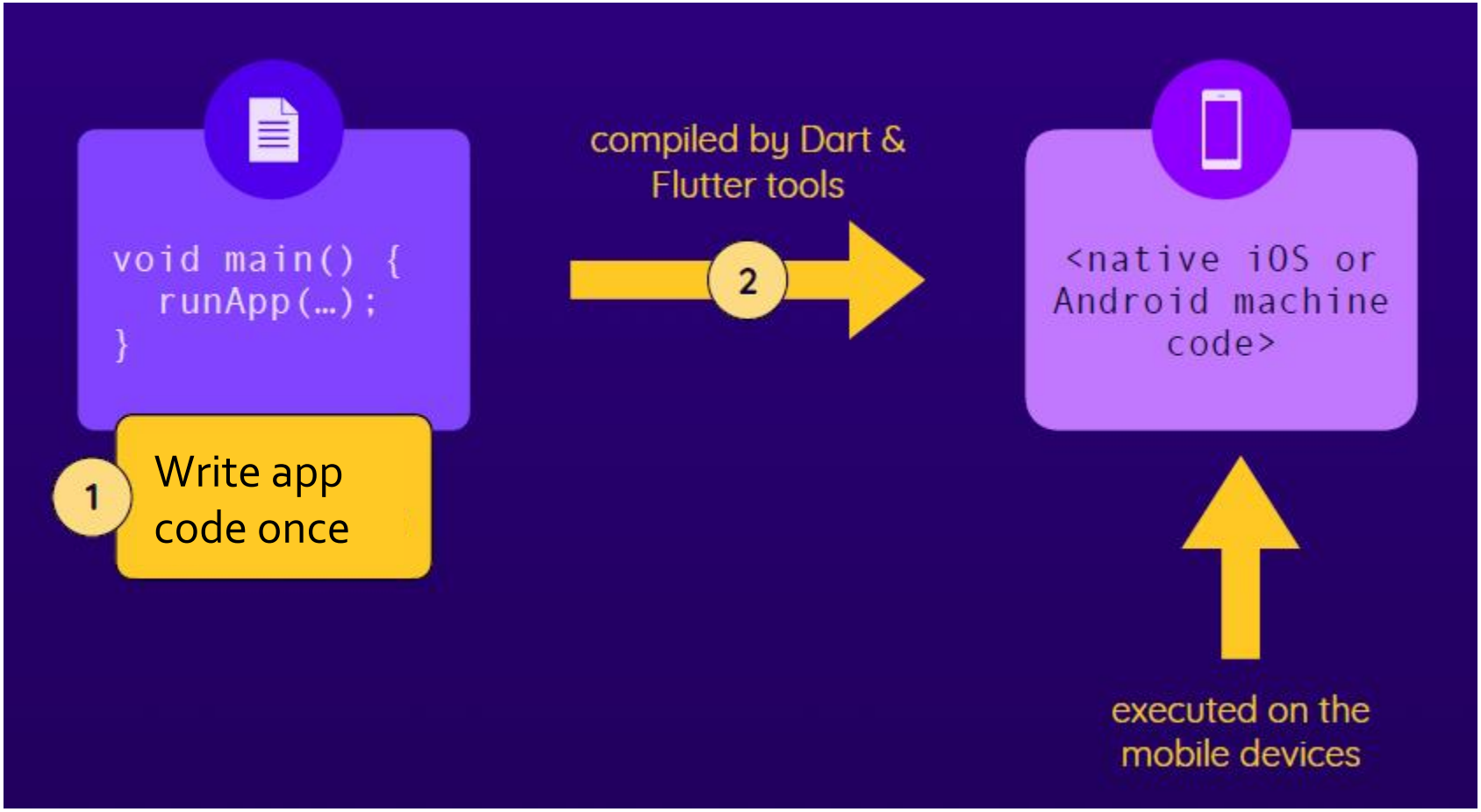
- Dart compiler translates the app code to platform-specific machine code



- Whilst you can write code for all platforms on the same machine, you can **only test & run iOS and macOS apps on macOS machine**, **Windows apps on Windows machine** and **Linux apps on Linux machine**!
- Android and web apps can be built and test on all operating systems

