Dart – Day 2

- ## Const keyword

In Dart, const creates a compile-time constant value that never changes, and it’s evaluated during compilation.

Example:

```
void main()
{
  const pi = 3.14159;   // compile-time constant
  print(pi);
}
```

- ## Final keyword

In Dart, final is used to declare a variable that can be set only once at runtime, and after that, its value cannot be changed.

Example:

```
void main()
{
  final currentTime = DateTime.now();   // runtime constant
  print(currentTime);
}
```

- ## Arithmetic Operators

Used for mathematical operations.

```
void main()
{
  int a = 10, b = 3;
  print(a + b);   // 13 (Addition)
  print(a - b);   // 7  (Subtraction)
  print(a * b);   // 30 (Multiplication)
  print(a / b);   // 3.333... (Division → double result)
  print(a ~/ b);   // 3 (Integer Division)
  print(a % b);   // 1 (Remainder)
}
```
Arithmetic operators perform basic math like +, -, *, /, %, and integer division ~/.

- **Relational Operators**

Used to compare values (returns bool).

```
void main()
{
  int a = 5, b = 10;
  print(a < b);    // true
  print(a > b);    // false
  print(a <= b);   // true
  print(a >= b);   // false
  print(a == b);   // false
  print(a != b);   // true
}
```
Relational operators check equality and ordering between values.

- **Logical Operators**

Used with boolean values.

```
void main()
{
  bool x = true, y = false;
  print(x && y);   // false (AND)
  print(x || y);   // true  (OR)
  print(!x);    // false (NOT)
}
```
Logical operators combine boolean expressions with &&, ||, and !.

- **Assignment Operators**

Assign values and update variables.

```
void main()
{
  int a = 5;
  a += 2;   // 7
  a -= 1;   // 6
  a *= 2;   // 12
  a ~/= 3;   // 4 (integer division assignment)
  a %= 3;   // 1
  print(a);
}
```

Assignment operators update variables with shortcuts like +=, -=, *=, ~/=, %=.

- **Prefix and Postfix (Increment/Decrement)**

Used to increase/decrease a value by 1.

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

  print(++a);   // 6 (Prefix → increments before use)
  print(a++);   // 6 (Postfix → increments after use)
  print(a);     // 7 (a got incremented)

  print(--a);   // 6 (Prefix decrement)
  print(a--);   // 6 (Postfix decrement)
  print(a);     // 5
}
```

Prefix updates the value before use, while postfix updates after use.

- **Infix Operators**

In Dart, operators like +, -, *, /, ==, <, > etc. are actually just infix operators.

Infix means the operator is written between two operands.

Example:

```
void main()
{
  int a = 10;
  int b = 5;

  print(a + b);    // + is an infix operator
  print(a > b);    // > is an infix operator
}
```

So, in Dart, almost all arithmetic, relational, and logical operators are infix operators because they come between two values.

- ## Type Test Operators

Used to check or cast object types.

```
void main()

{
  var x = "Hello";

  print(x is String);     // true → checks if x is a String
  print(x is int);        // false → checks if x is an int
  print(x is! double);   // true → checks if x is NOT a double

  Object y = "World";
  String z = y as String;   // Cast Object to String
  print(z.toUpperCase());   // WORLD
}
```

Summary in one line:

- is → checks if a variable is of a certain type.
- is! → checks if a variable is NOT of a certain type.
- as → explicitly casts a variable to another type.

- ## Functions

A block of reusable code that performs a specific task.

```
void greet()
{
  print("Hello, Dart!");
}

void main()
{
  greet();   // Calling the function
}
```

- ## Function Parameters in Dart

In Dart, functions can take different types of parameters to make them flexible and easy to use. The main types are:

1. **Positional Parameters** → Passed in the same order as defined.
2. **Named Parameters** → Passed using names (order doesn’t matter).
3. **Named Parameters with Default Values** → Provide fallback values if not given.
4. **Named Parameters with Required Values** → Must be passed explicitly.
5. **Optional Positional Parameters** → Enclosed in [], can be skipped.

1. Positional Parameters

Parameters passed in the exact order they are defined.

```
void greet(String name, int age)
{
  print("Hello $name, you are $age years old.");
}

void main()

{
  greet("Chandini", 21);    // Passed in the same order as in function signature
}
```

2. Named Parameters

Parameters passed by name (order doesn’t matter).

```
void greet({String? name, int? age})
{
  print("Hello $name, age $age");
}

void main()
{
  greet(age: 21, name: "Chandini");    // Order doesn’t matter
}
```

3. Named Parameters with Default Values

Provide default values if not passed.

