Dart Language Walkthrough

Jumail Bin Taliba
School of Computing, UTM
February 2020

Outline

- Introduction to Dart
- Basic syntax
- Identifiers, Data Types, Variables and Naming Convention
- String Interpolation
- Enumeration
- Control Flow Statements
- Collections
- Generics
- Functions
- Object-Oriented Programming
- Asynchronous Programming

References:

https://dart.dev/guides/language/language-tour

Download the source code

1. Open `Git Bash`
2. Move to the directory where you want to store the code

   `cd` *<your_working_directory>*

   e.g : `cd d:/code/dart`

3. Download the code from my repository on github to your PC *(command below should be in one-line)*

   ```
   git clone https://github.com/jumail-utm/dart_walkthrough.git dart_walkthrough
   ```

4. Move into the directory

   ```
   cd dart_walkthrough
   ```

5. Open the source code into VS Code (*don't forget the dot*)

   ```
   code .
   ```

6. Update (and download) the Dart dependencies (aka packages) for the project.
   - Open Command Palette, `Ctrl Shift P`
   - Choose the menu `Pub:Get Packages`

Introduction to Dart

- Dart is an open-source, scalable programming language for building web, server and mobile apps
- Made by Google, Dart 1.0 was released on 2013
- Current version is 2.7 (2020)
- It is purely OOP, dynamic language with C style syntax
- It supports optional static typing and type checks
- Adopts single inheritance with mixins supports
- Influenced by Strongly type languages like Java, C++, C# and loosely type dynamic language like JavaScript

Basic Dart Program

```dart
void main(){
  print ('Hello World');
}
```

Variables

```
1   void main() {
2     int day;
3
4     day = 1;
5     print(day);
6     // day = 'Monday';
7   }
```

Uncommenting Line 6 will give an error. The variable *day* can only hold an integer value.

dynamic Types

```dart
void main() {
  dynamic day;

  day = 1;
  print(day);

  day = 'Monday';
  print(day);
}
```

Output:

```
1
Monday
```

var

```
1  void main() {
2    var day = 1;
3
4    print(day);
5    // day = 'Monday';
6  }
```

- Type of *day* determined by the type of the initializer.
- Uncommenting Line 5 will give an error. *day* is of type int.

var vs. dynamic

```
1   void main() {
2     var day;
3
4     day = 1;
5     print(day);
6
7     day = 'Monday';
8     print(day);
9   }
```

Dart infers the type of *day* as *dynamic*. Thus, the variable can hold any type of value.

const variables

```
1   void main() {
2     const double pi = 3.14;
3     //  pi = 3.151;
4     print(pi);
5   }
6
```

- *const* variables are compile-time constants
- Cannot be changed at all. Line 3 will produce an error

`final` variables (1)

```
1  void main() {
2    final int id = 1234;
3    //  id = 1001
4    print(id);
5  }
```

- *final* variables are run-time constants
- Can be set only once. Line 3 will produce an error

`final` variables (2)

```
1   void main(){
2     var oddNumbers = [1,3,5,7];
3     var evenNumbers = [2,4,6,8,10,12];
4     final numbers = oddNumbers;
5
6     numbers.add(11);
7     print('\nnumbers:');  print(numbers);
8     print('\noddNumbers:');  print(oddNumbers);
9
10    // numbers = evenNumbers;
11
12    print('\nevenNumbers:');  print(evenNumbers);
13  }
```

- Line 6 is fine as *numbers* remains pointing to the same list, *oddNumbers*.
- Line 10 will produce an error as we try to change numbers to another list, *evenNumbers*

final variables (3)

```
1   class Person {
2     final String name;
3     int age;
4
5     Person(String name, int age)
6         : this.name = name,
7           this.age = age;
8
9     void show() {
10      print(name);
11      print(age);
12    }
13  }
14
15  void main() {
16    var p = Person('Ahmad', 20);
17    p.show();
18
19    // p.name = 'Abu';
20    p.age = 21;
21    p.show();
22  }
```

- Instance variable *name* of object *p* can only be set once as it is *final*.
- Line 19 will produce an error

Constant Variables vs. Constant Values

```
1   void main(){
2     final oddNumbers  = [1,3,5];
3     final evenNumbers = const [2,4];
4
5     oddNumbers.add(7);
6     // evenNumbers.add(6);
7     print('\noddNumbers:');  print(oddNumbers);
8     print('\nevenNumbers:');  print(evenNumbers);
9   }
10
```

- Line 5 is fine as *oddNumbers* remains pointing to the same list.
- Line 6 will produce an error. evenNumbers is pointing to a constant list.
- *oddNumbers* and *evenNumbers* are called constant variables.
- *const [2,4]* are constant values.

Naming Convention

Functions and variables:

- Camel-case starting with small letter
- Each word begins with a capital letter

```
myFirstFunction()
variableName
```

Files and directories:

- Lower case
- Words are separated by underscores.

```
my_program.dart
folder
directory_name
```

User-defined Data Types

- Camel-case starting with Capital letter
- Each word begins with a capital letter

```
class Person{}
enum DayName{}
```

String Interpolation

```dart
void main(){
  var name ='Ali';
  var yearOfBirth = 1999;
  var score = 85.0;

  print ("Student's name: $name \t age: ${2020-yearOfBirth}");
  print ('Test score: $score \t Result: ${ testResult(score)} ');
}

String testResult(double s){
  if (s>50) return 'Pass';
  return 'Fail';
}
```

Output:

```
Student's name: Ali    age: 21
Test score: 85.0       Result: Pass
```

Enumeration

To define named constant values

```
1   enum Days{
2     monday, tuesday, wednesday,
3     thursday,friday, saturday, sunday
4   }
5
6   void main(){
7     var today = Days.thursday;
8
9     print(Days.values);
10    print(today);
11    print(today.index);
12    if (today==Days.thursday) print('Today, the office is half-day');
13  }
```

