What is the difference between Tuples and ValueTuples?

Is it possible to have a tuple with more than 8 elements?

What is the difference between "is" and "as" keywords?

What is the difference between regular casting and casting with "as" keyword?

Why can we only use the "as" keyword to cast objects to nullable types?

What is the use of the “using” keyword?

What are the global using directives?

What is the purpose of the “dynamic” keyword?

What is the difference between strongly-typed and weakly-typed programming languages?

What is the difference between statically-typed and dynamically-typed programming languages?

What are COM objects?

What are expression-bodied members?

The "is" keyword checks if the object is of a given type. It returns a boolean result. The "as" keyword casts an object to a given type (it's applicable only to casting to reference types or nullable types).

Tuples are limited to hold up to 8 elements, but we can bypass this by making the tuple nested. This is awkward for tuples, but for ValueTuples we get some help from the compiler - it allows us to use the tuple like it really contained more than 8 elements, for example by using Item12 field.

The differences between tuples and ValueTuples are that tuples are reference types and ValueTuples are value types. Also, ValueTuples fields can be named, while with tuples we are stuck with properties named Item1, Item2, etc. Also, tuples are immutable while ValueTuples are mutable.

The "using" keyword has two main uses: the using directive, which allows using types from other namespaces and to create aliases for namespaces, and the using statement that defines the scope in which the IDisposable object will be used, and that will be disposed at the scope's end.

Because if casting with "as" fails, null will be returned. Null can only be assigned to nullable types.

When casting with "as" fails, it will return null. When regular casting fails, an InvalidCastException will be thrown.

In weakly-typed languages, variables are automatically converted from one type to another. In strongly-typed languages, rhey are not. In C#, which is a strongly-typed language, the "2"+8 expression will not compile, while in weakly-typed Perl it will give 10 as a result.

The "dynamic" keyword allows us to bypass static type checking that is done by default by the C# compiler. We can call any operations on dynamic variables and the code will still compile.

When a type is imported in any file with the global using directive, it is like it was imported in all files in the project. This is convenient when some namespace (like, for example, System.Linq) is used in almost every file in the project.

Expression-bodied members of a type are members defined with expression body instead of the regular body with braces. Using them allows us to shorten the code significantly.

COM means "Component Object Model" and it's a binary-interface standard for Windows software components.A COM object is something that can be understood by different Windows programs, and for example, it can allow communication between Excel and C# programs.

In statically-typed languages, the type checks are done at the compile time, while in dynamically-typed languages they are done at runtime. For example, in C# we can't pass an integer to a method expecting a string. In Python, which is dynamically typed, we can, but the execution could result in a runtime error.

What is an expression?

What is a statement?

What are Funcs and lambda expressions?

What is the signature of a function that could be assigned to the variable of type Func<int, int, bool>?

What is an Action?

What are delegates?

What is the difference between a Func and a delegate?

What is a multicast delegate?

How does the Garbage Collector decide which objects can be removed from memory?

What is the Mark-and-sweep algorithm?

How many stacks are there in a running .NET application?

What two main algorithms of identifying used and unused objects are implemented by tools similar to .NET Garbage Collector?

The Func and Action types allow us to represent functions. Lambda expressions are a special way of declaring anonymous functions. They allow us to define functions in a concise way.

A statement is a piece of code that does something but does not evaluate to a value. For example, Console.Writeline("abc") is a statement. It does not evaluate to any value, as the Console.Writeline is a void method.

An expression is a piece of code that evaluates to some value. For example "2 + 5" evaluates to 7.

A delegate is a type whose instances hold a reference to a method with a particular parameter list and return type.

Action is a type used to represent void functions. It works similarly to Func, but Func can only represent non-void functions.

It would be a function that takes two integers as parameters and returns a bool.

Garbage collector removes those objects, to which no references point. To decide whether a reference pointing to some object exists, the Garbage Collector builds a graph of all objects reachable from root objects of the application.

It's a delegate holding references to more than one function.

Func is a kind of delegate. To be more precise, Func is a generic delegate used to represent any function with given parameters and returned type. A delegate is a broader concept than Func - we can define any delegate we want, and it doesn't need to be generic at all.

