



Topic 3 – Dart Programming Language



EGL 208 Mobile Application Development

MAD

Explore DART

Explore DART Developer Site: <https://dart.dev/>

- Developer site provides the most resources into the language
- Language tour provides the introduction to language specification and a lead to more in-depth study
- Language samples provide a wealth of samples and tutorials for references
- Effective Dart provides the best practices for programming
- Blog and community provide peer supports for learning



References

Detailed Dart Language Specification

<https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf>

API Reference

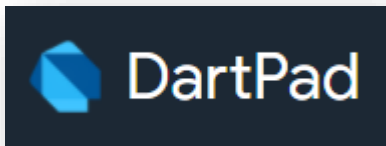
<https://api.dart.dev/stable/2.14.3/index.html>

DART IDE

- Visual Studio Code IDE provides the following :
 - Explore application structure
 - Editor (documentation)
 - Linking of dependencies
 - Compilation
 - Debug (system and console output)
 - Test
 - Deploy (* iOS / Android devices)



Visual Studio Code




<https://dartpad.dev/>

an on-line tool that lets you play with the Dart language in any browser.

Application Structure

➤ Explore our first program

```
void main() {  
    print('Hello world');  
}
```

- **main()** is the starting **Function** of the application that system automatically call. Function must be followed by curly braces { }
- { } is the **Scope** to enclose other program statements
- **void** is a **keyword** that specify no return value for the function
- **print()** function send a text message to the debug console
- Text message (a.k.a **String**) is enclosed in " " double quotes or ' ' single quotes  *preferred*
- ; at the end of each statement indicates end of statement

Explore Program

➤ Explore our second program

```
var temp = 'NYP';  
print('Hello world');  
print('Hello ' + temp); // the two strings are concatenated together
```

- In the first statement, **=** symbol is to store data value indicated on the right into a variable (memory space) indicated by the name *'temp'*
- **var** is a keyword used to indicate Inferred Data Type based on the data value which in this case temp is inferred to be a String.
- If we want to join two strings together, we can use the **+** symbol which is used as a symbol to concatenate two strings.

Data Types

- **VARIABLES** are memory spaces that store data.
- Variables can be classified into different **DATA TYPES** based on requirement for size, precision and type of data to be stored.
- Data types can be **SIMPLE** data types like

int: Integer e.g., 1, 2, 3, -1	double: Floating point e.g., 1.1, 3.0, -2.1	String: Alphanumeric e.g., 'NYP', 'C123'	bool: Boolean e.g., true, false
---	--	---	--

- Or **COMPLEX** data types like **list**, **maps**, **sets** or even **class** (e.g., **Person**)

Note: All data types in DART are **Objects** → values are by default **null**

Declaration

- Variable must be DECLARED to the system for it to be usable.
- Syntax of declaration is:

```
<Data Type> <name of variable> [= <values>];
```

- The data type can be INFERRED by the VALUES set on it:
e.g., **var temp = 'NYP'** → temp is inferred as String type
- Or EXPLICITLY defined, for example,

```
int a;
```

```
String temp = 'NYP';
```


Declaration

- Variables can be declared without setting initial values and setting values at a later part of the application.

for example,

```
int a;  
:  
:  
a = 2;
```

- If you don't explicitly state the type:

```
var a;    // a is a dynamic type    or    dynamic a = 'xyz';  
//The following will not have any errors  
a = 'hello';  
a = 8;  
a = true;
```

ARITHMETIC Operators

➤ Programming Language's **Arithmetic Operators** are very similar to our familiar mathematics symbols:

+ Add	- Subtract	* Multiply	/ Divide	() Parenthesis
~/ Divide and return integer value	% Get remainder of int division	++ Increment by 1	-- Decrement by 1	

for example,

```
c = a + b;      c++;      // increase value of c by 1
```

```
d = (a + b) * 2;    c = a % 2;    // c will be remainder
                                // after a is divided by 2
```

LAB Exercise - 2

- **PART 1**
INTRODUCTION TO DART DEVELOPER SITE
- **PART 2**
MY FIRST DART PROGRAM
- **PART 3**
DECLARING VARIABLES AND ASSIGNMENT OF VALUES
- **PART 4**
PERFORMING SIMPLE ARITHMETICS

RELATIONAL Operators

- **Relational Operators** tests or defines the kind of relationship between two entities.
- It return a **Boolean** value i.e., **true/ false**.
- Relational operators are:

> greater than

< less than

>= greater than or equal to

<= less than or equal to

== equal to

!= not equal to

RELATIONAL Operators

➤ Assume the value of A is 10 and B is 20.