Output:

```
[Days.monday, Days.tuesday, Days.wednesday, Days.thursday,
Days.friday, Days.saturday, Days.sunday]
Days.thursday
3
Today, the office is half-day
```

Control Flow Statements

- If-else
- For loops
- While and do-while loops
- Break and continue
- Switch and case

If and else

```
 6       if (score >=80) {
 7         grade = 'A';
 8       }
 9       else if (score >=70) {
10         grade = 'B';
11       }
12       else if (score >=60) {
13         grade = 'C';
14       }
15       else if (score >=50) {
16         grade = 'D';
17       }
18       else{
19         grade = 'E';
20       }
21     }
```

For loops

```
26  var sum = 0;
27  for (var n=10; n<100; n += 10 ){
28      sum += n;
29  }
30  print (sum);
```

While and do-while loops

```
while (!isDone()) {
  doSomething();
}
```

```
do {
  printLine();
} while (!atEndOfPage());
```

break and continue

```
38    for (var i=0; i<100; i++){
39      if (i>=50) break; // stop printing at i=50
40      print(i);
41    }
```

```
45
46    // Iterate all numbers but print only the odd numbers
47    for (var i=1; i<100; i++){
48      if ((i % 2)==1) continue;
49      print(i);
50    }
51
```

Switch and case

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}
```

Collections

- Lists
- Sets
- Maps

List Collection

A list represents an array

```
1  void main() {
2    var list = [10, 20, 30]; // List literals
3    print(list);
4
5    // Iterating the list
6    var sum =0;
7    for (var i=0; i<list.length; i++){
8      sum += list[i];
9    }
10
11   print ('The sum is $sum');
12
13   list.add(40);       // list = [10,20,30,40]
14   list.insert(0, 9); // list = [9,10,20,30,40]
15   list.removeAt(2);  // list = [9,10,30,40]
16   print(list);
17 }
```

Output:

[10, 20, 30]
The sum is 60
[9, 10, 30, 40]

Set Collection

A set contains unique items

```
1  void main() {
2    var a = {'milk','egg','bread'};
3    var b = {'egg','rice'};
4    var c = a.intersection(b);
5    var d = a.union(b);
6
7    print('Set C: $c \t count: ${c.length}');
8    print('Set D: $d \t count: ${d.length}');
9
10   c.add('milk');
11   d.add('milk');
12   print(c);
13   print(d);
14 }
```

Output:

Set C: {egg} count: 1
Set D: {milk, egg, bread, rice} count: 4
{egg, milk}
{milk, egg, bread, rice}

Map Collection

A map contains items in a form of key and value pairs

```dart
void main() {
  var wordNumbers = {
    'one' : 1, 'three' : 3, 'seven' : 7, 'ten' : 10
  };

  var numberWords = {
    1: 'one', 3: 'three', 7: 'seven', 10: 'ten'
  };

  var firstWord  = 'three', secondWord = 'seven';
  var firstNumber = wordNumbers[firstWord];
  var secondNumber = wordNumbers[secondWord];
  var result = firstNumber + secondNumber;

  print('$firstWord plus $secondWord is ${numberWords[result]}');
}
```

Output:

three plus seven is ten

Advanced Operations on Collections

- Collection If
- Collection for
- Spread operator
- High-order methods

Collection if Operations

Conditionally add items to collections

```
1   void main() {
2     var online = true;
3     var signedin = false;
4
5     var components = [
6       'Menu',
7       'Navigation Bar',
8       if (signedin) 'Show Avatar' else 'Show Random Image',
9       if (online) 'Show network'
10    ];
11
12    print (components);
13  }
```

Output:

[Menu, Navigation Bar, Show Random Image, Show network]

Collection for Operations

Add items to collections using loops

```dart
void main() {
  var users = ['User 1', 'User 2', 'User 9'];
  var buttons = [
    'Ok',
    'Cancel',
    for (var user in users) 'Add $user'
  ];

  print (buttons);
}
```

Output:

[Ok, Cancel, Add User 1, Add User 2, Add User 9]

Spread operator

Add multiple items into collections

```
1   void main() {
2     var list1 = ['A', 'B'];
3     var list2 = ['P', 'Q', 'R'];
4     var list3 = [list1, 'D','E', list2];
5     var list4 = [...list1, 'D','E', ...list2];
6
7     print (list3);
8     print (list4);
9   }
```

Output:

[[A, B], D, E, [P, Q, R]]
[A, B, D, E, P, Q, R]

Function Binding

A variable can hold a function

```
1   int ten() {
2     return 10;
3   }
4
5   void main() {
6     dynamic f;
7
8     f = ten(); // a normal function call
9     print (f) ;
10
11    f = ten; // this is function binding, not a function call
12    print (f) ;
13    print ( f() ) ;
14  }
```

Output:

10

Closure: () => int from Function 'ten': static.

10

High-order Functions vs Callbacks (1)

- Functions can also be sent as parameters to other functions.
-
- These functions are called callback functions
- The receiving functions (or the called functions) are called high-order functions.
- These are some common characteristics of functional programming paradigm.

High-order Functions vs Callbacks (2)

- In the following example, functions add, times and substract are callback functions
- doCalculation is a high-order function

```
1   int add(int a, int b){return a + b;}
2   int times(int a, int b){return a * b;}
3   int substract(int a, int b){return a - b;}
4
5   void doCalculation(int x, int y, Function callback){
6     int result = callback(x,y);
7
8     print('Result: $result');
9   }
10
11  void main() {
12    doCalculation(1,2, add);
13    doCalculation(5,4, times);
14    doCalculation(6,9, substract);
15  }
```

Output:

Result: 3
Result: 20
Result: -3

Lambda Functions

- A callback can be directly written to the high-order function.
- This is called Lambda function (or Anonymous function, i.e. no name)

