OOAD

What is Liskov Substitution principle?

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

What the LSP indicates is that subtype behavior should match base type behavior as defined in the base type specification. If the rectangle base type spec says that height and width can be set independently, then LSP says that square cannot be a subtype of rectangle. If the rectangle spec says that a rectangle is immutable, then a square can be a subtype of rectangle. It's all about subtypes maintaining the behavior specified for the base type

If a base class satisfies a contract, then by LSP all the derived classes must satisify LSP.

<http://stackoverflow.com/questions/9444818/how-to-create-the-perfect-oop-application>

<http://en.wikibooks.org/wiki/Relational_Database_Design>

<http://en.wikibooks.org/wiki/C%2B%2B_Programming/Chapter_Advanced_Features_Summary>

What is OO?

The important relationships in OO are:

**Has A: (**aggregation, composition)

Class declares the object in its declaration. Class determines the lifetime of the object.

Composition: lifetime of member object depends on object that has it

Aggregation: lifetime of member object depends on "new" and "deletion" from the object that has it.

**Knows A:**

Class declares an object in its declaration, class doesn't determine the lifetime of the object. The object will be passed to this class via constructor of a set method

**Uses a:**

Class will create an object temporarily and either pass it off to a client or end its lifetime. It could also be an object passed to the class that it uses temporarily in a function. No member variable to save the object in the class

**Is a:**

Class derives another class

Beginner advice:

Nouns are objects

Verbs are methods

Adjectives are members.

The Open closed principle states that software entities should be open for extension, but closed for modification

What does it mean and why is it important for good OOAD

It is important to support future unforeseen changes without rewriting its code.

It means that you should put new code in new classes/modules. Existing code should be modified only for bug fixing. New classes can reuse existing code via inheritance.

Open/closed principle is intended to mitigate risk when introducing new functionality. Since you don't modify existing code you can be assured that it wouldn't be broken. It reduce maintenance cost and increase product stability.

What are the SOLID Principles in OOAD

* [The Single Responsibility Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=2&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Fsrp.pdf&ei=1C4HSu3uEODJtgfs8uWeBw&usg=AFQjCNHQQ1Aw-2yCciEbERpJn3VdHBCmQw&sig2=Pf7Z4GNwRnAiaar%5FyCk%5Fdg): A class (or method) should have only one reason to change.
* [The Open Closed Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Focp.pdf&ei=9i4HStWBH5XhtgfSltWNBw&usg=AFQjCNGzBuNcaA1IXUx0tijZoTW7rlzcRQ&sig2=K-zyAaPRTp0CG28mtuO9jg): A class (or method) should be open for extension and closed for modification.
* [The Liskov Substitution Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Flsp.pdf&ei=By8HSoDJJd6ptger2uSdBw&usg=AFQjCNFnNI0DmzofjWDQEGILAT-W1L8Mtw&sig2=ozd86zYF9ZMqLfBS2tQJdg): Subtypes must be substitutable for their base types.
* [The Interface Segregation Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Fisp.pdf&ei=Gy8HSpH-CqK%5Ftwfv6OCXBw&usg=AFQjCNHuKx3fPObvzYaQDZ2etcZHYtg57g&sig2=aWFqTuyl%5FiCAsGk8SnwljQ): Clients should not be forced to depend upon methods that they do not use. Interfaces should belong to clients.
* [The Dependency Inversion Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Fdip.pdf&ei=Mi8HSsbyLdKJtgemroCfBw&usg=AFQjCNEpBGziZw6APHj0rMG9pp1LJt9FHA&sig2=y4EFfBIw-cC6hocfVEeTJg): Abstractions should not depend on details. Details should depend on abstractions.

As a software engineer, I have a strong bias towards writing business logic in the application layer, while typically relying on the database for little more than CRUD (Create Retrieve Update and Delete) operations. On the other hand, I have run across applications (typically older ones) where a large amount of the business logic was written in stored procedures, so there are people out there that prefer to write business logic in the database layer.

