Importance of null safety in Dart & flutter

Avoiding the Billion-Dollar Mistake in Flutter: Understanding Dart Null Safety

As a Flutter developer, found that understanding null safety is crucial to writing robust and bug-free code.

Null safety is a major feature in Dart that helps us avoid null reference errors, which are common runtime errors in many programming languages. By understanding and properly implementing Dart Null Safety, we can write null-safe code, reducing the possibility of null errors and making our code more predictable and easier to maintain.

Null safety is a concept that helps us to avoid one of the most common issues in programming - null reference errors. These errors occur when we try to access properties or methods on a null value. With Dart's null safety, we can prevent these errors at compile time, making our code safer and more predictable.

What is Null Safety?

Null safety is a feature in Dart that helps us distinguish between nullable variables and non-nullable variables. A nullable variable is one that can hold either a non-null value or a null value. On the other hand, a non-nullable variable is one that must always hold a non-null value.

```
1   void main () {
2    int nonNullableVariable = 10; // Non-nullable variable
3    int? nullableVariable = null; // Nullable variable
4   }
```

In the above code, **nonNullableVariable** is a non-nullable variable and **nullableVariable** is a nullable variable. If we try to assign a null value to **nonNullableVariable**, the Dart compiler will throw a compile-time error.

The Problem of Null Reference Errors:

Null reference errors, also known as the billion-dollar mistake, are a common type of runtime error that occurs when we try to access properties or methods on a null value. These errors can be hard to debug and can lead to unexpected behavior in our code.

```
1   void main () {
2    String? nullableVariable = null;
3    print (nullableVariable.length);                    // This will throw a runtime error
4   }
```

In the preceding code, we are attempting to access the nullableVariable's length property, which is null. This will result in a runtime error.

How Dart Null Safety Helps:

Dart Null Safety helps us to avoid null reference errors by distinguishing between nullable and non-nullable variables. It also provides null safety support through null-aware operators, which allow us to perform operations on nullable variables without causing null reference errors.

```
1   void main () {
2    String? nullableVariable = null;
3    print (nullableVariable?.length);                   // This will not throw a runtime error
4   }
```

In the above code, we're using the '?.' null aware operator to access the **length** property on **nullableVariable**. If **nullableVariable** is null, the expression will evaluate to null and will not throw a runtime error.

Why Null Safety in Dart ?:

As developers, we often encounter null reference errors in our code. These errors occur when we try to access properties or methods on a null value. Dart's null safety helps us avoid these errors and write more robust and predictable code.

The Need for Null Safety in Dart:

In most existing Dart codes, variables can be either null or non-null. This can lead to null reference errors, which are a common type of runtime error. These errors can be hard to debug and can lead to unexpected behaviour in our code.

```
1   void main () {
2    String? name;
3    print (name.length);                    // This will throw a runtime error
4   }
```

In the preceding code, we are attempting to access the name's length property, which is null. This will result in a runtime error.

How Null Safety Improves Dart Code :

Dart Null Safety improves our Dart code in several ways:

- Elimination of Null Reference Errors :

  Null safety ensures that variables cannot be null unless explicitly declared with a ? or late modifier.

  Example : String? nullableString = "Hello";

String nonNullableString = nullableString! ; // This usage requires explicit null check or assertion

- Improved Code Clarity :

  Null safety encourages explicit handling of nullable values, making code more self-explanatory.

Example : String? userName = getUserFromApi();

```
if (userName != null) {
  print("Welcome, $userName!");
} else {
  print("User not found.");
}
```

- Better Integration with External APIs :

  Null safety facilitates smoother integration with APIs that may return nullable values.

Example : Future<String?> fetchData() async {

```
  // Fetch data from API
}
```

// Usage

String data = await fetchData() ?? "Default Data";

- Non-nullable by Default :

  New variables are non-nullable by default, reducing the chance of null-related bugs.

Example : String nonNullableString = "Flutter is awesome!" ;

- Null Safety and Widgets :

  Null safety plays a significant role in Flutter widgets, ensuring that UI elements are built with non-null values.

Example : Text(getUserName() ?? "Guest"); // Providing a default value for a potentially null username

- Migration and Analysis Tools:

  Utilize tools like Dart's migration tool and static analysis to easily migrate existing codebases to null safety and catch potential issues.

Example : // Run Dart migration tool

dart migrate

- Sound Null Safety :

  Null safety in Dart is "sound," meaning it's enforced throughout the type system, providing stronger guarantees.

Example : int? nullableInt = 42; // This will result in a compilation error

int nonNullableInt = nullableInt;

- **Makes Code More Readable**: With Dart Null Safety, we can see at a glance whether a variable can be null or not. This makes our code more readable and easier to understand.

- **Improves Performance**:

  Dart Null Safety can also improve the performance of our Flutter app. By catching null errors at compile time, we can avoid unnecessary runtime checks for null values.

Best Practices for Null Safety in Flutter

Now that we have a basic understanding of null safety in Dart, let's explore some best practices for ensuring null safety in your Flutter applications.

- **Always Initialize Non-Nullable Variables**

  When declaring a non-nullable variable, always provide an initial value or make sure it's initialized before being used. This practice ensures that non-nullable variables never hold a null value, preventing null-related errors.

```
int nonNullableInt = 0; // Initialized with a value
String? nullableString; // Nullable, so it can be null
```

- **Use Late Variables Wisely**

  In some cases, you may need to declare a non-nullable variable without initializing it right away. The `late` keyword allows you to do this, but use it with caution, as it might lead to runtime errors if not initialized before being accessed.

```
late int lateInitializedInt;
void initializeLateInt() {
  lateInitializedInt = 10;
}
```

- **Leverage Null-Aware Operators**

Dart provides several null-aware operators that allow you to work with nullable variables safely. These operators help you avoid null reference exceptions in your code.

```
int? nullableInt;
int nonNullableInt = nullableInt ?? 0; // Using the null-aware ?? operator
```

- **Handle Nullable Callbacks and Functions**

When working with callbacks and functions that can be null, ensure you check for null before invoking them. You can use the null-aware `?.` operator for this purpose.

```
typedef VoidCallback = void Function();

class MyButton extends StatelessWidget {
  final VoidCallback? onPressed;
  MyButton({required this.onPressed});
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed != null ? () => onPressed!() : null,
      child: Text('Press me'),
    );
  }
}
```

- **Embrace the Power of Null Safety in Libraries and Packages**

When selecting libraries and packages for your Flutter project, ensure they support null safety. This practice helps maintain the null safety guarantees throughout your application.