100 C# INTERVIEW QUESTIONS

By Kristijan Kralj

Table of Contents

Introduction

Welcome, dear colleague developer. I'll keep this introduction short since no one reads this anyway.

The purpose of this eBook is to collect at one place all C# interview questions that might pop up at your next interview.

I have a favor to ask: if you see I made a mistake anywhere in this ebook or would like to suggest a question, please let me know at kristijan@blueinvader.com.

I'll be happy to correct that in the next version of this book.

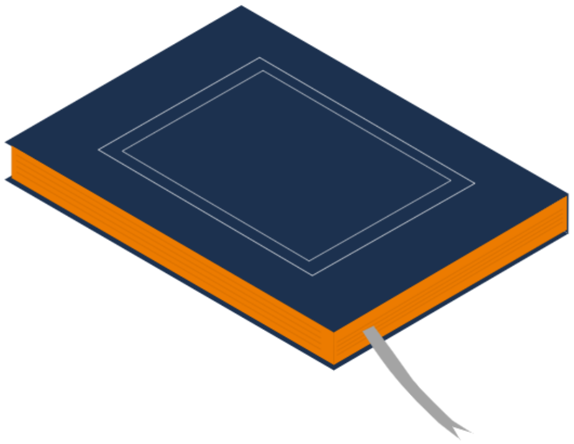
Ok, enough with this introduction! Let's start with the questions.

About the author

Hi,

My name is Kristijan. After more than 10 years of development experience, I've decided to start giving back to the community.

The C# Questions



1. What is a reference type?

Reference type means that a variable that is a reference type has a reference to the address where the value is stored. With reference type, two variables can reference the same object. If you change one variable, you will also change another variable.

The following keywords are used to define reference types: `class`, `interface`, and `delegate`. C# also has built-in reference types: `dynamic`, `object` and `string`.

2. What is the value type?

Value type means that a variable that is a value type directly contains the value. So, if a variable is `int number = 3;`

then 3 in memory is stored directly in the *number* variable. When you pass the value type variable to a method or assign it to another variable, then its value is copied. This means that change to one variable does not affect the value of another variable.

Some examples of value types are `bool`, `int`, `decimal`, `double`, `enum` and `struct`.

3. What is the difference between class and struct?

Classes are reference types, while structs are value types. Classes support inheritance and can be null, and should be used for complex objects. Structs don't support inheritance, they are value types and are typically used to represent small objects.

4. What is an interface?

An interface defines a contract. Any class or struct that implements that contract must provide an implementation of the members defined in the interface. Beginning with C# 8.0, an interface may define a default implementation for members. It may also define static members in order to provide a single implementation for common functionality.

5. What is the difference between an interface and an abstract class?

The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. So, they are pretty similar, and before C# 8.0, the main difference was that interfaces couldn't have a default implementation for methods and properties. Starting with C# 8.0, an interface may define a default implementation for methods and properties.

The other difference is that a class can inherit only one abstract class, while it can implement multiple interfaces.

6. Where do we use async and await in C#?

`async` and `await` are keywords used to create and execute asynchronous code. Asynchronous execution means that the code execution is performed on a thread that is not the main thread. This enables us to have a responsive UI.

`async` keyword is used while declaring an asynchronous method, while `await` is used to call that kind of method.

7. What are generics?

Generic class, struct, method, or interface is used to define a code that doesn't have a concrete type.

Example:

```
// Declare the generic class.
public class GenericClass<T>
{
    public void SomeMethod(T input) { }
}
```

This enables us to write one functionality, but reuse it for multiple types. Instead of using a specific data type, we use a generic type parameter, usually T.

8. Can we use this keyword within a static method?

We can't use `this` in a static method because that keyword returns a reference to the current instance of the class containing it. Static methods do not belong to a particular instance. They exist without creating an instance of the class and are called with the name of a class, not by instance.

9. What is a constructor?

A constructor is a special type of method that is used to create a new instance of an object. The constructor has the same name as the type it creates. When you create a new class/struct, by default you get a default constructor that can be used to create a new instance of that class/struct.

10. How many constructors can a class have?

There is no limit on how many constructors can one class have. The only thing it matters is that the parameters, or the order of parameters, can't be the same between two constructors.

