FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Unit Of Work*? ☆☆☆

Topics: ADO.NET OOP Design Patterns

Answer:

Unit of Work is referred to as a single transaction that involves multiple operations of `insert/update/delete` and so on kinds. To say it in simple words, it means that for a specific user action (say registration on a website), all the transactions like insert/update/delete and so on are done in one single transaction, rather than doing multiple database transactions.

Q2: How can you prevent your class to be *inherited* further? ☆☆☆

Topics: OOP

Answer:

You can prevent a class from being inherited further by defining it with the `sealed` keyword.

Q3: Can you specify the accessibility modifier for methods *inside* the interface? ☆☆☆

Topics: OOP

Answer:

All the methods inside an interface are always `public`, by default. You cannot specify any other access modifier for them.

Q4: Is it possible for a class to *inherit the constructor* of its *base* class? ☆☆☆

Topics: OOP

Answer:

No, a class cannot inherit the constructor of its base class.

Q5: What are *similarities* between a `class` and a `structure`? ☆☆☆

Topics: OOP

Answer:

The following are some of the similarities between a class and a structure:

- Access specifiers, such as *public*, *private*, and *protected*, are identically used in structures and classes to restrict the access of their data and methods outside their body.
- The access level for class members and struct members, including nested classes and structs, is private by default. Private nested types are not accessible from outside the containing type.
- Both can have constructors, methods, properties, fields, constants, enumerations, events, and event handlers.
- Both structures and classes can implement interfaces to use multiple-inheritance in code.
- Both structures and classes can have constructors with parameter.
- Both structures and classes can have delegates and events.

Q6: State the features of an *Interface* ☆☆☆

Topics: OOP

Answer:

An interface is a template that contains only the signature of methods. The signature of a method consists of the numbers of parameters, the type of parameter (value, reference, or output), and the order of parameters. An interface has no implementation on its own because it contains only the definition of methods without any method body. An interface is defined using the *interface* keyword. Moreover, you cannot instantiate an interface. The various features of an interface are as follows:

- An interface is used to implement multiple inheritance in code. This feature of an interface is quite different from that of abstract classes because a class cannot derive the features of more than one class but can easily implement multiple interfaces.
- It defines a specific set of methods and their arguments.
- Variables in interface must be declared as *public*, *static*, and *final* while methods must be *public* and *abstract*.
- A class implementing an interface must implement all of its methods.
- An interface can derive from more than one interface.

Q7: What do you mean by *Data Encapsulation*? ☆☆☆

Topics: OOP

Answer:

Data encapsulation is a concept of binding data and code in single unit called object and hiding all the implementation details of a class from the user. It prevents unauthorized access of data and restricts the user to use the necessary data only.

Q8: What are the different ways a method can be *Overloaded*? ☆☆☆

Topics: OOP

Answer:

The different ways to overload a method are given as follows:

- By changing the number of parameters used
- By changing the order of parameters
- By using different data types for the parameters

Q9: How could you define *Abstraction* in OOP? ☆☆☆

Topics: OOP

Answer:

Abstraction is a technique of taking something specific and making it less specific.

In OOP we achieve the abstraction by **separating the implementation from the interface**. We take a implemented class and took only those method signatures and properties which are required by the class client. We put these method signatures and properties into the interface or abstract class.

Q10: *Interface* or an *Abstract* class: which one to use? ☆☆☆

Topics: OOP

Answer:

- Use an *interface* when you want to force developers working in your system (yourself included) to implement a set number of methods on the classes they'll be building.

- Use an *abstract class* when you want to force developers working in your system (yourself included) to implement a set numbers of methods and you want to provide some base methods that will help them develop their child classes.

Q11: What's the difference between a `method` and a `function` in OOP context? ☆☆☆

Topics: OOP

Answer:

A **function** is a piece of code that is called by name. It can be passed data to operate on (i.e. the parameters) and can optionally return data (the return value). All data that is passed to a function is explicitly passed.

