Null Safety

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.

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.

```
void main() {
 String message = 'Hello';
 print(message.length);
}
```

The program declares a string variable called message, initializes its value to 'Hello', and displays the length property.

Before Dart 2.12, you can assign null to the message variable and access the length property. However, it’ll cause a runtime error:

```
void main() {
 String messge = null;
 print(message.length); // runtime error
}
```

With null safety, types in the code are non-nullable by default. If you attempt to assign null to a variable, the code editor will issue an error. In other words, null safety turns the runtime errors into edit-time errors.

To specify that a variable can be null, you add a question mark (?) to the type in variable declaration. For example:

```
void main() {
 String? message = 'Hello';
 print(message);
 message = null; // OK
}
```

```
}
```

Nullable types

A nullable type contains null in addition to its own values of the type.

- int? – a nullable integer such as 1, 2, and null.
- double? – a nullable double such as 3.14, 2.5, and null.
- bool? – nullable boolean such as true, false, and null.
- String? – a nullable string such as ‘Hello’, ‘Bye’, and null.
- Point? a nullable user-defined class Point. For example, point(10,20) and null.

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

Before null safety, it’s easy to forget to add the code that holds null. However, with null safety, Dart makes it impossible to forget. Because you really cannot do much with null unless you deal with the null possibility.

For example, Dart will not allow you to run the following code:

```
void main() {
 String message = 'Hello';
 print(message.length);
}
```

It issues a compile-time error

Type promotion

Type promotion allows you to assign a value to a nullable variable without requiring anythig else.

```
void main() {
 String? message ;
 message = 'Hello';    //type promotion
 print(message.length);
}
```

,

In this example, the variable message is a nullable string type. However, Dart can see that the message is not null because we assign a value to it before accessing the length property.
Therefore, Dart implicitly promotes the type of the message variable from String? to String automatically.

Flow analysis

Besides type promotion, Dart uses a sophisticated flow analysis to check every possible case the code would take. And if none of these cases come up with the possibility of being null, it promotes the variable to a non-nullable type using type promotion.

```
bool isEven(int? x) {
 if (x == null) {
 return false; }
 return x.isEven;
 }
```

In this example, once the last line of the function is reached, the parameter x cannot be null. Therefore, Dart promotes the x to int instead of using the int? type.

Dart Null-aware Operators

To deal with null values, Dart uses flow analysis and type promotion. In addition, it provides you with various null-aware operators.

- ?? if-null operator
- ??= null-aware operaor
- ?. null-aware access & method invocation operator
- ! null assertion operator
- ?.. null-aware cascade operaor
- ?[ ] null-aware index operaor
- ...? null-aware spread operaor

The following example uses an if statement to check if a variable is null and assign an error message if it is:

```
void main() {
 String? input;
 String message;
 if (input != null) {
 message = input;
```

```
} else {
 message = 'Error';
}
 print(message);
}
```

with if-null operator (??)

```
void main() {
 String? input;
 String message = input ?? 'Error';
 print(message);
 }    //Output: Error
```

The null-aware assignment operator (??=)
Sometimes, you have a single nullable variable:

```
 String? input;
```

And you want to assign it a value if it is null like this:

```
 input = input ?? 'Error';
```

This code works well. However, it is quite redundant to reference the input variable on both sides of the expression.

To avoid this, you can use the null-aware assignment operator:

```
input ??= 'Error';
```

```
void main() {
 String? input;
 if (input != null) {
 print(input.length);
 print(input.toLowerCase());
 }
}
```

with null-aware access operator (?.)

```
void main() {
 String? input;
print(input?.length); // null
print(input?.toLowerCase()); // null
}
```

The following code causes a compile-time error:

```
 bool? isTextFile(String? filename) {
if (filename != null) {
return filename.endsWith('.txt') ? true : false;
}
 return null;
}

void main() {
 bool result = isTextFile('readme.txt');
 print(result);
 }
```

with null-assertion operator

```
bool? isTextFile(String? filename) {
 if (filename != null) {
 return filename.endsWith('.txt') ? true : false;
}
 return null;
}
void main(){...}
```