Operator	Description	Example
$>$	Greater than	$(A > B)$ is False
$<$	Lesser than	$(A < B)$ is True
$>=$	Greater than or equal to	$(A >= B)$ is False
$<=$	Lesser than or equal to	$(A <= B)$ is True
$==$	Equality	$(A == B)$ is False
$!=$	Not equal	$(A != B)$ is True

TYPE TEST Operators

- These operators are handy for checking **DATA TYPES** at runtime.

Operator	Meaning
is	True if the object has the specified data type
is!	False if the object has the specified data type

- Assume variable ***a*** is declare as integer (**int**),
***a* is int** will be True.

LOGICAL Operators

- **Logical operators** are used to combine two or more comparison expressions.
- Logical operators return a Boolean (**True** or **False**) value.
- Assume the value of variable **A** is 10 and **B** is 20,

Operator	Description	Example
&&	And – The operator returns true only if all the expressions specified return true	(A > 10 && B > 10) is False.
	OR – The operator returns true if at least one of the expressions specified return true	(A > 10 B > 10) is True.
!	NOT – The operator returns the inverse of the expression's result. For E.g.: !(7>5) returns false	!(A > 10) is True.

Key Learning Points to recap

- Relational Operator used on String is acceptable
- Multiple Logical Operator is acceptable
- Logical Operator can be further encapsulated in Parenthesis to determine priority of evaluation of expression

Decision Program Flow

- A conditional/decision-making construct evaluates a condition before the instructions are executed.
- **If-statements** control flow **evaluate expression that yield Boolean** (True or False) values.
- The type of if-statements are

if statement

An if statement consists of a Boolean expression followed by one or more statements.

if...else Statement

An if can be followed by an optional else block. The else block will execute if the Boolean expression tested by the if block evaluates to false.

else...if Ladder

The else...if ladder is useful to test multiple conditions. Following is the syntax of the same.

if ... else Syntax

➤ Syntax of **if** statement

```
if (boolean_expression) {  
    // statement(s) will execute if the  
    // boolean expression is TRUE.  
}
```

➤ Syntax of **if-else** statement

```
if (boolean_expression) {  
    // statement(s) will execute if the  
    // boolean expression is TRUE.  
} else {  
    // statement(s) will execute if the  
    // boolean expression is FALSE.  
}
```

if ... ladder Syntax

➤ Syntax of else...if ladder statement

```
if (boolean_expression1) {  
    //statements if the expression1 evaluates to true  
}  
else if (boolean_expression2) {  
    //statements if the expression2 evaluates to true  
}  
else {  
    //statements if both expression1 and expression2 results are false  
}
```

switch Statement

- The **switch** statement evaluates an expression, matches the expression's value to a case clause and executes the statements associated with that case:

```
switch (variable_expression) {  
    case constant_expr1: {  
        // statements; }  
    break;  
  
    case constant_expr2: {  
        // statements; }  
    break;  
  
    default: {  
        // statements; }  
}
```

switch Statement

- The value of the ***variable_expression*** is tested against all "cases" in the switch.
- If the variable matches one of the **CONSTANT VALUE** in any of the cases, the corresponding code block is executed.
- If no case expression matches the value of the ***variable_expression***, the code within the "**default**" block is associated.

Conditional Expression

- Dart has two operators that let you evaluate expressions that might otherwise require *if else* statements.
- If condition is true, then the expression evaluates **expr1** (and returns its value); otherwise, it evaluates and returns the value of **expr2**.

