



Dart

<https://dart.dev>

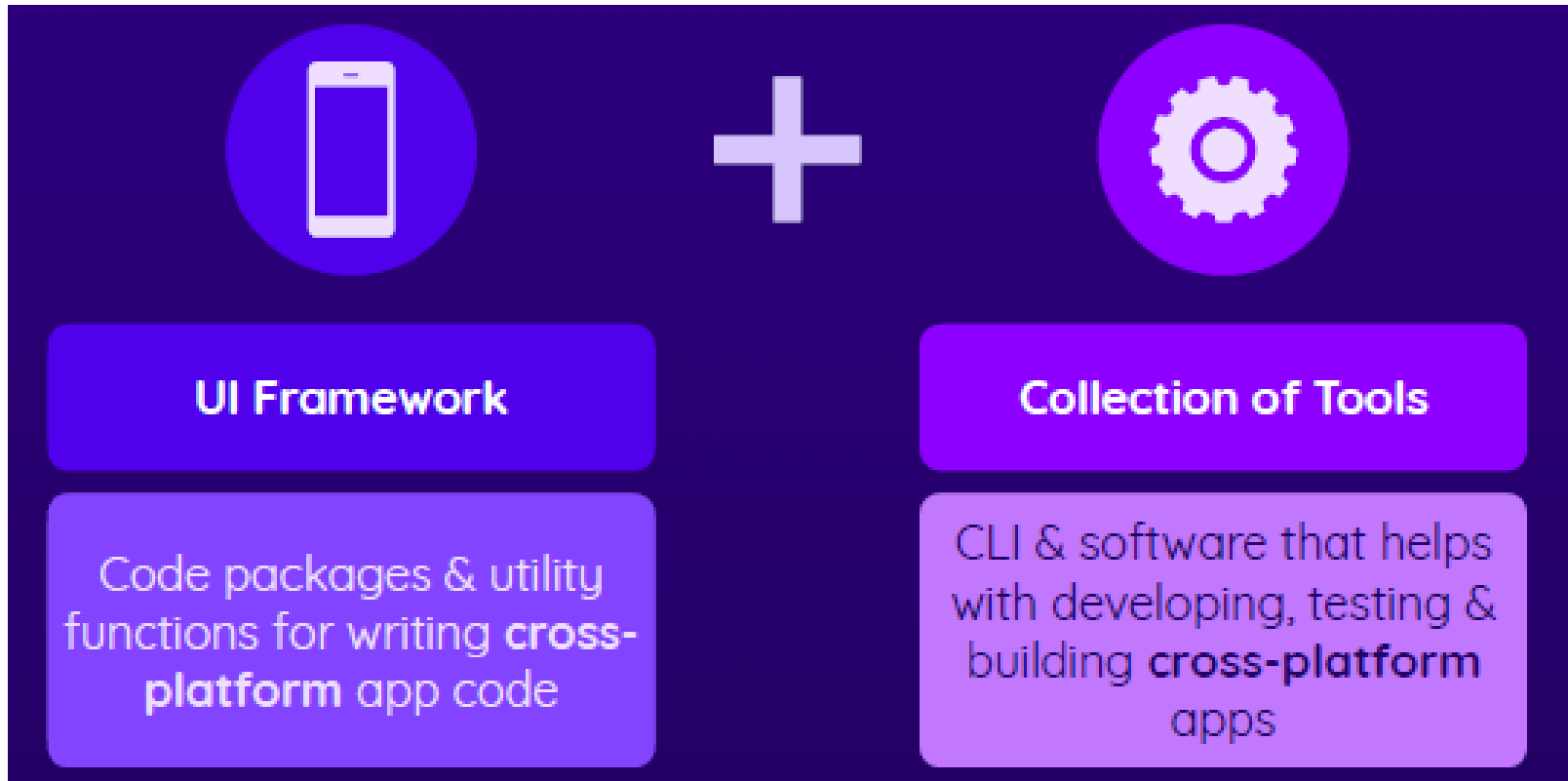
Table of Contents

1. Introduction to Flutter and Dart
2. Declaring Variables
3. Conditional statements: If & switch
4. Loops
5. Functions
6. OOP

Some of the slides are based on Flutter Complete Course [content](#)

