1. What Are Variables in Dart?

In Dart, a variable is a named reference to an object stored in memory. Unlike some languages where variables might directly hold primitive values, Dart variables hold references. This means that even when a variable seems to store a simple value (like an integer or a string), it actually points to an object that holds that value.

---

2. Type Inference vs. Explicit Typing

Dart is statically typed but supports both explicit type annotations and type inference:

- **Type Inference with var:**
  When you declare a variable with var, the compiler infers the type from the assigned value at compile time:

```
var name = 'Bob'; // Dart infers type String.

var age = 25;     // Dart infers type int.
```

Although concise, once inferred, the type cannot change. This means:

```
var name = 'Alice';

// name = 42; // Error: Cannot assign an int to a String variable.
```

Explicit Typing:

You may declare the type explicitly, which enhances code clarity, especially for public APIs or class members:

```
String greeting = 'Hello';

int year = 2025;

Object flexible = 'Can hold any object type';
```

Explicit typing is also useful when the initial value might be ambiguous or when you plan to use a broader type (like Object or dynamic) later.

Dart provides a rich type system that includes:

- **Numbers:** int, double, and num for numeric operations.
- **Boolean:** bool for true/false values.
- **String:** For text data.
- **Collections:** List, Set, and Map for various data grouping needs.
- **Runes:** To work with Unicode code points.
- **Symbols:** For identifier names used in reflection.

- **Dynamic:** For flexible types with runtime type changes.
- **Object:** As the base type for all Dart types.

---

1. Numbers

Dart has three main numeric types: **int**, **double**, and **num**.

int:

Represents whole numbers (both positive and negative).

```
int wholeNumber = 42;

print('Int: $wholeNumber'); // Output: Int: 42
```

double:

Represents numbers with a decimal point (floating-point numbers).

```
double pi = 3.14159;

print('Double: $pi'); // Output: Double: 3.14159
```

num:

A common superclass for both int and double. It allows you to store either type.

```
num someNumber = 10;     // Could be an int

print('Num as int: $someNumber'); // Output: Num as int: 10


someNumber = 10.5;       // Now a double

print('Num as double: $someNumber'); // Output: Num as double: 10.5
```

2. Boolean

The **bool** type represents a true/false value.

```
bool isDartFun = true;

print('Boolean: $isDartFun'); // Output: Boolean: true
```

Booleans are often used in control flow statements (if, while, etc.).

3. Strings

A **String** is a sequence of characters used to represent text.

```
String greeting = 'Hello, Dart!';
print('String: $greeting'); // Output: String: Hello, Dart!
```

Strings can be manipulated using various built-in methods such as substring, split, and replace.

---

4. Collections

Dart provides several collection types:

a. List

A **List** is an ordered collection of items. You can specify the type of elements in the list.

Lists support indexing, iteration, and many useful methods (e.g., add, remove).

```
List<String> fruits = ['Apple', 'Banana', 'Cherry'];
print('List: $fruits'); // Output: List: [Apple, Banana, Cherry]
 print('List: ${fruits.join(', ')}'); // Output: List: Apple, Banana, Cherry
```

b. Set

A **Set** is an unordered collection of unique items. Duplicate entries are automatically removed.

Sets are useful when you need to ensure no duplicates are present.

```
Set<int> uniqueNumbers = {1, 2, 5, 3, 2}; //don't print repet ative values
  print('Set: $uniqueNumbers'); // Output: Set: {1, 2, 3}
  print('Set: ${uniqueNumbers.join(',')}');
```

c. Map

A **Map** is a collection of key-value pairs. The keys must be unique.

Maps are excellent for lookups and representing relationships between keys and values.

```
Map<String, int> ages = {
  'Alice': 30,
  'Bob': 25,
  'Charlie': 35,
};
print('Map: $ages'); // Output: Map: {Alice: 30, Bob: 25, Charlie: 35}
```

---

5. Runes

Runes represent Unicode code points of a string. They are useful for handling characters outside the basic multilingual plane (e.g., emojis).

```
String heart = ' 💙 ';//print decimal valure of characters

Runes runes = heart.runes;

print('Runes: ${runes.toList()}'); // Output: Runes: [10084, 65039]
```

Runes allow you to work with the numerical representation of Unicode characters.

---

6. Symbols

A **Symbol** represents an operator or identifier name. It is mainly used in APIs that rely on reflection.

```
Symbol mySymbol = #example;

print('Symbol: $mySymbol'); // Output: Symbol: Symbol("example")
```

Symbols are rarely used in everyday Dart programming but can be useful in advanced scenarios.

---

7. Dynamic

The **dynamic** type allows a variable to hold any type of value, and its type can change over time. However, using dynamic bypasses compile-time type checking.

```
dynamic variable = 'Hello, world!';

print('Dynamic (as String): $variable'); // Output: Dynamic (as String): Hello, world!


variable = 123;

print('Dynamic (as int): $variable'); // Output: Dynamic (as int): 123
```

Use **dynamic** sparingly as it can make code less predictable and harder to maintain.

