Lecture 2 - Dart Basics

📊 Data Types

Numbers, Strings, and Booleans:

- `int` : for integers
- `double` : for fractional values
- `String` : for text
- `bool` : for true/false values

```
void main() {
  String name = 'Ibrahim';
  int age = 30;
  double weight = 75.5;
  String msg = '''My name is $name, and
  I'm $age years old.
  My weight is $weight''';
  print(msg);
}
```

Methods for Numbers, Strings, and Booleans:

- `isEven` , `isOdd` , `isNegative`
- `abs()` , `ceil()` , `floor()` , `round()`
- `gcd(other)` , `compareTo(other)` , `remainder(other)`
- `toDouble()` , `toInt()` , `toString`
- `isEmpty` , `length` , `toUpperCase()` , `toLowerCase()`
- `int.parse(String)` , `double.parse(String)`
- `int.tryParse(String)` , `double.tryParse(String)`

```
int x = 20;
double y = 6.5;
String course = 'FLUTTER';

// Check if x is even
print(x.isEven); // true

// Get the runtime type of x
print(x.runtimeType); // int

// Find the greatest common divisor of x and 12
```

```
print(x.gcd(12)); // 4

// Compare x to 12
print(x.compareTo(12)); // 1 (returns 1 if x > 12, 0 if x == 12, -1 if x < 12)

// Convert x to a double
print(x.toDouble()); // 20.0

// Parse a string '3' to an integer
print(int.parse('3')); // 3

// Try to parse a string 'A' to an integer (returns null if parsing fails)
print(int.tryParse('A')); // null

// Round up y to the nearest integer
print(y.ceil()); // 7

// Round down y to the nearest integer
print(y.floor()); // 6

// Check if the string course is empty
print(course.isEmpty); // false

// Convert the string course to lowercase
print(course.toLowerCase()); // flutter
```

💡 *dynamic & var*

- `dynamic` :
  - Used for variables whose type depends on the value.
  - Allows reassigning with new types.

```
void main() {
  dynamic x = "tom";
  print(x);
  x = 5;
  print(x);
}
```

- `var` :
  - Infers type from value.
  - Type cannot be reassigned.

```
void main() {
  var n = 42;
  // n = 'abc'; // Error: A value of type 'String' can't be assigned to a
variable of type 'int'.
}
```

📚 Collections

- `List` : Ordered group of objects.
- `Set` : Collection of unique objects.
- `Map` : Key/value pairs.

```
void main() {
  // Creating a fixed-length list with 5 elements, all initialized to 2
  var fixedList = List.filled(5, 2);
  print(fixedList);

  // Creating a list of strings
  List<String> studNames = ['Ali', 'Ahmed', 'Mostafa'];
  print(studNames);

  // Creating a set of strings (ignores duplicates)
  Set<String> studNamesSet = {};
  studNamesSet.add('Ali');
  studNamesSet.add('Ahmed');
  studNamesSet.add('Ali'); // Ignored, as it's a duplicate
  print(studNamesSet);

  // Creating a map with key/value pairs
  var details = {'Username': 'admin', 'Password': 'admin@123'};
  details['Uid'] = 'U1001';
  print(details);
}
```

🔒 Final and Const

- Used to declare constants.
- `const` represents a compile-time constant.
- `final` can be determined at compile and run time.

```
void main() {
  // Declaring a final variable username
  final String username = 'admin';

  // Declaring a const variable password
  const String password = 'admin@123';

  // Declaring a final dynamic variable anything
  final dynamic anything = getFromDatabase();
}

// A function to simulate fetching data from a database
String getFromDatabase() {
  return 'Dynamic Data';
}
```

💬 Comments

- Dart supports `single-line` and `multi-line` comments.

```
void main() {
  // This is a single line comment

  /*
  This is a
  multi-line comment
  */
}
```

Let's break down and provide examples for each group of operators:

Operators

💡 Arithmetic Operators

- **Addition (`+`):** Adds two operands together.
- **Subtraction (`-`):** Subtracts the right operand from the left operand.
- **Unary Minus (`-expr`):** Reverses the sign of the expression.

```
void main() {
  int a = 5;
  int b = 3;

  print(a + b); // Output: 8
  print(a - b); // Output: 2
  print(-a);    // Output: -5
}
```