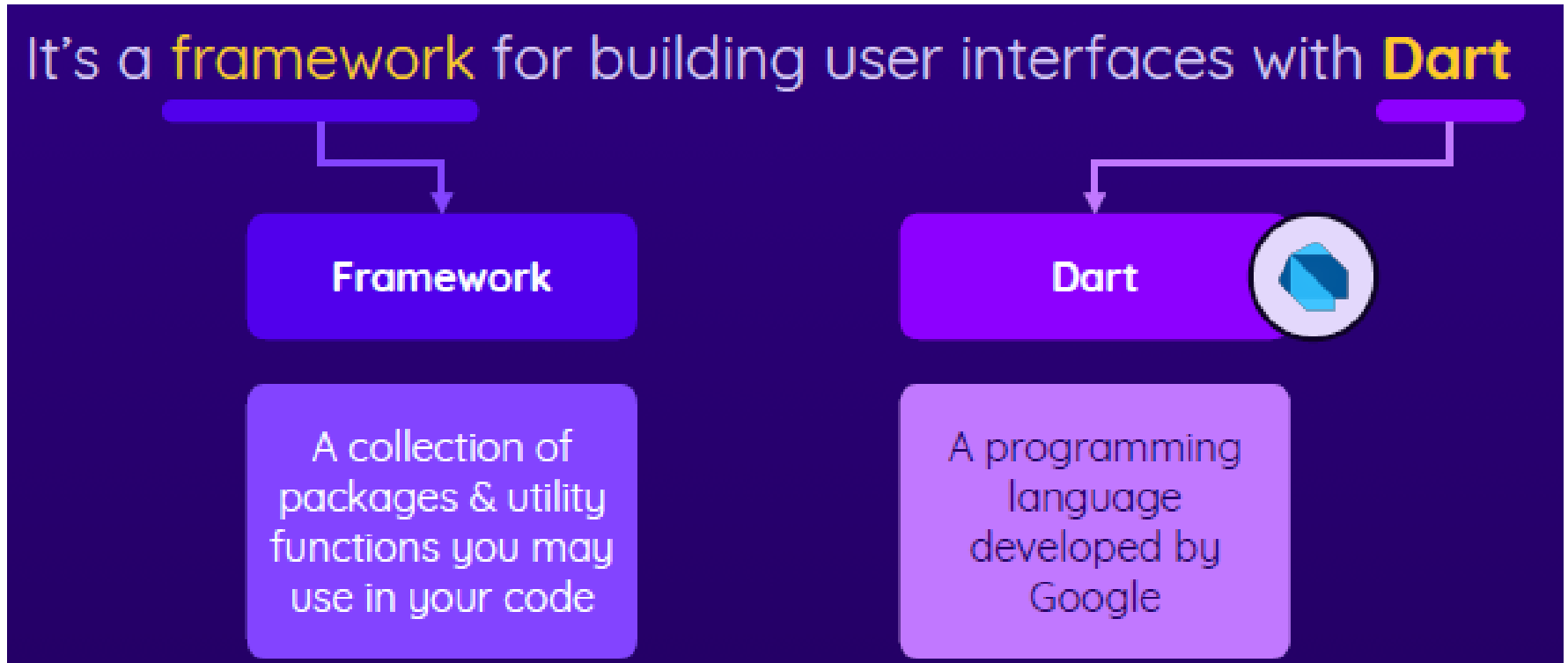
Introduction to Flutter and Dart

What is Flutter?



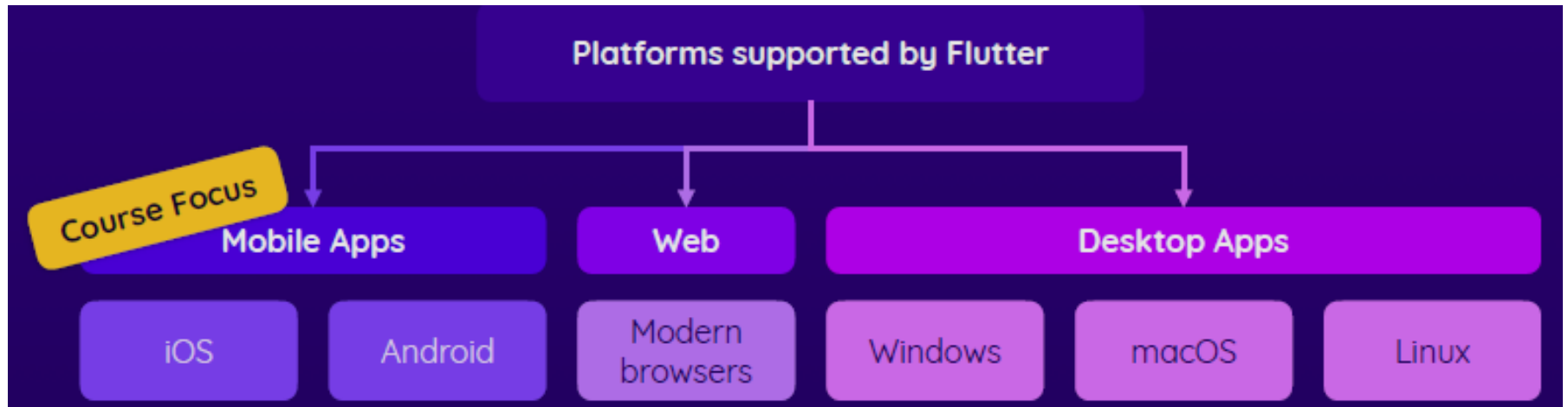
- Flutter uses **Dart** programming language to build **natively compiled** apps for **multiple platforms** from a **single codebase**

Flutter Is Not A Programming Language!