First is reference counting, which associates a count of references pointing to an object with each object. An example of a language using it is Swift. Another algorithm is tracing (this one is used in .NET) which builds a graph of reachability starting from the application roots.

As many as threads. Each thread has its own stack.

It's the algorithm that the Garbage Collector implements. According to this algorithm, the GC first marks objects that can be removed (mark phase) and then actually removes them (sweep phase).

What are generations?

What is the Large Objects Heap?

What does it mean that the object is pinned?

What is the difference between Dispose and Finalize methods?

What is the difference between a destructor, a finalizer, and the Finalize method?

Does the Garbage Collector call the Dispose method?

When should we write our own destructors?

What are managed and unmanaged resources?

What are default implementations in interfaces?

What can be the reason for using default implementations in interfaces?

What is deconstruction?

What is the difference between the destructor and the Deconstruct method?

It means it will not be moved during the memory defragmentation that the Garbage Collector is executing. It is an optimization, as large objects are expensive to move, and it's hard to find a chunk of memory large enough for them.

It's a special area of the heap reserved for objects larger than 85 000 bytes. Such objects logically belong to generation 2 from the very beginning of their existence and are pinned.

The Garbage Collector divides objects into three generations - 0, 1, and 2 - depending on their longevity. The Garbage Collector collects objects from generation 0 most often, and from generation 2 least often. This feature is introduced in order to improve Garbage Collector's performance.

No. The Garbage Collector is not aware of this method. We must call it ourselves, usually by using the using statement.

There is no difference, as they are the same thing. During the compilation, the destructor gets changed to the Finalize method which is commonly called a finalizer.

The Dispose method is used to free unmanaged resources. The Finalize method is the same thing as the destructor, so it's the method that is called on an object when it is being cleaned up by the Garbage Collector.

Starting with C# 8, we can provide methods implementations in interfaces. This feature was mostly designed to make it easier to add new methods to existing interfaces without breaking the existing code.

The managed resources are managed by the Common Language Runtime. Any objects we create with C# are managed resources. Unmanaged resources are beyond the realm of the CLR. Examples of unmanaged resources are database connections, file handlers, COM objects, opened network connections, etc.

The safest answer is "almost never". Destructors are very tricky and we don't even have a guarantee that they will run. Use IDisposable instead.

The destructor is a method that's called on an object when it's being removed from memory by the Garbage Collector. The Deconstruct method allows objects to be deconstructed into single variables. It is by default generated for tuples, ValueTuples, and positional records, but we can also define it in custom types.

Deconstruction is a mechanism that allows breaking a tuple or a positional record into individual variables. It is also possible to define how deconstruction should work for user-defined types by implementing the Deconstruct method.

Default implementations in interfaces are mostly designed to make it easier to add new methods to existing interfaces without breaking the existing code. Without it, if we add a method to an interface we release it as a public library, we will force everyone who updates this library to provide the implementation immediately.

How can we define deconstruction for types that we did not create and we don't have access to their source code?

Why is "catch(Exception)" almost always a bad idea (and when it is not?)?

What are the acceptable cases of catching any type of exception?

What is the global catch block?

What is the difference between "throw" and "throw ex"?

What is the stack trace?

Should we use "throw" or "throw ex", and why?

What is the difference between typeof and GetType?

What is the purpose of the GetType method?

Where is the GetType method defined?

What is reflection?

What are the downsides of using reflection?

The acceptable use cases for catching any type of exceptions are: 1) The global catch block that is catching all exceptions not handled elsewhere, and shows them to the user. 2) Any catch block in which we rethrow an exception without handling it.

Using “catch(Exception)” should be avoided, because it catches every kind of exception. When we decide to catch an exception, we should know how to handle it, and it’s not feasible if the exception’s type is unknown.

We can define the Deconstruct method as an extension method for this type.

The stack trace is a trace of all methods that have been called, that lead to the current moment of the execution. At the top of the stack trace we have the method that has been called most recently, and at the bottom - the one that has been called first.

“throw” preserves the stack trace (the stack trace will point to the method that caused the exception in the first place) while ”throw ex” does not preserve the stack trace (we will lose the information about the method that caused the exception in the first place.)