For the people that have and/or enjoy written/writing business logic in a stored procedure, what were/are your reasons for using this method?

Two good reasons for putting the business logic in the database are:

* It secures your logic and data against additional applications that may access the database that don't implement similar logic.
* Database designs usually outlive the application layer and it reduces the work necessary when you move to new technologies on the client side.

I try to seriously limit my business logic in the DB to only procs that have to do alot of querying and updating to perform a single application operation. Some may argue that even that should be in the app, but I like to keep the IO down if I can.

Databases are great for CRUD but if they get bloated with logic:

1. It becomes confusing where the logic is,
2. Typically databases are a silo and do not scale horizontally nearly as well as the app servers.
3. t\_sql/PLsql is hard to read and procedural in nature
4. You forfeit all of the benefits of OOAD.

Limiting the business logic to the application layer is short-sighted at best. Experienced professional database designers rarely allow it on their systems. Database need to have constraints and triggers and stored procs to help define how the data from any source will go into it.

If the database is to maintain its integrity and to ensure that all sources of new data or data changes follow the rules, the database is the place to put the required logic. Putting it the application layer is a data nightmare waiting to happen. Databases do not get information just from one application. Business logic in the application is often unintentionally bypassed by imports (assume you got a new customer who wanted their old historical data imported to your system or a large number of target records, no one is going to enter a million possible targets through the interface, it will happen in an import.) It is also bypassed by changed made through the query window to fix one-time issues (things like increasing the price of all products by 10%). If you have application layer logic that should have been applied to the data change, it won't be. Now it's ok to put it in the application layer as well, no sense sending bad data to the database and wasting network bandwidth, but to fail to put it in the database will sooner or later cause data problems.

Another reason to keep all of this in the database has to to with the possibility of users committing fraud. If you put all your logic in the application layer, then you must grant the users access directly to the tables. If you encapsulate all your logic in stored procs, they can be limited to doing only what the stored procs allow and not anything else. I would not consider allowing any kind of access by users to a database that stores financial records or personal information (such as health records) as I would not allow anyone except a couple of dbas to directly access the production records in any way shape or form. More fraud is committed than many developers realize and almost none of them consider the possibility in their design.

If you need to import large amount of data, going through a data access layer could slow down the import to a crawl becasue it doesn't take advanatge of the set-based operations that databases are designed to handle.

I think Specially for older applications which i working on (Banking) where the Bussiness logic is huge, it's almost next to impossible to perform all these business logic in application layer, and also It's a big performenance hit when we put these logic in Application layer where the number of fetch to the database is more, results in more resource utilization(more java objects if it's done in java layer) and network issues and forget abt performenance.

So there are many ways of structuring objects (I'm talking of OOP here). For the question, I will use the classic "Car" example of OOP. Basically, How do I know when to make the car an object, or the wheel of a car an object, when both program structures would accomplish the goal?

How do I classify and categorize the parts of an object to determine whether or not they are better suited as simple attributes or variables of an object, or if they really need to be an object themselves?

Well the first thing you have to realize is the OOAD ("Object-oriented analysis and design") is a tool and not a means to an end. What you get out of that process is a model, not a true representation of what you're modelling. That model makes certain assumptions. The purpose of that model is to solve a problem you have.

So how do you know how to design objects? How do you know if you've done it right? By the end result: has it solved your problem?

So, for the Car example, in some models a car count could simply be an integer count, for example the car traffic through an intersection in a traffic model. In such a model rarely do you care about the make, model or construction of cars, just the number. You might care about the type of vehicle to the point of is it a truck or car (for example). Do you model that as a Vehicle object with a type of Car or Truck? Or just separate carCount and truckCount tallies?

The short answer is: whichever works best.

The normal test for something being an object or not is does it have behaviour?

Remember that ultimately objects = data + behaviour.

So you might say that cars have the following state:

# of wheels;

Height of suspension;

Left or right drive;

Colour;

Width;

Weight;

Length;

Height;

# of doors;

Whether it has a sunroof;

Whether it has a stereo, CD player, MP3 player and/or satnav;

Size of the petrol tank;

Number of cylinders;

# of turbo charges and/or fuel injection;

Maximum torque;

Maximum brake-horsepower;

and so on.