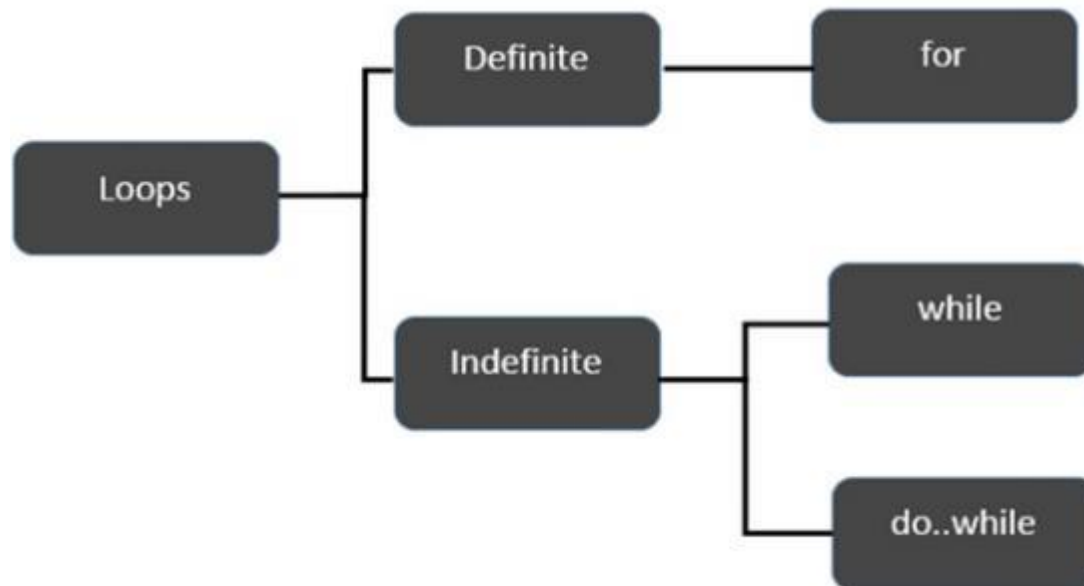
condition ? expr1 : expr2

- **Null** is a key word that means nothing. If **expr1** is non-null, returns its value; otherwise, evaluates and returns the value of **expr2**

expr1 ?? expr2

Loop Program Flow

- At times, certain instructions require repeated execution. Loops are an ideal way to do the same.
- A loop represents a set of instructions that must be repeated.
- In a loop's context, a repetition is termed as an **iteration**.



Definite Loop

- A loop whose number of iterations are definite/fixed is termed as a **definite loop**.
- The **for** loop is an implementation of a definite loop. The **for** loop executes the code block for a specified number of times.
- It can be used to iterate over a fixed set of values, such as an array.
- syntax of the **for** loop.

```
for ( initial_count_value; continuation-condition; step) {  
    // statements  
}
```


Indefinite Loop

- An **indefinite loop** is used when the number of iterations in a loop is indeterminate or unknown.
- The **while** loop executes the instructions each time the condition specified evaluates to true.
- The loop evaluates the condition before the block of code is executed.
- It can be used to iterate over a fixed set of values, such as an array.
- syntax of the **WHILE** loop.

```
while (expression) {  
    // Statement(s) to be executed  
    // if expression is true  
}
```

Indefinite Loop

INDEFINITE LOOP

- The **do...while** loop is similar to the while loop except that it doesn't evaluate the condition for the first time the loop executes.
- However, the condition is evaluated for the subsequent iterations.
- Syntax of the **DO-WHILE** loop.

```
do {  
    // Statement(s) to be executed;  
} while (expression);
```

Lab Exercise 3

➤ PART 1

USE OF OPERATORS

➤ PART 2

USAGE OF CONDITIONAL EXPRESS, IF ... ELSE, & SWITCH ... CASE STATEMENTS

➤ PART 3

USAGE OF FOR, DO...WHILE & WHILE STATEMENTS

➤ PART 4

LOOP CONTROL STATEMENTS

List

- Commonly known in programming as **array**.
- Array are memory spaces used to store data.
- **List** is simply an ordered group of objects
- The **dart:core** library provides the List class that enables creation and manipulation of lists
- Lists can be classified as –
 - **Fixed Length** List
 - **Growable** List

List

➤ Example: *test_list*

<i>value</i>	26	68	39
<i>index</i>	0	1	2

➤ A logical representation of a list in Dart

➤ *test_list* – is the identifier that references the collection.