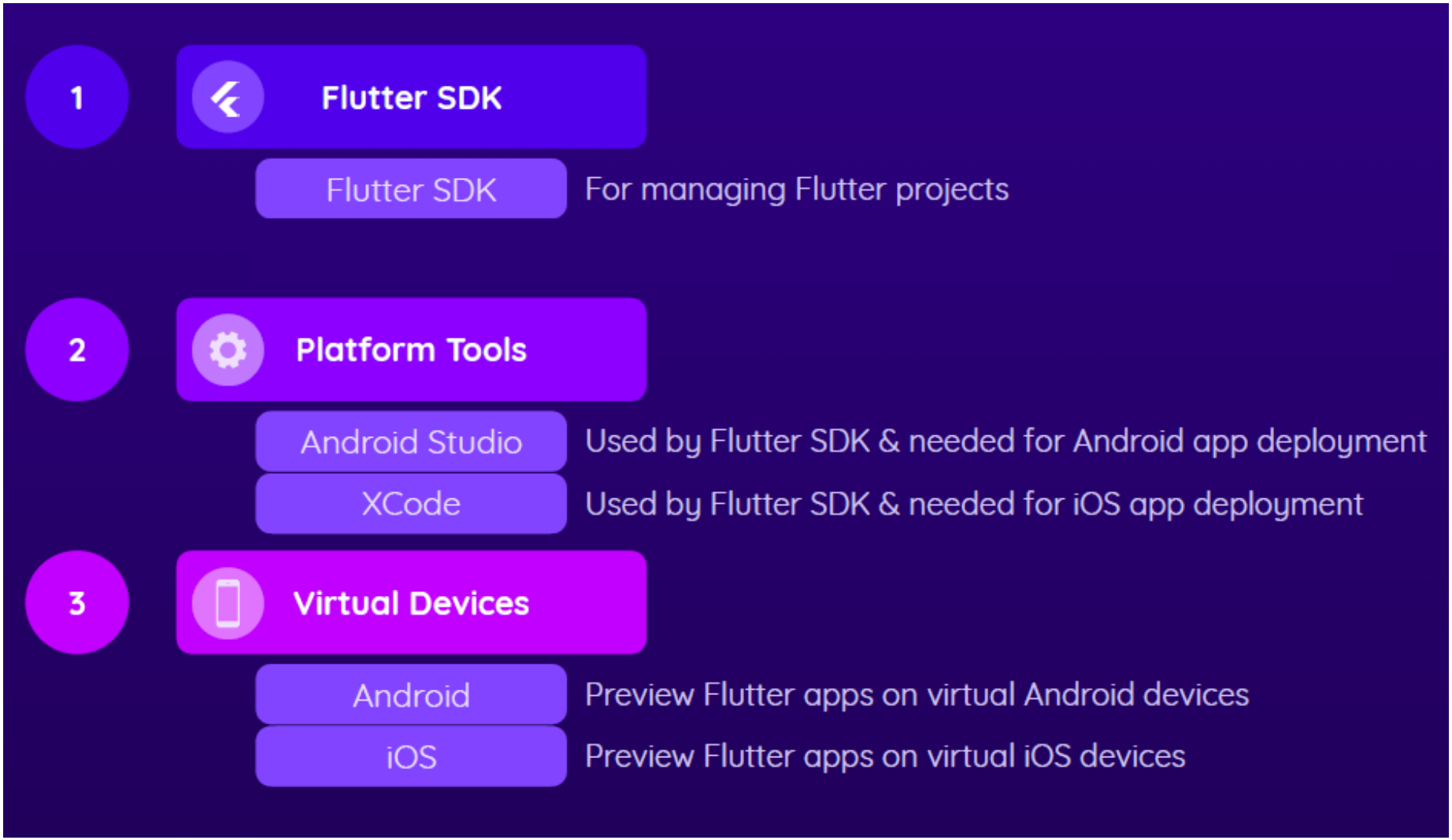
Source: Flutter Complete Course slides

Dart & Flutter Code Is Compiled



Source: Flutter Complete Course slides

Flutter Setup



https://docs.flutter.dev/get-started/install

Target Platform Tools & Devices Setup

https://docs.flutter.dev/get-started/install

- You will setup your dev environment and create your GitHub account during Lab 1

Source: Flutter Complete Course slides

Dart Features (1 of 2)

- Dart is an open-source general-purpose programming language developed by Google (Dart 1.0 Nov 2013, current version Dart 3.5)
- Platform-independent (Windows, Mac, Linux, and Web)
- **Strongly Typed Language**: type **validation** at compile time, ensuring both safety. Plus, code completion by IDE.
- Supports **Type Inference:** type automatically determined from the context
- Sound null safety
- **Just-in-Time (JIT) Compilation** in development: allows for hot reloads during development, enabling developers to see changes instantly without restarting the app
- **Ahead-of-Time (AOT) Compilation** in production: compiles code into native machine code for mobile, web and desktop

Dart Features (2 of 2)

- Rich Standard Library: provides a wide range of utilities for collections, file I/O, networking, and more
- **Object-oriented** programming (encapsulation, inheritance, polymorphism) with **functional** programming features
- **Asynchronous Programming**: with features like **async** and **await**, making it easier to write non-blocking code, particularly useful for I/O-bound tasks
- Auto memory management with **Garbage Collection** (GC)
- Easy to learn and use: concise and readable code
  - Dart has a syntax inspired from languages like JavaScript, Java, C#
- Strong community and plenty of resources available for learning https://dart.dev/ and development https://pub.dev/