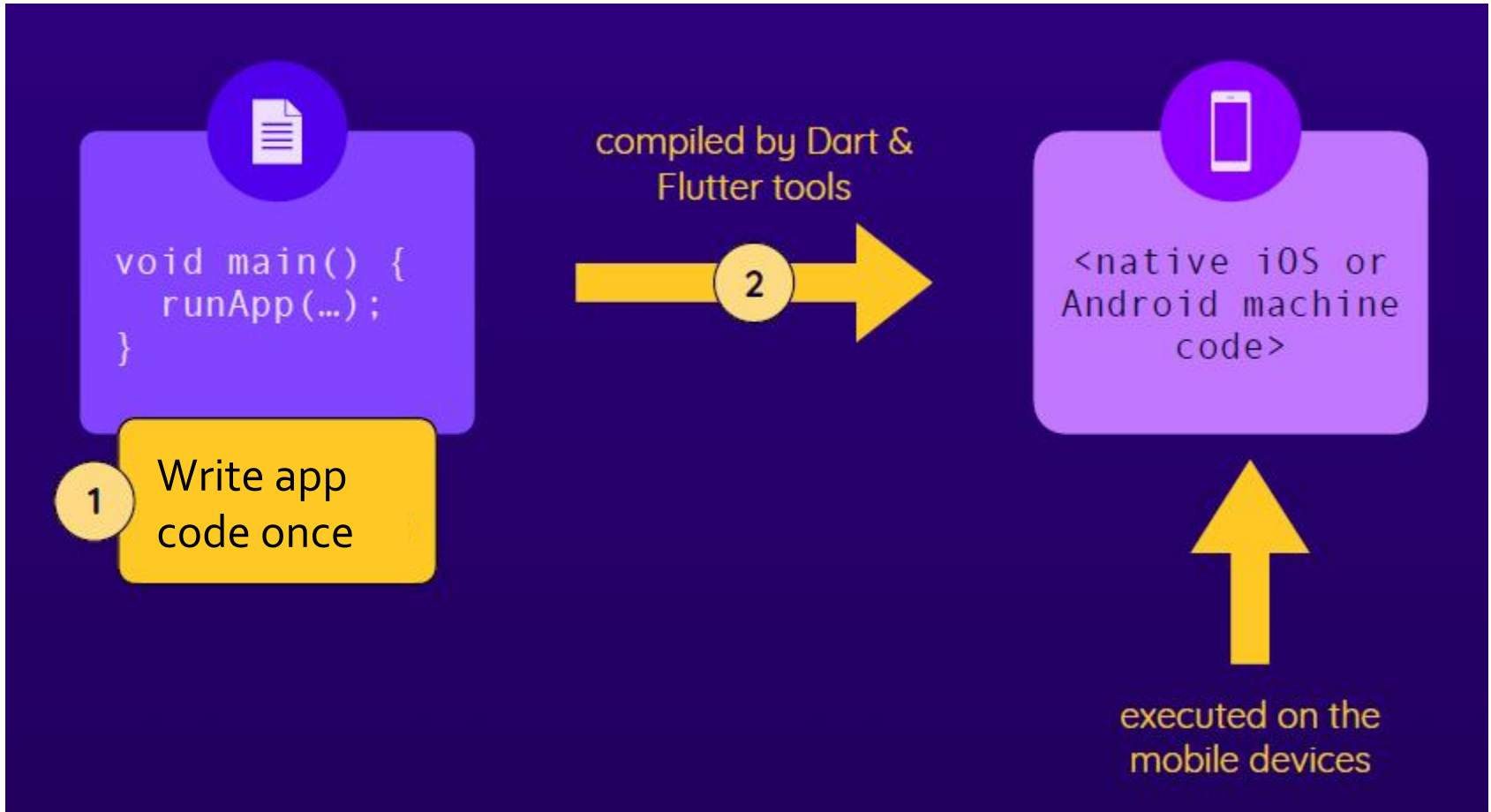
One Codebase, Multiple Apps

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

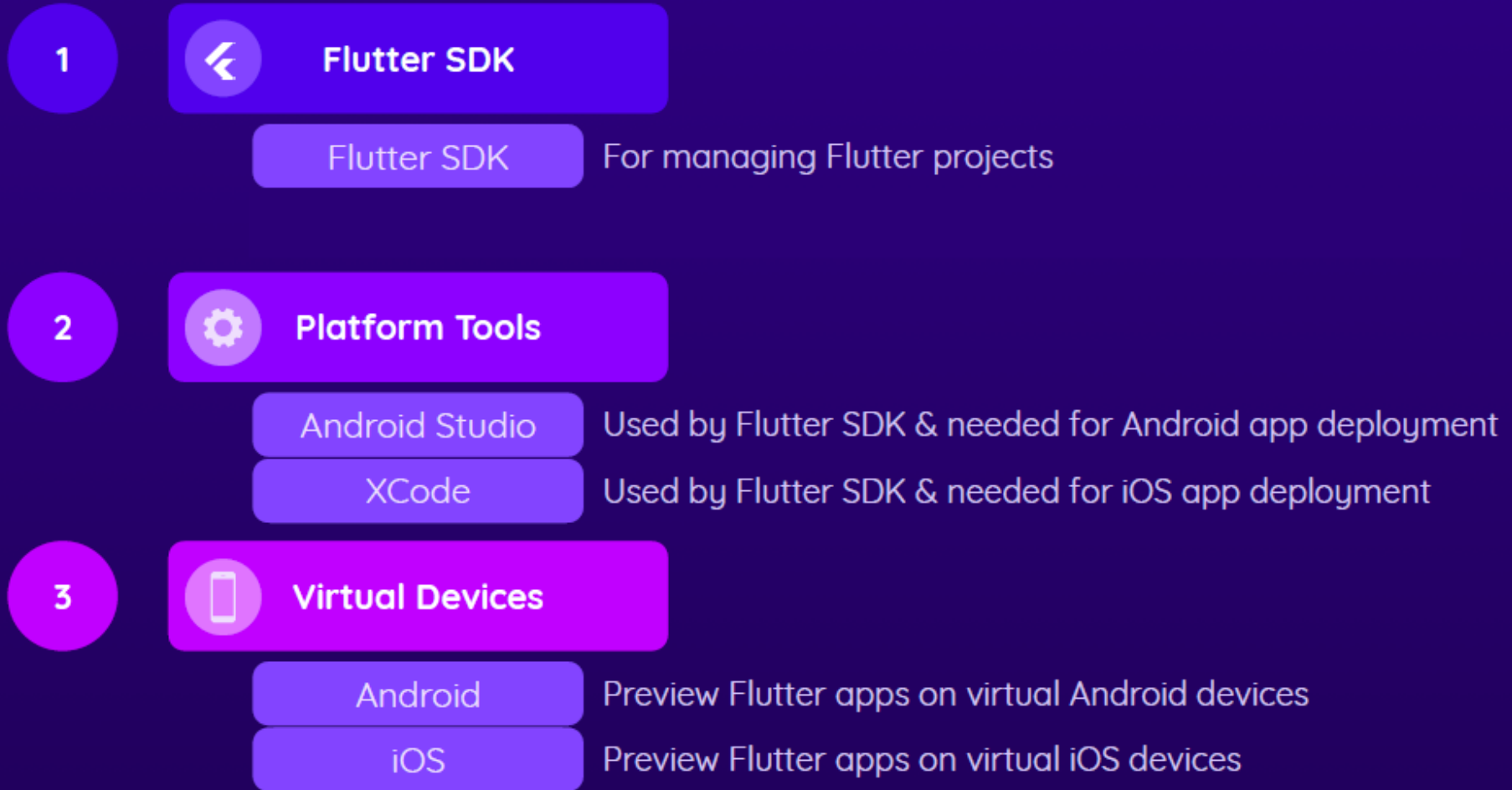


- Whilst you can write code for all platforms on the same machine, you can **only test & run iOS and macOS apps on macOS machine, Windows apps on Windows machine and Linux apps on Linux machine!**
