FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Inheritance*? ☆

Topics: OOP

Answer:

Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the IS-A relationship. For example, mammal IS A animal, dog IS-A mammal hence dog IS-A animal as well, and so on.

Q2: What is *Object-Oriented Programming* (OOP)? ☆

Topics: OOP

Answer:

OOP is a technique to develop logical modules, such as classes that contain properties, methods, fields, and events. An object is created in the program to represent a class. Therefore, an object encapsulates all the features, such as data and behavior that are associated to a class. OOP allows developers to develop modular programs and assemble them as software. Objects are used to access data and behaviors of different software modules, such as classes, namespaces, and sharable assemblies. .NET Framework supports only OOP languages, such as Visual Basic .NET, Visual C#, and Visual C++.

Q3: What is *Encapsulation*? ☆☆

Topics: OOP

Answer:

Encapsulation is defined *as the process of enclosing one or more items within a physical or logical package*. Encapsulation, in object oriented programming methodology, prevents access to implementation details.

Q4: What is *Polymorphism*? ☆☆

Topics: OOP

Answer:

The word polymorphism means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as *one interface, multiple functions*.

Q5: What is a `class`? ☆☆

Topics: OOP

Answer:

A class describes all the attributes of objects, as well as the methods that implement the behavior of member objects. It is a comprehensive data type, which represents a blue print of objects. It is a template of object.

A class can be defined as the primary building block of OOP. It also serves as a template that describes the properties, state, and behaviors common to a particular group of objects.

A class contains data and behavior of an entity. For example, the aircraft class can contain data, such as model number, category, and color and behavior, such as duration of flight, speed, and number of passengers. A class inherits the data members and behaviors of other classes by extending from them.

Q6: What is an `object`? ☆☆

Topics: OOP

Answer:

Objeects are instance of classes. It is a basic unit of a system. An object is an entity that has attributes, behavior, and identity. Attributes and behavior of an object are defined by the class definition.

Q7: What is the relationship between a `class` and an `object`? ☆☆

Topics: OOP

Answer:

A class acts as a blue-print that defines the properties, states, and behaviors that are common to a number of objects. An object is an instance of the class. For example, you have a class called *Vehicle* and *Car* is the object of that class. You can create any number of objects for the class named *Vehicle*, such as *Van*, *Truck*, and *Auto*.

The *new* operator is used to create an object of a class. When an object of a class is instantiated, the system allocates memory for every data member that is present in the class.

Q8: Explain the basic features of OOPs ☆☆

Topics: OOP

Answer:

The following are the four basic features of OOP:

- **Abstraction** - Refers to the process of exposing only the relevant and essential data to the users without showing unnecessary information.
- **Polymorphism** - Allows you to use an entity in multiple forms.
- **Encapsulation** - Prevents the data from unwanted access by binding of code and data in a single unit called object.
- **Inheritance** - Promotes the reusability of code and eliminates the use of redundant code. It is the property through which a child class obtains all the features defined in its parent class. When a class inherits the

common properties of another class, the class inheriting the properties is called a derived class and the class that allows inheritance of its common properties is called a base class.

Q9: What is the difference between a `class` and a `structure`? ☆☆

Topics: OOP

Answer:

Class:

- A class is a reference type.
- While instantiating a class, CLR allocates memory for its instance in heap.
- Classes support inheritance.
- Variables of a class can be assigned as null.
- Class can contain constructor/destructor.

Structure:

- A structure is a value type.
- In structure, memory is allocated on stack.
- Structures do not support inheritance.
- Structure members cannot have null values.
- Structure does not require constructor/destructor and members can be initialiazed automatically.

Q10: Why is the `virtual` keyword used in code? ☆☆

Topics: OOP

Answer:

The `virtual` keyword is used while defining a class to specify that the methods and the properties of that class can be overridden in derived classes.

Q11: Explain the concept of *Constructor* ☆☆

Topics: OOP

Answer:

Constructor is a special method of a class, which is called automatically when the instance of a class is created. It is created with the same name as the class and initializes all class members, whenever you access the class. The main features of a constructor are as follows:

- Constructors do not have any return type.
- Constructors can be overloaded.
- It is not mandatory to declare a constructor; it is invoked automatically by .NET Framework.

Q12: Can you *inherit* `private` members of a class? ☆☆

Topics: OOP

Answer:

No, you cannot inherit `private` members of a class because `private` members are accessible only to that class and not outside that class.