---

8. Object

The **Object** type is the base type for all Dart objects. Every type in Dart is a subclass of Object.

```
Object obj = 'I am an object';

print('Object: $obj'); // Output: Object: I am an object
```

Even though **Object** is a supertype of all classes, explicit casting may be necessary when you need a more specific type.

Full Example: Putting It All Together

```
void main() {
 // 1. Numbers
 // Dart supports whole numbers (int), floating-point numbers (double), and a common type (num) that can hold either.

 int wholeNumber = 42;
 double pi = 3.14159;
 num flexibleNumber = 10; // Initially an int
 flexibleNumber = 10.5;   // Now holds a double

 print('Int: $wholeNumber');        // Output: Int: 42
 print('Double: $pi');              // Output: Double: 3.14159
 print('Num (int then double): $flexibleNumber'); // Output: Num (int then double): 10.5

 // 2. Boolean
 // The bool type represents true/false values and is commonly used in conditions.

 bool isDartFun = true;
 bool isLearning = false;
 print('Boolean isDartFun: $isDartFun'); // Output: Boolean isDartFun: true
 print('Boolean isLearning: $isLearning'); // Output: Boolean isLearning: false

 // 3. Strings
 // Strings are sequences of characters. You can use either single or double quotes.

 String greeting = 'Hello, Dart!';
 String message = "Dart is awesome.";
 print('String greeting: $greeting'); // Output: String greeting: Hello, Dart!
 print('String message: $message');   // Output: String message: Dart is awesome.

 // 4. Collections
 // Dart provides various collection types like List, Set, and Map.

 // List: An ordered collection of items.
```

```
List<String> fruits = ['Apple', 'Banana', 'Cherry'];
print('List of fruits: $fruits');  // Output: List of fruits: [Apple, Banana, Cherry]

// Set: An unordered collection of unique items.
Set<int> uniqueNumbers = {1, 2, 3, 2};
print('Set of unique numbers: $uniqueNumbers'); // Output: Set of unique numbers: {1, 2, 3}

// Map: A collection of key-value pairs.
Map<String, int> ages = {
  'Alice': 30,
  'Bob': 25,
  'Charlie': 35,
};
print('Map of ages: $ages'); // Output: Map of ages: {Alice: 30, Bob: 25, Charlie: 35}

// 5. Runes
// Runes represent Unicode code points. They are useful for handling characters outside the basic multilingual plane.

String heartEmoji = '💙';
Runes heartRunes = heartEmoji.runes;
print('Runes for heart emoji: ${heartRunes.toList()}'); // Output might be: [10084, 65039]

// 6. Symbols
// Symbols are used to refer to identifiers by name. They are mostly useful in reflection and meta-programming.

Symbol mySymbol = #myExample;
print('Symbol: $mySymbol'); // Output: Symbol("myExample")

// 7. Dynamic
// Dynamic allows the variable's type to change at runtime, but it sacrifices compile-time type checking.

dynamic variable = 'Hello, world!';
print('Dynamic as String: $variable'); // Output: Dynamic as String: Hello, world!
variable = 123; // Changing type to int
print('Dynamic as int: $variable');    // Output: Dynamic as int: 123
```

```
// 8. Object
// Object is the base type for all Dart objects. Every type in Dart is a subclass of Object.

Object genericObject = 'I can be any object';
print('Object: $genericObject'); // Output: Object: I can be any object
}
```

NULL

What is Null?

- **Null is a Unique Object:**
  In Dart, null is the sole instance of the Null class. It represents a lack of value, similar to how "nothing" is represented in other languages.
- **Before Null Safety:**
  Historically, any variable in Dart could be assigned null, which sometimes led to runtime errors when methods or properties were accessed on a null value.
- **After Null Safety:**
  With the introduction of sound null safety (starting with Dart 2.12), variables are non-nullable by default. This means a variable cannot hold a null value unless it is explicitly declared as nullable by appending a ? to its type.

How Null Works in Dart

- **Non-nullable Variables:**
  By default, if you declare a variable without marking it as nullable, Dart expects it to always have a valid, non-null value. For example:

```
String name = 'Alice';
// name cannot be assigned null later.
```

- **Nullable Variables:**
  If you want a variable to be able to hold null, you must declare it with a ?. This tells Dart that the variable might be null at some point.

```
String? name;
print(name); // Outputs: null
name = 'Bob';
print(name); // Outputs: Bob
```

- **Null-Aware Operators:**
  To safely work with nullable variables, Dart provides several null-aware operators:

- **Null-Aware Access (?.):**
  Safely access a property or method on an object that might be null.

```
String? name;

print(name?.length); // If name is null, this evaluates to null.
```

  - When you try to access name?.length:
    - **If name is not null**: The length property of the string would be accessed, and the length of the string would be printed.
    - **If name is null**: The ?. operator prevents a NullPointerException and instead evaluates the expression as null.

    Since name is null in this case, the expression name?.length evaluates to null, and that's what gets printed.

