FullStack.Cafe - Kill Your Tech Interview

Q1: Could you explain some benefits of *Repository Pattern*? ☆☆☆☆

Topics: ADO.NET OOP Design Patterns

Answer:

Very often, you will find SQL queries scattered in the codebase and when you come to add a column to a table you have to search code files to try and find usages of a table. The impact of the change is far-reaching. **With the repository pattern, you would only need to change one object and one repository**. The impact is very small.

Basically, **repository** hides the details of how exactly the data is being fetched/persisted from/to the database. Under the covers:

- for reading, it creates the query satisfying the supplied criteria and returns the result set
- for writing, it issues the commands necessary to make the underlying *persistence* engine (e.g. an SQL database) save the data

Q2: Can you declare an *overridden* method to be `static` if the original method is *not static*? ☆☆☆☆

Topics: OOP

Answer:

No. Two virtual methods must have the same signature.

Q3: Does .NET support *Multiple Inheritance*? ☆☆☆☆

Topics: OOP .NET Core

Answer:

.NET does not support multiple inheritance directly because in .NET, a class cannot inherit from more than one class. .NET supports multiple inheritance through interfaces.

Q4: What is a `static` *constructor*? ☆☆☆☆

Topics: OOP C#

Answer:

Static constructors are introduced with C# to **initialize the static data of a class**. CLR calls the static constructor before the first instance is created.

The static constructor has the following features:

- No access specifier is required to define it.

- You cannot pass parameters in the static constructor.
- A class can have only one static constructor.
- It can access only static members of the class.
- It is invoked only once when the program execution begins.

Q5: What is *Cohesion* in OOP? ☆☆☆☆

Topics: OOP

Answer:

In OOP we develop our code in modules. Each module has certain responsibilities. Cohesion shows how much module responsibilities are strongly related. Higher cohesion is always preferred. Higher cohesion benefits are:

- Improves maintenance of modules
- Increase reusability

Q6: What is the difference between an `abstract` function and a `virtual` function? ☆☆☆☆

Topics: OOP

Answer:

- **An abstract function cannot have functionality.** You're basically saying, any child class MUST give their own version of this method, however it's too general to even try to implement in the parent class.
- **A virtual function**, is basically saying look, here's the functionality that may or may not be good enough for the child class. So if it is good enough, use this method, if not, then override me, and provide your own functionality.

Q7: What's the advantage of using *getters* and *setters* - that only `get` and `set` - instead of simply using public fields for those variables? ☆☆☆☆

Topics: OOP

Answer:

There are actually *many good reasons* to consider using accessors rather than directly exposing fields of a class - beyond just the argument of encapsulation and making future changes easier.

Here are the some of the reasons I am aware of:

- Encapsulation of behavior associated with getting or setting the property - this allows additional functionality (like validation) to be added more easily later.
- Hiding the internal representation of the property while exposing a property using an alternative representation.
- Insulating your public interface from change - allowing the public interface to remain constant while the implementation changes without affecting existing consumers.
- Controlling the lifetime and memory management (disposal) semantics of the property - particularly important in non-managed memory environments (like C++ or Objective-C).
- Providing a debugging interception point for when a property changes at runtime - debugging when and where a property changed to a particular value can be quite difficult without this in some languages.

- Improved interoperability with libraries that are designed to operate against property getter/setters - Mocking, Serialization, and WPF come to mind.
- Allowing inheritors to change the semantics of how the property behaves and is exposed by overriding the getter/setter methods.
- Allowing the getter/setter to be passed around as lambda expressions rather than values.
- Getters and setters can allow different access levels - for example the get may be public, but the set could be protected.

Q8: What is the difference between *Cohesion* and *Coupling*? ☆☆☆☆

Topics: OOP

Answer:

- **Cohesion** refers to what the class (or module) can do.
- *Low cohesion* would mean that the class does a great variety of actions - it is broad, unfocused on what it should do.
- *High cohesion* means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

- As for **coupling**, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together, making a change could require an entire system revamp.

Good software design has **high cohesion** and **low coupling**.

Q9: How to solve *Circular Reference*? ☆☆☆☆

Topics: C# OOP

Problem:

How do you solve circular reference problems like `Class A` has `Class B` as one of its properties, while `Class B` has `Class A` as one of its properties?

Solution:

First I would tell you needs to rethink your design. Circular references like you describe are often a code smell of a design flaw. In most cases when I've had to have two things reference each other, I've created an interface to remove the circular reference. For example:

BEFORE

```
public class Foo
{
    Bar myBar;
}

public class Bar
{
    Foo myFoo;
}
```

Dependency graph:

```
Foo     Bar
 ^       ^
 |       |
Bar     Foo
```

Foo depends on Bar, but Bar also depends on Foo. If they are in separate assemblies, you will have problems building, particularly if you do a clean rebuild.