Terms Revisited

- ***Statement***: command that ends with ";"

```
print('Hello world!');
```

- ***Expression***: command evaluated to a single value

```
'Hello ' + 'world!'
```

- ***Keyword***: word reserved for compiler

```
int, String, if, for, static, final, etc.
```

- ***Identifier***: name of variable, function, class, etc.

```
int age;
```

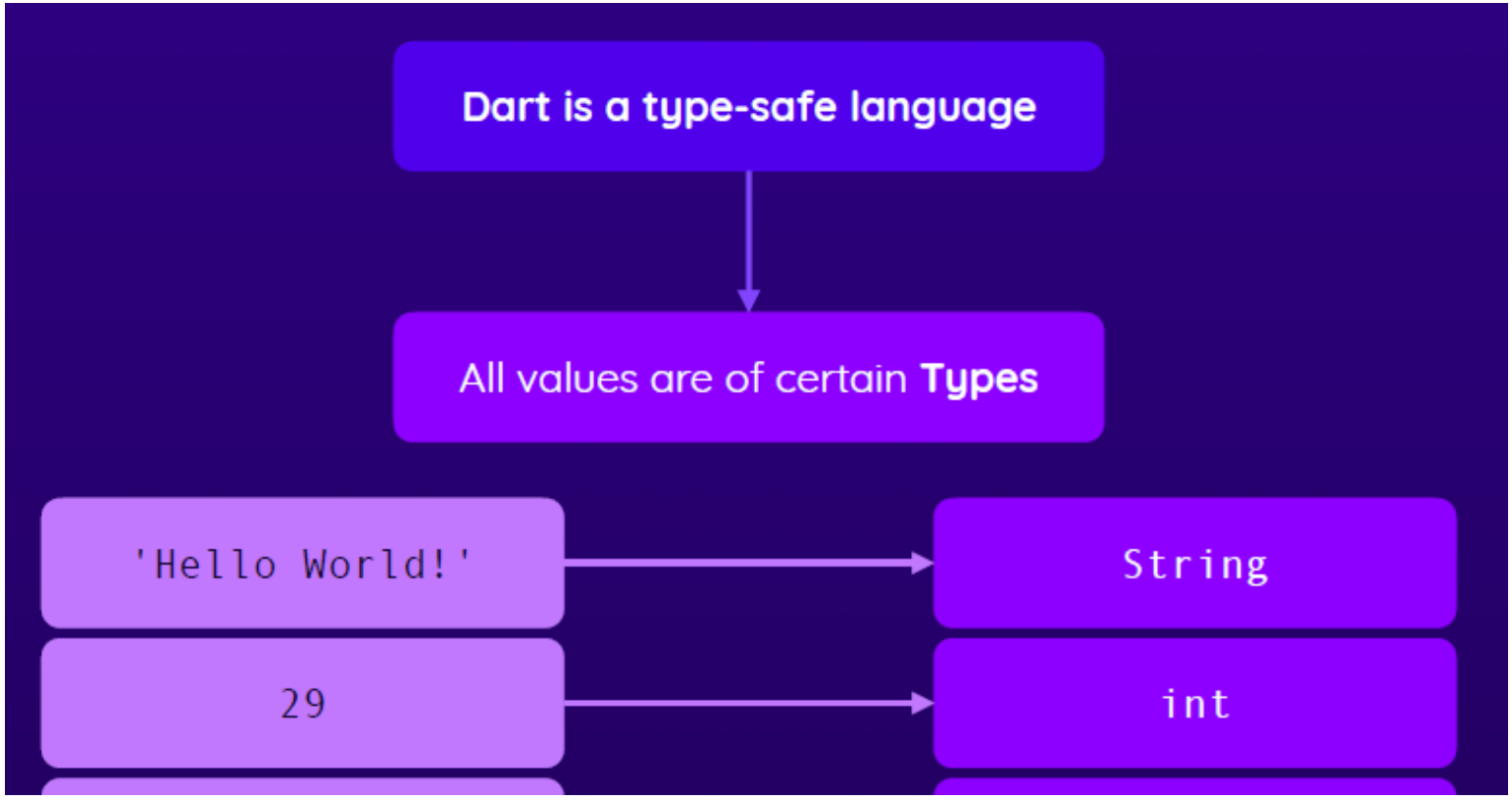
- ***Literal***: value directly written in source code

```
double pi = 3.14;
```

Declaring Variables

Understanding Data Types

- Variable is named storage location (i.e., a container for values in a program)
- Data types simply refers to the type and size of data than can stored in a variable



Source: Flutter Complete Course slides

Some Core Types

`int`	**Integer numbers**	Numbers **without** decimal places	`29, -15`
`double`	**Fractional numbers**	Numbers **with** decimal places	`3.91, -12.81`
`num`	**Integer or fractional numbers**	Numbers **with or without** decimal places	`15, 15.01, -2.91`
`String`	**Text**	Text, wrapped with single or double quotes	`'Hello World'`
`bool`	**Boolean values**	true or false	`true, false`
`Object`	**Any kind of object**	The base type of all values	`'Hi', 29, false`

