Dart for Java/JavaScript Developers

Key Differences
& Unique Dart Features

- List Creation & Manipulation
- Cascade Operator (..)
- Named Parameter
- Function Signatures
- Extension Methods
- Immutable Data Updates in Dart: copyWith()
- Collection Spreads & If/For in Collections
- Lazy Initialization
- Factory Constructors
- Mixins
- Pattern Matching
- Null Safety
  - Null Coalescing Operator (??)
  - Null Aware Operation (??=)
- "is" and "as" operator
- Getters and Setters

List Creation & Manipulation

- Java: Verbose with multiple statements

```java
var list = new ArrayList<String>();  // Create
list.add("Hello");                   // Add returns boolean
list.add("World");                   // Add returns boolean
list.sort();                         // Sort returns void
return list;                         // Return separately
```

- **JavaScript** (Simple in creation, but verbose in adding elements)

```
let list = [];           // Create
list.push("Hello");      // Push returns new length
list.push("World");      // Push returns new length
list.sort();             // Sort returns array
return list;             // Return separately
```

Cascade Operator (..)

- Dart supports the cascade operator to return the original object.

```
var list = <String>[]
  ..add("Hello")
  ..add("World")
  ..sort();
```

- This is useful even with constructors.

```dart
var paint = Paint()
  ..color = Colors.blue
  ..strokeWidth = 5.0
  ..style = PaintingStyle.stroke;
```

Java

```java
Paint paint = new Paint();                  // Create object
paint.setColor(Colors.BLUE);                // Set color (void return)
paint.setStrokeWidth(5.0f);                 // Set width (void return)
paint.setStyle(PaintStyle.STROKE);          // Set style (void return)
```

JavaScript

```javascript
let paint = new Paint();          // Create object
paint.color = Colors.blue;        // Set property
paint.strokeWidth = 5.0;          // Set property
paint.style = PaintStyle.stroke; // Set property
```

Named Parameter

Java:

```java
new User("John", 25, "john@email.com", true);
// Hard to remember parameter order
```

JavaScript:

```javascript
let user = new User("John", 25, "john@email.com", true);
```

Dart

- Use `{...}` for named parameters in Dart; use `required` to make them mandatory so the compiler enforces their presence.
- Here, `x` must be provided, but `y` is optional.

```
void foo({required int x, int y = 0}) ...
foo(x:10) // same as foo(x:10, y:10)
```

- `name` is required – must be provided.
- `email` is nullable, defaults to `null` if omitted.
- `age` and `isActive` have default values, so they’re optional.

```
class User {
  String name; int age; String? email; bool isActive;
  User({required this.name, this.age = 0,
        this.email, this.isActive = false});
}
```

```dart
class User {
  String name; int age; String? email; bool isActive;
  // Constructor
  User({required this.name, this.age = 0,
        this.email, this.isActive = false});
}

// Clear and flexible
// this.email is null as it is nullable
// age = 0 from the given default value
var user = User(
  name: "John", // Not compile if not provided
  isActive: true
);
```

Function Signatures

```
Future<RecordModel?> getRecord([int page = 1, int perPage = 5])
```

- 🔄 An **asynchronous function** signature
- 📦 Returns a **Future** containing either a RecordModel or null
- ⚙️ Takes **optional parameters** for pagination
- 🌐 Commonly used in **database queries** and API calls

Return Type: Future<RecordModel?>

- `Future<T>` - Represents an asynchronous operation
- `RecordModel` - Custom class representing data structure
- `?` - Nullable type (can be null)

Parameters: [int page = 1, int perPage = 5]

Square Brackets `[]` = Optional Positional Parameters

- `page = 1`: Default page number is 1
- `perPage = 5`: Default records per page is 5

- You can call this function with **0, 1, or 2 arguments**

```
// All these calls are valid:
getRecord()           // page=1, perPage=5
getRecord(2)          // page=2, perPage=5
getRecord(3, 10)      // page=3, perPage=10
```

This is a syntax error in Dart.

```
Future<RecordModel?> getRecord(int page = 1, int perPage = 5)
getRecord(1); // ??? which is given: page or perPage?
```

