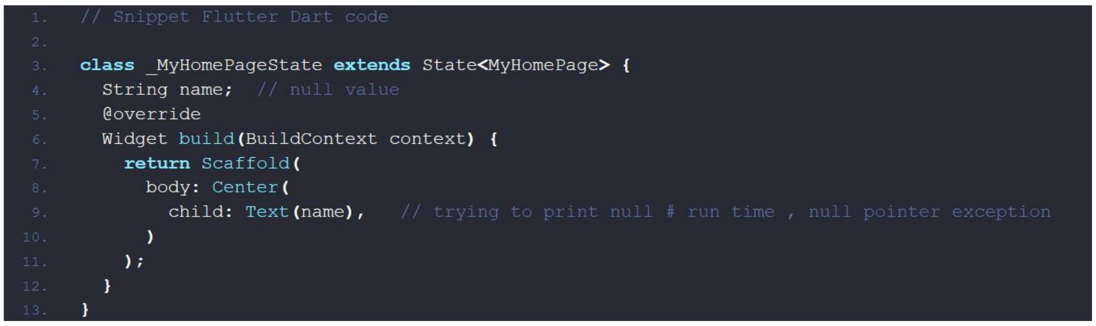
What is null safety in dart?

A Flutter Null Safety feature is a secured way of developing a flutter application, In any programming language, a variable can have a value assigned to it or might be **null**.

because in dart variable can be **null** & if a variable is null we cannot perform an action with it like, printing the **null value** on user screen or we cannot execute at string method to a **null value** & if we do so this may lead to **null pointer exception** & your flutter application will crash in run-time.

Example without null safety flutter

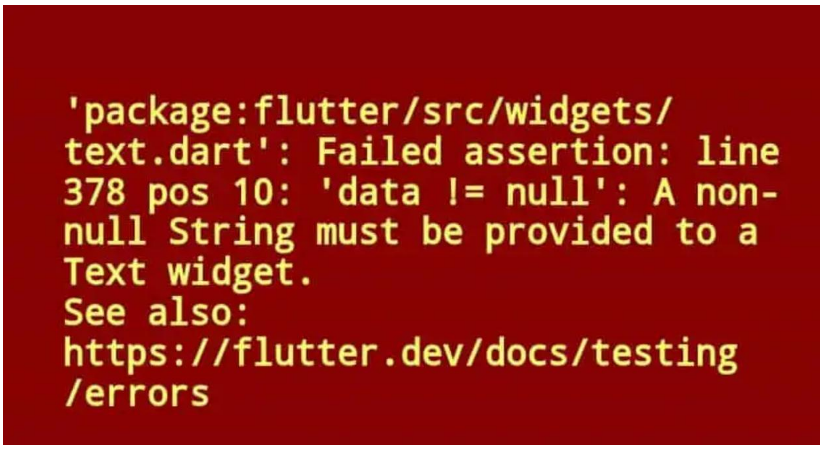
#code



```
// Snippet Flutter Dart code

class _MyHomePageState extends State<MyHomePage> {
  String name;  // null value
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Text(name),   // trying to print null # run time , null pointer exception
      )
    );
  }
}
```

Output example

Here you can see that when i tried to print null in text widget it showing Failed assertion – A non-null string must be provided.



```
'package:flutter/src/widgets/
text.dart': Failed assertion: line
378 pos 10: 'data != null': A non-
null String must be provided to a
Text widget.
See also:
https://flutter.dev/docs/testing
/errors
```