```
1   void doCalculation(int x, int y, Function callback){
2     int result = callback(x,y);
3
4     print('Result: $result');
5   }
6
7   void main() {
8     doCalculation(1,2,  (a,b){return a+b;}  );
9
10    // using shorthand notation using arrow function
11    doCalculation(10,20,  (a,b)=>a+b  );
12  }
```

Collections and High-order Methods (1)

Collections come with several high order methods

```dart
void callback(int item){
  print('Number $item');
}

void main() {
  var numbers = [10, 1, 5, 7];

  numbers.forEach( callback);

  // or using lambda function
  // numbers.forEach((int item)=>print('Number $item') );
}
```

Output:

Number 10
Number 1
Number 5
Number 7

Collections and High-order Methods (2)

```dart
void main() {
  var list = [1, 3, 6, 7];

  // sum all the numbers in the list
  var sum = list.reduce((total, item)=>total+item );

  // how many odd numbers in the list?
  var odd = list.reduce((count, item)=> (item % 2==1) ? count + 1 : count );

  print('Sum = $sum');
  print('Number odds = $odd');
}
```

Output:

Sum = 17
Number odds = 3

Generics (1)

Generics are useful for type safety

```
2   var list1 = <int>[1, 2, 0];
3   var list2 = List<String>();    Use collection literals when p
4   list2.add('Generics');
5   // list2.add(10);
6
7   var pages = <String, String>{
8     'index.html': 'Homepage',
9     'robots.txt': 'Hints for web robots',
10    'humans.txt': 'We are people, not machines'
11  };
12
13  var errors = Map<int, String>();    Use collection literals w
14  errors.addAll({200: 'Success', 404: 'Server unreachable'});
15  // errors.addAll({'Error#404' : 'Server error'});
16
```

Line 5 and 15 will produce errors

Generics (2)

Generics are also useful for reducing code duplication

Without generics: code duplication in two classes

```
1   class IntegerAddition{
2       int first,second;
3
4       IntegerAddition(int f, int s)
5       { first = f;
6         second = s;
7       }
8
9       int get add =>first + second;
10  }
11
```

Generics (3)

Generics are also useful for reducing code duplication

Without generics: code duplication in two classes

```
12    class StringAddition{
13      String first,second;
14
15      StringAddition(String f, String s)
16      { first = f;
17        second = s;
18      }
19
20      String get add =>first + second;
21    }
```

Generics (4)

Generics are also useful for reducing code duplication

Without generics: code duplication in two classes

```
23  void main() {
24    var i=IntegerAddition(1,2);
25    var s=StringAddition('Hello', 'World');
26    print (i.add);
27    print (s.add);
28  }
29
```

Generics (5)

Generics are also useful for reducing code duplication

With generics: reduce code duplication

```
1   class Addition<T> {
2     T first;
3     T second;
4
5     Addition(T f, T s){
6       first=f;
7       second=s;
8     }
9
10    T get add => (first as dynamic) + (second as dynamic);
11  }
12
13  void main() {
14    var i=Addition<int>(1,2);
15    var s=Addition<String>('Hello', 'World');
16
17    print (i.add);
18    print (s.add);
19  }
```

Functions

- Define functions
- Shorthand
- Function parameters
- Positional parameters
- Named parameters
- Default parameter values

Defining Functions

- Function declaration and definition are in the same place
- Dart supports hoisting. You may write the function call on top of the function definition

```
1   void main(){
2     var area = rectangleArea(10,20);
3     print ("Rectangle's area is $area");
4   }
5
6   double rectangleArea(double width, double height){
7     return width * height;
8   }
```

Output:

Rectangle's area is 200.0

Shorthand notations

Use arrow syntax for shorthand. However, applicable only to an expression.

```
5
6   double rectangleArea(double width, double height) => width * height;
7
```

Function Parameters

- A function can have required and optional parameters.
- The required parameters are listed first, then followed by the optional.
- An optional parameter can be either a named or positional parameter.
- A parameter can be specified with a default value

Positional Parameters

Positional parameters are enclosed in [ ]

```
1
2   double area( [double width=1, double height=2] ) => width * height;
3
4   void main(){
5     print ("Rectangle #1's area is ${area()}");
6     print ("Rectangle #2's area is ${area(2,3)}");
7     print ("Rectangle #3's area is ${area(5)}");
8   }
9
```

Output:

Rectangle #1's area is 2.0
Rectangle #2's area is 6.0
Rectangle #3's area is 10.0

Named Parameters

- Named parameters are enclosed in { }
- To make parameters mandatory, annotate them with @required

```dart
import 'package:meta/meta.dart';

double area( {@required double width, double height=2} ) => width * height;

void main(){
  print ("Rectangle #1's area is ${area(width:2, height: 3)}");
  print ("Rectangle #2's area is ${area(height:5, width: 3)}");
  print ("Rectangle #3's area is ${area(width:5)}");