11. What is method overloading?

Method overloading is having several methods with the same name, but they have different signatures (different parameters). Multiple methods can have the same name as long as the number and/or type of parameters are different.

12. What is LINQ?

LINQ stands for Language-Integrated Query. This is a query syntax to retrieve data from different sources and formats. It is already integrated into the C# language, you just need to import the `System.Linq` namespace.

Example of querying list of development languages:

```
List<string> developmentLanguages = new List<string> { "C#", "Java",
"Python", "Go" };

// LINQ Query
var result = from name in developmentLanguages
                    where name.Contains('a')
                    select name;

foreach (var name in result)
    Console.Write(name + " "); // “Java” is the output
```

The alternative way to perform LINQ queries is through method chaining:

```
// the same query as above
var result = developmentLangugages.Where(name => name.Contains('a'));
```

As you can see, this is a very concise way of filtering data.

13. What is the difference between ref & out parameters?

An argument passed as `ref` must be initialized before passing to a method, while `out` parameter needs not to be initialized before passing to a method.

14. What is the difference between “as” and “is” operators?

The `as` operator is used to cast an instance of an object to a particular type. If it fails, it will return null.

The `is` operator is used to check whether an object can be cast to a particular type. It returns `true` if the object can be cast to a particular type, `false` if it can’t.

15. What is the difference between “throw” and “throw ex” in C#?

When called in the `catch` block, the `throw` statement preserves the original stack trace. `throw` ex will have stack trace only from that catch block. You should try to use the `throw` keyword whenever you can because it provides more error information.

16. Write an example of a try-catch block?

To catch an exception, try using try-catch blocks. The catch block can specify what kind of exception will it catch:

```
try {
      DoSomethingThatMightThrow();
}
catch (Exception ex) {
}
```

17. What are the most commonly used types of exceptions?

The most commonly occurring types of exceptions are:

- NullReferenceException
- StackOverflowException
- ArgumentOutOfRangeException
- OutOfMemoryException
- DivideByZeroException
- IndexOutOfRangeException

18. What are custom exceptions?

In some cases, we want to throw exceptions that are specific to our application domain. In order to do that we can define a custom exception type. That type inherits from some exception type.

19. Why do we use reflection?

You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them.

20. What are extension methods?

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are static methods, but they're called as if they were instance methods on the extended type.

The next example shows how to extend `DateTime` type to provide a new method to it, to check whether or not the day of the week for a date is the weekend:

```
public static class DateTimeExtensions
{
    public static bool IsWeekend(this DateTime dateTime)
    {
        return dateTime.DayOfWeek == DayOfWeek.Saturday ||
                dateTime.DayOfWeek == DayOfWeek.Sunday;
    }
}
```

And the usage:

```
public bool CanScheduleEvent()
{
    bool canSchedule = DateTime.Now.IsWeekend();
    return canSchedule;
}
```

21. What is a virtual method?

A virtual method is a method that can be overridden in the subclasses. A virtual method has an implementation in the base class but the subclass can choose to replace that implementation. We use the `virtual` keyword to indicate that the method can be overridden.

22. What are anonymous types?

Anonymous types allow us to create new types without defining them. This is a way of defining read-only properties in a single object without having to define each type explicitly. The type is generated by the compiler and is accessible only in the current block of code. The type of properties is also inferred by the compiler. We can create anonymous types by using the `new` keyword together with the object initializer.

23. What are nullable value types?

A nullable value type is a type that can have a value type or null. We define nullable value types by adding the "?" to the type: `var int? = null`.

24. What is the var keyword?

`var` is an implicit type that can be used for variables at method scope. An implicitly typed variable has a strong type as any other variable, but this syntactic sugar is used to omit the variable type to make the code shorter.

25. What are nullable reference types?

Nullable reference types are available beginning with C# 8.0, in code that has opted into a *nullable aware context*. Nullable reference types, the null static analysis warnings, and the null-forgiving operator are optional language features. All are turned off by default. A nullable context is controlled at the project level using build settings, or in code using pragmas.

In a nullable aware context:

- A variable of a reference type T must be initialized with non-null, and may never be assigned a value that may be null.

