TIPS to WRITE CLEAN CODE in FLUTTER

Obaid Ullah
Expert Flutter Developer

SWIPE

USE MEANINGFUL AND DESCRIPTIVE NAMES

Use meaningful and descriptive names for your variables, functions, and classes.
Avoid abbreviations and single-letter variable names.

```
// Bad example
String s;    // What does s stand for?

// Good example
String username;   // Descriptive and meaningful name
```

WRITE SHORT AND CONCISE FUNCTIONS

Write short and concise functions that do one thing and do it well. Aim for a maximum of 20 lines of code per function.

```
// Bad example
void validateAndSaveForm() {
// A long function that does too many things
// spanning multiple pages or screens
}

// Good example
void saveForm() {
// A short and focused function that does one thing
// such as saving a form data to a database or API
}
```

AVOID REDUNDANT CODE

Avoid redundant code by using inheritance, composition, and other object-oriented design patterns.

```
// Bad example
void fetchUserData() async {
  // Fetch user data from the server
  // ...
  // Fetch user data again to update the UI
  // ...
}

// Good example
void fetchUserData() async {
  // Fetch user data from the server
  // ...
}

void updateUI() {
  // Update the UI with the latest user data
  // ...
}
```

COMMENT SPARINGLY AND EFFECTIVELY

Comment your code sparingly and only when necessary. Your code should be self-explanatory, and comments should add value by providing context or explaining complex logic.

```
// Bad example
// This is a function that saves a user's profile
// data to the server using an HTTP POST request
void saveUserData() {
  // Code that saves the user's data
  // ...
}


// Good example
void saveUserData() {
  // Saves the user's profile data to the server
  // using an HTTP POST request
  // ...
}
```

FORMAT YOUR CODE CONSISTENTLY

Format your code consistently and use whitespace to make it more readable. Use a linter like Dartfmt to enforce consistent formatting across your project.

```
// Bad example
void fetchUserData() async {
var response = await
http.get(Uri.parse('https://example.com/userdata'));
if(response.statusCode == 200) {
 print('Success');
} else {
  print('Failed');
}
}

// Good example
void fetchUserData() async {
 var response = await
http.get(Uri.parse('https://example.com/userdata'));

 if (response.statusCode == 200) {
  print('Success');
 } else {
  print('Failed');
 }
}
```

HANDLE ERRORS GRACEFULLY

Handle errors gracefully and provide meaningful error messages. Use try-catch blocks to catch and handle exceptions.

```
// Bad example
void fetchData() async {
  try {
    // Code that fetches data from the server
    // ...
  } catch (e) {
    // Do nothing
  }
}


// Good example
void fetchData() async {
  try {
    // Code that fetches data from the server
    // ...
  } catch (e) {
    // Handle the error gracefully
    print('Error fetching data: $e');
  }
}
```

USE CONSTANTS AND ENUMS

Use constants and enums instead of hardcoding values in your code. This makes your code more flexible and easier to maintain.

```
// Bad example
void setColor(String color) {
  if (color == 'red') {
    // Do something
  } else if (color == 'blue') {
    // Do something else
  }
}


// Good example
enum Color { red, blue }

void setColor(Color color) {
  if (color == Color.red) {
    // Do something
  } else if (color == Color.blue) {
    // Do something else
  }
}
```

KEEP YOUR CODE MODULAR AND ORGANIZED

Keep your code modular and organized. Use packages and libraries to separate your code into logical components.

```
// Bad example
void validateEmail(String email) {
  // Code that validates the email
  // ...
}

void validatePassword(String password) {
  // Code that validates the password
  // ...
}

void validateForm(String email, String password) {
  validateEmail(email);
  validatePassword(password);
}
```

```
/ Good example
class EmailValidator {
 static void validate(String email) {
 // Code that validates the email
 // ...
 }
}

class PasswordValidator {
 static void validate(String password) {
 // Code that validates the password
 // ...
 }
}

class FormValidator {
 static void validate(String email, String password) {
 EmailValidator.validate(email);
 PasswordValidator.validate(password);
 }
}
```

WRITE UNIT TESTS

Write unit tests for your code to ensure that it works as expected. Use a testing framework like Flutter Test to automate your tests.

```
// Bad example
void calculateTotalPrice(int price, int quantity) {
  // Code that calculates the total price
  // ...
}


// Good example
int calculateTotalPrice(int price, int quantity) {
  return price * quantity;
}

void main() {
  test('calculateTotalPrice', () {
    expect(calculateTotalPrice(10, 2), equals(20));
    expect(calculateTotalPrice(5, 3), equals(15));
  });
}
```

CONTINUOUSLY REFACTOR YOUR CODE

Continuously refactor your code to keep it clean and maintainable. Refactoring involves improving the design of existing code without changing its behavior.

```
// Bad example
void submitOrder() async {
  // Code that submits the order to the server
  // ...
}

// Later, the requirements change to include sending a
confirmation email
void submitOrder() async {
  // Code that submits the order to the server
  // ...
  sendConfirmationEmail();
}
```

```
// Good example
Future<void> submitOrder() async {
 // Code that submits the order to the server
 // ...
}

Future<void> sendConfirmationEmail() async {
 // Code that sends the confirmation email
 // ...
}

void main() {
 test('submitOrder', () async {
 // Test the original functionality
 // ...

 // Test the new functionality
 await submitOrder();
 await sendConfirmationEmail();
 });
}
```

These tips should help you write clean and maintainable code in Flutter. Good luck, and happy coding!

Obaid Ullah
Expert Flutter Developer

SWIPE

If you want

FLUTTER

APP DEVELOPMENT

to execute your ideas into real-time mobile apps

DM ME for a Free Consultation Call

OBAID ULLAH

SAVE
FOR LATER