Multiplicative Operators

- **Multiplication (`*`):** Multiplies two operands.
- **Division (`/`):** Divides the left operand by the right operand.
- **Integer Division (`~/`):** Divides two operands and returns an integer result.
- **Modulus (`%`):** Returns the remainder of the division operation.

```
void main() {
  int a = 10;
  int b = 3;

  print(a * b); // Output: 30
  print(a / b); // Output: 3.3333...
  print(a ~/ b); // Output: 3
  print(a % b); // Output: 1
}
```

Comparison Operators

- **Greater than (`>`):** Checks if the left operand is greater than the right operand.
- **Less than (`<`):** Checks if the left operand is less than the right operand.
- **Greater than or equal to (`>=`):** Checks if the left operand is greater than or equal to the right operand.
- **Less than or equal to (`<=`):** Checks if the left operand is less than or equal to the right operand.
- **Equal to (`==`):** Checks if two operands are equal.
- **Not equal to (`!=`):** Checks if two operands are not equal.

```
void main() {
  int a = 5;
  int b = 3;

  print(a > b); // Output: true
  print(a < b); // Output: false
  print(a >= b); // Output: true
  print(a <= b); // Output: false
  print(a == b); // Output: false
  print(a != b); // Output: true
}
```

Type Test Operators

- `is` **:** Checks if the object has a specific type.
- `is!` **:** Checks if the object does not have a specific type.

```
void main() {
  double n = 2.20;
  var num = n is! int;
  print(num); // Output: true
}
```

Assignment Operators

- **Equal to (`=`):** Used to assign values to expressions or variables.
- **Null-aware Assignment (`??=`):** Assigns the value only if it is null.

```
void main() {
  int a = 5;
  int b = 7;
  var d;

  d ??= a + b; // Value is assigned as it is null
  print(d); // Output: 12

  d ??= a - b; // Value is not assigned as it is not null
  print(d); // Output: 12
}
```

Logical Operators

- **And Operator (`&&`):** Returns true if both conditions are true.
- **Or Operator (`||`):** Returns true if at least one condition is true.
- **Not Operator (`!`):** Reverses the result.

```
void main() {
  bool isRaining = true;
  bool isSunny = false;

  print(isRaining && isSunny); // Output: false
  print(isRaining || isSunny); // Output: true
  print(!isRaining); // Output: false
}
```

🚦 Control Statements

Decision Making Statements:

1. If Statement:

- The `if` statement is used to execute a block of code if a specified condition is true.
- If the condition evaluates to true, the code inside the block is executed; otherwise, it is skipped.
- Nested `if` statements can be used for more complex conditions.

```
void main() {
  var a = 10;
  var b = 20;
  var c = 30;

  if (a > b) {
    if (a > c) {
      print("a is greater");
    }
  } else if (b > c) {
    print("b is greater");
  } else {
    print("c is greater");
  }
}
```

2. Switch Case Statement:

- The `switch` statement is used to perform different actions based on different conditions.
- It evaluates an expression and compares it with case labels.
- If a case label matches the value of the expression, the corresponding block of code is executed.

```
void main() {
  int n = 3;
  switch (n) {
    case 1:
      print("Value is 1");
      break;
    case 2:
      print("Value is 2");
      break;
    case 3:
      print("Value is 3");
      break;
    default:
      print("Out of range");
  }
}
```

Looping Statements:

1. While Loop:

- The `while` loop executes a block of code as long as a specified condition is true.
- It evaluates the condition before executing the loop body, so the loop may not execute at all if the condition is initially false.

```
void main() {
  var a = 1;
  var max_num = 10;

  while (a <= max_num) {
    print(a);
    a = a + 1;
  }
}
```

2. Do While Loop:

- The `do-while` loop is similar to the `while` loop, but it executes the block of code first and then evaluates the condition.
- This ensures that the block of code is executed at least once, even if the condition is false initially.

```
void main() {
  var a = 1;
  var max_num = 10;

  do {
    print("The value is: ${a}");
    a = a + 1;
  } while (a < max_num);
}
```

3. For Loop:

- The `for` loop is used to execute a block of code a specified number of times.
- It consists of an initialization statement, a condition, and an iteration statement, all of which are optional.
- It is commonly used when the number of iterations is known before entering the loop.

```
void main() {
  for (int num = 1; num <= 10; num++) {
    print(num);
  }
}
```