- A variable of a reference type `T?` may be initialized with null or assigned null, but is required to be checked against null before dereferencing.
- A variable `m` of type `T?` is considered to be non-null when you apply the null-forgiving operator, as in `m!`.

Nullable reference types aren't new class types, but rather annotations on existing reference types. The compiler uses those annotations to help you find potential null reference errors in your code.

26. What is boxing?

Boxing is a process of converting value type to a reference type of type `object`.

27. What is unboxing?

Unboxing is a process of explicit conversion of the reference type back to the value type.

28. What is the difference between Continue and Break keyword?

In loops, the `break` keyword is used to finish the loop execution. The `continue` keyword is used to only finish the current iteration and continue program execution with the next iteration.

29. Which class acts as a base class for all classes in .NET?

The Object type is used as a base class for all other classes in .NET. An `object` is used as an alias for System.Object class. To an instance of the `object` class may be assigned values of any other type, whether it’s reference or value type.

30. What are the methods that System.Object provides to all other classes?

Those methods are:

- Equals

- Finalize
- GetHashCode
- ToString

31. Can multiple catch blocks be executed?

There is no way multiple catch blocks with a similar type can't be executed. Only one catch block can be executed at the time.

32. What are the differences between System.String and System.Text.StringBuilder classes?

System.String is immutable. When we adjust the value of the string variable, the current memory will be allocated to the new value, and the previous memory allocation will be removed. System.StringBuilder was developed to provide a notion of a mutable string where a number of operations may be done without specifying a different memory position to the adjusted string.

33. What is the difference between Finalize() and Dispose() methods?

`Dispose()` object is only called when an object is needed to release unmanaged resources. Also, `Finalize()` can also be used for the same reason, but it doesn't assure the garbage collection of an object.

34. Is C# managed or unmanaged code?

C# is definitely a managed code since common language runtime (CLR) can compile C# code to Intermediate language.

35. What is a Console application?

A console application is an application that is run via the command prompt in Windows. It is a common practice for new developers in .Net to start with building console applications.

36. What is a jagged array?

A jagged array is an array that consists of elements of a type array. A jagged array is also called an array of arrays.

37. What is serialization?

The process of converting an instance of a class to a stream of bytes is called serialization. This is commonly used when we want to transport an object across a network but first have to transform an object into a stream of bytes.

38. What are sealed classes?

When it's important to prevent the class from being inherited, we define it as a sealed class. This sealed modifier is what is used to eliminate class derivation. If we try to define a sealed class as a base class, a compile error is displayed.

39. What is the namespace?

A namespace helps in providing an alternative to keep one set of names separated from another. The class names declared in one namespace won't conflict with the same class names declared in another.

40. What is the purpose of an access specifier in C#?

The purpose of the access specifier is to clearly define the scope and visibility of a class member.

41. What is the scope of a public member variable of a C# class?

Public access describes a class to expose its member variables and member functions to other objects. Any member of the public is accessible in and out of the class.

42. What is the scope of a private member variable of a C# class?

The scope of a private member variable of a C# class is to hide its member variables and methods from another method. Only methods of the same class are accessible by its private members. Even an instance of a class is just an instance and can't be accessed by its private members.

43. What is the scope of a protected member variable of a C# class?

Protected access specifies a child class to access the member variables and member functions of its base class.

44. What is the default access modifier for a class?

In C#, the default access modifier is always internal.

45. What is the use of Null Coalescing Operator (??) in C#?

The null coalescing operator is used for nullable value classes and comparison types. It is used to transform an operand to another nullable (or non-nullable) type of operand type, where an implicit conversion is necessary.

If the value of the first operand is empty, the operator returns the value of the second operand; otherwise, it returns the value of the first operand.

46. Can you create a method in C# that can accept a varying number of arguments?

By using the `params` keyword, you can define a method parameter that requires a variable number of arguments or even no argument.

47. Can you pass additional types of parameters after using params in the function definition?

No. Additional parameters are not allowed after the keyword `params`. Only one keyword `params` is allowed in the declaration of the method.

48. What is an enumeration in C#?

An enumerated type is declared using the keyword `enum`.

C# enumerations are a type of data value. In other words, the enumeration contains its very own values and therefore cannot pass inheritance.