The global catch block is the catch block defined at the upper-most level of the application, that is supposed to catch any exceptions that hadn’t been handled elsewhere. It usually logs the exception and shows some information to the user, before stopping the application.

This method returns the Type object which holds all information about the type of the object it was called on. For example, it contains the type name, list of the constructors, attributes, the base type, etc.

Both are used to get the information about some type. Typeof takes the name of the type we want to inspect, and is is resolved at compile time. GetType is a method executed on an object. Because of that, it is resolved at runtime. It comes from the System.Object base class, so it is available in any object in C#.

We should use “throw” as it preserves the stack trace and helps us find the original source of the problem.

Using reflection has a relatively big impact on performance. Also, it makes the code hard to understand and maintain. It may also tempt some programmers to “hack” some code, for example, to access private fields at runtime, which may lead to unexpected results and hard-to-understand bugs.

Reflection is a mechanism that allows us to write code that can inspect types used in the application. For example, using reflection, we can list all fields and their values belonging to a given object, even if at compile time we don’t know what type it is exactly.

It is defined in the System.Object type, which is a base type for all types in C#. This is why we can call the GetType method on objects of any type.

What are attributes?	**What is metadata?**	**How to define a custom attribute?**
What is serialization?	**What are the uses of serialization?**	**What does the Serializable attribute do?**
What is deserialization?	**What is pattern matching?**	**How can we check if an object is of a given type, and cast to it this type in the same statement?**
How does the binary number system work?	**What is the decimal representation of number 101?**	**Why arithmetic operations in programming can give unexpected results, like for example adding two large integers can give a negative number?**

To define a custom attribute we must define a class that is derived from the Attribute base class.

Metadata is data providing information about other data. For example, when working with databases, the data stored inside the database is the actual data, while the structure of tables and relations between them is metadata. In programming, metadata describes types used in an application.

Attributes add metadata to a type. In other words, they are a way to add information about a type or method to the metadata which describes that type or method.

This attribute indicates that instances of a class can be serialized with BinaryFormatter or SoapFormatter. It is not required for XML or JSON serialization.

It can be used to send objects over a network, or to store objects in a file for later reconstruction, or even to store them in a database - for example to save a "snapshot" of an object every time a user makes some changes to it, so we can log the history of the changes.

Serialization is the process of converting an object into a format that can be stored in memory or transmitted over a network. For example, the object can be converted into a text file containing JSON or XML, or a binary file.

We can use pattern matching for that. For example, we could write “if obj is string text”. This way, we will cast the object to the string variable called text, but only if this object is of type string.

Pattern matching is a technique where you test an expression to determine if it has certain characteristics.

Deserialization is the opposite of serialization: it’s using the content of a file to recreate objects.

Because there is a limited number of bits reserved for each numeric type, for example for integer it’s 32 bits. If the result of the arithmetic operation is so large that it doesn’t fit on this amount of bits, some of the bits of the result will be trimmed, giving an unexpected result that is not valid.

It’s 5 because it’s 2 to the power of zero plus two to the power of 2, which gives 1 + 4 = 5.

The binary number system is used to represent numbers using only two digits - 0 and 1. For example, the number 13 (in the decimal number system) is 1101 in the binary number system. All data in a computer’s memory is stored as sequences of bits, and so are all numbers.

What is the purpose of the “checked” keyword?

What is the purpose of the "unchecked" keyword?

What is a silent failure?

What is the BigInteger type?

What is the difference between double and decimal?

What is the difference between double and float?

What is the NaN?

What numeric type should we use to represent money?

What is an Array?

What is a jagged array?

What are the advantages of using arrays?

What are the disadvantages of using arrays?

It’s a kind of failure that happens without any notification to the users or developers - they are not informed that something went wrong, and the application moves on, possibly in an invalid state.

This keyword defines a scope in which check of arithmetic overflow is disabled. It makes sense to use it in projects in which the checking for overflow is enabled for an entire project (can be set on the project level settings).

The “checked” keyword is used to define a scope in which arithmetic operations will be checked for overflow.