Flutter null safety enable

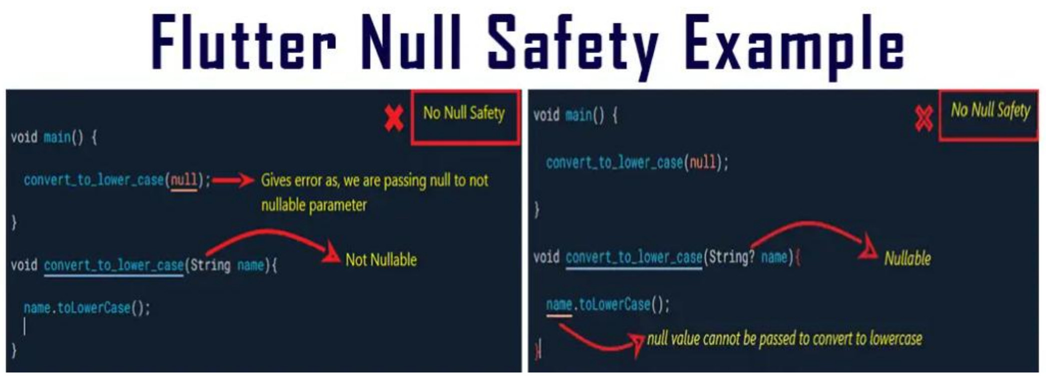
To solve null pointer exception flutter 2 came with null safety feature.

So now **all of our datatype are by default “Not Nullable”**

Not nullable means they requires some value to be assigned to them. A String, Int, or any date type must have some value.

Flutter null Safety Examples

1. **Null Safety Example 1**

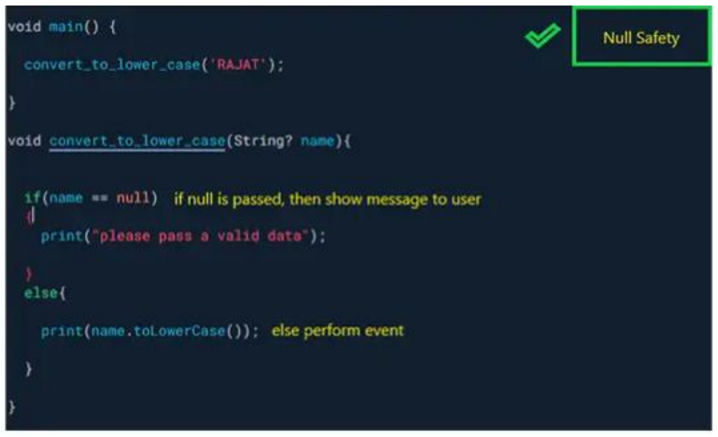


Flutter Null Safety Example
No Null Safety
Gives error as, we are passing null to not nullable parameter
Not Nullable
No Null Safety
Nullable
null value cannot be passed to convert to lowercase

```
void main() {

  convert_to_lower_case(null);

}

void convert_to_lower_case(String name){

  name.toLowerCase();

}
```

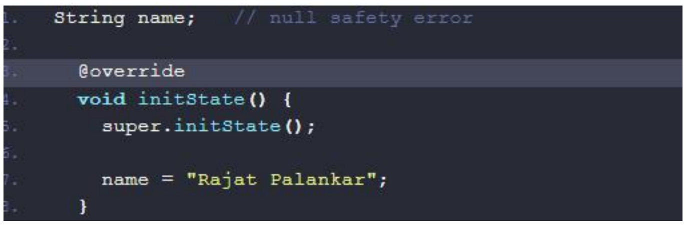
```
void main() {

  convert_to_lower_case(null);

}

void convert_to_lower_case(String? name){

  name.toLowerCase();

}
```

Solution:



```
void main() {

  convert_to_lower_case('RAJAT');

}

void convert_to_lower_case(String? name){

  if(name == null)
  {
    print("please pass a valid data");
  }
  else{
    print(name.toLowerCase());
  }
}
```

Late Modifier to achieve null safety



```
  String name;    // null safety error

    @override
    void initState() {
      super.initState();

      name = "Rajat Palankar";
    }
```

The above code will give you a null safety compilation error because dart is not clever enough to understand that you have initialized the value to the variable afterwords.

Therefore, to future initialization or late initialization value to variable you can make use of **“late modifier”**, that tells compiler that value will be initialized later on.

what is late modifier?