49. What is the difference between constants and read-only?

Constant variables are declared and initialized by the compiler. The value at run time will be fixed. Read-only only used when we want to assign the value at run time.

50. Can a private virtual method be overridden?

No, and the reason being that private virtual methods are not accessible outside the class. That being said, it really doesn't make sense to declare a method as private virtual.

51. Describe the accessibility modifier "protected internal".

Protected internal members are available within the same class and the same project, but not outside of the project.

52. What are the circular references?

The circular references happens when multiple references rely on each other and therfore trigger a lock state.

53. What is an object pool in .NET?

The object pool is a container that houses objects ready to be used. It records the total number of items actually in use in the pool, thereby reducing the overhead for making and re-creating objects.

54. What are delegates?

In C#, delegates are the same as method pointers, but the only difference is that they are type-safe, unlike method pointers. Delegates are needed due to their reusable property to write much more generic type-safe functions.

55. How do you inherit a class in C#?

In C#, the colon is used as an inheritance operator in C#. Just place a colon and then the class name will do the trick of inheriting a class into another class.

```
public class DerivedClass : BaseClass
```

56. What are indexers in C# .NET?

Smart arrays are known as indexers in C#. This indexer allows the instances of a class to be indexed exactly like an array. Example of using an indexer:

```
public int this[int index]    // Indexer declaration
```

57. What are C# attributes and their significance?

C# provides developers a way to define declarative tags on certain types, like classes or methods. These are known as attributes. The attribute's information can be retrieved at runtime with the use of reflection.

58. What is a preprocessor directive in C#?

The preprocessor in C# instructs the compiler to process the information before the actual compilation begins.

The main advantage of a preprocessor like #ifdef is generally to make source programs easy to change, compile, and run in different execution environments.

59. Is operator overloading supported in C#?

Most built-in operators usable in C# may be redefined or overloaded.

Overloaded operators are methods with a unique identity or specifier, the keyword operator, preceded by the operator symbol being specified. Similar to every other method, the overloaded operator has a fixed return form and a set of parameters.

60. Are multiple inheritances supported in C#?

No, it's not. You can only inherit from one base class. But a class can implement multiple interfaces.

61. How many ways can you pass parameters to a method?

There are three ways that parameters can be passed to a method:

- Reference parameters – The reference copies directly to the memory location of an argument into the formal parameter which means the argument will be affected by the change.
- Value parameters – What makes this method different is that it only copies the actual value of an argument into the formal parameter of the function. This means the changes made in the parameter inside the function will have no effect on the argument.
- Output parameters – This method helps in returning multiple values from a method.

There is also a `params` keyword, to pass a variable number of parameters to a method.

62. What are the different ways a method can be overloaded?

There are a few ways for a method to be overloaded like using different data types for a parameter, different order of parameters, and also a different number of parameters.

Architecture

63. What are the Object-Oriented Programming principles?

The object-oriented programming principles are:

- **Abstraction** - Objects only provide access to methods and properties that are relevant to the use of other objects, and hide implementation details.
- **Encapsulation** - The implementation and state of each object should not be exposed to the public.
- **Inheritance** - Code can be reused through the hierarchy.
- **Polymorphism** - At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays.

64. How is encapsulation implemented in C#?

Encapsulation is implemented by directly using the access specifiers.

65. What are the SOLID principles?

S.O.L.I.D is an acronym for the 5 object-oriented design principles by Robert C. Martin (also known as Uncle Bob).

The principles are:

- **Single-responsibility principle** - An object should only have a single responsibility, that is, only changes if one part of the application changes.
- **Open-closed principle** - Objects should be open for extension, but closed for modification.
- **Liskov substitution** - you should be able to replace objects in an application with instances of their subtypes without changing the other code that uses that supertype.
- **Interface segregation principle** - Many smaller interfaces are better than one big interface.
- **Dependency inversion principle** - An object should depend upon interfaces, not concrete types.

If a developer follows these principles, then it should be easy for him to create applications that are easy to maintain and extend.

66. What is a Dependency Injection?

Dependency injection is a technique where you give an object the dependencies it needs. Dependencies are other objects that our object needs to fulfill its functionality.