In Dart, we must provide arguments to avoid confusion when using optional parameters.

```
// int page is required arguments
Future<RecordModel?> getRecord(int page, int perPage = 5)
getRecord(1); // same as getRecord(1,5)
```

Extension Methods

- We can add methods to existing classes in JavaScript/Dart.

```
extension StringExtension on String {
  String reverse() => split('').reversed.join('');
  bool get isEmail => contains('@') && contains('.');
}

// Usage
print("hello".reverse());     // "olleh"
print("test@email.com".isEmail); // true
```

JavaScript: (Prototype modification - not recommended)

```javascript
String.prototype.reverse = function() {
  return this.split('').reverse().join('');
};

// Usage
console.log("hello".reverse()); // Output: "olleh"
let name = "JavaScript";
console.log(name.reverse()); // Output: "tpircSavaJ"
```

- JavaScript: Modifying `String.prototype` affects all strings globally and can break other code.
- Dart: Extensions are non-invasive, scoped, and don’t alter the original class—safe and clean
    - They're just syntactic sugar for calling helper functions on objects.

Immutable Data Updates in Dart: copyWith()

Problem: What if you change a part of an object?

```dart
class User {
  // final - can be assigned only in the constructor
  final String name;
  final int age;
  final String email;

  const User({required this.name,
    required this.age,
    required this.email});
  // copyWith creates a new instance with some fields changed
  User copyWith({String? name, int? age, String? email}) {
    return User(
      // if name is not given, use this.name instead
      name: name ?? this.name,
      // if age is not given, use this.age instead
      age: age ?? this.age,
      // if email is not given, use this.email instead
      email: email ?? this.email,
    );
  }
}
```

Key Idea: Create a new object with selective changes, keeping other fields unchanged.

Usage Examples

```
var user = User(name: "John", age: 25, email: "john@email.com");

// Change only the age
var olderUser = user.copyWith(age: 26);
print(olderUser); //
  print(olderUser.name);  // "John" (unchanged)
  print(olderUser.age);   // 26 (changed)
  print(olderUser.email); // "john@email.com" (unchanged)

  // Change nothing (creates an identical copy)
  var copy = user.copyWith();

  print(user == olderUser);        // false (different objects)
  print(user.name == copy.name);   // true (same values)
}
```

Why copyWith() is Essential

Without `copyWith()` (Problematic):

```dart
// ❌ Can't modify - fields are final
user.age = 26;  // Compile error!

// ❌ Verbose and error-prone
var olderUser = User(
  name: user.name,        // Easy to forget fields
  age: 26,               // Only this should change
  email: user.email,     // Repetitive
);
```

With `copyWith()` (Clean):

```
// ✅ Clear intent, less error-prone
var olderUser = user.copyWith(age: 26);
```

Benefits:

- 🔒 **Immutability**: Objects never change (thread-safe)
- 🎯 **Selective Updates**: Change only what you need
- 📝 **Less Boilerplate**: No need to repeat all fields
- 🐛 **Fewer Bugs**: Can't accidentally miss fields
- 🧠 **Clear Intent**: Obvious which fields are changing

Perfect for state management, data classes, and functional programming! 🚀

Collection Spreads & If/For in Collections

JavaScript:

```
const list1 = [1, 2];
const list2 = [3, 4];
const combined = [...list1, ...list2];
```

Dart (More Powerful):

```dart
var list1 = [1, 2];
var list2 = [3, 4];
bool condition = true;
var range = [10, 20, 30];

// Dart's powerful collection syntax
var combined = [
  ...list1,                    // Spread list1: [1, 2]
  ...list2,                    // Spread list2: [3, 4]
  if (condition) 5,            // Conditional element: 5 (if true)
  for (var i in range) i * 2, // [20, 40, 60]
];

// => [1,2,3,4,5,20,40,60]
```

Lazy Initialization

- Delaying the creation or computation of a value until it's needed.