- Dart is strongly typed language: it uses static type checking to ensure that a variable's value *always* matches the variable's static type

Source: Flutter Complete Course slides

Type inference

- Type inference allows the compiler to **automatically determine the type** of a variable based on the value assigned to it
  - Making the code more concise and easier to read without explicitly specifying types
  - Dart infers the type at compile-time, ensuring type safety
  - The inferred type is final and can't be changed to another type later

```
var name = 'Ali’;    // Inferred as String
var age = 18;        // Inferred as int
var height = 1.8;    // Inferred as double

print('$name is $age years old and $height meters tall.');
```

Strings

```
//Strings and String Template
var firstName = "Ali"
var lastName = "Faleh"
```

- **String Template** (aka String Interpolation) allow creating dynamic templated string with placeholders (instead of string concatenation!)
  - Simple reference uses **$** and an expression uses **${}**

```
val fullName = "$firstName $lastName"
val sum = "2 + 2 = ${2 + 2}"

//Multiline Strings
val multiLinesStr = """
  First name: $firstName
  Last name: $lastName
"""
```

Convert a number to a string

- Use number's ***toString*** method

```
var num = 10
var str = num.toString()
```

Convert a string to a number

- Use string's `int.`***parse*** method

```
num = int.parse(str)
```

var vs. const vs. final

- **var** is **mutable** and can be reassigned
- **const** is **compile-time constant** and **immutable** (read-only) can only assign a value to it exactly one time at compile time
    - **compile-time constant**: The value must be known at compile-time and cannot be changed
- **final** is **immutable** (read-only) can only be set once either at compile time or at runtime
    - **Runtime Constant**: it doesn't have to be known at compile-time => value can be determined at runtime

See 02.2_var_const_final.dart example

Nullable Types

- By default, variables in Drat are **non-nullable**

unless explicitly declared as nullable using a **?** after the data type

- **Syntax:**

```
String iCannotBeNull = "Not Null"
String? iCanBeNull = null
```

- `String iCannotBeNull = null`
  - Compilation Error: Can't assign null to a non-nullable variable
- `String? iCanBeNull = null`
  - Compiles ok

Null safety (1 of 2)

- **Null-aware Operator (?.)**: Safely accesses a property or method on an object that might be null
  - If the object is null, the expression evaluates to null instead of throwing an error

```
String? name;
// Output: null, safe access even if 'name' is null
print(name?.length);
```

- **Null-coalescing Operator (??)**: Provides a default value if the expression on the left is null

Null safety (2 of 2)

- **Null-aware Assignment Operator (??=):** Assigns a value to a variable only if the variable is currently null

```
String? email = 'mrcool@dart.dev';
// Email is only assigned if 'email' is null
email ??= 'info@dart.dev';
print(email);
```

- **Using switch expression for null-safe access**

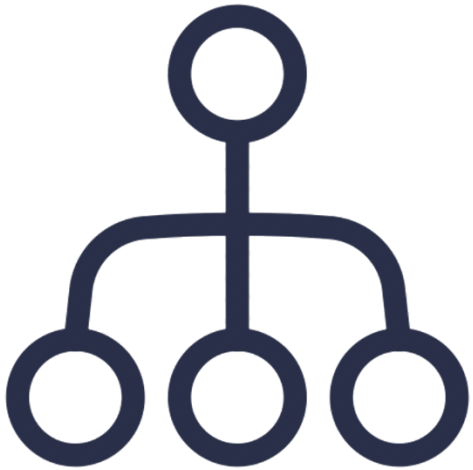
```
var greeting = switch (name) {
  null => 'Hello, Guest!',       // Handle null value
  '' => 'Hello, Anonymous!',     // Handle empty string
  _ => 'Hello, $name!',   // Handle non-null, non-empty string
};
print(greeting); // Output: Hello, Guest!
```

Comments

```
// slash slash line comment

/*
 slash star
 block comment
*/
```

Control Flow: if, when expressions



if-else statement

```
var age = 20
var ageCategory = ""

if (age < 18) {
      ageCategory = "Teenager"
} else {
      ageCategory = "Young Adult"
}
```

if-else expression using the ternary operator ?:

The ternary operator ?: (condition ? expr1 : expr2)

```
var ageCategory = age < 18 ?
    "Teenager" : "Young Adult";
print('Age category: $ageCategory');
```