Chances are you'll only care about a small subset of that: pick whatever is relevant. A racing game might go into more detail about the wheels, such as how hot they are, how worn, the width and tread type and so on. In such a case, a Wheel object could be said to be a collection of all that state (but little behaviour) because a Car has a number of Wheels and the Wheels are interchangeable.

So that brings up the second point about objects: an object can exist because of a relationship such that the object represents a complete set of data. So a Wheel could have tread, width, temperature and so on. You can't divide that up and say a Car has tread but no wheel width so it makes sense for Wheel to be an object since a Wheel in it's entirety is interchangeable.

But again, does that make sense for what're doing? That's the key question.

What is the difference between composition and aggregation? can anybody give me a sample of this OOAD?

Both aggregation and composition are special kinds of associations. Aggregation is used to represent ownership or a whole/part relationship, and composition is used to represent an even stronger form of ownership. With composition, we get coincident lifetime of part with the whole. The composite object has sole responsibility for the disposition of its parts in terms of creation and destruction. In implementation terms, the composite is responsible for memory allocation and deallocation

Composition can be used to model by-value aggregation, which is semantically equivalent to an attribute. In fact, composition was originally called aggregation-by-valuein an earlier UML draft, with “normal” aggregation being thought of asaggregation-by-reference. The definitions have changed slightly, but the general ideas still apply. The distinction between aggregation and composition is more of a design concept and is not usually relevant during analysis.

Consider a student, the student's brain, and the school the student attends.

The brain *is a part of* the student. If the student is destroyed, so is the brain. This is **composition**.

The student *has a* school. The student survives the school's destruction, and vice versa. This is**aggregation**

What are the steps that you follow when approaching a problem statement, which needs to be converted in to a OO design.

I would suggest following steps. 1. Write use case. It will help you to understand problem, it should contain basic execution flow (correct path) and alternate paths. 2. Do linguistic analysis and find out class and methods. (This is optional if you know any other good method then use it.) 3. Use SOLID principal for designing your classes. 4. Keep encapsulation, inheritance, polymorphism in mind. If you can't remember them write it on paper or stick it on your desk. 5. One more thing you should keep in mind "What varies encapsulate it". 6. Use design patterns when they are required. Don't enforce design pattern to your code. Try to map your design problem to any design pattern.

When should I use an interface and when should I use a base class?

Should it always be an interface if I don't want to actually define a base implementation of the methods?

If I have a Dog and Cat class. Why would I want to implement IPet instead of PetBase? I can understand having interfaces for ISheds or IBarks (IMakesNoise?), because those can be placed on a pet by pet basis, but I don't understand which to use for a generic Pet.

Let's take your example of a Dog and a Cat class, and let's illustrate using C#:

Both a dog and a cat are animals, specifically, quadruped mammals (animals are waaay too general). Let us assume that you have an abstract class Mammal, for both of them:

public abstract class Mammal

This base class will probably have default methods such as:

* Hunt
* Feed
* Mate

All of which are behavior that have more or less the same implementation between either species. To define this you will have:

public class Dog : Mammal

public class Cat : Mammal

Now let's suppose there are other mammals, which we will usually see in a zoo:

public class Giraffe : Mammal

public class Rhinoceros : Mammal

public class Hippopotamus : Mammal

This will still be valid because at the core of the functionality, Hunt(), Feed() and Mate() will still be the same.

However, giraffes, rhinoceros, and hippos are not exactly animals that you can make pets out of. That's where an interface will be useful:

public interface IPettable

{

IList<Trick> Tricks{get; set;}

void Bathe();

void Train(Trick t);

}

The implementation for the above contract will not be the same between a cat and dog; putting their implementations in an abstract class to inherit will be a bad idea.

