Tutorial: Express your design intent more clearly with nullable and non-nullable reference types

Article • 11/03/2022

Nullable reference types complement reference types the same way nullable value types complement value types. You declare a variable to be a **nullable reference type** by appending a `?` to the type. For example, `string?` represents a nullable `string`. You can use these new types to more clearly express your design intent: some variables *must always have a value*, others *may be missing a value*.

In this tutorial, you'll learn how to:

- ✓ Incorporate nullable and non-nullable reference types into your designs
- ✓ Enable nullable reference type checks throughout your code.
- ✓ Write code where the compiler enforces those design decisions.
- ✓ Use the nullable reference feature in your own designs

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Incorporate nullable reference types into your designs

In this tutorial, you'll build a library that models running a survey. The code uses both nullable reference types and non-nullable reference types to represent the real-world concepts. The survey questions can never be null. A respondent might prefer not to answer a question. The responses might be `null` in this case.

The code you'll write for this sample expresses that intent, and the compiler enforces that intent.

Create the application and enable nullable reference types

Create a new console application either in Visual Studio or from the command line using `dotnet new console`. Name the application `NullableIntroduction`. Once you've created the application, you'll need to specify that the entire project compiles in an enabled **nullable annotation context**. Open the *.csproj* file and add a `Nullable` element to the `PropertyGroup` element. Set its value to `enable`. You must opt in to the **nullable reference types** feature in projects earlier than C# 11. That's because once the feature is turned on, existing reference variable declarations become **non-nullable reference types**. While that decision will help find issues where existing code may not have proper null-checks, it may not accurately reflect your original design intent:

```XML
<Nullable>enable</Nullable>
```

Prior to .NET 6, new projects do not include the `Nullable` element. Beginning with .NET 6, new projects include the `<Nullable>enable</Nullable>` element in the project file.

Design the types for the application

This survey application requires creating a number of classes:

- A class that models the list of questions.
- A class that models a list of people contacted for the survey.
- A class that models the answers from a person that took the survey.

These types will make use of both nullable and non-nullable reference types to express which members are required and which members are optional. Nullable reference types communicate that design intent clearly:

- The questions that are part of the survey can never be null: It makes no sense to ask an empty question.
- The respondents can never be null. You'll want to track people you contacted, even respondents that declined to participate.
- Any response to a question may be null. Respondents can decline to answer some or all questions.

If you've programmed in C#, you may be so accustomed to reference types that allow `null` values that you may have missed other opportunities to declare non-nullable

instances:

- The collection of questions should be non-nullable.
- The collection of respondents should be non-nullable.

As you write the code, you'll see that a non-nullable reference type as the default for references avoids common mistakes that could lead to NullReferenceExceptions. One lesson from this tutorial is that you made decisions about which variables could or could not be `null`. The language didn't provide syntax to express those decisions. Now it does.

The app you'll build does the following steps:

1. Creates a survey and adds questions to it.
2. Creates a pseudo-random set of respondents for the survey.
3. Contacts respondents until the completed survey size reaches the goal number.
4. Writes out important statistics on the survey responses.

Build the survey with nullable and non-nullable reference types

The first code you'll write creates the survey. You'll write classes to model a survey question and a survey run. Your survey has three types of questions, distinguished by the format of the answer: Yes/No answers, number answers, and text answers. Create a `public SurveyQuestion` class:

C#

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

The compiler interprets every reference type variable declaration as a **non-nullable** reference type for code in an enabled nullable annotation context. You can see your first warning by adding properties for the question text and the type of question, as shown in the following code:

C#

```
namespace NullableIntroduction
{
```

```
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Because you haven't initialized `QuestionText`, the compiler issues a warning that a non-nullable property hasn't been initialized. Your design requires the question text to be non-null, so you add a constructor to initialize it and the `QuestionType` value as well. The finished class definition looks like the following code:

C#

```
namespace NullableIntroduction;

public enum QuestionType
{
    YesNo,
    Number,
    Text
}

public class SurveyQuestion
{
    public string QuestionText { get; }
    public QuestionType TypeOfQuestion { get; }

    public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
        (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
}
```

Adding the constructor removes the warning. The constructor argument is also a non-nullable reference type, so the compiler doesn't issue any warnings.

Next, create a `public` class named `SurveyRun`. This class contains a list of `SurveyQuestion` objects and methods to add questions to the survey, as shown in the following code:

C#

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new
List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) =>
surveyQuestions.Add(surveyQuestion);
    }
}
```

As before, you must initialize the list object to a non-null value or the compiler issues a warning. There are no null checks in the second overload of `AddQuestion` because they aren't needed: You've declared that variable to be non-nullable. Its value can't be `null`.

