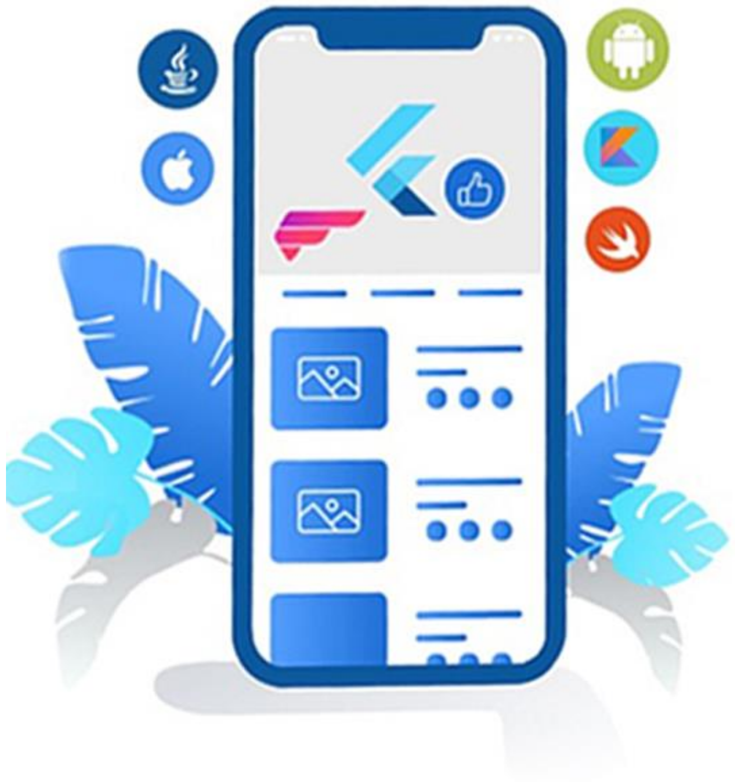
1301304 - VISUAL PROGRAMMING



- Academic Year/Semester: 2023/1
- Lecturer: Dr. Mahmoud Aljawarneh
- Email: ma_jawarneh@asu.edu.jo
- Office Number: 1217

CHAPTER 1: DART LANGUAGE OVERVIEW

INTRODUCTION TO DART LANGUAGE

OVERVIEW

- We use the Dart language when writing Flutter, but Dart isn't very popular (yet).
- Most developers jump right into Flutter with no prior knowledge of the language.
- In this chapter, we're making no attempt to teach you everything about Dart. Our goal here is to get you just enough Dart to be effective as you write Flutter. So, this chapter is brief and to the point.
- We are only dealing with the things that would otherwise have slowed you down while writing Flutter.

WHAT IS DART?

- Dart is a compiled, statically typed, object-oriented, procedural programming language.
- It has a very mainstream structure much like other OO languages, making it extremely easy to learn for developers who have experience with Java, C#, C++, or other OO, C-like languages.
- It adds some features that developers in other languages would not expect, but which are very cool and make the language more than elegant.

EXPECTED FEATURES – DART CHEAT SHEET

- This quick reference assumes that you’re an experienced OO developer and ignores the stuff that would be painfully obvious to you.
- For a more indepth and detailed look at Dart, please visit:
  - https://dart.dev/guides/language/language-tour

DATA TYPES

```
int x = 10;           // Integers
double y = 2.0;       // IEEE754 floating point numbers
bool z = true;        // Booleans
String s = "hello";// Strings
dynamic d;            // Dynamic variables can change types at any time.
d = x;                // Use moderately!
d = y;
d = z;
```

TYPE INFERENCE

- If I said “x=10.0”, what data type would you guess that x is?
  - Double
- How did you know?
  - Because you looked to the right of the equal sign and inferred its type based upon the value being assigned to it. Dart can do that too.
- If you use the keyword var instead of a data type, Dart will infer what type it is and assign that type.

TYPE INFERENCE (EXAMPLES)

```
var i = 10;     // i is now defined as an int.
i = 12;             // Works, because 12 is an int.
i = "twelve";       // No! "twelve" is a String and not an int.
var str = "ten";    // str is now defined as a String.
str = "a million"; // Yep, works great.
str = 1000000.0;    // Nope! 1000000.0 is a double, not a string.
```

- This is often confused with dynamic. ***Dynamic*** can hold any data type and can change at runtime. ***Var*** is strongly and statically typed.