AFTER

```
public interface IBar
{
}

public class Foo
{
    IBar myBar;
}

public class Bar : IBar
{
    Foo myFoo;
}
```

Dependency graph:

```
Foo, IBar     IBar
    ^          ^
    |          |
   Bar        Foo
```

Both Foo and Bar depend on IBar. There is no circular dependency, and if IBar is placed in its own assembly, Foo and Bar being in separate assemblies will no longer be an issue.

Q10: You have defined a *destructor* in a class that you have developed by using the C#, but the destructor *never executed*. Why? ☆☆☆☆☆

Topics: OOP C#

Answer:

The runtime environment automatically invokes the destructor of a class to release the resources that are occupied by variables and methods of an object. However, in C#, programmers cannot control the timing for invoking destructors, as Garbage Collector is only responsible for releasing the resources used by an object. Garbage Collector automatically gets information about unreferenced objects from .NET's runtime environment and then invokes the *Finalize()* method.

Although, it is not preferable to force Garbage Collector to perform garbage collection and retrieve all inaccessible memory, programmers can use the *Collect()* method of the Garbage Collector class to forcefully execute Garbage Collector.

Q11: Can you declare a `private` class in a *namespace*? ☆☆☆☆☆

Topics: OOP

Answer:

The classes in a namespace are *internal*, by default. However, you can explicitly declare them as *public* only and not as *private*, *protected*, or *protected internal*. The nested classes can be declared as *private*, *protected*, or *protected internal*.

Q12: What is the difference between *Association*, *Aggregation* and *Composition*? ☆☆☆☆☆

Topics: OOP

Answer:

- **Association** is a relationship where all objects have their own lifecycle and there is no owner.

  Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers, but there is no ownership between the objects and both have their own lifecycle. Both can be created and deleted independently.

- **Aggregation** is a specialised form of Association where all objects have their own lifecycle, but there is ownership and child objects can not belong to another parent object.

  Let's take an example of Department and teacher. A single teacher can not belong to multiple departments, but if we delete the department, the teacher object will *not* be destroyed. We can think about it as a “has-a” relationship.

- **Composition** is again specialised form of Aggregation and we can call this as a “death” relationship. It is a strong type of Aggregation. Child object does not have its lifecycle and if parent object is deleted, all child objects will also be deleted.

  Let's take again an example of relationship between House and Rooms. House can contain multiple rooms - there is no independent life of room and any room can not belong to two different houses. If we delete the house - room will automatically be deleted.

  Let's take another example relationship between Questions and Options. Single questions can have multiple options and option can not belong to multiple questions. If we delete the questions, options will automatically be deleted.

Q13: Can you provide a simple explanation of *methods* vs. *functions* 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

Q14: Why prefer *Composition* over *Inheritance*? What trade-offs are there for each approach? When should you choose *Inheritance* over *Composition*? ☆☆☆☆☆

Topics: OOP

Answer:

Think of containment as a has a relationship. A car "has an" engine, a person "has a" name, etc. Think of inheritance as an is a relationship. A car "is a" vehicle, a person "is a" mammal, etc.

Prefer composition over inheritance as it is more malleable / easy to modify later, but do not use a compose-always approach. With composition, it's easy to change behavior on the fly with Dependency Injection / Setters. Inheritance is more rigid as most languages do not allow you to derive from more than one type. So the goose is more or less cooked once you derive from TypeA.

My acid test for the above is:

- Does TypeB want to expose the complete interface (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected? Indicates **Inheritance**.

e.g. A Cessna biplane will expose the complete interface of an airplane, if not more. So that makes it fit to derive from Airplane.

- Does TypeB want only some/part of the behavior exposed by TypeA? Indicates need for **Composition.**

e.g. A Bird may need only the fly behavior of an Airplane. In this case, it makes sense to extract it out as an interface / class / both and make it a member of both classes.

Q15: What does it mean to *Program to an Interface*? ☆☆☆☆☆

Topics: OOP

Answer:

Programming to an interface has absolutely nothing to do with abstract interfaces like we see in Java or .NET. It isn't even an OOP concept. It means just interact with an object or system's public interface. Don't worry or even anticipate how it does what it does internally. Don't worry about how it is implemented. In object-oriented code, it is why we have public vs. private methods/attributes.

And with databases it means using views and stored procedures instead of direct table access.

Using interfaces is a key factor in making your code easily testable in addition to removing unnecessary couplings between your classes. By creating an interface that defines the operations on your class, you allow classes that want to use that functionality the ability to use it without depending on your implementing class directly. If later on you decide to change and use a different implementation, you need only change the part of the code where the implementation is instantiated. The rest of the code need not change because it depends on the interface, not the implementing class.