Java:

```java
private String expensiveValue;
// expensiveValue is computed only when it is needed.
public String getExpensiveValue() {
  if (expensiveValue == null) {
    expensiveValue = computeExpensive();
  }
  return expensiveValue;
}
```

Dart:

- In Dart, non-nullable variables cannot be initialized with null.
- But using `late` allows you to defer their initialization.

```
String errorName; // Error as no assignment value
late String name; // No error
void initName() {
  // it's OK to assign late before using it
  name = 'John';
  if (name == 'John') ...
}
```

We can use `late` to make the code simple.

```dart
// computedExpensive is not called now
late String expensiveValue = computeExpensive();
// It is computed now
if (expensiveValue == 10) ...
```

- Combining with `final`, we can make a computed once and a mutable variable.

```dart
late final String config = loadConfig();
// Computed once, then immutable
```

Without Lazy Initialization:

```
class VideoPlayer {
  final VideoCodec codec = loadHeavyCodec();        // 2 seconds
  final AudioProcessor audio = initAudio();         // 1 second
  final NetworkBuffer buffer = allocateBuffer();    // 500ms

  VideoPlayer() {
    // Total: 3.5 seconds startup time!
    // Even if the user wants to check video info
  }
}

var player = VideoPlayer(); // 3.5 second delay
print(player.getVideoInfo()); // Just wanted metadata!
```

With Lazy Initialization

```
class VideoPlayer {
  late VideoCodec codec = loadHeavyCodec();      // Only if playing
  late AudioProcessor audio = initAudio();       // Only if audio needed
  late NetworkBuffer buffer = allocateBuffer();  // Only if streaming

  VideoPlayer() {} // Instant creation!

  String getVideoInfo() => "Video: 1080p, 60fps"; // No heavy loading
  void play() => codec.decode(buffer.getData());  // Now they load
}

var player = VideoPlayer();        // Instant!
print(player.getVideoInfo());      // Fast!
player.play();                     // Now loads codec & buffer
```

Factory Constructors

For flexibility, we use a factory that uses a constructor.

Java: uses static function

```
public static User createGuest() {
  return new User("Guest", 0, null, false);
}
User u = createGuest(); // factory
```

Dart supports factory.

```
class User {
  User({required this.name, this.age = 0});

  factory User.guest() => User(name: "Guest");
  factory User.fromJson(Map<String, dynamic> json) {
    return User(name: json['name'], age: json['age']);
  }
}

var guest = User.guest();       // Cleaner syntax
var user = User.fromJson(data); // Named constructors
```

Singleton implemented with a factory.

```dart
class Logger {
  static Logger? _instance;
  final String name;

  // Private constructor
  Logger._(this.name);

  // Factory constructor that returns a singleton
  factory Logger(String name) {
    // if _instance is null, private constructor is called
    _instance ??= Logger._(name);
    return _instance!;
  }
}
var logger1 = Logger("App");
// Still returns the same instance
// name is still "App"
var logger2 = Logger("Database");
```

Mixins

Multiple Inheritance Alternative

Java: (Interface with default methods)

```java
interface Flyable {
  void fly();
}
```

- A class should implement the interface.

Dart:

```dart
mixin Flyable {
  void fly() => print("Flying");
}

mixin Swimmable {
  void swim() => print("Swimming");
}

class Duck with Flyable, Swimmable {
  void quack() => print("Quack");
}

var duck = Duck()..fly()..swim()..quack();
```

- They are about the contract.
  - defines what a class can do (behavior), but can’t provide instance fields or maintain state.
- Dart mixins are about sharing behavior/code between classes.
  - allowing concrete implementations to be composed into classes without the need for inheritance.

Pattern Matching

- Pattern matching empowers you to concisely identify, extract, and act on complex data structures.
- It allows us to recognize regularities or shapes, making problem-solving cleaner, faster, and more expressive

Dart supports switch/case

```dart
double getArea(Shape shape) {
  switch (shape.runtimeType) {
    case Circle:
      var r = (shape as Circle).radius;
      return r * r * 3.14;
    case Rectangle:
      var rect = shape as Rectangle;
      return rect.width * rect.height;
    case Square:
      var s = (shape as Square).side;
      return s * s;
    default:
      throw Exception('Unknown shape');
  }
}
```

- We can make the code easier with a Pattern matching.
- `=> f` operator is a syntactic sugar of { return f }

```
double getArea(Shape shape) =>
  switch (shape) {
    Circle(radius: var r) =>
      r * r * 3.14,
    Rectangle(width: var w, height: var h) =>
      w * h,
    Square(side: var s) =>
      s * s,
    Shape() => throw UnimplementedError(),
};
```

Pattern matching is a better approach than if/else.