TYPE INFERENCE (EXAMPLES)

```
void main() {
  dynamic x = 'hal';
  x = 123;
  print(x);
  var a = 'hal';
  a = 123;
  //Error: A value of type 'int' can't be assigned to a variable of type
  'String'.
  print(a);
}
```

dynamic is a type underlying all Dart objects. You shouldn't need to explicitly use it in most cases.

var is a keyword, meaning "I don't care to notate what the type is here." Dart will replace the var keyword with the initializer type or leave it dynamic by default if there is no initializer.

ARRAYS/LISTS

- Square brackets means a list/array. In Dart, arrays and lists are the same thing.

```
List<dynamic> list = [1, "two", 3];
// Optional angle brackets show the type - Dart supports Generics
```

- How to iterate a list?

```
for (var d in list) {
   print(d);
}
```

- Another way to iterate a list

```
list.forEach((d) => print(d));     // Both of these would print "1", then "two", then "3"
```

ARRAYS/LISTS

- Add items to an existing list you can use the add() method:

```
list.add(5);      // Append new item at the end of the list
```

- Use the length method to get the number of items in a list:

```
list.length;
```

- To remove an element from a list, use remove

```
list.remove(5);      // Removes the first occurrence of value from the list.
```

- To insert an element at position in the list, use insert

```
list.insert(list.length, 10);      // Insert the value at the end of the list.
```

ARRAYS/LISTS

- To sort the elements of the list, use sort() method:

```
list.sort((a, b) => a.compareTo(b));       // Append new item at the end of the list
```

- Removes all objects from this list that satisfy test:

```
List<String> numbers = ['one', 'two', 'three', 'four'];
numbers.removeWhere((item) => item.length == 3);
print(numbers);
```

CONDITIONAL EXPRESSIONS

- Traditional if/else statement

```
int x = 10;
if (x < 100) {
      print('Yes');
} else {
      print('No');
}
// Would print "Yes"
```

LOOPING

- A for loop

```
for (int i=1 ; i<10 ; i++) {
      print(i);
}// Would print 1 thru 9
```

- A while loop

```
int i=1;
while(i<10) {
   print(i++);
}// Would print 1 thru 9
```

FINAL AND CONST

- `final` and `const` are Dart variable modifiers:

```
final int x = 10;

const double y = 2.0;
```

- They both mean that once assigned, the value can’t change.
- But **const** goes a little farther – the value is set at compile time and is therefore embedded in the installation bundle.

FINAL AND CONST

- `final` means that the variable can't be reassigned. It does not mean that it can't change. For example, this is allowed:

```
final Employee e = Employee();
e.employer = "The Bluth Company";
```

- In the above example `e` object changed, but it wasn't reassigned so that's okay. This, however, is not allowed using `const`:

```
const Employee e = Employee();
```

const is not allowed at all because this particular class has properties that could potentially change at runtime.

FINAL AND CONST

- **final** marks a variable as unchangeable, but **const** marks a value as unchangeable.
- So, in summary:
  - **dynamic** – Can store any data type. The data type can change at any time.
  - **var** – The data type is inferred from the value on the right side of the "=". The data type does not change.
  - **final** – The variable, once set, cannot be reassigned.
  - **const** – The value is set at compile time, not runtime.

SOUND NULL SAFETY

- Types in your code are non-nullable by default, meaning that variables can’t contain null unless you say they can.
- With null safety, your runtime null-rereference errors turn into edit-time analysis errors.
  - *Sound null safety is available in Dart 2.12 and Flutter 2.*
- To indicate that a variable might have the value null, just add **?** to its type declaration:

```
int? x;     double? y;     bool? z;     String? s;     dynamic d;
```

STRING INTERPOLATION WITH $

- Interpolation saves devs from writing string concatenations. This …

```
String fullName = '$first $last, $suffix';
```

- … is effectively the same thing as this …

```
String fullName = first + " " + last + ", " + suffix;
```

- When the variable is part of a map or an object, the compiler can get confused, so you should wrap the interpolation in curly braces.

```
String fullName = '${name['first']} ${name['last']}';
```

MULTILINE STRINGS

- You can create multiline strings with three single or double quotes:

```
String introduction = """
Now the story of a wealthy family
who lost everything
And the one son who had no choice
but to keep them all together.
""";
```