➤ The list contains in it the values 26, 68, and 39. The memory blocks holding these values are known as **elements**.

➤ Each element in the List is identified by a unique number called the **index**. The index starts from **zero** and extends up to **n-1** where **n** is the total number of elements in the List. The index is also referred to as the **subscript**.

Fixed Length List

- List's length cannot change at runtime.
- Syntax for creating a fixed length list

Step 1 – Declaring a fixed length list

```
var list_name = new List(initial_size)
```

The above syntax creates a list of the specified size. The list cannot grow or shrink at runtime. Any attempt to resize the list will result in an exception.

Step 2 – Initializing a list

```
list_name[index] = value;
```

Growable Length List

- List's length can change at runtime.
- Syntax for creating a growable length list

Step 1 – Declaring a List

creates a list containing the specified values

```
var list_name = [val1, val2, val3]
```

or creates a list of no elements (size zero)

```
var list_name = new List()
```

Growable Length List

Step 2 – Add element to a List

```
list_name.add(value);
```

- New elements can be added at the end of the list.

Step 3 – Referencing a List element

```
list_name[index];
```

- The index/subscript is used to reference the element that should be populated with a value.

List Properties

➤ Commonly used properties of the List class in the **dart:core** library

Methods & Description	
first	Returns the first element case.
isEmpty	Returns true if the collection has no elements.
isNotEmpty	Returns true if the collection has at least one element.
length	Returns the size of the list.
last	Returns the last element in the list.
reversed	Returns an iterable object containing the lists values in the reverse order.
single	Checks if the list has only one element and returns it.

Function

- Building blocks of readable, maintainable, and REUSABLE code.
- A set of statements to perform a specific task.
- Once DEFINED, functions may be CALLED to access code.
- Function DEFINITION tells the compiler about a function's NAME, RETURN TYPE, and PARAMETERS.

Function

	DECLARING Function	CALLING Function
No return value No parameter	<pre>void <function name>() { // logic statements }</pre> <p>e.g., void <i>printTitle</i>() { print('Mobile Application Development'); }</p>	<p>e.g., <i>printTitle</i>();</p>
Return value No parameter	<pre>Return-Type <function name>() { // logic statements return value; }</pre> <p>e.g., String <i>sayHello</i>() { var msg = 'Hello, World!'; return msg; }</p>	<p>e.g., var <i>m</i> = <i>sayHello</i>(); or print(<i>sayHello</i>());</p>

Function

	DECLARING Function	CALLING Function
Fixed (Required) parameters	<pre><i>Return-Type</i> <function name>(<param1, param2 ...>) { // logic statements return <i>value</i>; }</pre> <p>e.g., double calAreaOfCircle(double <i>radius</i>) { var area = 3.142 * radius * radius; return <i>area</i>; }</p>	<p>e.g.,</p> <pre>double a = calAreaOfCircle(4.0);</pre>
Optional Positional parameters	<pre><i>Return-Type</i> <function name>(<param1>, [<param2>]) { // logic statements return <i>value</i>; }</pre> <ul style="list-style-type: none">- <i>params inside []</i> are optional- must declare after any required parameters <p>e.g., void setRec (String <i>name</i>, [String <i>comment</i>]) { print (comment == null ? '\$name' : '\$name: \$comment'); }</p>	<p>e.g.,</p> <pre>setRec('Joe');</pre> <p>or</p> <pre>setRec('Bob', 'no comment');</pre>

Function

	DECLARING Function	CALLING Function
Optional Named parameters	<pre>Return-Type <function name>(<param1>, {<param2>}) { // logic statements return value; }</pre> <ul style="list-style-type: none">- <i>params inside { }</i> are optional- must declare after any required parameters- have to use parameter name to pass value (separated by :)- avoid confusion or error while passing values to a function with many parameters <p>e.g., int calVolume(int length, int width, {int height}) { var volume = length * width * height; return volume; }</p>	<p>e.g.,</p> <pre>var v = calVolume(5, 8, height: 6);</pre> <p>or</p>
Default parameter	<p>e.g., int calVolume(int length, int width, {int height=10}) { var volume = length * width * height; //default height of 10 return volume; }</p>	<pre>var v = calVolume(5, 8); //default height of 10</pre>

Lambda Function

Although most of the functions should be named, there is an option of creating nameless function called **Lambda** (*anonymous* or *Fat Arrow*) function.