A **method** is a piece of code that is called by a name that is associated with an object. In most respects it is identical to a function except for two key differences:

1. A method is implicitly passed the object on which it was called.
2. A method is able to operate on data that is contained within the class (remembering that an object is an instance of a class - the class is the definition, the object is an instance of that data).

Q12: Explain different types of *Inheritance* ☆☆☆☆

Topics: OOP .NET Core

Answer:

Inheritance in OOP is of four types:

- **Single inheritance** - Contains one base class and one derived class
- **Hierarchical inheritance** - Contains one base class and multiple derived classes of the same base class
- **Multilevel inheritance** - Contains a class derived from a derived class

- **Multiple inheritance** - Contains several base classes and a derived class

All .NET languages support single, hierarchical, and multilevel inheritance. They do not support multiple inheritance because, in these languages, a derived class cannot have more than one base class. However, you can implement multiple inheritance in .NET through interfaces.

Q13: Explain the concept of *Destructor* ☆☆☆☆

Topics: OOP

Answer:

A destructor is a special method for a class and is invoked automatically when an object is finally destroyed. The name of the destructor is also same as that of the class but is followed by a prefix tilde (~).

A destructor is used to free the dynamic allocated memory and release the resources. You can, however, implement a custom method that allows you to control object destruction by calling the destructor.

The main features of a destructor are as follows:

- Destructors do not have any return type
- Similar to constructors, destructors are also always public
- Destructors cannot be overloaded.

Q14: Differentiate between an `abstract` class and an `interface` ☆☆☆☆

Topics: OOP

Answer:

Abstract Class:

1. A class can extend only one abstract class
2. The members of abstract class can be private as well as protected.
3. Abstract classes should have subclasses
4. Any class can extend an abstract class.
5. Methods in abstract class can be abstract as well as concrete.
6. There can be a constructor for abstract class.
7. The class extending the abstract class may or may not implement any of its method.
8. An abstract class can implement methods.

Interface

1. A class can implement several interfaces
2. An interface can only have public members.
3. Interfaces must have implementations by classes
4. Only an interface can extend another interface.
5. All methods in an interface should be abstract
6. Interface does not have constructor.
7. All methods of interface need to be implemented by a class implementing that interface.
8. Interfaces cannot contain body of any of its method.

Q15: What is *Coupling* in OOP? ☆☆☆☆

Topics: OOP

Answer:

OOP Modules are dependent on each other. **Coupling** refers to **level of dependency between two software modules**. Two modules are **highly dependent** on each other if you have changed in one module and for supporting that change every time you have to change in the dependent module. **Loose Coupling is always preferred**.

Inversion of Control and **Dependency Injection **are some techniques for getting loose coupling in modules.

Q16: What exactly is the difference between an *Interface* and `abstract` class? ☆☆☆☆

Topics: OOP

Answer:

- An interface is a **contract**: The person writing the interface says, "*hey, I accept things looking that way*", and the person using the interface says "*OK, the class I write looks that way*".
- *An interface is an empty shell**. There are only the signatures of the methods, which implies that the methods do not have a body. The interface can't do anything. It's just a pattern. Implementing an interface consumes very little CPU, because it's not a class, just a bunch of names, and therefore there isn't any expensive look-up to do. It's great when it matters, such as in embedded devices.
- Abstract classes, unlike interfaces, are classes. They are more expensive to use, because there is a look-up to do when you inherit from them. Abstract classes look a lot like interfaces, but they have something more: You can define a behavior for them. It's more about a person saying, *"these classes should look like that, and they have that in common, so fill in the blanks!"*.

Q17: When should I use an *Interface* and when should I use a *Base Class*? ☆☆☆☆

Topics: OOP

Answer:

Modern style is to define IPet *and* PetBase. The advantage of the interface is that other code can use it without any ties whatsoever to other executable code. Completely "clean." Also interfaces can be mixed. But base classes are useful for simple implementations and common utilities. So provide an abstract base class as well to save time and code.