Switch expression

- Switch expression provides a concise and expressive way to handle conditional logic
- Assign a value based on matching condition

```
var month = 8;
var season = switch (month) {
  12 || 1 || 2 => "Winter",
  >= 3 && <= 4 => "Spring",
  >= 6 && <= 8 => "Summer",
  >= 9 && <= 11 => "Autumn",
  _ => "Invalid Month",
};

print("The season is $season.");
```

while (…)
do { … }
for { … }
Loops

Execute Blocks of Code Multiple Times

While Loop

- While Loop:

```
while (condition) {
    statements
}
```

- Do-While Loop:

```
do {
    statements
}
while (condition)
```

for Loop Example

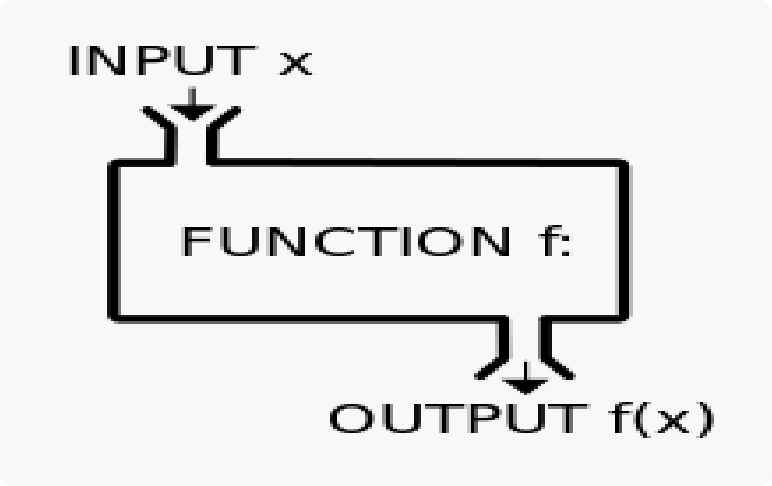
```
// List of names
var names = ["Sara", "Fatima", "Ali"];

// Loop through the list
for (var name in names) {
  print(name);
}

// Loop with index and value
for (var i = 0; i < names.length; i++) {
  print("$i -> ${names[i]}");
}

names.forEach((name) => print(name));
names.forEach(print);
```

Functions



Functions

- Can be declared at the **top level** of a file (without belonging to a class)
- Can have a **block or expression body**
- Can have **named** parameters
- Can have default parameter values to avoid method overloading

```
// Function with a block body
int max(int a, int b) {
  return a > b ? a : b;
}
// Function with a block body and named parameters
int max({required int a, required int b}) {
  return a > b ? a : b;
}
// Function with an expression body (Lambda Expression)
int max(int a, int b) => a > b ? a : b;
// Function assigned to a variable
var max = (int a, int b) => a > b ? a : b;
```

Functions

```
// Function with block body
fun sum(a: Int, b: Int): Int {
    return a + b
}

// Function with expression body
// Omit return type
fun sum(a: Int, b: Int) = a + b

//Arrow function - called Lambda expression
var sum = { a: Int, b: Int -> a + b }
```

void return type

- When defining a function that doesn't return a value, we can use **void** as the return type

```
void display(dynamic value) => print(value);
```

- If the return type is omitted, then return type of the function is **dynamic** type

Use default parameters for function overloading

- Dart doesn't support function overloading (i.e., having multiple functions with the same name but different parameters)
- You default parameters instead

```
void displayLine({String character = '*', int length = 20}) {
  var line = character * length;
  print(line);
}

void main() {
  displayLine(); // Uses default character '*' and length 20
  // Uses provided character '=' and default length 20
  displayLine(character: '=');
  // Uses provided character '~' and length 5
  displayLine(character: '~', length: 5);
}
```

Extension Method

- Enable adding functions and properties to existing classes

```
// Extension method extending String class
extension NumberParsing on String {
  int parseInt() {
    return int.parse(this);
  }
}

// Extension method extending int class
extension IntExtensions on int {
  bool get isEven => this % 2 == 0;
}

void main() {
  var number = "123".parseInt();
  print("Parsed number: $number");

  var num = 10;
  print("Is $num even: ${num.isEven}");
}
```

Extension Function Example

```
extension StringExtensions on String {
  String lastChar() {
    return this.substring(this.length - 1);
  }

  String lastChar() {
    return substring(length - 1);
  }
}
```


```
var name = "Fatima";
name.lastChar();
```

```
lastChar() → String
lastIndexOf(…) → int
length int
toLowerCase() → String
padLeft(…) → String
trimLeft() → String
```

Exceptions

- Throw:

```
throw Exception("Invalid input")
```

- Handling

```
try {
}
catch (e) {
}
finally {
}
```

```
// Example
int? parseInt(String number) {
  try {
    return int.parse(number);
  } catch (e) {
    print(e);
    return null;
  }
}
```

OOP

Class Example

👍 **Concise**

```dart
void main() {
  var person = Person(firstName: 'John', lastName: 'Doe', age: 17);
  print('Full name: ${person.fullName}');
  print('Is minor: ${person.isMinor()}');
}
```