- **Lambda** is a short and concise manner to represent small functions.
- Lambda function's syntax can only return one expression.
- No need a **return** statement explicitly.
- **=>** notation is used as a shorthand to return a single expression (Fat Arrow).

```
void main() {  
    print(addNumbers(3, 6));  
}  
  
int addNumbers( int x, int y ) {  
    var sum = x + y;  
    return sum;  
}
```

```
void main() {  
    print(addNumbers(3, 6));  
    //or  
    addNumbers(3, 6);  
}  
  
//function expression using shorthand syntax or Fat arrow '=>'  
int addNumbers( int x, int y ) => x + y;  
  
//or  
void addNumbers( int x, int y ) => print(x + y);
```

Library & Packages

- Library in a programming language represents a COLLECTION OF ROUTINES (set of programming instructions)
- Dart library comprises of a set of classes, constants, functions, typedefs, properties, and exceptions.
- Dart library categories similar routines into PACKAGES
- To make use of library, PACKAGES need to be IMPORTED using either file system path or <PACKAGE>: <SCHEME> convention to specify its URI.
- Syntax :

```
import '<URI>'
```

e.g., `import 'dart:io';` `// Dart standard library's io package`
 `import 'package:lib1/libfile.dart'` `// file system path`

Commonly Used Packages

Library & Description

dart:io

File, socket, HTTP, and other I/O support for server applications. This library does not work in browser-based applications. This library is imported by default.

dart:core

Built-in types, collections, and other core functionality for every Dart program. This library is automatically imported.

dart: math

Mathematical constants and functions, plus a random number generator.

dart: convert

Encoders and decoders for converting between different data representations, including JSON and UTF-8.

dart: typed_data

Lists that efficiently handle fixed sized data (for example, unsigned 8 byte integers).