The only difference is that double occupies 64 bits of memory while float occupies 32, giving double a larger range. Except for that, they work exactly the same.

Double is a floating-point binary number, while decimal is a floating-point decimal number. Double is optimized for performance, while decimal is optimized for precision. Doubles are much faster, they occupy less memory and they have a larger range, but they are less precise than decimals.

It’s a numeric type that can represent an integer of any size - it is limited only by the application’s memory. It should be used to represent gigantic numbers (remember that max long is over 16 quantillions, so BigInteger should be used instead of long only to represent unthinkably large numbers).

Array is the basic collection type in C#, storing elements in an indexed structure of fixed size. Arrays can be single-dimensional, multi-dimensional, or jagged.

When representing money we should always use decimals.

NaN is a special value that double and float can be. It means Not a Number, and it’s reserved for representing results of undefined mathematical operations, like dividing infinity by infinity.

Arrays are of fixed size, so they can’t be resized; they are not good for representing collections that grow or shrink over time. If we want to allocate the memory for all elements that may be stored, we may allocate too much and waste it, or we can allocate not enough.

They are fast when it comes to accessing an element at the given index. They are basic and easy to use and great for representing simple data of size that is known upfront.

A jagged array is an array of arrays, which can be all of the different lengths.

How to resize an array?

What is a List?

Why it is a good idea to set the Capacity of the List in the constructor if we know the expected count of elements upfront?

What's the time complexity of the Insert method from the List class?

What is an ArrayList?

What is the difference between an array, a List, and an ArrayList?

When to use ArrayList over a generic List<T>?

What is the purpose of the GetHashCode method?

Can two objects of the same type, different by value, have the same hash codes?

Why it may be a good idea to provide a custom implementation of the GetHashCode method for structs?

What is a Dictionary?

What is a hash table?

Because this way we will avoid the performance-costly operation of copying the underlying array into a new, larger one, which happens when we exceed the count of 4, 8, 16… elements.

List<T> is a strongly-typed, generic collection of objects. Lists are dynamic, which means we can add or remove the elements from them. It uses an array as the underlying collection type. As it grows, it may copy the existing array of elements to a new, larger array.

It’s not possible. An array is a collection of a fixed size and once created, it can’t be resized.

An array is a basic collection of fixed size that can store any declared type of elements. The List is similar but its size can change. An ArrayList is a dynamic collection that can store various types of elements at the same time, as it treats everything it stores as instances of the System.Object type.

It's a collection that can store elements of any type (all as objects). They were widely used in old C#, where the generics were not yet available. Nowadays they should not be used, as their performance is impacted by the fact that they need to box value types.

This method needs to move some of the elements of the underlying array forward, to make room for the new element. In the worst-case scenario, when we insert an element at the beginning, we will need to move all existing elements, so the complexity of this operation is O(N).

Yes. Hash code duplications (or “hash code conflicts”) can happen, simply because the count of distinct hash codes is equal to the range of the integer, and there are many types that can have much more distinct objects than this count.

The GetHashCode method generates an integer for an object, based on this object’s fields and properties. This integer, called hash, is most often used in hashed collections like HashSet or Dictionary.

Never, unless you work with a very old version of C#, which did not support generics. Even if you do, you should rather upgrade .NET to a higher version than work with ArrayLists.

A hash table is a data structure that stores values in an array of collections. The index in the array is calculated using the hash code. It allows quick retrieval of objects with given hashcode. A hash table is the underlying data structure of Dictionary.

A Dictionary is a data structure representing a collection of key-value pairs. Each key in the Dictionary must be unique.

Because the default implementation uses reflection, and because of that is slow. A custom implementation may be significantly faster, and if we use this struct as a key in hashed collections extensively, it may improve the performance very much.

Will the Dictionary work correctly if we have hash code conflict for two of its keys?

Why should we override the Equals method when we override the GetHashCode method?

What are indexers?

Is it possible to have a class with an indexer accepting a string as a parameter?

Can we have more than one indexer defined in a class?

What is caching?

What are the benefits of using caching?

What are the downsides of using caching?

What are immutable types and what's their purpose?

What are pure functions?

What are the benefits of using immutable types?