Instantiate an object

```dart
class Person {
  // Use final for properties that
  // are initialized once and won't change
  final String firstName;
  final String lastName;
  final int age;

  // Constructor with named parameters
  // Use required for non-nullable properties to ensure that
  // a value is provided during object creation
  Person({required this.firstName, required this.lastName, required this.age});

  // Computed property (getter)
  String get fullName => '$firstName $lastName';

  // Method to check if the person is a minor
  bool isMinor() => age < 18;
}
```

Properties

Primary constructor

Computer Property

Method

Class with a computed property

```
class Rectangle(double width, double height) {
    val isSquare
        get() = width == height

}
```

Named Constructor

```dart
class Conference {
  final String name;
  final String city;
  final bool isFree;
  double fee;

  // Primary constructor with optional parameters
  Conference({
    required this.name,
    required this.city,
    this.isFree = true,
    this.fee = 0.0,
  });

  // Named constructor for non-free conferences
  Conference.withFee({
    required this.name,
    required this.city,
    required this.fee,
  }) : isFree = false;

}

void main() {
  // Using the named constructor for a conference with a fee
  var conference = Conference.withFee(name: "Flutter Conference", city: "Doha", fee: 300);
  print('Conference: ${conference.name}, City: ${conference.city}, Fee: ${conference.fee},
          Is Free: ${conference.isFree}');
}
```

Static Properties and Methods

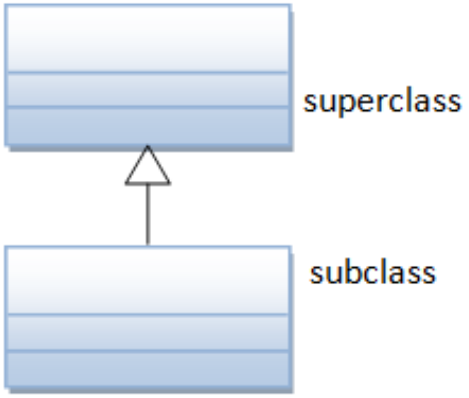
- Static properties/methods belong to the class rather than to any particular object
  - They can be called on the class itself, without creating an instance

```
class Car {
  // Static property to keep track of the number of cars created
  static int carCount = 0;
  // Instance property
  String model;

  // Constructor
  Car(this.model) {
    carCount++; // Increment car count whenever a new car is created
  }

  // Static method to get the total number of cars
  static int getCarsCount() {
    return carCount;
  }
}
```

Inheritance

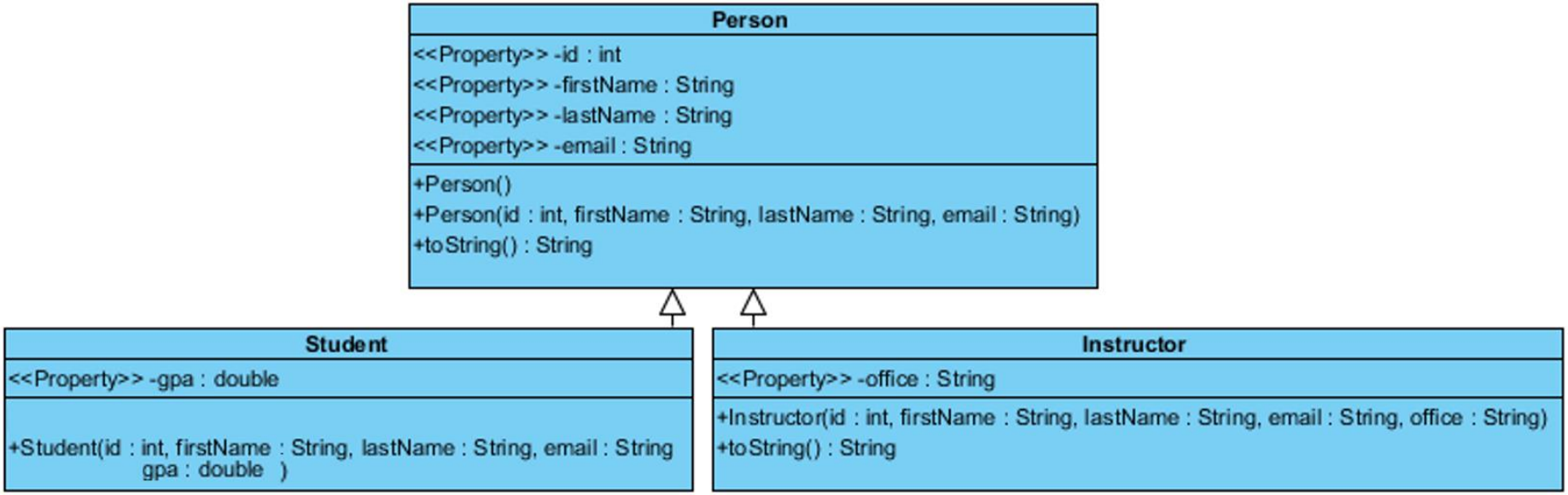


- **Ideas**
  - Common properties and methods are place in a **superclass** (also called ***parent class*** or ***base class***)
  - You can create a subclass that **inherits** the properties and methods of the super class
    - Subclass also called ***child class***, ***subclass*** or ***derived class***
  - Subclass can extend the superclass by adding new properties/methods and/or overriding the superclass methods
- **Syntax**

```
class SubClass extends SuperClass { … }
```