This is very useful in creating unit tests. In the class under test you have it depend on the interface and inject an instance of the interface into the class (or a factory that allows it to build instances of the interface as needed) via the constructor or a property settor. The class uses the provided (or created) interface in its methods. When you go to write your tests, you can mock or fake the interface and provide an interface that responds with data configured in your unit test. You can do this because your class under test deals only with the interface, not your concrete implementation. Any class implementing the interface, including your mock or fake class, will do.

Q16: What is *LSP (Liskov Substitution Principle)* and what are some examples of its use (good and bad)? ☆☆☆☆☆

Topics: OOP

Answer:

The Liskov Substitution Principle (LSP, lsp) is a concept in Object Oriented Programming that states:

> Functions that use pointers or references to base classes **must be able** to use objects of derived classes without knowing it. IN other words substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S.

Consider **the bad example**:

```
public class Bird{
    public void fly(){}
}
public class Duck extends Bird{}
public class Ostrich extends Bird{}
```

The duck can fly because of it is a bird. Ostrich is a bird, but it can't fly, Ostrich class is a subtype of class Bird, but it can't use the fly method, that means that we are breaking LSP principle.

So **the right example**:

```
public class Bird{
}
public class FlyingBirds extends Bird{
    public void fly(){}
}
public class Duck extends FlyingBirds{}
public class Ostrich extends Bird{}
```

Q17: In terms that an OOP programmer would understand (without any functional programming background), what is a `monad`? ☆☆☆☆☆

Topics: OOP

Answer:

A monad is an *"amplifier" of types* that *obeys certain rules* and *which has certain operations provided*.

First, what is an "amplifier of types"? By that I mean some system which lets you take a type and turn it into a more special type. For example, in C# consider `Nullable<T>`. This is an amplifier of types. It lets you take a type, say `int`, and add a new capability to that type, namely, that now it can be null when it couldn't before.

As a second example, consider `IEnumerable<T>`. It is an amplifier of types. It lets you take a type, say, `string`, and add a new capability to that type, namely, that you can now make a sequence of strings out of any number of single strings.

Q18: What is the difference between a *Mixin* and *Inheritance*? ☆☆☆☆☆

Topics: OOP

Answer:

A **mix-in** is a base class you can inherit from to provide additional functionality. The name "mix-in" indicates it is intended to be mixed in with other code. As such, the inference is that you would not instantiate the mix-in class on its own. Frequently the mix-in is used with other base classes. Therefore **mixins are a subset, or special case, of inheritance**.

The advantages of using a mix-in over single inheritance are that you can write code for the functionality one time, and then use the same functionality in multiple different classes. The disadvantage is that you may need to look for that functionality in other places than where it is used, so it is good to mitigate that disadvantage by keeping it close by.

Q19: Why doesn't C# allow *static methods* to implement an *interface*? ☆☆☆☆☆

Topics: OOP C#

Answer:

My (simplified) technical reason is that static methods are not in the vtable, and the call site is chosen at compile time. It's the same reason you can't have override or virtual static members. For more details, you'd need a CS grad or compiler wonk - of which I'm neither.

You pass an interface to someone, they need to know how to call a method. An interface is just a virtual method table (vtable). Your static class doesn't have that. The caller wouldn't know how to call a method. (Before i read this answer i thought C# was just being pedantic. Now i realize it's a technical limitation, imposed by what an interface is). Other people will talk down to you about how it's a bad design. It's not a bad design - it's a technical limitation.

Q20: Could you elaborate *Polymorphism* vs *Overriding* vs *Overloading*? ☆☆☆☆☆

Topics: OOP

Answer:

- **Polymorphism** is the ability of a class instance to behave as if it were an instance of another class in its inheritance tree, most often one of its ancestor classes. For example, in .NET all classes inherit from Object. Therefore, you can create a variable of type Object and assign to it an instance of any class.

- An **override** is a type of function which occurs in a class which inherits from another class. An override function "replaces" a function inherited from the base class, but does so in such a way that it is called even when an instance of its class is pretending to be a different type through polymorphism. Referring to the previous example, you could define your own class and override the toString() function. Because this function is inherited from Object, it will still be available if you copy an instance of this class into an Object-type variable. Normally, if you call toString() on your class while it is pretending to be an Object, the version of toString which will actually fire is the one defined on Object itself. However, because the function is an override, the definition of toString() from your class is used even when the class instance's true type is hidden behind polymorphism.

- **Overloading** is the action of defining multiple methods with the same name, but with different parameters. It is unrelated to either overriding or polymorphism.