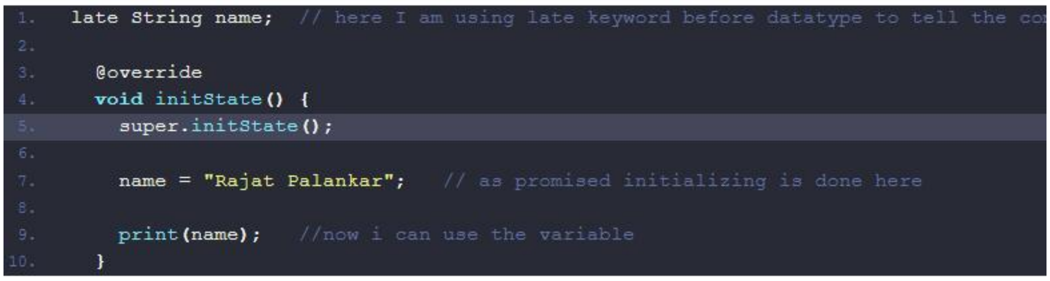
In Flutter **late keyword** will lets you use non-nullable datatype. Now i dart 2.12 has add late modifier, late keyword is used in two case:

- While migrating flutter project to **null safety**.
- **Lazy** initializing of a variable.

Late keyword is used just to promise dart compiler that you are going to intialize the value to variable later on in future.

Note: if you don’t full fill as promised using late modifier then your flutte app will crash at runtime.

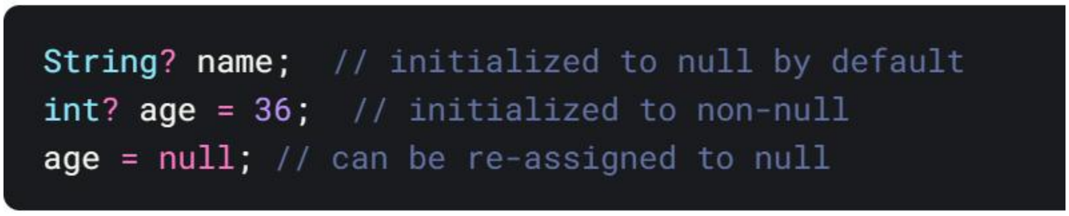
Late modifier example



```
late String name;  // here I am using late keyword before datatype to tell the co

  @override
  void initState() {
    super.initState();

    name = "Rajat Palankar";    // as promised initializing is done here

    print(name);   //now i can use the variable
  }
```

Declaring Nullable Variables

The ? symbol is what we need:



```
String? name;  // initialized to null by default
int? age = 36;  // initialized to non-null
age = null; // can be re-assigned to null
```

The assertion operator

We can use the assertion operator ! to assign a nullable expression to a non-nullable variable:

```dart
int? maybeValue = 42;
int value = maybeValue!; // valid, value is non-nullable
```

By doing this, we're **telling** Dart that maybeValue is not null, and it's safe to assign it to a non-nullable variable.

Note that applying the assertion operator to a null value will throw a runtime exception:

```dart
String? name;
print(name!); // NoSuchMethodError: '<Unexpected Null Value>'
print(null!); // NoSuchMethodError: '<Unexpected Null Value>'
```

Sometimes we need to work with APIs that return nullable values. Let's revisit the lastName function:

```dart
String? lastName(String fullName) {
  final components = fullName.split(' ');
  return components.length > 1 ? components.last : null;
}
```

Flow Analysis: Promotion

Dart can make your life easier by taking into account null checks on nullable variables:

```dart
int absoluteValue(int? value) {
  if (value == null) {
    return 0;
  }
  // if we reach this point, value is non-null
  return value.abs();
}
```

Here we use an if statement to return early if the value argument is null.