- **Motivation**
  - Allow **code reuse**. **Common properties and methods are placed in a super class** then inherited by subclasses (i.e., avoids writing the same code twice to ease maintenance)

Inheritance – Person Example



- The Person class has the common properties and methods
- Each subclass can add its own specific properties and methods (e.g., `office` for Instructor and `gpa` for Student)
- Each subclass can **override** (redefine) the parent method (e.g., Instructor class overrode the toString() method).

Inheritance – Person Example

```
class Person { ... }

class Student extends Person {
  final double gpa;

  Student(String firstName, String lastName, DateTime dob, this.gpa)
      : super(firstName, lastName, dob);

   /*
    - Override a base class method
    - super keyword to call the implementation of the base class
   */
  @override
  String toString() => '${super.toString()}. GPA: $gpa';
}
```

Abstract Classes

- Idea
  - Use an abstract class when you want to define a **template** to guarantee that all **subclasses** in a hierarchy will have certain common methods
  - Abstract classes can contain implemented methods and **abstract methods** that are NOT implemented
- Syntax

```
abstract class SomeClass() {
    fun abstract method1(…): SomeType // No body
    fun method2(…): SomeType  { … } // Not abstract
}
```

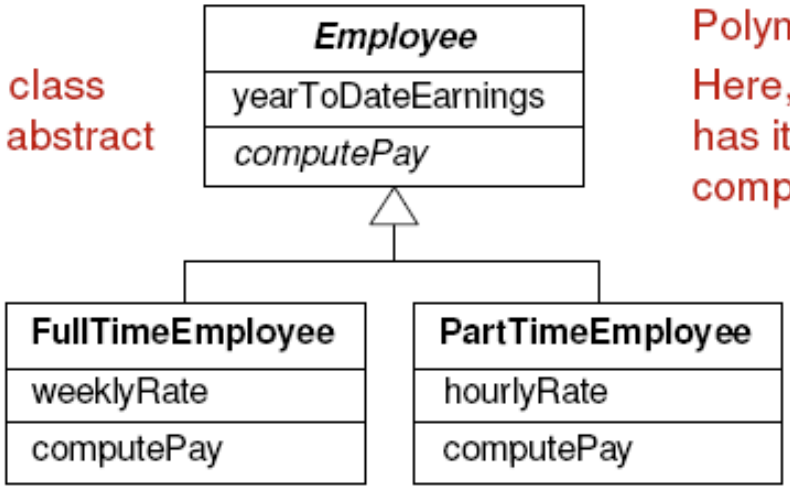
- Motivation
  - Guarantees that all subclasses will have certain methods => **enforce a common design**.
  - Lets you make collections of mixed type objects that can processed polymorphically

Abstract Classes

- An abstract class has one or more abstract properties/methods that subclasses MUST override
  - Abstract properties/methods do not provide implementations because they **cannot be implemented in a general way**
- An abstract class cannot be instantiated

Abstraction:

Employee is an abstract class and *computePay()* is an abstract operation (italicized)



Polymorphism:

Here, each type of Employee has its own version of computePay()

Abstract Class Example

Shape.kt

```
abstract class Shape {
    abstract val area: Double
    open val name
        get() = "Shape"
}
```

Rectangle.kt

```
class Rectangle(val width: Double,
                val height: Double) : Shape() {
    override val area
        get() = width * height

    override val name
        get() = "Rectangle"
}
```

Circle.kt

```
class Circle(val radius: Double) : Shape() {
    override val area
        get() = Math.PI * radius.pow(2)

    override val name
        get() = "Circle"
}
```