SPREAD OPERATOR

- The “…” operator will spread out the elements of an array, flattening them.
- This will be very familiar to JavaScript developers:

```
List fiveTo10 = [ 5, 6, 7, 8, 9, 10, ];
// Spreading the inner array with "...":
List numbers = [ 1, 2, 3, 4, ...fiveTo10, 11, 12];
// numbers now has [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

MAP<KEY, VALUE>

- Maps are like a hash or dictionary. They're merely an object with a set of key-value pairs. The keys and values can be of any type:

```
// You set the value of a Map with curly braces:
Map<String, dynamic> person = {
"first": "Mahmoud",
"last": "Aljawarneh",
"employmentDate": DateTime.parse("2014-09-21"),
"email": "ma_jawarneh@asu.edu.jo",
};
```

Angle brackets on a Map set the data types of the keys and values. They're not required but are a good practice.

MAP<KEY, VALUE>

- You can reference a map member with square brackets as follows:

```
String introduction = "${person['first']} was hired on
${person['employmentDate']}.";
```

- You can add a new key, value pair as follows:

```
person["salary"] = 1000;
```

- You can add a new key, value pair as follows:

```
for(var key in person.keys)
    print('$key = ${person[key]}');
```

FUNCTIONS ARE OBJECTS

- Functions can be passed around like data, returned from a function, passed into a function as a parameter, or set equal to a variable.
- You can do just about anything with a function that you can do with an object in Java or C#:

```
void sayHi (String name) {
  print('Hello, ' + name);
}
// You can pass sayHi around like data; it's an object!
Function meToo = sayHi;
meToo("Mahmoud");
```

NAMED FUNCTION PARAMETERS

- Positional parameters are great, but it can be less error-prone (albeit more typing) to have named parameters. Instead of calling a function like this:

```
sendEmail(‘ma_jawarneh@asu.edu.jo','Lets Learn Dart');
```

- You can call it like this:

```
sendEmail(subject:'Lets Learn Dart', toAddress:’ma_jawarneh@asu.edu.jo');
```

- Now the order of parameters is unimportant.

NAMED FUNCTION PARAMETERS

- Here is how you'd write the function to use named parameters. *Note the curly braces*:

```
void sendEmail({String toAddress, String subject}) {
    // send the email here
}
```

LAMBDA FUNCTIONS (FAT ARROW)

- Lambda functions are also called **Fat Arrow functions**.
- Lambda is a short and concise manner to represent small functions.
- In short, if your function has only one expression to return then to quickly represent it in just one line you can use lambda function.
  - **Syntax:** `return_type function_name(arguments) => expression;`
- Example:
  - `int ShowSum(int numOne, int numTwo) => numOne + numTwo;`

LAMBDA FUNCTIONS (FAT ARROW)

- These functions are all the same:

```
int triple(int val) {
    return val * 3;
}
Function triple = (int val) {
    return val * 3;
};
Function triple = (int val) => val * 3;
```

CREATE CLASSES

```
class Name {
  String first;
  String last;
  String suffix;
}

class Person {
  // Classes have properties
  int id;
  // Another class can be used as a
  type
  Name name;
  String email;
  int age;
  // Classes have methods
  void save() {
    // Write to a database somehow.
  }
}
```

PUBLIC AND PRIVATE CLASS MEMBERS

Dart does not use class visibility modifiers such as public, private, protected, package, or friend like other OO languages.

All members are public by default.

PUBLIC AND PRIVATE CLASS MEMBERS

- To make a class member private, put an **underscore** in front of the name:

```
class Person {
  int id;
  String name;
  String _password;
  set password(String value) {
    _password = value;
  }
  String get hashedPassword {
    return sha512.convert(utf8.encode(_password)).toString();
  }
}
```

In this example, **id** and **name** are **public**. However, the _password is **private** because the first character in the **name** is "_", the underscore character.

GETTERS AND SETTERS

- Getter Method:
  - It is used to retrieve a particular class field and save it in a variable.
  - All classes have a default getter method, but it can be overridden explicitly.
  - The getter method can be defined using the get keyword as:

```
return_type get field_name{
    ...
}
```

```
int get stud_age {
        return age;
    }
```

GETTERS AND SETTERS

- Setter Method:
  - It is used to set the data inside a variable received from the getter method.
  - All classes have a default setter method, but it can be overridden explicitly.
  - The setter method can be defined using the set keyword as:

```
set field_name{
    ...
}
```

```
void set stud_age(int age) {
      if(age<= 0) {
        print("Age should be greater than 5");
      }  else {
          this.age = age;
      }
   }
```

CLASS CONSTRUCTORS