4. For-In Loop:

- The `for-in` loop is used to iterate over the elements of a collection (such as a list or a map).
- It assigns each element of the collection to a variable for processing.

```
void main() {
  var list1 = [10, 20, 30, 40, 50];

  for (var i in list1) {
    print(i);
  }
}
```

Break and Continue Statements:

Break:

- The `break` statement is used to exit a loop prematurely.
- When the `break` statement is encountered inside a loop, the loop is terminated immediately, and the program continues execution after the loop.

```
int count = 0;
while (count <= 6) {
  print(count);
  count++;
  if (count == 4) {
    break;
  }
}
```

Continue:

- The `continue` statement is used to skip the remaining code inside a loop for the current iteration and proceed to the next iteration.
- When the `continue` statement is encountered inside a loop, the remaining code in the loop for the current iteration is skipped, and the loop proceeds to the next iteration.

```
int count = 0;
while (count <= 6) {
  print(count);
  count++;
  if (count == 4) {
    continue;
  }
}
```

Let's break down each part of the Functions section with explanations:

🔧 Functions

Function Signature and Calling:

- A function is a `set of code` that `performs a specific task`.
- It allows breaking large code into `smaller`, `reusable modules`, `enhancing readability` and `debuggability`.

```
void main() {
  var name = fullName('John', 'Doe');
  print(name);
}

String fullName(String firstName, String lastName) {
  return "$firstName $lastName";
}
```

- **Function Signature**: `return_type func_name(parameter_list)`
- **Function Calling**: `func_name(arguments)`

Arrow Function:

- Arrow functions allow creating functions consisting of a `single expression`, omitting curly brackets and the return keyword.

```
void main() {
  print(add(3, 5));
  print(sub(5, 4));
}

int add(int x, int y) => x + y;
int sub(int x, int y) => x - y;
```

Positional Parameters and Optional Positional Parameters:

- Square brackets `[ ]` specify optional positional parameters.
- Optional parameters should be the last parameters.

```dart
void displayMessage(String msg, [int ntimes = 3]) {
  for (int i = 0; i < ntimes; i++) {
    print(msg);
  }
}

displayMessage('Hello', 5);
displayMessage('Hello');
```

Named Parameters and Optional Parameters:

- Curly brackets `{ }` specify optional named parameters.
- Named parameters are by default optional.

```dart
void displayMsg({String msg = 'Test', int ntimes = 3}) {
  for (int i = 0; i < ntimes; i++) {
    print(msg);
  }
}

displayMsg(ntimes: 4, msg: 'Welcome');
displayMsg(msg: 'Welcome', ntimes: 4);
displayMsg(msg: 'Welcome');
displayMsg(ntimes: 5);
```

Anonymous Function:

- Dart provides the facility to specify a `nameless function` or `function without a name`.

```dart
void main() {
  var list = ["James", "Patrick", "Mathew", "Tom"];
  print("Example of anonymous function");
  list.forEach((item) {
    print("${list.indexOf(item)}: $item");
  });
}
```

🛠️ Exception Handling

Understanding Exceptions:

- Exceptions are `runtime errors` that `occur during program execution`.
- They are `not reported` at compile time.
- These errors terminate program `execution abruptly`, often due to inappropriate conditions like dividing by zero.

Handling Exceptions in Dart:

- Dart treats every exception as a subtype of the pre-defined class `Exception`.
- Dart offers various techniques to handle exceptions:
  - `try/on/catch/finally` Blocks:
    - `try` block holds the code that might throw an exception.
    - `on` block specifies the exceptions to be handled.
    - `catch` block handles the exception object.
    - `finally` block always executes, regardless of whether an exception occurs or not.

Example:

```
void main() {
  try {
    int y = 5 ~/ 0; // Trying to divide by zero
  } on UnsupportedError {
    // Handling specific exceptions
    print("Can't Divide by Zero");
  } catch (e) {
    // Handling other exceptions
    print('Other Errors');
  } finally {
    // Code that always executes
    print('Done Finally');
  }
}
```

Throwing an Exception:

- Exceptions can be raised explicitly to force handling.
- It's essential to handle explicitly raised exceptions to prevent sharp program exits.

Example:

```
void main() {
  try {
    checkMarks(-10); // Trying to check negative marks
  } catch (e) {
    print('The marks cannot be negative');
  }
}

void checkMarks(int marks) {
  if (marks < 0) {
    throw FormatException(); // Raising an exception for negative marks
  }
}
```