- Android and web apps can be built and test on all operating systems

Dart & Flutter Code Is Compiled

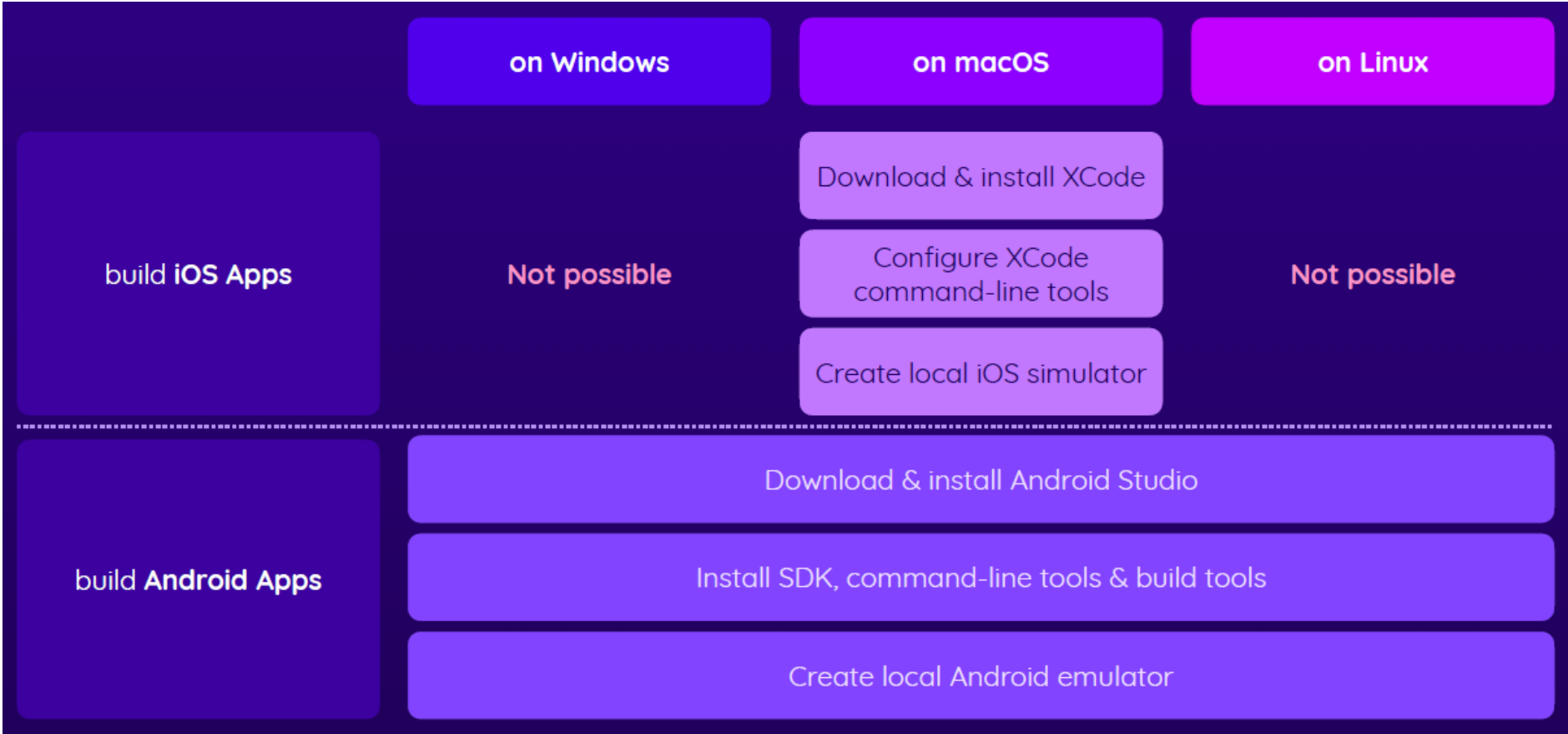


Flutter Setup



<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

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.