Your Dog and Cat definitions should now look like:

public class Dog : Mammal, IPettable

public class Cat : Mammal, IPettable

Theoretically you can override them from a higher base class, but essentially an interface allows you to add on only the things you need into a class without the need for inheritance.

Consequently, because you can usually only inherit from one abstract class (in most statically typed OO languages that is... exceptions include C++) but be able to implement multiple interfaces, it allows you to construct objects in a strictly *as required* basis.

In general, you should favor interfaces over abstract classes. One reason to use an abstract class is if you have common implementation among concrete classes. Of course, you should still declare an interface (IPet) and have an abstract class (PetBase) implement that interface.Using small, distinct interfaces, you can use multiples to further improve flexibility. Interfaces allow the maximum amount of flexibility and portability of types across boundaries. When passing references across boundaries, always pass the interface and not the concrete type. This allows the receiving end to determine concrete implementation and provides maximum flexibility. This is absolutely true when programming in a TDD/BDD fashion.

The Gang of Four stated in their book "Because inheritance exposes a subclass to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation". I believe this to be true.

Interfaces and base classes represent two different forms of relationships.

**Inheritance** (base classes) represent an "is-a" relationship. E.g. a dog or a cat "is-a" pet. This relationship always represents the (single) **purpose** of the class (in conjunction with the "single responsibility principle"[1]).

**Interfaces**, on the other hand, represent **additional features** of a class. I'd call it an "is" relationship, like in "Foo is disposable", hence the IDisposable interface in C#.

Also well explained in the <http://www.javaworld.com/javaworld/javaqa/2001-04/03-qa-0420-abstract.html> atrciule

Why prefer composition over inheritance? What trade-offs are there for each approach? When should you choose inheritance over composition?

*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 Class A.  
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 only 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.

Inheritance is easily overused, though, and creates additional complexity, with hard dependencies between classes. Also understanding what happens during execution of a program gets pretty hard due to layers and dynamic selection of method calls.

I would suggest using composing as the default. It is more modular, and gives the benefit of late binding (you can change the component dynamically). Also it's easier to test the things separately. And if you need to use a method from a class, you are not forced to be of certain form (Liskov Substitution Principle).

With all the undeniable benefits provided by inheritance, here's some of its disadvantages.

**Disadvantages of Inheritance:**

1. You can't change the implementation inherited from super classes at runtime (obviously because inheritance is defined at compile time).
2. Inheritance exposes a subclass to details of its parent's class implementation, that's whey it's often said that inheritance breaks encapsulation (in a sense that you really need to focus on interfaces only not implementation, so reusing by sub classing is not always preferred).
3. The tight coupling provided by inheritance makes the implementation of a subclass very bound up with the implementation of a super class that any change in the parent implementation will force the sub class to change.
4. Excessive reusing by sub-classing can make the inheritance stack very deep and very confusing too.

On the other hand **Object composition** is defined at runtime through objects acquiring references to other objects. In such a case these objects will never be able to reach each-other's protected data (no encapsulation break) and will be forced to respect each other's interface. And in this case also, implementation dependencies will be a lot less than in case of inheritance.

I think inheritance should be used if your answer is an affirmative to any of these questions.

* Is your class part of a structure that benefits from polymorphism ? For example, if you had a Shape class, which declares a method called draw(), then we clearly need Circle and Square classes to be subclasses of Shape, so that their client classes would depend on Shape and not on specific subclasses.
* Does your class need to re-use any high level interactions defined in another class ? The [template method](http://en.wikipedia.org/wiki/Template_method_pattern) design pattern would be impossible to implement without inheritance. I believe all extensible frameworks use this pattern.

However, if your intention is purely that of code re-use, then composition most likely is a better design choice.

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

<http://pragprog.com/articles/tell-dont-ask>

**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 effecting 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.

There are reasons to use getters and setters, but if those reasons don't exist, making getter/setter pairs in the name of false encapsulation gods is not a good thing. Valid reasons to make getters or setters include the things often mentioned as the potential changes you can make later, like validation or different internal representations. Or maybe the value should be readable by clients but not writable (for example, reading the size of a dictionary), so a simple getter is a nice choice. But those reasons should be there when you make the choice, and not just as a potential thing you may want later. This is an instance of YAGNI (You Ain't Gonna Need It).