  // print ("Rectangle #4's area is ${area(height:1)}"); // error: width is required
}

```

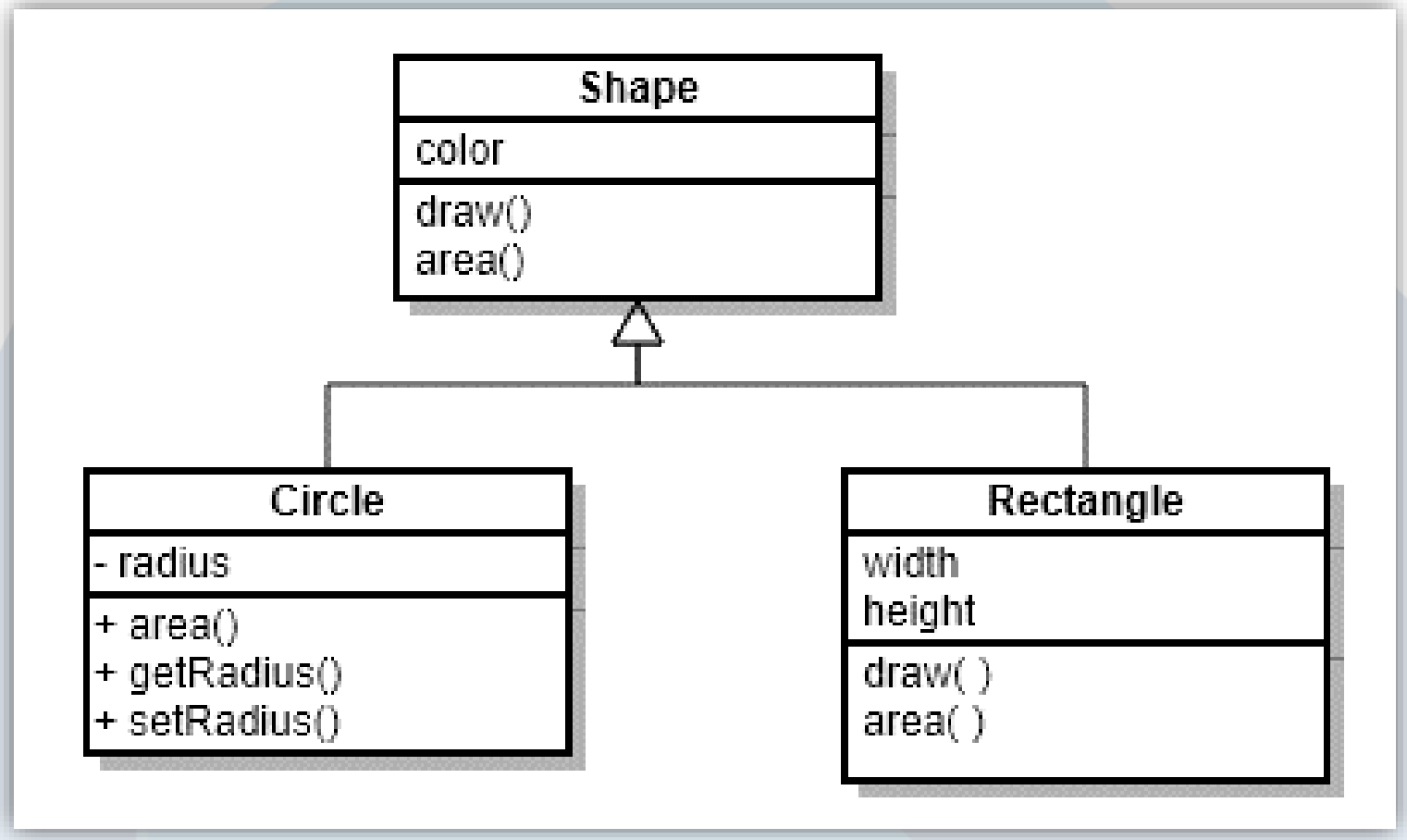
Output:

Rectangle #1's area is 6.0
Rectangle #2's area is 15.0
Rectangle #3's area is 10.0

Object-Oriented Programming

- Define classes and create objects
- Abstract Classes
- Constructors
- Inheritances
- Overridden Methods
- Cascade Notations
- Interfaces
- Mixins

Example problem



Abstract Classes

- An abstract class cannot be instantiated.
- It can have methods with or without definition.

```
3   abstract class Shape {
4     String color;
5
6     Shape([this.color]);
7     void draw() => print('Painted in $color');
8     double get area; // This method is not defined yet here
9   }
```

Inheritance (1)

- Dart adopts single inheritance model.
- Use the keyword *extends*.
- Child classes inherit all members from the parent class.
- Child classes can access to parent's members with *super*

```dart
class Circle extends Shape {
  double _radius;

  Circle(this._radius, String color) : super(color);
  double get radius => _radius;
  set radius(double value) => _radius = value;

  @override
  double get area => pi * _radius * _radius;
}
```

Inheritance (2)

```
25    Rectangle({this.width, this.height, String color}) : super(color);
26    Rectangle.square(double size) : this(width: size, height: size, color:'White');
27
28    @override
29    double get area => width * height;
30
31    @override
32    void draw() {
33      print('It is a rectangle');
34      super.draw();
35    }
36  }
```

Overridden Methods

- Override a method if a child class need to have a new version of the method.
- Annotate each method to be overridden with *@override*

```
11  class Circle extends Shape {
12      double _radius;
13
14      Circle(this._radius, String color) : super(color);
15      double get radius => _radius;
16      set radius(double value) => _radius = value;
17
18      @override
19      double get area => pi * _radius * _radius;
20  }
```

Creating Objects (1)

- Call to the class constructor when creating objects.
- No need to explicitly write the *new* operator.

```
38  void main() {
39    var c = Circle(20,'Yellow');
40    var r = Rectangle(height: 2, color: 'Green', width: 5);
41    var s = Rectangle.square(3);
42
43    print("Circle's area is ${c.area} ");
44    c.draw();
45
46    print("\nRectangle r's area is ${r.area} ");
47    r.draw();
48
49    print("\nSquare s's area is ${s.area} ");
50    s.draw();
51  }
```

Creating Objects (2)

Output:

```
Circle's area is 1256.6370614359173
Painted in Yellow

Rectangle's area is 10.0
It is a rectangle
Painted in Green

Square's area is 9.0
It is a rectangle
Painted in White
```

Cascade Notations

- Cascades are shorthand to make sequence of operations on the same object in a single statement.
- Use the notation . . (*two dots*)

```
55    Rectangle.square(20)
56      ..color = 'Yellow'
57      ..draw();
```

This code is equivalent to:

```
61    var square = Rectangle.square(20);
62    square.color='Yellow';
63    square.draw();
```

Interface (1)

- Interface allows a class to have method names from other classes
- The class must implement the methods (i.e. re-define the methods)
- Use the keyword *implements*.