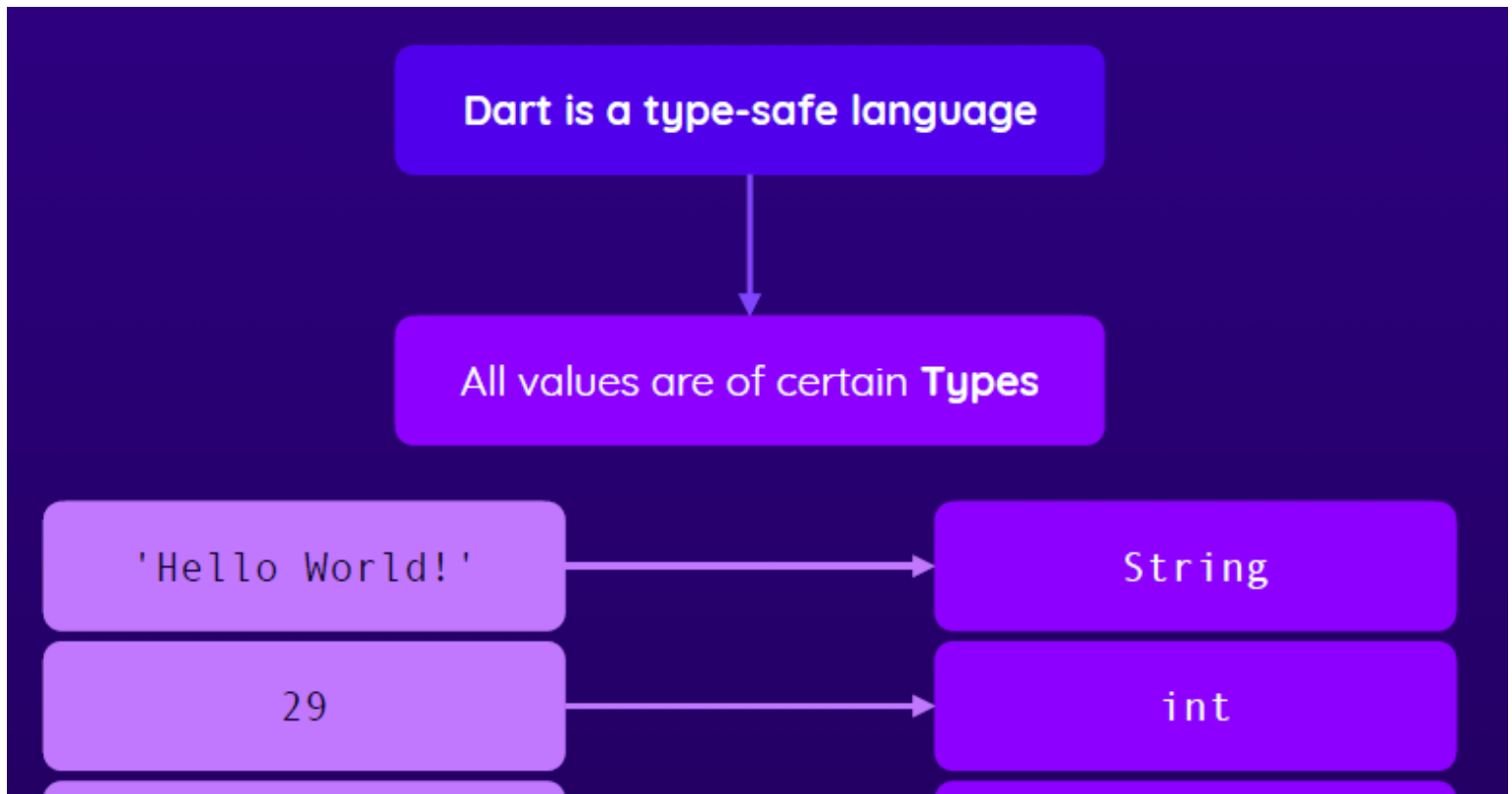
```
int age;
```
- **Literal**: value directly written in source code

```
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



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

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

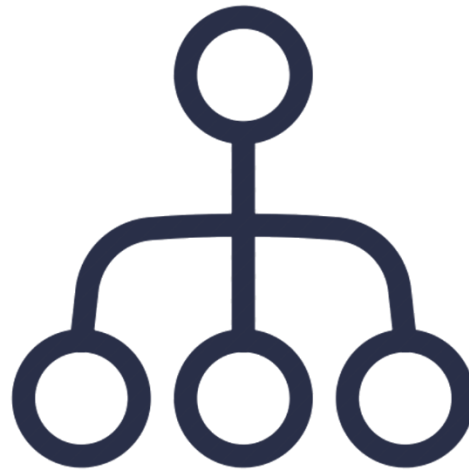
*/**

slash star

block comment

**/*

Control Flow: if, when expressions



if-else statement

```
var age = 20
var ageCategory = ""
// Using 'if' as an expression
if (age < 18) {
    ageCategory = "Teenager"
} else {
    ageCategory = "Young Adult"
}
```

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

```
var names = listOf("Sara", "Fatima", "Ali")
```

```
for (name in names) {  
    println(name)  
}
```

// Loop with index and value

```
for ( (idx, value) in names.withIndex()) {  
    println("$idx -> $value")  
}
```

Ranges

- Usually defined by: `1..100`
- `1 until 100` // Range excludes 100
- Negative step: `100 downTo 40`
- Decrement by 3
`100 downTo 40 step 3`

Caution!