- Concise & Clear: Map each shape to its logic—easy to read and follow.
- Exhaustiveness Checking: The compiler makes sure all shapes are handled—fewer bugs.

- Less Boilerplate: No repetitive type checks or casts, code is cleaner.
- Safe Extraction: Variables are extracted for you—less error-prone.
- Expressive: Business logic is central, not hidden in conditionals.

Destructive via Patterns

- Without Pattern

```
if (user is User && user.age > 18) {
  var n = user.name;
  print('Adult: $n');
}
```

- With Pattern & destructive

```
if (user case User(name: var n, age: > 18)) {
  print('Adult: $n');
}
```

```
user case User(name: var n, age: > 18)
```

- Check if `user` is of type `User`.
- Destructure (i.e., extract) its `name` and `age` fields.
- Apply a condition: `age > 18`.
- Assign the user's name to the variable `n` if the match is successful.

Null Safety

Null: the billion-dollar mistake

- Unlike Java/JavaScript, Null Safety is built into Kotlin.

Java:

```
String name; // Can be null
String getName() { return name; } // Runtime Error!
```

Dart:

- We should specify if a variable can be null using `?`.

```dart
String name;        // Compile error!
String? name;       // Nullable
String name = 'Hi'; // Non-nullable

String? getName() => name; // name can be null
print(name!);       // Force unwrap (use carefully)
```

Use Force unwrap carefully

- When you’re sure a nullable variable isn’t null, use `!` (null assertion) so Dart treats it as non-null—but beware: if you’re wrong, it throws a runtime error.

```
void main() {
  String? name;
  print(name!);
  // Uncaught Error: Null check operator used on a null value
}
```

1. The variable `name` is declared with a nullable type: `String?`, meaning it can hold `null` or a `String` value.
2. It is never assigned a value, so its default value is `null`.

3. The line `print(name!)` uses the null assertion operator `!` to tell the compiler: "Trust me, `name` is not null."
4. However, since `name` is null, Dart will throw a runtime error when you run the code:

Null Coalescing Operator (??)

Provides default values for null cases:

```
// name can be null
String? name = null;
// displayName cannot be null
String displayName = name ?? 'Anonymous';
```

Chaining:

```
String? first = null;
String? second = null;
String? third = "Found!";

String result = first ??
                second ??
                third ??
                'Default'; // Not reached
// result = "Found!"
```

- Use the null-coalescing operator (`??`) when you’re unsure if a variable could be null—it safely provides a default value.
- The `??` operator is especially handy in database programming, where values might often be null or missing.

Java:

```java
String userName = "Guest";
int userAge = 18;

if (userData != null && userData.get("name") != null) {
    userName = (String) userData.get("name");
}
if (userData != null && userData.get("age") != null) {
    userAge = (int) userData.get("age");
}
```

Dart:

```dart
String userName = userData?['name'] as String? ?? 'Guest';
int userAge = userData?['age'] as int? ?? 18;
```

```dart
String userName = userData?['name'] as String? ?? 'Guest';
```

- `userData?['name']` is potentially `null` because the value at the `'name'` key may not exist, or `userData` itself could be `null`.
- `as String?` tells Dart: “Treat whatever I get here as a `String?` (nullable string).”
- When it is `null`, use 'Guest' instead.

Null Aware Operation (??=)

- `variable ??= value;` only assigns `value` if `variable` is currently `null`.
- This is handy for providing default values without overwriting any existing non-null values.

- Without ??=

```
if (userData["name"] == null) {
  userData["name"] = expand;
}
if (userData["age"] == null) {
  userData["name"] = field;
}
```

- With ??=

```
userData["expand"] ??= expand;
userData["fields"] ??= fields;
```

Super Parameters in Dart

Super Parameters: Making your constructors simpler!