Interfaces

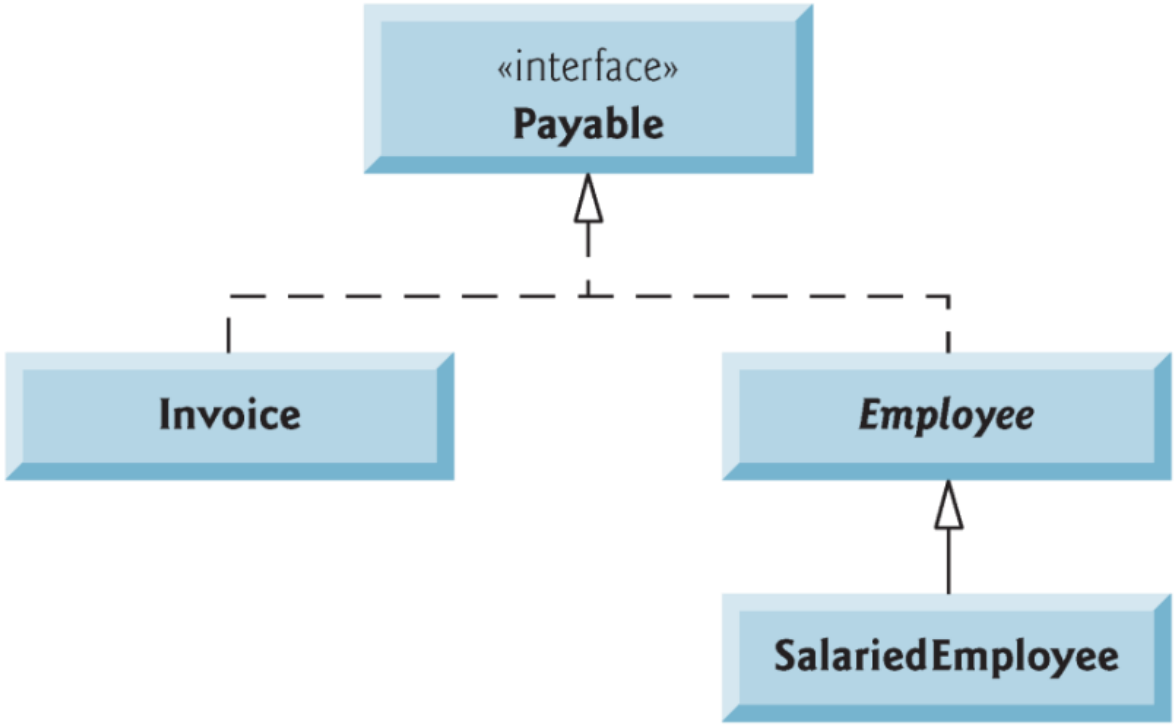
- Idea
  - **Interfaces** are used to define a set of common properties and methods that must be implemented by **classes not related by inheritance**
  - The interface specifies ***what*** methods a class must perform but does not specify ***how*** they are performed
- Syntax

```
interface SomeInterface {
   fun method1(…): SomeType  // No body
   fun method2(…): SomeType  // No body
}
class SomeClass() : SomeInterface {
  // Real definitions of method1 and method 2
}
```

- Motivation
  - Interfaces enables requiring that **unrelated classes implement a set of common methods**
  - Ensure consistency and guarantee that classes has certain methods:
    - Interface defines a **contract** that implementing classes must adhere to
  - Let us make **collections of mixed type** objects that can processed polymorphically

Interface Example

- A finance system has Employees and Invoices
- `Employee` and `Invoice` are not related by inheritance
- But to the company, they are both ***Payable***



Interface Example

Payable.kt

```
interface Payable {
 fun getPayAmount(): Double
}
```

Employee.kt

```
class Employee ( ... ) : Payable {
    ...
    override fun getPayAmount() = salary
    ...
}
```

Invoice.kt

```
class Invoice ( ... ) : Payable {
   ...
   override fun getPayAmount() = totalBill
   ...
}
```

Polymorphism Using interfaces

- A way of coding generically
  - way of referencing many related objects as one generic type
    - Cars and Bikes can both `move()` → refer to them as **`Transporter`** objects
    - Phones and Teslas can both `charge()` → refer to them as *`Chargeable`* objects, i.e., objects that implement **`Chargeable`** interface
    - Employees and invoices can both getPayAmount() → refer to them as *`Payable`* objects

```
for (payable : payables ) {
    println ( payable.getPayAmount() )
}
```

Abstract Class vs. Interface

- Abstract classes and interfaces cannot be instantiated
- Abstract classes and interfaces may have abstract methods that must be implemented by the subclasses
- Classes that implement an interface **can be from different inheritance hierarchies**
  - An interface is often used when unrelated classes need to provide **common properties and methods**
  - When a class implements an interface, it establishes a '**IS-A**' relationship with the interface type, enabling interface references to invoke polymorphic methods in a manner similar to how an abstract superclass reference can
- Concrete subclasses that extend an abstract superclass are **all related to one other by inheriting from a shared superclass**
- Classes can extend only ONE abstract class but they may implement more than one interface

Enum class

- Represents an enumeration

```
enum class Gender {
    FEMALE, MALE
}

enum class Direction {
    LEFT, RIGHT, UP, DOWN
}
```

Summary

- Inheritance = “factor out” the common properties and methods and place them in a single superclass

  => Removing code redundancy will result in a smaller, more flexible program that is easier to maintain.

- Interfaces are contracts, can’t be instantiated
  - force classes that implement them to define specified methods
- Polymorphism allows for generic code by using superclass/interface type variables to manipulate objects of subclass type
  - make the client code more **generic** and ease extensibility

Dart Resources

- Draft Language
  - Dart language tour https://dart.dev/language
- Dart learning resources
  - https://dart.dev/guides
  - https://dart.dev/tutorials
- Online Dart dev https://dartpad.dev/