Lab Exercise 4

➤ PART 1

USING LIST PROPERTIES

➤ PART 2

DEFINING & CALLING FUNCTION

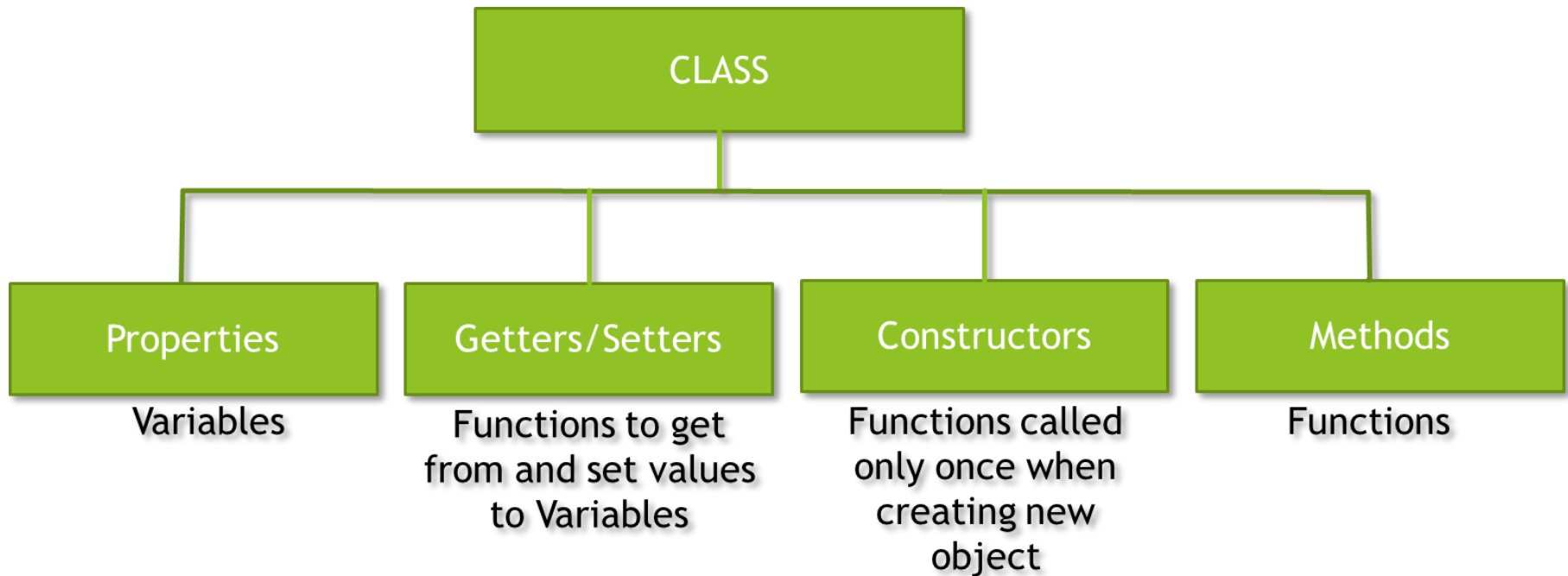
➤ PART 3

USING LIBRARY

Object-Oriented Programming

- OBJECT-ORIENTED PROGRAMMING is to model **OBJECTS** in real world.
- A **Person** object, for example, possesses data like *first name, last name, nric, address, handphone number*, etc and can perform action like *run, jump, eat, sleep*.
- An **Aircraft** object possesses data like *number of seats, number of engine, number of wheels, model of engine* and can perform actions like *run, take off, land, park*.
- OOP model real life object as **CLASS** which is a blueprint of objects to be created in future, like following blueprint of "Boeing 747" to create "SQ741", "SQ742", "MI737" etc.

Object-Oriented Programming



Object-Oriented Programming

CLASS DECLARATION

➤ Syntax:

```
class <Class_Name> {  
    <properties>  
    <getters/setters>  
    <constructors>  
    <methods>  
}
```

```
class Person {  
    // property  
    String name;  
  
    // getters & setters  
    String get pName() {  
        return name;  
    }  
    set pName(String name) {  
        this.name = name;  
    }  
  
    // constructor  
    Person (String name) {  
        this.name = name; //note this keyword  
    }  
  
    // method  
    void dispName() {  
        print('$name is a person.');    }  
}
```

Object-Oriented Programming

INSTANTIATING CLASS

➤ Syntax

```
var <object name> = <Class Name>([arguments])
```

e.g., **var** *john* = **Person**('John');

john

➤ Use dot notation (.) to access properties and methods of a class

e.g., *john.name* → refers to ***name*** property

e.g., *john.displayName*() → call ***displayName*** method which print the name on the console

name – 'John'

Object-Oriented Programming

CLASS DECLARATION with INHERITANCE

➤ Syntax:

```
class <Child_class_name> extends <Parent_class_name>
{
    @override
    // same method declaration
}
```

```
class Employee extends Person
{
    @override
    void dispName() {
        super.name();    // note super keyword
        print('$name is also an employee');
    }
}
```

Exception Handling

- Problem arises during the execution of a program
- e.g., **FormatException**
 - Exception thrown when a string or some other data does not have an expected format and cannot be parsed or processed.
- e.g., **IntegerDivisionByZeroException**
 - Thrown when a number is divided by zero.
- Every exception is a child of **Exception** class.
- Exceptions must be handled to prevent the application from terminating abruptly

Exception Handling

TRY/ON/CATCH BLOCKS

➤ Syntax

```
try {  
    // code that might throw an exception  
}  
on <Exception1> [catch (e)] {  
    // code for handling specific exception  
}  
on <Exception2> [catch (e)] {  
    // code for handling specific exception  
}  
catch (e) {  
    // code for handling generic exception  
}  
finally {  
    // code needed to be executed  
}
```


Lab Exercise 5

➤ PART 1

CLASS & INHERITANCE

➤ PART 2

EXCEPTION HANDLING