```
var notARange = 100 to 40  
// => Pair(100, 40)
```

- To check if a value belongs to the range:
`var is5inRange = 5 in range`

Ranges

```
if (i in 1..10) { // 1 <= i && i <= 10    for (i in 1..4 step 2) print(i) // "13"
    println(i)
}

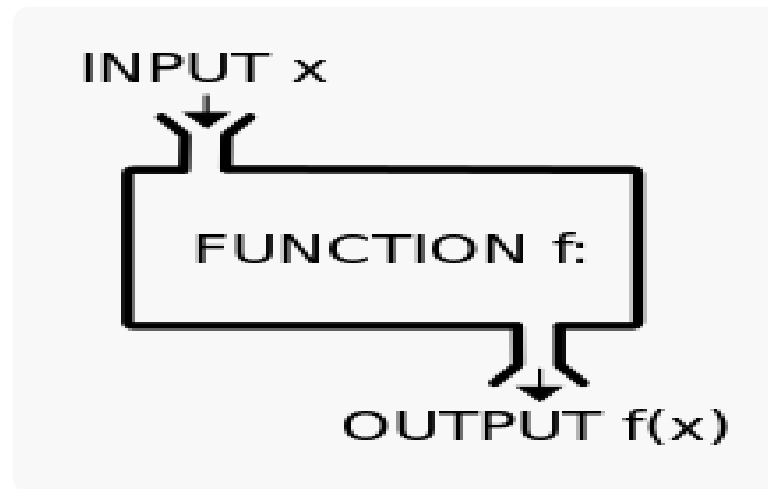
for (i in 1..4) print(i) // "1234"

for (i in 4 downTo 1 step 2)
    print(i) // "42"

for (i in 4..1) print(i) // No Output
for (i in 4 downTo 1)
    print(i) // "4321"

// i in [1, 10), 10 is excluded
for (i in 1 until 10) {
    println(i)
}
```

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 default parameter values to avoid method overloading
- Can use **named** arguments in a function call

```
fun max(a: Int, b: Int): Int { //name - parameters - return type
|   return if(a>b) a else b    //function block body
}
```

```
fun max(a: Int, b: Int) = if(a>b) a else b //expression body
```

```
max(a = 1, b = 2) //call with named arguments
max( a: 1, b: 2)
```

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 }
```

Unit return type

- When defining a function that doesn't return a value, we can use **Unit** as the return type (Unit is equivalent to void in Java)
 - Specifying Unit as a return type is NOT mandatory can omit it

```
fun display(value : Any) : Unit {  
    println(value)  
}
```

Use default parameters instead of function overloading

Function overloading

```
fun displayBorder() {  
    displayBorder('*', 20)  
}  
  
fun displayBorder(character: Char) {  
    displayBorder(character, 20)  
}  
  
fun displayBorder(character: Char, length: Int) {  
    for (i in 1..length) {  
        print(character)  
    }  
}
```

Default parameters

```
fun displayBorder(character: Char = '*', length: Int = 20)  
{  
    for (i in 1..length) {  
        print(character)  
    }  
}
```

```
fun main() {  
    displayBorder()  
    displayBorder('=')  
    displayBorder('=', 5)  
}
```

Extension Function

- Enable adding functions and properties to existing classes

// Extension method extending Int class

```
fun Int.isEven() = this % 2 == 0
```

```
fun main() {  
    var num = 10  
    println("Is $num even: ${num.isEven()}")  
}
```

Extension Function Example

```
fun String.lastChar() = this.get(this.length - 1)
```



this can be omitted

```
fun String.lastChar() = get(length - 1)
```

```
val c: Char = "abc".l
```

- λ lastChar() for String in com
- λ last {...} (predicate: (Char
- λ last() for String in kotlin
- λ lastOrNull {...} (predicate:
- λ lastOrNull() for String in k
- ✓ length

Infix function calls

- Functions marked with the **infix** keyword can be called using the infix notation (omitting the dot and the parentheses for the call)
- Infix function must satisfy 3 requirements:
 - Must be member function or extension function.
 - Must have a single parameter.
 - The parameter must not accept a variable number of arguments

```
infix fun Int.add(b : Int) : Int = this + b
```

```
fun main() {  
    var x = 10.add(20)  
    var y = 10 add 20    // infix call  
}
```

Exceptions

- Throw:

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

- Handling

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

- Expression

```
var num = try { input.toInt() }  
           catch (e: NumberFormatException){ null }
```


OOP


Java Class vs. Kotlin Class

```
public class Person {  
    private final String name;  
    private final int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

person.getName() 


 Concise primary constructor

```
class Person(  
    val name: String,  
    val age: Int  
)
```

person.name 

Class

```
class Person(val firstName: String,  
             val lastName: String,  
             val age: Int) {  
    val fullName  
        get() = "$firstName $lastName"  
    fun isUnderAge() = age < 18  
}
```



Properties

- **Instantiate:**

```
val student = Person ("Fatima", "Ali", 18)
```

- **Named arguments:**

```
val student = Person (firstName = "Fatima",  
                      lastName = "Ali", 18)
```

Properties are directly accessible without getters / setters

- `val` – read only properties
- `var` – read/write properties
- The primary constructor **cannot** contain any code