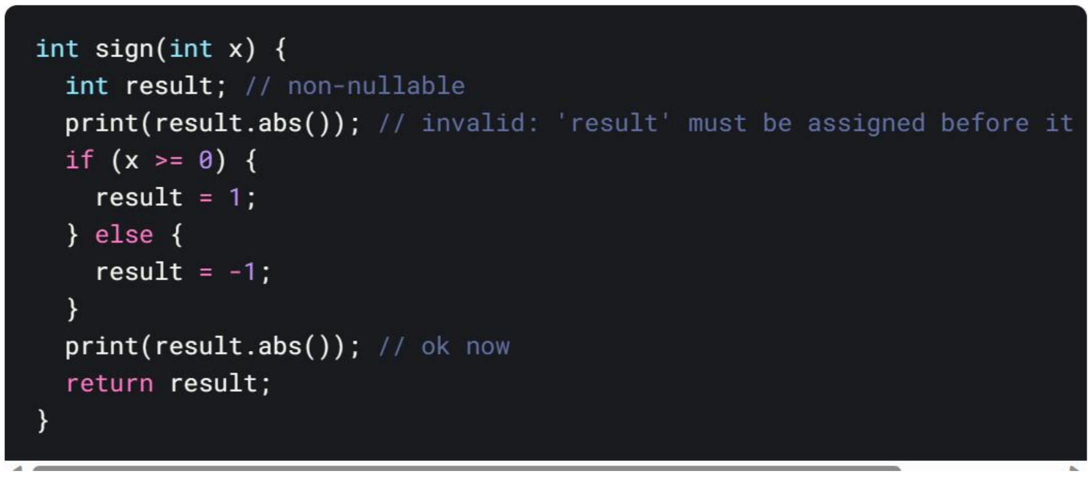
Beyond that point, value cannot be null and is treated (or **promoted**) to a non-nullable value. Hence we can safely use value.abs() rather than value?.abs() (with the null-aware operator).

Similarly, we could throw an exception if the value is null:

Flow Analysis: Definite Assignment

Dart knows where variables are **assigned** and where they're **read**.

This example shows how to initialize a non-nullable variable **after** checking for a condition:



```
int sign(int x) {
  int result; // non-nullable
  print(result.abs()); // invalid: 'result' must be assigned before it
  if (x >= 0) {
    result = 1;
  } else {
    result = -1;
  }
  print(result.abs()); // ok now
  return result;
}
```

As long as a non-nullable variable is given a value **before** it's used, Dart is happy.

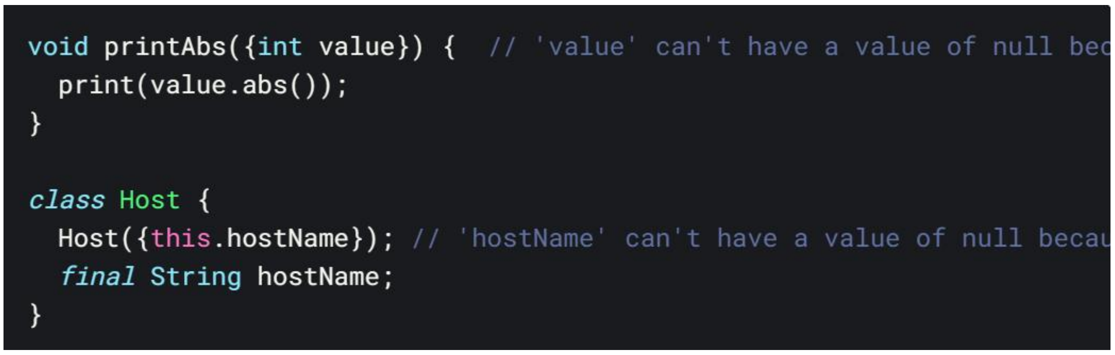
Using non-nullable variables with classes

Instance variables in classes must be initialized if they are non-nullable:

Non-nullable named and positional arguments

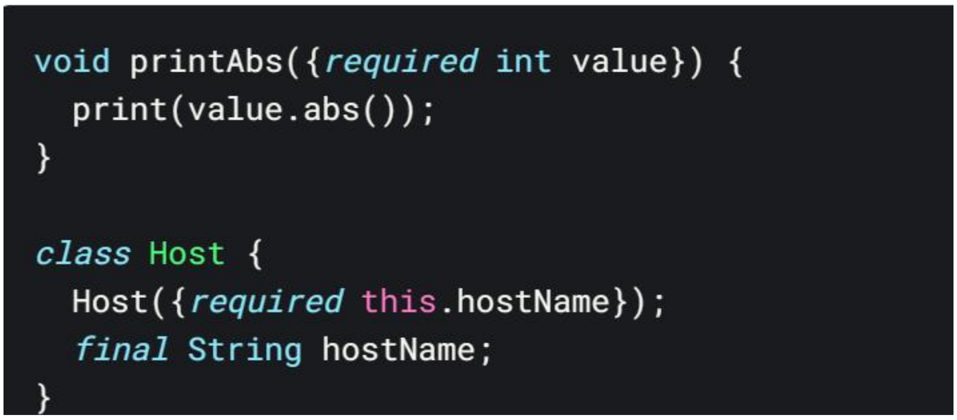
With Null Safety, non-nullable **named** arguments must always be **required** or have a **default value**.