What is the non-destructive mutation?

Indexers allow instances of a type to be indexed just like arrays. In this way, they resemble properties except that they take parameters. For example, a Dictionary<string, int> has an indexer that allows calling “dictionaryVariable[“some key”]” to access the value under some key.

Because the Equals method is needed for the Dictionary to distinguish two keys in case of the hash code conflict, so its implementation should be in line with the implementation of the GetHashCode method. For example, if GetHashCode returns the ID for a Person object, then the Equals method should also only compare the IDs

Yes. The Dictionary still can tell which key is which using the Equals method, so it will not mistake them only because they have the same hash codes.

Caching is a mechanism that allows storing some data in memory, so next time it is needed, it can be served faster.

Yes. Just like with method overloading, we can have as many indexers as we want, as long as they differ by the type, count, or order of parameters.

Yes. We can define indexers with any parameters. An example of such a class can be a Dictionary<string, int> as we access its elements like “dict[“abc”]”.

Immutability of a type means that once an object of this type is created none of its fields of properties can be updated. Using immutable types over mutable ones gives a lot of benefits, like making the code simpler to understand, maintain and test, as well making it thread-safe.

Cache occupies memory. It may grow over time, so some kind of cleanup mechanism should be introduced (usually it baes on the expiration time of the data). The data in the cache may become stale (not up-to-date). Because of that, caching is most useful when retrieving data that doesn’t change often.

Caching can give us a performance boost if we repeatedly retrieve data identified by the same key. It can help not only with data retrieved from an external data source but even calculated locally if the calculation itself is heavy (for example some complex mathematical operations).

It is an operation of creating a new object based on another immutable object. The immutable object won’t be modified, but the result of “modification” will become a new object. For example adding 7 days to a date of January the 1st will not change this date, but it will produce a new date of January the 8th.

The code using them is simple to understand. They make it easy to create pure functions and to work with multithreaded applications, as there is no risk that one thread will modify a value that the other thread is using. Immutable objects retain their identity and validity.

Pure functions are functions whose results only depend on the input parameters, and they do not have any side effects like changing the state of the class they belong to or modifying the objects passed as an input.

What are records and record structs?	**What is the purpose of the "with" keyword?**	**What are positional records?**
Why does string behave like a value type even though it is a reference type?	**What is interning of strings?**	**What is the size of the stack in megabytes?**
What is the underlying data structure for strings?	**What is the difference between string and StringBuilder?**	**What does it mean that strings are immutable?**
What is operator overloading?	**What is the purpose of the "operator" keyword?**	**What is the difference between explicit and implicit conversion?**

They are records with no bodies. The compiler generates properties, constructor, and the Deconstruct method for them. They are a shorter way of defining records, but we can't add custom methods or writable properties to a positional record.

The "with" keyword is used to create a copy of a record object with some properties set to new values. In other words, it's used to perform a non-destructive mutation of records.

Records and record structs are new types introduced in C# 9 and 10. They are mostly used to define simple types representing data. They support value-based equality. They make it easy to create immutable types.

It's 1 MB for 32-bit processes and 4 MB for 64-bit processes.

Interning means that if multiple strings are known to be equal, the runtime can just use a single string, thereby saving memory. This optimization wouldn't work if strings were mutable, because then changing one string would have unpredictable results on other strings.

String is a reference type with the value type semantics. String has value-type semantics as this is more convenient for developers, but it can't be a value type because string objects can be large, and value types are stored on the stack which has a limited size. All strings are immutable.

It means once a string is created, it can't be modified. When we modify a string, actually a brand-new string is created and the variable that stored it simply has a new reference to this new object.

String is a type used for representing textual data. StringBuilder is a utility class created for optimal concatenation of strings.

It's an array of chars. Arrays by definition have fixed size, which is a reason why strings are immutable - we couldn't modify a string by adding new characters to it, because they wouldn't fit in the underlying array.

Implicit conversion happens when we assign a value of one type to a variable of another type, without specifying the target type in the parenthesis. For example, when assigning an int to a double. Explicit conversion requires specifying the type in parenthesis, for example when assigning a double to an int.

It is used when overloading an operator for a type.