```
class Person(val firstName: String,  
             val lastName: String,  
             var age: Int) {  
    val fullName: String  
        get() = "$firstName $lastName"  
    fun isUnderAge() = age < 18  
}
```

```
val student = Person ("Fatima", "Ali", 18)  
student.age = 20
```

Class with a computed property

```
class Rectangle(val width: Double, val height: Double) {  
    val isSquare  
        get() = width == height  
}
```

Secondary Constructor

```
class Conference(val name: String,  
                 val city: String,  
                 val isFree: Boolean = true) {  
    var fee : Double = 0.0  
  
    // Secondary Constructor  
    constructor(name: String,  
                 city: String,  
                 fee: Double) : this(name, city, false) {  
        this.fee = fee  
    }  
}  
  
fun main() {  
    val conference = Conference("Kotlin Conf.", "Doha", 300)  
}
```

Data Classes

- Data classes provide autogenerated implementations of **equals()**, **hashCode()**, **copy()** and **toString()** methods

```
data class User(val name: String, val age: Int)
val ali = User(name = "Ali", age = 18)
```

//Copy:

```
val olderAli = ali.copy(age = 19)
```

//Destructuring

```
val (name, age) = ali
```

// prints "Ali, 18 years of age"

```
println("$name, $age years of age")
```

Use 'copy' method for data classes

```
class Person(val name: String,  
             var age: Int)  
  
fun happyBirthday(person: Person) {  
    person.age++  
}
```

```
data class Person(val name: String,  
                 val age: Int)  
  
fun happyBirthday(person: Person) =  
    person.copy(  
        age = person.age + 1)
```


object = singleton

- Once instance for the whole app

```
object Util {  
    fun getNumberOfCores() = Runtime.getRuntime().availableProcessors()  
  
    val randomInt  
        get() = Random().nextInt()  
}  
  
fun main() {  
    println(Util.getNumberOfCores())  
    println(Util.randomInt)  
}
```

No static keyword -> alternatives

- **Top-level** functions and properties
(e.g. placed outside the class)
- **Companion objects:** special object inside a class to place static properties and methods
- **object** declaration used to create a **Singleton** (i.e., a single instance for the whole app):
 - Used for declaring the class
 - And providing a single instance of it

```
class Foo {  
    companion object {  
        fun bar() {  
            //...  
        }  
    }  
}
```

```
object Singleton {  
    fun doSomething() {  
        //..  
    }  
}
```

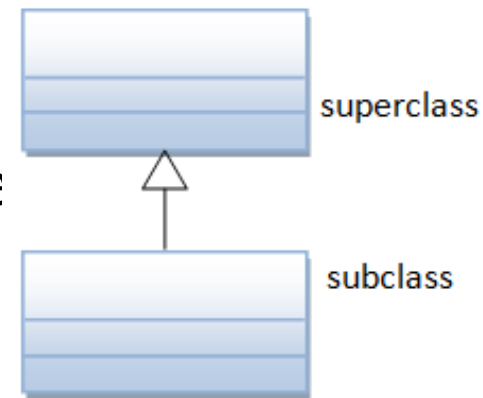
Foo.bar()

Singleton.doSomething()

Inheritance

- **Ideas**

- Common properties and methods are placed 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**



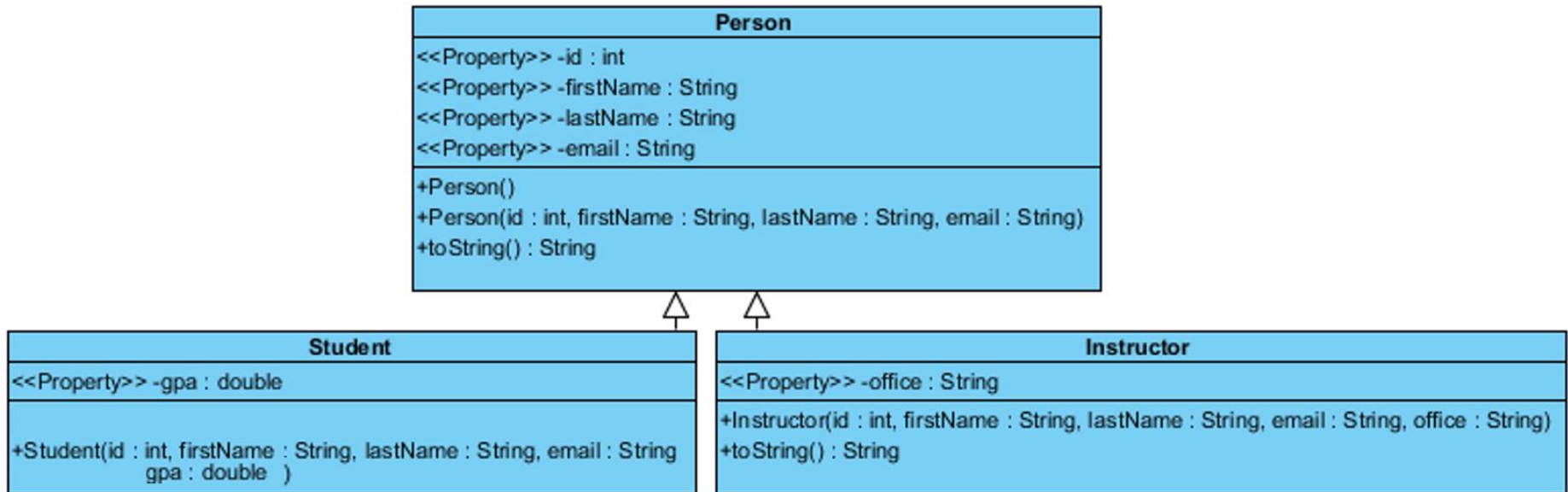
- **Syntax**