```
1   abstract class Controller{
2     void turnOn();
3     void turnOff();
4   }
5
6   class Setting{
7     void setLanguage() {print ('The language is set to English');}
8   }
```

Interface (2)

```dart
class Television implements Controller, Setting {
  String _channel;

  Television(this._channel);
  @override
  void turnOn() {
    print('The TV is currently turned ON on channel $_channel');
  }

  @override
  void turnOff() {
    print('The TV is OFF');
  }

  @override
  void setLanguage() {
    print('The language is set to Bahasa Melayu');
  }

  set channel(String newChannel) {
    print('Switching channel from $_channel to ${_channel = newChannel}');
  }
}
```

Interface (3)

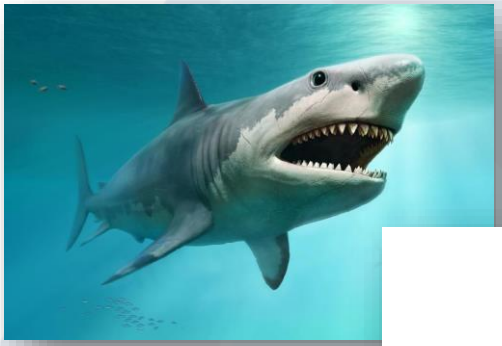
```
36  void main() {
37    var tv = Television('TV3');
38    tv.turnOff();
39    tv.turnOn();
40    tv.channel = 'NTV7';
41    tv.setLanguage();
42  }
```

Output:

The TV is OFF
The TV is currently turned ON on channel TV3
Switching channel from TV3 to NTV7
The language is set to Bahasa Melayu

Mixin

- Mixin allows adding features to a class.
- Adding features to a class can be done with inheritance.
- So why mixin?

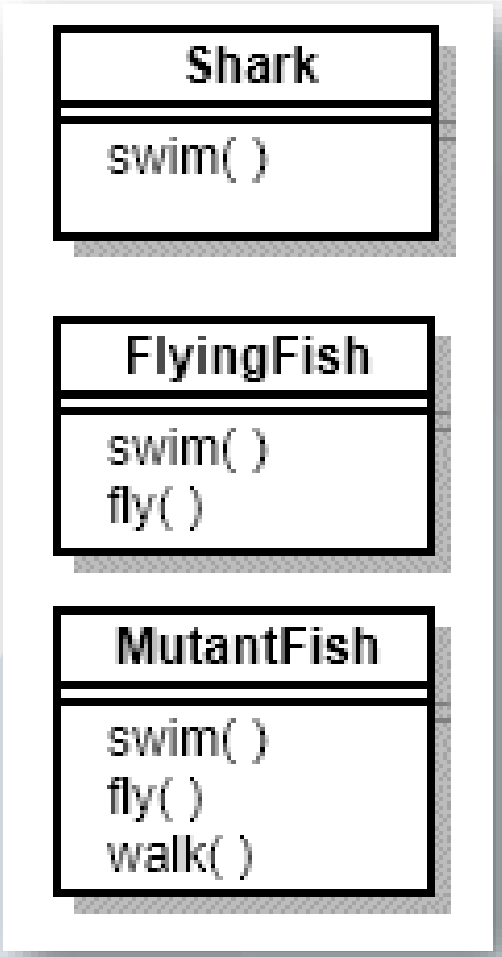


Shark swims

Flying fish swims and flies

Mutant fish swims, flies and walks

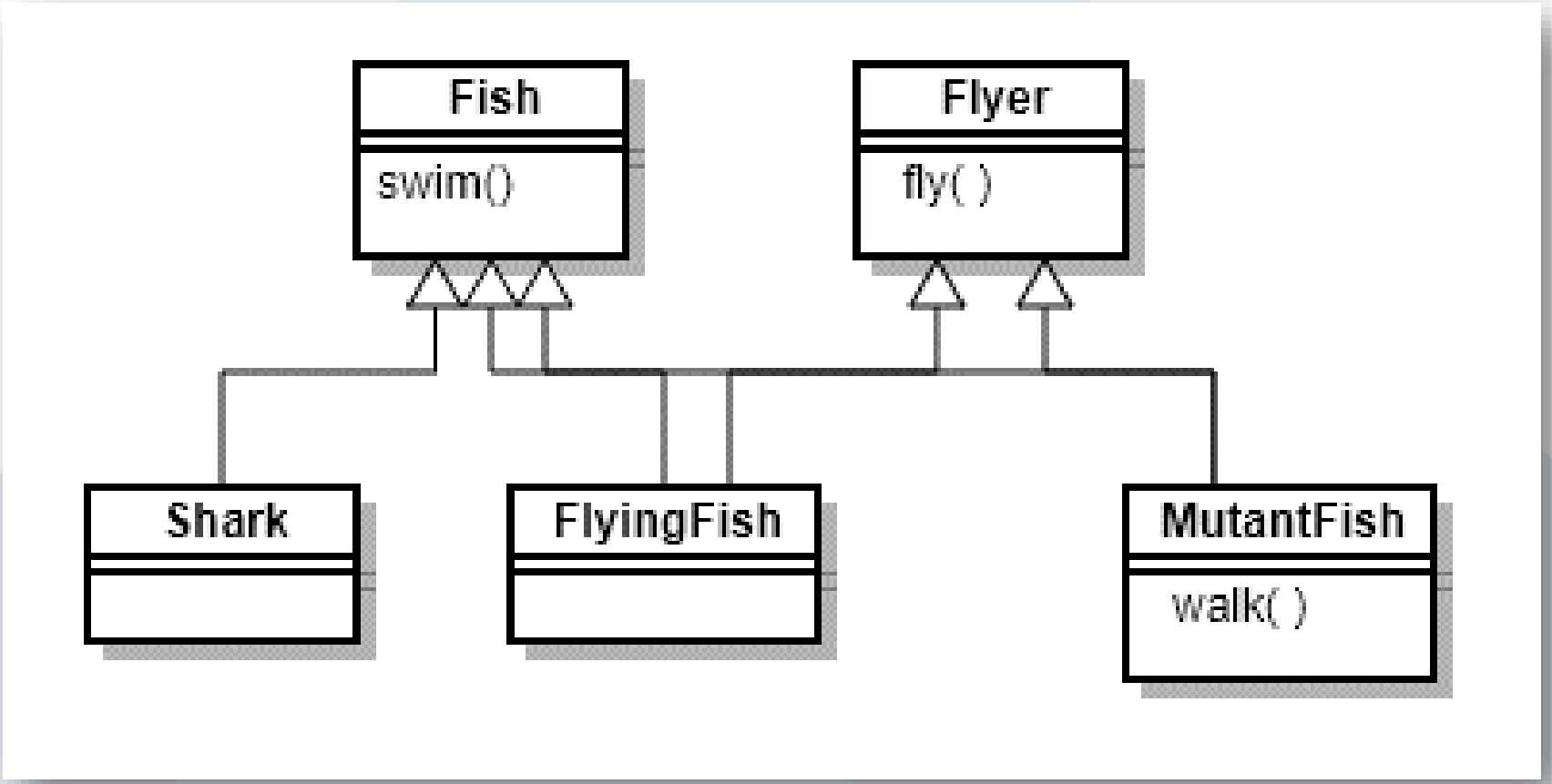
Solution 1: without inheritance