Operator overloading is a mechanism that allows us to provide custom behavior when objects of the type we defined are used as operands for some operators. For example, we can define what will "obj1+obj2" do.

What are anonymous types?	**Can we modify the value of an anonymous type property?**	**When should we, and when should we not use anonymous types?**
Are anonymous types value or reference types?	**What is cohesion?**	**Is following the Single Responsibility Principle and keeping high cohesion the same thing?**
What is coupling?	**How to recognize strongly couples types?**	**Which of the SOLID principles allow us to reduce coupling?**
What is the Strategy design pattern?	**What are the benefits of using the Strategy design pattern?**	**What is the Dependency Injection design pattern?**

We should use them when the type we want to use is simple and local to some specific context and it will not be used anywhere else. Also, anonymous types can only provide read-only properties; they can't have methods, fields, events, etc, so we sshouldn't use them if we need any of those features.

No. All properties of anonymous types are read-only.

Anonymous types are types without names. They provide a convenient way of encapsulating a set of read-only properties into a single object without having to explicitly define a type first.

No, but it's common that a highly cohesive class meets the SRP and vice versa. If following only the SRP, we could keep splitting classes into smaller pieces until every class would have only one public method. Each of them would meet the SRP, but they wouldn't be cohesive, as those methods should belong together.

Cohesion is the degree to which elements of a module belong together. In simpler words, it measures how strong the relationship is between members of a class. High cohesion is a desirable trait of the classes and modules.

They are reference types since they are classes, but they support value-based Equality with the Equals method. In other words, two anonymous objects with the same values of properties will be considered equal by the Equals method even if their references are different.

The Dependency Inversion Principle, which says that classes shouldn't depend on concrete implementations, but rather on abstractions. When following this principle we remove the direct way of communication between classes, making them more independent from each other.

One type uses another type directly, without having any abstraction in between. We often recognize strong coupling the hard way: when we see that even a small change in a class leads to a cascade of changes all around the project. It proves that the types are not independent.

Coupling is the degree to which one module depends on another module. In other words, it's a level of "intimacy" between modules. If a module is very close to another, knows a lot about its details, and will be affected if the other changes, it means they are strongly coupled.

Dependency Injection is providing the objects some class needs (its dependencies) from the outside, instead of having it construct them itself.

It helps to reduce code duplications, makes the code cleaner and more easily testable. It separates the code that needs to be changed often (the particular strategy) from the code that doesn't change that much (the code using the strategy).

The Strategy Design pattern is a pattern that allows us to define a family of algorithms to perform some tasks. The concrete strategy can be chosen at runtime.

What are Dependency Injection frameworks?

What are the benefits of using Dependency Injection?

What is the Template Method design pattern?

What is the difference between the Template Method design pattern and the Strategy design pattern?

What is the Decorator design pattern?

What are the benefits of using the Decorator design pattern?

What is the Observer design pattern?

In the Observer design pattern, what is the Observable and what is the Observer?

What are events?

What is the difference between an event and a field of the delegate type?

Why is it a good practice to unsubscribe from events when a subscribed object is no longer needed?

What is Inversion of Control?

Template Method is a design pattern that defines the skeleton of an algorithm in the base class. Specific steps of this algorithm are implemented in derived classes.

It decouples a class from its dependencies. The class doesn't decide what concrete type it will use, it only declares in the constructor what interfaces it will need. Thanks to that, we can easily switch the dependencies according to our needs, for example to inject a mock in unit tests.

They are mechanisms that automatically create dependencies and inject them into objects that need them. They are configurable, so we can decide what concrete types will be injected into objects depending on some abstractions. Some of the popular Dependency Injection frameworks in C# are Autofac or Ninject.

This pattern allows us to easily add functionality to objects, without touching the original classes, so it's in line with the Open-Closed Principle. It makes it easy to stack functionalities together. It also helps us to be in line with the Single Responsibility Principle, as each class now has a very focused responsibility.

Decorator is a design pattern that dynamically adds extra functionality to an existing object, without affecting the behavior of other objects from the same class.

