```
// variables by default are non-nullable -> cannot be null / must have a value
int x = 8;

// ? after a variable means it CAN be null
int? y;

// ?? provides a fallback value in case the variable is null

// (case 1)
String? nameFromDatabase; // null
String nameInApp = nameFromDatabase ?? "No name";
void main(){
  print(nameInApp); // No name
}

// (case 2)
String? nameFromDatabase = "Yousef"; // valued
= nameFromDatabase ?? "No name";
void main(){
  print(nameInApp); // Yousef
}
```

```
/*

!  after a variable means you are CERTAIN this variable is NOT null
?. null aware operator -> used to access a property/method of an object

- if the object before ?. is not null, it will return like normal
- if the object before ?. is null, it will just return null

Without the null ?. operator, you will get errors for using properties/method on null values
Simply, in other words...

?. help you gracefully handle null values without your app crashing.


BEFORE NULL SAFETY:
if (nameFormDatabase != null){
  do safe code since we manually checked that it's not null
}
```

By Antony Hany

```
WHEN TO USE ?. VS !
__________________________________________________________________________________________
Advanteges of ?.

- Saftey: Using ?. is safe when dealing with nullable objects.
          If the object is null, the expression will gracefully
          return null without throwing an error

- Cleaner Code: It can simply conditional checks
                Instead of using longer conditions

                if (student != null){
                  return student.name;
                } else {
                  return null;
                }

                you can just say:
                student?.name;
// Example
void main(){
  print(nameFromDatabase?.length); // 100% it is not goiong to be null
}


__________________________________________________________________________________________
Advanteges of !

- Explicitness: By using ! after a variable, you're explicitly stating that you
                expect the vale to be non-null
                If it does end up being null, the code will throw an ERROR,
                which can actually make debugging straight-forward since the error
                will point directly to the line with !
__________________________________________________________________________________________

PRACTICAL EXAMPLE..

- Imagine a school where students take an exam.
- the exam is out of 15
- At the end of the year, the school wants to print out the marks of every student.
- However, not all students took the exam

*/
class Student {
  String name; // every student has a name
  int? score; // score can be null because the student was absent
```

By Antony Hany

```
  Student({required this.name, this.score});

}

String scoreAsPercentage(int? score){
  int totalMarks = 15;
  double percentage = (score?.toDouble() ?? 0) * 100 / totalMarks;

  return score == null ? 'Absent' : percentage.toStringAsFixed(0) + "%";
}
void main(){
  print(nameFromDatabase?.length); // 100% it is not goiong to be null
  Set<Student> students ={
    Student(name: "hamdy", score: 2),
    Student(name: "Henery", score: 6),
    Student(name: "sara"), // absent
    Student(name: "ahmend", score: 5),
    Student(name: "quiet kid", score: 15),
    Student(name: "populat kid", score: 20),
  };

  // print student marks
  for (var student in students ){
    print("${student.name}'s marks: ${scoreAsPercentage(student.score)} ");
  }


}
/* output
null
hamdy's marks: 13%
Henery's marks: 40%
sara's marks: Absent
ahmend's marks: 33%
quiet kid's marks: 100%
populat kid's marks: 133%
*/
```

By Antony Hany