```
1  class Shark{
2      void swim(){print('It swims');}
3  }
4
5  class FlyingFish{
6      void swim(){print('It swims');}
7      void fly(){print('It flies');}
8  }
9
10 class MutantFish{
11     void swim(){print('It swims');}
12     void fly(){print('It flies');}
13     void walk(){print('It walks');}
14 }
```

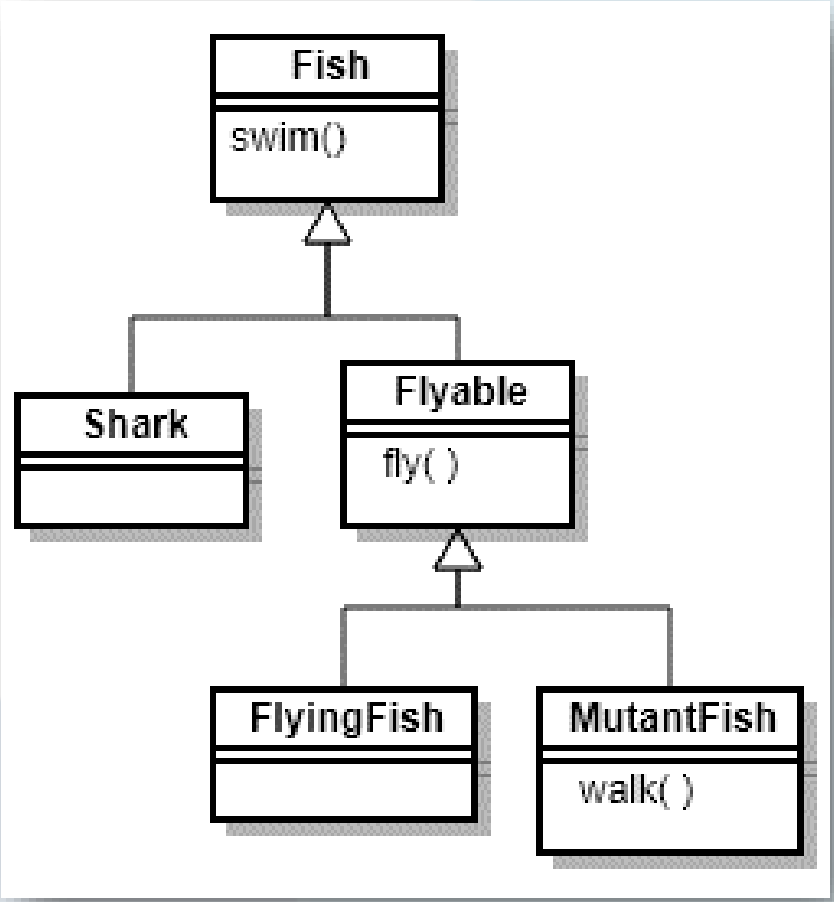
Problem: Code duplication

Solution 2: Multiple Inheritance



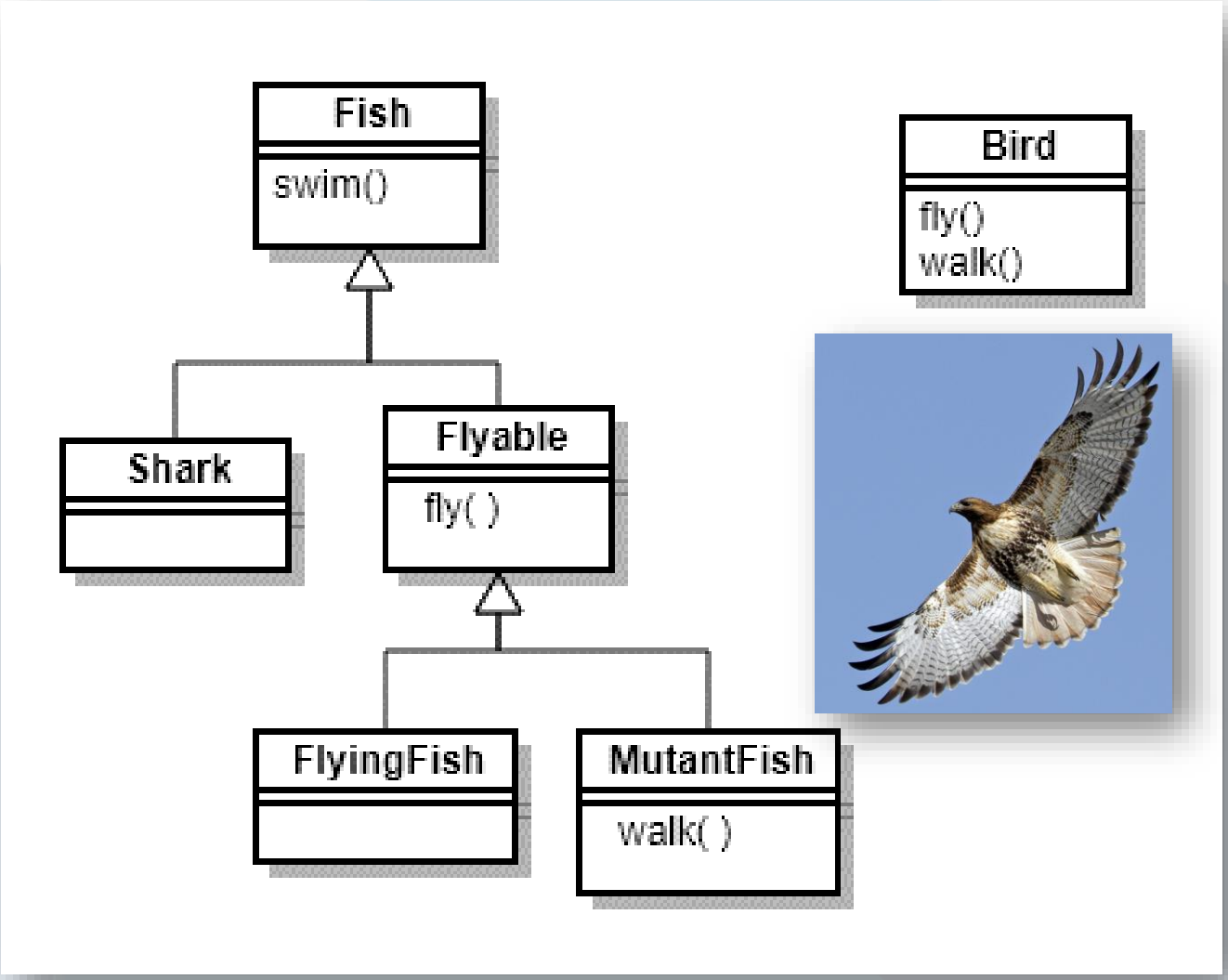
- Solve the problem of code duplication.
- However, multiple inheritance is not supported in Dart

Solution 3: Multi-level Single Inheritance