This applies to regular methods as well as class constructors:



```
void printAbs({int value}) {  // 'value' can't have a value of null bec
  print(value.abs());
}

class Host {
  Host({this.hostName}); // 'hostName' can't have a value of null becau
  final String hostName;
}
```

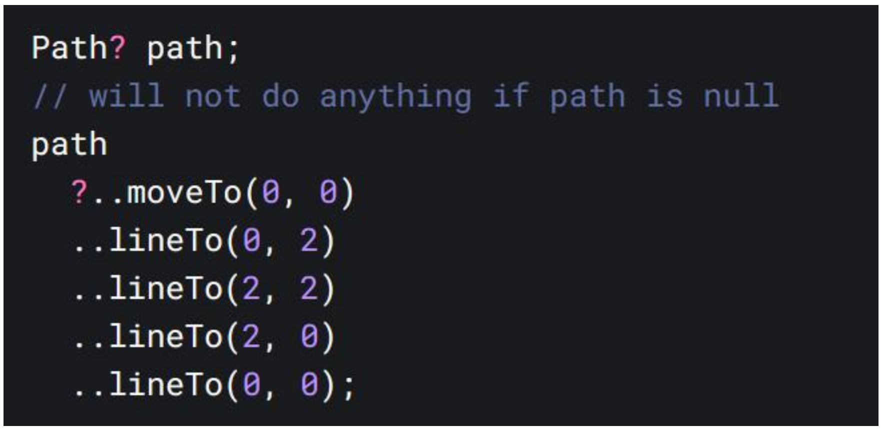
We can fix the code above with the new required **modifier**, which replaces the old @required **annotation**:



```dart
void printAbs({required int value}) {
  print(value.abs());
}

class Host {
  Host({required this.hostName});
  final String hostName;
}
```

Null-aware cascade operator

To deal with Null Safety, the cascade operator now gains a new null-aware variant: ?... Example:



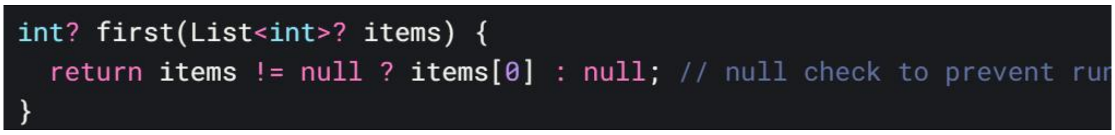
```dart
Path? path;
// will not do anything if path is null
path
  ?..moveTo(0, 0)
  ..lineTo(0, 2)
  ..lineTo(2, 2)
  ..lineTo(2, 0)
  ..lineTo(0, 0);
```

The cascade operations above will only be executed if path is not null.

The null-aware cascade operator can **short-circuit**, so only one ?.. operator is needed at the beginning of the sequence.

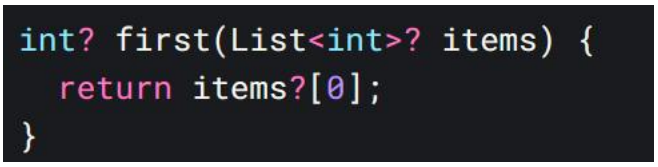
Null-aware subscript operator

Up until now, checking if a collection was null before using the subscript operator was verbose:



```dart
int? first(List<int>? items) {
  return items != null ? items[0] : null; // null check to prevent ru
}
```

Dart 2.9 introduces the null aware operator ?[], which makes this a lot easier:



```dart
int? first(List<int>? items) {
  return items?[0];
}
```