```
void greet({String name = "Guest", int age = 18})
{
  print("Hello $name, age $age");
}

void main()
{
  greet(); // Uses default values → Guest, 18
  greet(name: "Chandini");   // Overwrites default for name
}
```

4. Named Parameters with Required Values

Force user to pass specific parameters using required.

```
void greet({required String name, required int age})
{
  print("Hello $name, age $age");
}

void main()
{
  greet(name: "Chandini", age: 21);   // Must provide both
}
```

5. Optional Positional Parameters

Enclosed in square brackets [], can be skipped.

```
void greet(String name, [int? age])
{
  print("Hello $name, age $age");
}

void main()
{
  greet("Chandini");   // Age skipped → null
  greet("Sneha", 22);
}
```

- ## String

→ A String in Dart is a sequence of characters used to represent text.
→ Strings are enclosed in single quotes ' ' or double quotes " ".

1. Declaring Strings

```
void main()
{
  String name = 'Chandini';
  String message = "Hello, Dart!";
  print(name);     // Chandini
  print(message);     // Hello, Dart!
}
```

2. Multi-line Strings

- Use triple quotes (''' or """) for multi-line strings.

```
void main()
{
  String note = '''This is
                         a multi-line
                         string.''';
  print(note);        // This is a multi-line string.
}
```

3. String Interpolation (with $)

- Insert variable values inside strings using $variable or ${expression}.

```
void main()
{
  String city = "Bangalore";
  int age = 21;
  print("I live in $city and I am $age years old.");
  print("Next year, I will be ${age + 1} years old.");
}
```

4. String Concatenation

- Combine strings using + or by writing them next to each other.

```
void main()
{
  String first = "Hello";
  String second = "World";
  print(first + " " + second);    // Using +
  print("$first $second");        // Using interpolation
}
```

5. Common String Methods

```
void main()
{
  String text = " Dart Programming ";

  print(text.length);          // 18
  print(text.toUpperCase());   // " DART PROGRAMMING "
  print(text.toLowerCase());   // " dart programming "
  print(text.trim());          // "Dart Programming" (removes spaces)
  print(text.contains("Dart")); // true
  print(text.replaceAll("Dart", "Flutter"));   // " Flutter Programming "
  print(text.substring(1, 5)); // "Dart"
}
```

6. Escape Characters

```
void main()
{
  String s = 'It\'s a sunny day';     // use \ to escape
  String path = "C:\\Users\\Files";    // backslash
  print(s);     // It's a sunny day
  print(path);    // C:\Users\Files
}
```

7. Raw String (with r)

Treats the string literally → escape characters (\n, \t, \) are not processed.

```
void main()
{
  String rawText = r"Hello\nWorld\tDart";
  print(rawText);   // Hello\nWorld\tDart
}
```

- ## Record

A record is a fixed-size, ordered collection of values, which can hold multiple types. Records are lightweight and immutable by default.

1. Positional Records

Values are stored by position.

```
void main()
{
  var record = (1, "Chandini", true);   // int, String, bool
  print(record.$1); // 1
  print(record.$2); // Chandini
  print(record.$3); // true
}
```

Access values using $1, $2, $3, …

2. Named Records

Values are stored with names instead of numeric positions.

```
void main()
{
  var record = (name: "Chandini", age: 21, city: "Bangalore");
  print(record.name); // Chandini
  print(record.age);  // 21
  print(record.city); // Bangalore
}
```

Access values using their names.

3. Mixed Records

You can combine positional and named fields.

```
void main()
{
  var record = (1, "Dart", language: "Flutter", version: 3.0);
  print(record.$1);         // 1
  print(record.$2);         // Dart
  print(record.language);   // Flutter
  print(record.version);    // 3.0
}
```

Summary in one line:

- **Records** group multiple values together.
- **Positional** → access via $index.
- **Named** → access via name.
- **Mixed** → both positional & named fields.

- ## Returning Multiple Values

Dart allows functions to return multiple values easily using records instead of creating a class or list.

```
// Function returning multiple values as a record
(int, String, bool) getUser()
{
  return (1, "Chandini", true);
}

void main()
{
  (int, String, bool) user = getUser();     // OR  var user = getUser();
  print(user.$1);   // 1
  print(user.$2);   // Chandini
  print(user.$3);   // true
}
```

This avoids creating extra classes or arrays just to return multiple pieces of data.

Using Named Fields for Clarity

```
({String name, int age, bool isActive}) getUser()
{
  return (name: "Chandini", age: 21, isActive: true);
}

void main() {
  ({String name, int age, bool isActive}) user = getUser();
  print(user.name);     // Chandini
  print(user.age);      // 21
  print(user.isActive); // true
}
```
Named records make the returned values easier to read and access.