```
1   class Fish{
2     void swim(){print('It swims');}
3   }
4
5   class Shark extends Fish{
6   }
7
8   class Flyable extends Fish{
9     void fly(){print('It flies');}
10  }
11
12  class FlyingFish extends Flyable{
13  }
14
15  class MutantFish extends Flyable{
16    void walk(){print('It walks');}
17  }
```

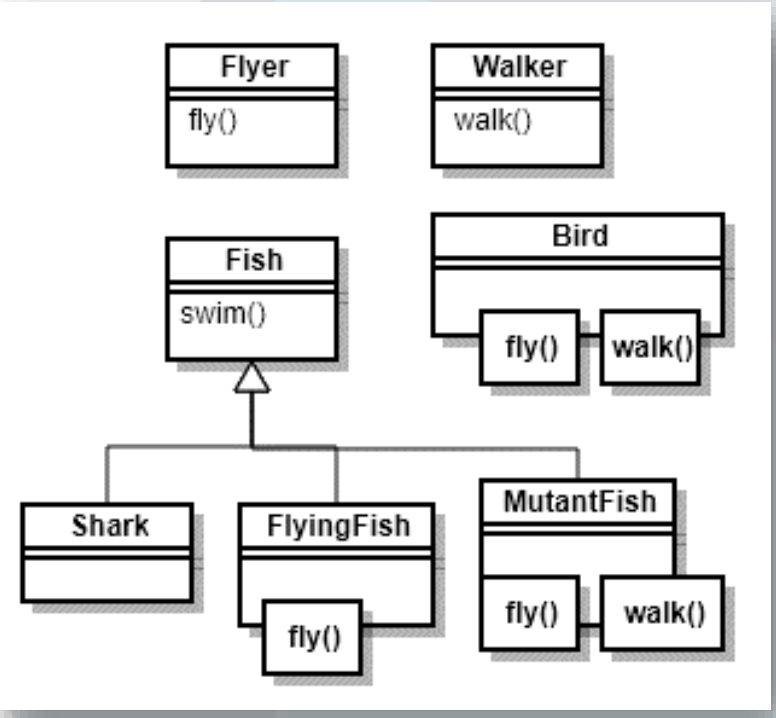
But what if there is a new class?



Code duplication on the methods *fly()* and *walk()*

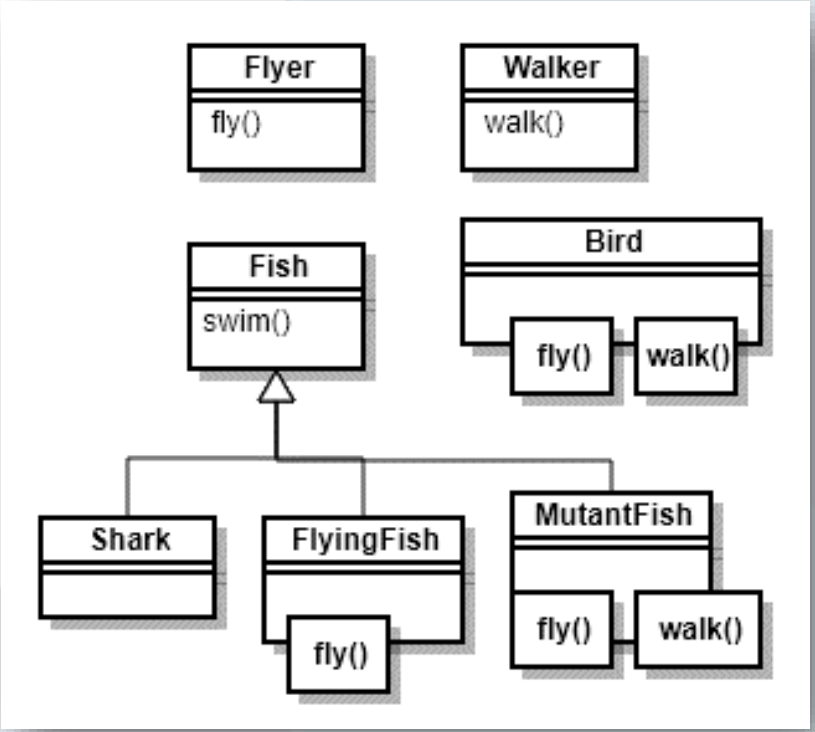
Solution 4: Mixin to the rescue (1)

- Define classes with *mixin*
- A mixin class has no declared constructor, no instance, no parent class and no child classes.