```
class SubClass( ... ) : 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

```
open class Person( ... ) { ... }
```

```
class Student(firstName: String,  
              lastName: String,  
              age: Int,  
              val gpa: Double  
              ) : Person(firstName, lastName, age) {  
  
    /*  
    - Override a base class method  
    - super keyword to call the implementation of the base class  
    */  
    override fun toString() = "${super.toString()}. GPA: ${gpa}"  
}
```

- Add **open** keyword to the base class and to properties and methods to be overridden

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

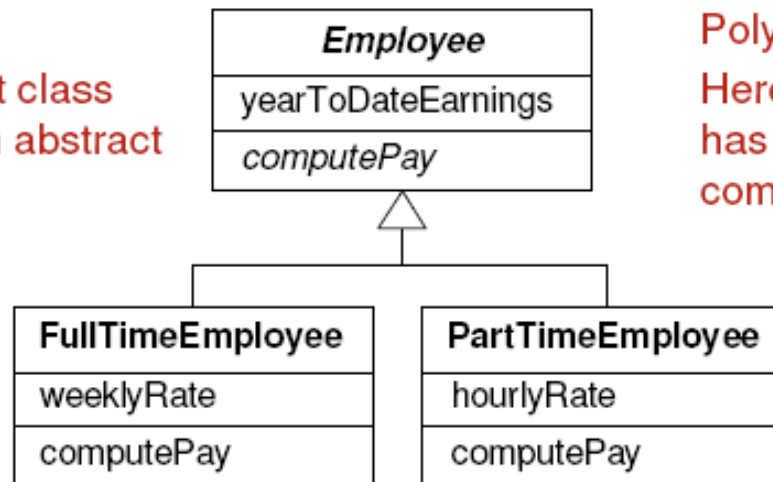
```
abstract class SomeClass() {  
    fun abstract method1(...): SomeType // No body  
    fun method2(...): SomeType { ... } // Not abstract  
}
```
- Motivation
 - Guarantees that all subclasses will have certain methods => **enforce a common design.**
 - Lets you make collections of mixed type objects that can be 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

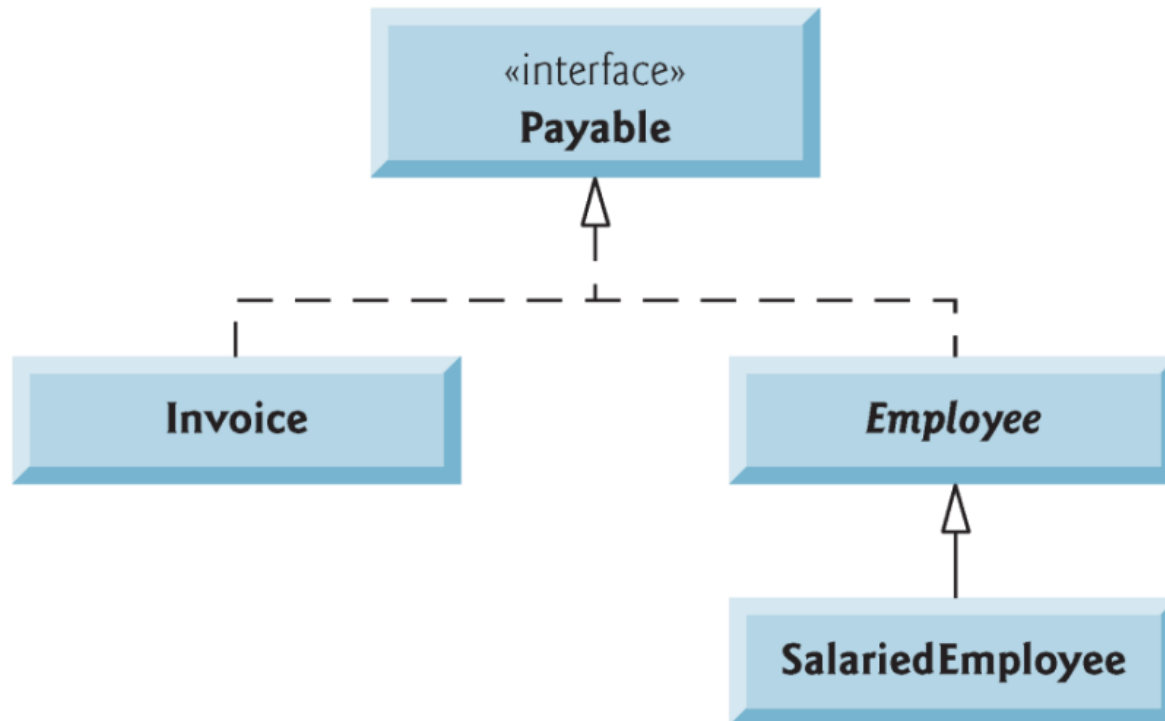
- Idea
 - **Interfaces** are used to define a set of common properties and methods that must be implemented by **classes not related by inheritance**
 - The interface specifies **what** methods a class must perform but does not specify **how** they are performed
- Syntax

```
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/>