Switch to *Program.cs* in your editor and replace the contents of `Main` with the following lines of code:

C#

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a
NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many
times (to the nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Because the entire project is in an enabled nullable annotation context, you'll get warnings when you pass `null` to any method expecting a non-nullable reference type. Try it by adding the following line to `Main`:

C#

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Create respondents and get answers to the survey

Next, write the code that generates answers to the survey. This process involves several small tasks:

1. Build a method that generates respondent objects. These represent people asked to fill out the survey.
2. Build logic to simulate asking the questions to a respondent and collecting answers or noting that a respondent didn't answer.
3. Repeat until enough respondents have answered the survey.

You'll need a class to represent a survey response, so add that now. Enable nullable support. Add an `Id` property and a constructor that initializes it, as shown in the following code:

C#

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Next, add a `static` method to create new participants by generating a random ID:

C#

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

The main responsibility of this class is to generate the responses for a participant to the questions in the survey. This responsibility has a few steps:

1. Ask for participation in the survey. If the person doesn't consent, return a missing (or null) response.
2. Ask each question and record the answer. Each answer may also be missing (or null).

Add the following code to your `SurveyResponse` class:

C#

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}
```

The storage for the survey answers is a `Dictionary<int, string>?`, indicating that it may be null. You're using the new language feature to declare your design intent, both to the compiler and to anyone reading your code later. If you ever dereference

`surveyResponses` without checking for the `null` value first, you'll get a compiler warning. You don't get a warning in the `AnswerSurvey` method because the compiler can determine the `surveyResponses` variable was set to a non-null value above.

Using `null` for missing answers highlights a key point for working with nullable reference types: your goal isn't to remove all `null` values from your program. Rather, your goal is to ensure that the code you write expresses the intent of your design. Missing values are a necessary concept to express in your code. The `null` value is a clear way to express those missing values. Trying to remove all `null` values only leads to defining some other way to express those missing values without `null`.

Next, you need to write the `PerformSurvey` method in the `SurveyRun` class. Add the following code in the `SurveyRun` class:

C#

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Here again, your choice of a nullable `List<SurveyResponse>?` indicates the response may be null. That indicates the survey hasn't been given to any respondents yet. Notice that respondents are added until enough have consented.

The last step to run the survey is to add a call to perform the survey at the end of the `Main` method:

C#

```
surveyRun.PerformSurvey(50);
```

Examine survey responses

The last step is to display survey results. You'll add code to many of the classes you've written. This code demonstrates the value of distinguishing nullable and non-nullable reference types. Start by adding the following two expression-bodied members to the `SurveyResponse` class:

C#

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index)
?? "No answer";
```

Because `surveyResponses` is a nullable reference type, null checks are necessary before de-referencing it. The `Answer` method returns a non-nullable string, so we have to cover the case of a missing answer by using the null-coalescing operator.

Next, add these three expression-bodied members to the `SurveyRun` class:

C#

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ??
Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

The `AllParticipants` member must take into account that the `respondents` variable might be null, but the return value can't be null. If you change that expression by removing the `??` and the empty sequence that follows, the compiler warns you the method might return `null` and its return signature returns a non-nullable type.

Finally, add the following loop at the bottom of the `Main` method:

C#

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} :
{answer}");
        }
    }
    else
    {
```

```
            Console.WriteLine("\tNo responses");
        }
    }
```

You don't need any `null` checks in this code because you've designed the underlying interfaces so that they all return non-nullable reference types.

Get the code

You can get the code for the finished tutorial from our samples repository in the csharp/NullableIntroduction folder.

Experiment by changing the type declarations between nullable and non-nullable reference types. See how that generates different warnings to ensure you don't accidentally dereference a `null`.

Next steps

Learn how to use nullable reference type when using Entity Framework:

Entity Framework Core Fundamentals: Working with Nullable Reference Types