```
1   mixin Flyer{
2     void fly()=>print('It flies');
3   }
4
5   mixin Walker{
6     void walk()=>print('It walks');
7   }
8
```

Mixin to the rescue (2)

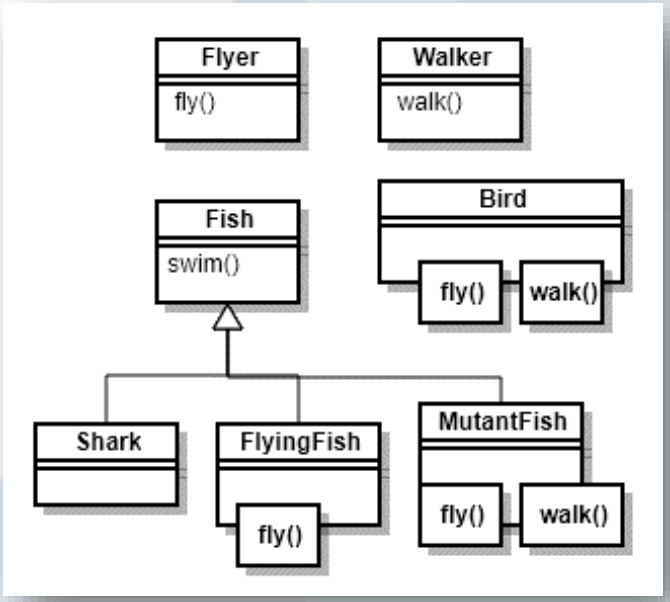



```dart
class Fish{
  void swim()=>print('It swims');
  void whatIs()=>print('It is a fish');
}

class Shark extends Fish{
  @override
  void whatIs()=>print('It is a shark');
}
```

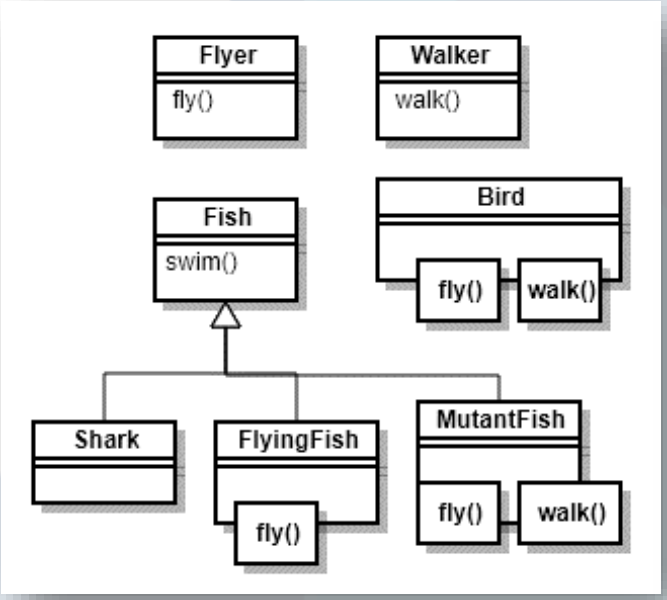
Mixin to the rescue (3)

- Define the consumer class with the keyword *with*
- A class can consume multiple mixins.




```
19  class FlyingFish extends Fish with Flyer{
20    @override
21    void whatIs()=>print('It is a flying fish');
22  }
23
24  class MutantFish extends Fish with Flyer, Walker{
25    @override
26    void fly()=>print('I believe I can fly');
27  }
28
```

Mixin to the rescue (4)




```
29  class Bird with Flyer, Walker{
30    String species;
31    Bird(this.species);
32
33    void showFact(){
34      print('It is a bird from $species species');
35      fly();
36      walk();
37    }
38  }
```

Mixin to the rescue (5)

```
40  void main(){
41
42    var shark = Shark();
43    shark.whatIs();
44    shark.swim();
45
46    print('');
47
48    var fish = FlyingFish();
49    fish.whatIs();
50    fish.swim();
51    fish.fly();
52    // fish.walk(); //error,
53    // FlyingFish has no walk() method
54    print('');
55
56    var mutant = MutantFish();
57    mutant.whatIs();
58    mutant.swim();
59    mutant.fly();
60    mutant.walk();
61
62    print('');
63    var bird = Bird('Bald Eagle');
64    bird.showFact();
65  }
```

Output:

```
It is a shark
It swims

It is a flying fish
It swims
It flies

It is a fish
It swims
I believe I can fly
It walks

It is a bird from Bald Eagle species
It flies
It walks
```

*Asynchronous Programming

- Future
- Async / Await

**This topic will be discussed later*