The advantage of dependency injection is that the object who wants to call some services doesn't have to know how to construct those services. Instead, the object delegates the responsibility of providing its services to an external code.

67. What are IoC Containers?

Inversion of Control (IoC) Containers are classes or packages which help us to register dependencies and then automatically resolve them when we need them.

68. What is the difference between inheritance and composition?

The composition is a type of relationship between objects where one object contains another object. For example, a `Worker` class might contain a property of type `Address`. Object composition is used to represent "*has-a*" relationships: every worker has an address.

Inheritance is a type of relationship between objects where they are arranged in a hierarchy. This is an "*is-a-type-of*" relationship. For example, a class `Worker` might inherit from the class `Person`. This means that the `Worker` class is a type of `Person`.

In general, preferring composition over inheritance is a design principle that gives your code higher flexibility.

69. What are Design Patterns?

A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

70. Explain the Factory Method design pattern.

Factory method is a design pattern that provides an interface for creating objects in a superclass but allows superclass to change the type of the object that will be created.

71. Explain Command design pattern.

The Command is a design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue the request's execution, and support undoable operations. The Command design pattern is used heavily in MVVM architectures such as Xamarin.Forms of WPF. It’s used to trigger action that should be executed in the view model after the user has performed an action (button tap, cell swipe...).

72. Explain the Decorator design pattern.

The Decorator is a design pattern that lets you attach new behavior to objects by placing these objects inside a special wrapper class that usually implements the same interface as the wrapped class.

73. Explain the Singleton design pattern.

Singleton is a design pattern that ensures that a class has only one instance while providing a global access point to this instance.

74. Explain the Observer design pattern.

The observer is a design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they are observing.

75. Explain the Adapter design pattern.

The adapter is a design pattern that allows objects with incompatible interfaces to collaborate. The adapter implements the compatible interface and wraps an object with an incompatible interface. The other object then communicates with the original object through the adapter and compatible interface.

76. Explain the Abstract Factory design pattern.

The Abstract Factory design pattern is a pattern that describes how to deal with creating families of related or dependent objects without specifying their concrete classes. It is used when a family of related objects is needed, but their concrete classes need to be determined by a third-party object.

77. Explain the Chain of Responsibility design pattern.

The Chain of Responsibility pattern is useful when you need to link a set of objects, where each object in the chain either performs a task or hands off the request to the next object in the chain. The request should be processed in sequence, but the request should be passed to the next object only if the current object in the chain can’t process the request.

78. Explain the Builder design pattern.

The Builder pattern is a way of simplifying object creation. Rather than instantiating the object directly, as if you were creating a regular class, you instead construct it through a special interface that hides almost all of your class's implementation details from the client. The "builder" interface defines a sequence of methods that the client can call to manipulate the object's state and get it ready for use. In this way, the client can customize an object's behavior and state without having to know how the class is implemented.

79. Explain the Facade design pattern.

The Facade pattern hides the complexity of the systems and creates an interface that is simpler to use. The facade pattern is a design pattern that helps you separate a large, complex system into smaller components. This can improve the understandability of your code since a user can now only focus on one component instead of having to learn a lot about the entire system.

80. Explain the Template Method design pattern.

The Template Method pattern is a behavioral design pattern, which means that it can be used to define the behavior of a class. The template method defines the skeleton of an algorithm in a method. This method is then specialized in subclasses to make it specific to the type of problem that they are trying to solve.

81. Explain the State design pattern.

The state design pattern is a software design pattern that encapsulates the data that varies over time and helps you define the different behavior of a class.

82. Explain the Composite design pattern.

The Composite design pattern is a structural design pattern that implements a tree structure of objects. The composite pattern lets clients treat individual objects and compositions uniformly. Composite objects can represent either an aggregation of other objects or a whole. The composite pattern ensures that clients interact with the composite (the whole) instead of individual objects that make up a composite (the parts).

83. Explain the Memento design pattern.

The Memento design pattern encapsulates the state of an object into a secondary object, the memento, which stores all the information about the object in an easy-to-access way. This way, you can save the object into persistent storage, and restore it into a new object when needed. The pattern is used when you need to serialize an object and deserialize it later on.