- **Default Value Operator (??):**
  Provide a default value if an expression is null.

```
String? name;

print(name ?? 'No name provided'); // Prints the default message if name is null.
```

- **Null Assertion Operator (!):**
  Assert that a value is non-null (and throw a runtime error if it is null).

```
String? name = 'Dart';

print(name!.length); // Asserts name is non-null.
```

Why Managing Null is Important

Handling null correctly is crucial because:

- **Preventing Runtime Errors:**
  Accessing methods or properties on a null value can lead to crashes (known as null dereference errors).
- **Improving Code Safety:**
  With null safety, many potential bugs are caught at compile time rather than at runtime.
- **Enhanced Code Clarity:**
  Declaring whether a variable can be null makes your code's intent clearer to both the compiler and other developers.

Example: Handling Null in Dart

Here's a small example that demonstrates nullable and non-nullable variables, and how to work safely with null values:

Summary

- **Null** represents “no value” and is an instance of the Null class.
- With **null safety**, variables are non-nullable by default unless explicitly declared as nullable (using ?).
- Null-aware operators (?., ??, !) are essential tools for writing safe code when dealing with nullable values.
- Properly managing null helps prevent runtime errors and makes your code more robust.

Late Variables

The late modifier is a key part of Dart’s null safety, and it addresses two main scenarios:

- **Deferred (Postponed) Initialization**
  Sometimes you know that a variable will be assigned a non-null value—but not at the point of declaration. For example, when you have a non-nullable instance variable that will be set later (perhaps in a method or during runtime logic), you can mark it as late:
  late String description;
  void main() {
  description = 'Feijoada!';
  print(description);
  }
  This tells Dart, “Trust me, I’ll set this variable before I use it.” However, if you try to read a late variable before it has been given a value, Dart will throw a runtime error. This adds a layer of safety by ensuring that a non-nullable variable is indeed non-null when accessed.
- **Lazy Initialization**
  If you provide an initializer along with the late keyword, that initializer doesn’t run immediately. Instead, it executes the first time the variable is used. This is very useful when the initialization is expensive or when the variable might not be needed during every execution path:
  late String temperature = readThermometer();
  Here, the function readThermometer() is called only if—and when—the variable temperature is accessed for the first time.

---

3. Final and Const

What They Mean

- **Final Variables:**
  When you declare a variable as final, you tell Dart that the variable’s value can be set only once. This means you can assign it at runtime—but after it’s been set, any further attempt to change it will result in an error. A key point is that a final variable doesn’t have to be known at compile time; it can be computed later, such as within a function or after some asynchronous work.

```
final currentTime = DateTime.now();

// Later in the code, trying to assign a new value to currentTime would be an error.
```

- **Const Variables:**
  The const keyword, on the other hand, is used to define compile-time constants. A const variable's value is determined during compilation, and it must be a literal or a compile-time expression. Because its value is fixed at compile time, a const variable is implicitly final.

```
const pi = 3.14159;

const list = [1, 2, 3];
```

Here, pi is known at compile time, and you can also declare collections as const so that the entire structure becomes deeply immutable.

Key Differences

1. **When the Value Is Set:**
   - *Final:* The value is set only once, but not necessarily at compile time. It's common to initialize a final variable during runtime (for example, reading the current date/time or the result of a computation).
   - *Const:* The value is set at compile time. You cannot use a const variable for values that are determined during program execution.
2. Mutability and Immutability:

Example:

```
const pi = 3.145;
void main() {
 final String nickname = 'Bobby';
 //nickname = 'hi'; show error
 print(nickname);
 print('PI = $pi');
 final name = 'Bob';
 final currentTime = DateTime.now();
// Later in the code, trying to assign a new value to currentTime would be an error.
}
```

---

4. Wildcard Variables

What Are Wildcard Variables?

Wildcard variables are a Dart language feature introduced in Dart 3.7. They allow you to declare a variable using an underscore (_) as a placeholder. When you use _ as a variable name, you're

indicating that the value is not needed or will not be used. Essentially, it creates a non-binding variable—one that is present only to satisfy syntax requirements but isn't accessible in your code.

Why Use Wildcard Variables?

Wildcard variables serve several important purposes:

- **Avoiding Unused Variables:**
  When you need to satisfy the syntax of the language (for instance, in a for-loop or a catch clause) but you don't actually care about the value, using _ makes your intent clear.

```
// In a loop where the index isn't needed:
for (var _ in list) {
  // Do something that doesn't require the element.
}
```

- **Reducing Clutter and Naming Collisions:**
  By using _, you avoid declaring and potentially clashing with other variable names. Multiple wildcard variables can coexist in the same scope without causing conflicts.
- **Expressing Intent:**
  When you see _ as a parameter or variable, it immediately signals to anyone reading the code that this value is intentionally being ignored.

Common Use Cases

1. **Local Variable Declarations:**

```
void main() {
  // Calling a function for its side effect, but we don't need the result.
  // By assigning the result to '_', we indicate that we intentionally ignore it.
  var _ = performSideEffect();

  // Continue with other code...
}

void performSideEffect() {
  print("Side effect executed");
}
```

2. **For loop variable**

```
void main() {
  // Suppose we want to print "Hello" 5 times:
```

```
  // We don't need to use the loop variable, so we use '_'.
  for (var _ in Iterable.generate(5)) {
    print("Hello");
  }
}
```

3.Catch Clauses

4.Generic and Function Type Parameters