Both allow specifying what concrete algorithm or a piece of the algorithm will be used. The difference is that with the Template Method, it is selected at compile-time, as this pattern uses the inheritance. With the Strategy pattern, the decision is made at runtime, as this pattern uses composition.

Events are the .NET way of implementing the Observer design pattern. They are used to send a notification from an object to all objects subscribed.

The Observable is the object that's being observed by Observers. The Observable notifies the Observers about the change in its state.

The Observer design pattern allows objects to notify other objects about changes in their state.

Inversion of Control is the design approach according to which the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the external sources (framework, services, other components) take control of it.

Because as long as it is subscribed, a hidden reference between the observable and the observer exists, and it will prevent the Garbage Collector from removing the observer object from memory.

A public field of a delegate type can be invoked from anywhere in the code. Events can only be invoked from the class they belong to.

What is a callback?

What is the difference between a framework and a library?

What is the “composition over inheritance” principle?

What is the problem with using composition only?

What are forwarding methods?

What are mocks?

What is Moq?

What is the relation between mocking and Dependency Injection?

What are NuGet packages?

What is the difference between Debug and Release builds?

How can we execute some piece of code only in the Debug, or only in the Release mode?

What are preprocessor directives?

“Composition over inheritance” is a design principle stating that we should favor composition over inheritance. In other words, we should reuse the code by rather containing objects within other objects, than inheriting one from another.

A **library** is a set of functions that you can call. A **framework** embodies some abstract design, with more behavior built in. In order to use it, you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework relies on Inversion of Control, but the library does not.

A callback is an executable code (a method in C#) that gets passed as an argument to some other code.

They are objects that can be used to substitute real dependencies for testing purposes. For example, we don’t want to use a real database connection in unit tests. Instead, we will replace the object connecting to a database with a mock that provides the same interface, but returns test data.

They are methods that don’t do anything else than calling almost identical methods from some other type. Forwarding methods indicate a very close relationship between types, which may mean that one type should be inherited from another.

Without inheritance it's hard to define types that are indeed in an “IS-A” relation. For example, a Dog IS an Animal. When implementing such hierarchy with the composition we create very similar types that wrap other types only adding a bit of new functionality, and they mostly contain forwarding methods.

NuGet packages contain compiled code that someone else created, that we can reuse in our projects. The tool used to install and manage them is called NuGet Package Manager.

Mocking is hard to implement without the Dependency Injection. DI allows us to inject some dependencies to a class, so we can choose whether we inject real implementations or mocks. If the dependency of the class would not be injected but rather created right in the class, it wouldn't be possible to make it a mock.

Moq is a popular mocking library for C#. It allows us to easily create mocks of interfaces, classes, Funcs, or Actions. It gives us the ability to decide what result will be returned from the mocked functions, or validate if some function has been called.

They help us control the compilation process from the level of the code itself. We can choose if some part of the code will be compiled or not, we can disable or enable some compilation warnings, or we can even check for the .NET version and execute different code depending on it.

By placing it inside a **#if DEBUG** or **#if RELEASE** conditional preprocessor directives.

During the Release build, the compiler applies optimizations it finds appropriate. Because of that, the result of the build is often smaller and it works faster. On the other hand, it’s harder to debug because the compiled result doesn’t match the source code exactly.

What is the preprocessor?	**How to disable selected warning in a file?**	**What are nullable reference types?**
What is the default value of non-nullable reference types?	**What is the purpose of the null-forgiving operator?**	**Is it possible to enable or disable compiler warnings related to nullable reference types on the file level? If so, how to do it?**

This feature enables explicit declaration of a reference type as nullable or not. The compiler will issue proper warnings for such types, for example if we assign nul to a non-nullable type. This feature doesn't change the actual way of executing C# code; it only changes the generated warnings.

By using the #pragma warning disable preprocessor directive. It takes the warning code as the parameter, so for example to disable the "Don't use throw ex" warning we can do "#pragma warning disable CA2200".

The preprocessor (also known as the "precompiler") is a program that runs before the actual compiler, that can apply some operations on code before it's compiled.

It is possible. We can do it by using #nullable enable and #nullable disable preprocessor directives.

It allows us to suppress a compiler warning related to nullability.

It is null.