Q13: What is the difference between *procedural* and *object-oriented* programming? ☆☆

Topics: OOP

Answer:

Procedural programming is based upon the modular approach in which the larger programs are broken into procedures. Each procedure is a set of instructions that are executed one after another. On the other hand, OOP is based upon objects. An object consists of various elements, such as methods and variables.

Access modifiers are not used in procedural programming, which implies that the entire data can be accessed freely anywhere in the program. In OOP, you can specify the scope of a particular data by using access modifiers - *public*, *private*, *internal*, *protected*, and *protected* internal.

Q14: What is the difference between *Interface* and *Abstract Class*? ☆☆☆

Topics: C# OOP

Answer:

There are some differences between **Abstract Class** and **Interface** which are listed below:

- interfaces can have no state or implementation
- a class that implements an interface must provide an implementation of all the methods of that interface
- abstract classes may contain state (data members) and/or implementation (methods)
- abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)
- interfaces may be multiple-inherited, abstract classes may not (this is probably the key concrete reason for interfaces to exist separately from abtract classes - they permit an implementation of multiple inheritance that removes many of the problems of general MI).

Consider using abstract classes if :

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields.

Consider using interfaces if :

1. You expect that unrelated classes would implement your interface. For example,many unrelated objects can implement `Serializable` interface.
2. You want to specify the behaviour of a particular data type, but not concerned about who implements its behaviour.
3. You want to take advantage of multiple inheritance of type.

Q15: What is the difference between *Virtual* method and *Abstract* method? ☆☆☆

Topics: C# OOP

Answer:

- A **Virtual method** must always have a default implementation. However, it can be overridden in the derived class, though not mandatory. It can be overridden using *override* keyword.
- An **Abstract method** does not have an implementation. It resides in the abstract class. It is mandatory that the derived class implements the abstract method. An *override* keyword is not necessary here though it can be used.

Q16: How is method `overriding` different from method `overloading`? ☆☆☆

Topics: OOP

Answer:

- **Overriding** involves the creation of two or more methods with the same name and same signature in different classes (one of them should be parent class and other should be child).
- **Overloading** is a concept of using a method at different places with same name and different signatures within the same class.

Q17: How can you prevent a class from *overriding* in C#? ☆☆☆

Topics: OOP C#

Answer:

You can prevent a class from overriding in C# by using the `sealed` keyword.

Q18: What are `abstract` classes? What are the distinct characteristics of an `abstract` class? ☆☆☆

Topics: OOP

Answer:

An abstract class is a class that cannot be instantiated and is always used as a base class.
The following are the characteristics of an abstract class:

- You cannot instantiate an abstract class directly. This implies that you cannot create an object of the abstract class; it must be inherited.
- You can have abstract as well as non-abstract members in an abstract class.
- You must declare at least one abstract method in the abstract class.
- An abstract class is always public.
- An abstract class is declared using the *abstract* keyword.

The basic purpose of an abstract class is to provide a common definition of the base class that multiple derived classes can share.

Q19: What is *Polymorphism*, what is it for, and how is it used? ☆☆☆

Topics: OOP

Answer:

Polymorphism describes a pattern in object oriented programming in which classes have different functionality while sharing a common interface.

The beauty of polymorphism is that the code working with the different classes does not need to know which class it is using since they're all used the same way. A real world analogy for polymorphism is a button. Everyone knows how to use a button: you simply apply pressure to it. What a button "does," however, depends on what it is connected to and the context in which it is used — but the result does not affect how it is used. If your boss tells you to press a button, you already have all the information needed to perform the task.

Q20: When should I use a `struct` instead of a `class`? ☆☆☆

Topics: OOP

Answer:

Do not define a structure unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (integer, double, and so on).
- It has an instance size smaller than 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.

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 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 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: 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.

Q13: 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.

Q14: Explain different types of *Inheritance* ☆☆☆☆

Topics: OOP

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 supports 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.

Q15: 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.

Q16: 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.

Q17: 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 dependent module. Loose Coupling is always preferred. Inversion of Control and dependency injections are some techniques for getting loose coupling in modules.

Q18: 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!"*.

Q19: 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.

Q20: 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**.

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

Topics: OOP

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 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.

Q3: 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

Q4: 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.

Q5: 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.

Q6: 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.

Q7: 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.

Q8: 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*.

Q9: 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.

Q10: 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

Q11: 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.

Q12: 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.

Q13: 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{}
```

Q14: 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.

Q15: 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`88.

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.

Q16: 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.

Q17: 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.