```
class Person {
  Name name;
  // Typical constructor
  Person() {
    name = Name();
    name.first = "";
    name.last = "";
  }
}
```

NO OVERLOADING IN DART – NAMED CONSTRUCTOR

Dart does not support overloading methods. This includes constructors.

Since we can’t have overloaded constructors, Dart supports a different way of doing essentially the same thing. They’re called named constructors.

NAMED CONSTRUCTORS

- Named constructors are happen when you write a typical constructor, but you tack on a dot and another name:

```
class Person {
   // Typical constructor
   Person() {
      name = Name()..first=""..last="";
   }
   // A named constructor
   Person.withName({String firstName, String lastName})
   {
      name = Name()
         ..first = firstName
         ..last = lastName;
   }
   // Another named constructor
   Person.byId(int id) {
   // Maybe go fetch from a service by the provided id
   }
}
```

NAMED CONSTRUCTORS

- And to use these named constructors, do this:

```
Person p = Person();
// p would be a person with a blank first and last name
Person p1 = Person.withName(firstName:"Mahmoud",lastName:"Aljawarneh");
// p1 has a first name of "Mahmoud" and a last name of "Aljawarneh"
Person p3 = Person.byId(100);
// p3 would be fetched based on the id of 100
```

NAMED CONSTRUCTORS

- Named parameters also work great with class constructors where they are very commonly used in Flutter:

```
class Person {
  Name name;
  // Named parameters
  Person({String firstName = '', String lastName = ''}) {
    name = Name();
    name.first = firstName;
    name.last = lastName;
  }
}
```

CLASS CONSTRUCTOR PARAMETER SHORTHAND

- Merely a shorter way of writing your Dart classes which receive parameters.
  - When you write the constructor to receive "this.something" and have a class-scoped property with the same name, the compiler writes the assignments, so you don't have to:

```
class Person {
   String email;
   String phone;
   Person(this.email, this.phone)
   {}
}
```

The parameters are assigned to properties automatically because the parameters say "this."

CLASS CONSTRUCTOR PARAMETER SHORTHAND

- The preceding code is equivalent to:

```
class Person {
   String email;
   String phone;
   Person(email, phone) {
       this.email = email;
       this.phone = phone;
   }
}
```

CREATE AN INSTANCE - OMITTING “NEW”

- In Dart, it is possible – and encouraged – to avoid the use of the new keyword when instantiating a class:

```
// No. Avoid this style.
Person p = new Person();
// Yes
Person p = Person();
```

OMITTING “THIS.”

- Inside of a class, the use of “this.” to refer to members of the class is not only unneeded because it is assumed, but it is also discouraged.
- The code is shorter and cleaner:

```
class Name {
  String first;
  String last;
  String getFullName() {
    String full = '${this.first} ${this.last}'; // Avoid "this."
    String full = '$first $last'; // This is Better
   return full;
  }
}
```

THE CASCADE OPERATOR (..)

- When you see two dots, it means “return this class, but before you do, do something with a property or method.” We might do this:

```
Person p = Person()..id=100..email='gob@bluth.com'..save();
```

- which would be a more concise way of writing

```
Person p = Person();
p.id=100;
p.email=‘ma_jawarneh@asu.edu.jo';
p.save();
```

INHERITANCE

- Dart, like Java and C#, only supports single inheritance.
- A class can only extend one thing.

```
class Animal {
  void makeSound() {
    print('I have no sound!');
  }
}

class Cat extends Animal {
  @override
  void makeSound() { print('meow meow!'); }
}

void main(List<String> arguments) {
  Animal animal = Animal();
  animal.makeSound();
  Cat cat = Cat();
  cat.makeSound();
}
```

COMPOSITION

- Composition is a more modular approach in which a class contains instances of other classes that bring their own abilities with them.
- Composition is used extensively in Flutter's UI framework.

```
class Room {
  int? capacity;
  String? code;
  Room({this.capacity,this.code});
}
class Building {
  int? id;
  String? title;
  Room? room;
  Building({this.id,this.title,roomCapacity,roomCode}){
    room = Room(capacity:roomCapacity,code:roomCode);
  }
}
void main() {
  Building building = Building(
    id:1,
    title:"IT",
    roomCapacity:20,
    roomCode:"1G00");
}
```

END OF CHAPTER