84. Explain the Prototype design pattern.

The Prototype design pattern is a creational design pattern in which an object is created from a duplicate of an existing object, called the prototype. In some ways, the prototype is an initial or "mockup" version of the object, which is then used to make new, confident copies. The prototypes are used to create new objects, with the expectation that these new objects will be similar, if not identical, to the original.

85. Explain the Proxy design pattern.

The Proxy design pattern is an object that acts as a delegate to other objects, providing the means to control access to these other objects. For those instances, the proxy provides a handle to invoke operations on the real object. The proxying object may perform additional operations, such as validation or conversion, that are not intrinsic to the real object.

86. Explain the Visitor design pattern.

The Visitor Pattern allows you to add new operations to a class without modifying the classes themselves. This makes the pattern particularly useful when you want to add new operations to an existing class without the need to modify the classes and recompile the code.

87. Explain the Bridge design pattern.

A bridge is a design pattern that helps decouple an abstraction from its implementation so the two can vary independently.

88. Explain the Iterator design pattern.

Iterator pattern is a creational pattern in which an iterator interface is implemented by a class that enumerates its elements. This iterator interface enables the elements to be accessed one at a time in a sequential manner, through the iterator.

89. Explain the Mediator design pattern.

The Mediator design pattern is used when there are multiple objects that need to communicate with each other. The mediator acts as a central point of control, receiving requests from the client object, communicating with the other objects on the subject's behalf, and sending responses back to the client.

Testing

90. What is unit testing?

Unit testing is a type of testing where individual components of the software are tested.

The purpose of unit testing is to confirm that each unit of the software performs as designed. A unit is a small piece in a codebase, usually a single method in a class, but it can be the class itself.

91. What is the structure of a unit test?

A unit test usually consists of three main parts:

- **Arrange** objects, create and set them up as necessary.
- **Act** on an object. Call the method you want to test.
- **Assert** (check) that something works as expected.

This structure is also known as 3A.

92. What is code coverage?

Code coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. An application with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to an application with low test coverage.

93. What is integration testing?

An integration test is a type of test where you test that several classes work together as expected. Integration tests usually involve interaction with a database, file system, or mail. Therefore their execution time is long since you need to set up the database first and clean it up after the test is completed.

94. What is UI testing?

UI tests are tests that make sure your app's user interface behaves correctly when expected actions are performed.

With UI testing we can find and interact with UI elements, and validate UI properties and state.

95. What is a testing pyramid?

The automated testing pyramid is a visual representation of how to use automated testing in your development process. It suggests that you should have the majority of your tests at the unit level, with fewer tests at the integration level and even fewer at the user interface level.

96. What are stubs?

A stub is a dummy implementation of a dependency your class under test needs to execute a testing scenario. Its main purpose is to return fake data. With stubs, you don't have to deal with network or database in your tests, which enables you to have fast unit tests.

97. What are mocks?

A mock is a fake implementation of a dependency your class under test needs to execute a testing scenario. In this, it's pretty similar to the stub.

However, the main difference between the mock and the stub is that the mock usually verifies that the class under test behaves as expected. The mock checks that some method was called and/or that it was called with correct parameters. In the assert part of the test, you check your assumptions against the mock.

98. What are mocking libraries?

The mocking library is a library that provides an automated way to define and create mocks and stubs.

The popular mocking libraries in .NET are moq, NSubstitute and FakeItEasy.

99. What is Test-Driven Development?

Test-driven development is a software development process where tests are written before the code (the actual implementation). The tests guide the developer to the path of the actual implementation. They drive the implementation, hence the name test-driven development (TDD).

The process of test-driven development follows the following phases:

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- Refactor to remove duplication.

The TDD process is also known as the Red-Green-Refactor cycle.

100. What is refactoring?

Refactoring is a change made to the code to make it easier to understand and cheaper to modify without changing its observable behavior. The advantages of refactoring your code are:

- You'll minimize the risk of introducing bugs while changing code.
- Refactoring helps you to structure the code so it's easy to add new features in the future.
- Refactoring organizes your code so you can easily understand and change it.

Conclusion

Congratulations! You've made it to the end! You are now one step closer to getting that C# job you want.

Remember to take plenty of sleep before your interview.

Best of luck!

Kristijan Kralj