- **What?** Forward constructor parameters from a subclass to its superclass automatically.
- **How?** Use `super.name` etc. right in the subclass constructor.
- **Why?**
  - Less boilerplate
  - Cleaner code
  - Parent class fields initialized transparently

- `MyApp({super.key})` keeps Flutter widget constructors clean and ensures keys are set up correctly for the widget tree—no extra boilerplate needed.

```
class MyApp extends StatelessWidget {
  MyApp({super.key});  // ← This line!
  @override
  Widget build(BuildContext context) {
    return MaterialApp(title: 'Flutter Demo',
                              home: MyHomePage());
  }
}
```

- **Translation:** "Accept an optional `key` parameter and forward it directly to the parent StatelessWidget constructor"

```
MyApp({super.key})
│     │    │   │
│     │    │   └─── Parameter name
│     │    └─────── Super parameter syntax
│     └──────────── Named parameters
└────────────────── Constructor name
```

```
StatelessWidget({Key? key})
```

- In `MyApp({super.key})`, the `key` parameter is automatically made optional because it’s defined as an optional parameter in the superclass constructor.
- Dart’s super parameter feature infers the type and nullability.

Traditional vs Super Parameters

- **Traditional Way** (Verbose)

```
class MyApp extends StatelessWidget {
  // Declare + forward manually
  MyApp({Key? key}) : super(key: key);
  //
  //                                  Pass to parent
  //                                  Call parent constructor
  //                                  Declare parameter
}
```

- **Super Parameters** (Concise)
- Same functionality, cleaner syntax - no repetition!

```
class MyApp extends StatelessWidget {
  MyApp({super.key});  // Direct forwarding!
}

var app1 = MyApp();                    // No key
var app2 = MyApp(key: ValueKey('app')); // With key
```

"is" and "as" operator

Mimicking JavaScript with dynamic

- JavaScript is typeless; we can use `dynamic` if we don't want to specify a type.

```
dynamic value = "hello";
print(value);
value = 42;
print(value);
value = 3.14;
```

is operator for typechecking

- We use the `is` operator for typechecking.

```
void typeCheck(value) {
  if (value is String) {
    print('Value is a String: $value');
  } else if (value is int) {
    print('Value is an int: $value');
  } else {
    print('Value is of unknown type: $value');
  }
}
```

as operator for a type cast (type conversion)

- If value is not actually a "String" at runtime, Dart throws a "TypeError".
- Use the as operator only if you are certain the value is of the target type to avoid runtime errors.

- Only if we are sure

```dart
dynamic value = 20;
// Cast dynamic to String
String text = value as String;
int length = text.length; // runtime error
```

- Better to be safe

```dart
// Cast dynamic to String
if (value is String) {
  String text = value as String;
}
typeCheck(text); // Value is a String: ...
```

Getters and Setters

JavaScript:

```javascript
class Temperature {
    constructor() { this._celsius = 0; }

    get celsius() { return this._celsius; }
    set celsius(value) { this._celsius = value; }

    get fahrenheit() {
        return this._celsius * 9/5 + 32;
    }
}
```

Dart: Property-like access with `get` and `set`

```dart
class Temperature {
    double _celsius = 0;
    // Getter: access like a property
    double get celsius => _celsius;
    // Setter: assign like a property
    set celsius(double value) => _celsius = value;
    // Computed property: calculated on demand
    double get fahrenheit => _celsius * 9/5 + 32;
}
```

Property Syntax

```
void main() {
    var temp = Temperature();

    temp.celsius = 25;             // Looks like property assignment
    print(temp.celsius);           // Looks like property access
    print(temp.fahrenheit);        // Computed property: 77.0
}
```

No difference in usage between:

- Real properties: `temp.celsius`
- Computed properties: `temp.fahrenheit`

When to Use Getters/Setters

Use getters for:

- Computed values (area, full name)
- Data formatting (currency, dates)
- Read-only access to private fields

Use setters for:

- Input validation
- Data transformation before storing
- Triggering updates when values change

Keep as regular fields when:

- Simple data storage with no logic needed