You've insulated your public interface from changes under the sheets - if you were designing an interface and weren't sure whether direct access to something was OK, then you should have kept designing.

 think the crucial argument is that, *"if you were designing an interface and weren't sure whether direct access to something was OK, then you should have kept designing."* That is the most important problem with getters/setters: They reduce a class to a mere container of (more or less) public fields. In *real* OOP, however, an object is more than a container of data fields. It encapsulates state and algorithms to manipulate that state. What's crucial about this statement is that the state is supposed to be *encapsulated* and only to be manipulated by the algorithms provided by the object.

<http://typicalprogrammer.com/?p=23>

<http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html?page=1>

When should you use structures and when should you use classes

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.

MSDN has the answer: [Choosing Between Classes and Structures](http://msdn.microsoft.com/en-us/library/ms229017.aspx).

1. Is the main responsability of the type data storage?
2. Is its public interface defined entirely by properties that access or modify its data members?
3. Are you sure your type will never have subclasses?
4. Are you sure your type will never be treated polymorphically?

If you answer 'yes' to all 4 questions: use a struct. Otherwise, use a class.

Difference between abstract class and interface

<http://codeofdoom.com/wordpress/2009/02/12/learn-this-when-to-use-an-abstract-class-and-an-interface/>

When we talk about abstract classes we are defining characteristics of an object type, specifying **what an object is** but in the case of an interface we define a capability and we bond to provide that capability, we are talking about establishing a contract about **what the object can do.**

An abstract class can have shared state or functionality. An interface is only a promise to provide the state or functionality. A good abstract class will reduce the amount of code that has to be rewritten because it's functionality or state can be shared. The interface has no defined information to be shared

--

Use an abstract class if you have some functionality that you want it's subclasses to have. For instance, if you have a set of functions that you want all of the base abstract class's subclasses to have.

Use an interface if you just want a general contract on behavior/functionality. If you have a function or object that you want to take in a set of different objects, use an interface. Then you can change out the object that is passed in, without changing the method or object that is taking it.

Interfaces are typically loose, compared to Abstract classes. You wouldn't want to use interfaces in a situation where you are constantly writing the same code for all of the interface's methods. Use an abstract class and define each method once.

Also, if you are trying to create a specific object inheritance hierarchy, you really wouldn't want to try to do that with just interfaces.

Also, again, in some languages you can only have a single base class, and if an object already has a base class, you are going to have to do some refactoring in order to use an abstract base class. This may or may not mean that you might want to use an inteface instead.

As @tvanfosson notes, it's not a bad idea to use a lot of interfaces, when you really understand abstract classes and interfaces, it's not really an either/or situation. A particular situation could use both abstract classes and interfaces or neither. I like to use interfaces sometimes simply to restrict what a method or object can access on a passed in parameter object.

--

In .NET (similar for Java):

* 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 itslef)
* 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).

As general OO terms, the differences are not necessarily well-defined. For example, there are C++ programmers who may hold similar rigid definitions (interfaces are a strict subset of abstract classes that cannot contain implementation), while some may say that an abstract class with some default implementations is still an interface or that a non-abstract class can still define an interface.

Indeed, there is a C++ idiom called the Non-Virtual Interface (NVI) where the public methods are non-virtual methods that 'thunk' to private virtual methods:

* <http://www.gotw.ca/publications/mill18.htm>
* <http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Non-Virtual_Interface>

I think a key point in the explanation Michael provided is that when implementing an interface you MUST implement all members in the interface, but when inheriting from an abstract class it's NOT REQUIRED by a child class to implement its parent's members

How about an analogy: when I was in the Air Force, I went to pilot training and became a USAF pilot. At that point I wasn't qualified to fly anything, and had to attend aircraft type training. Once I qualified, I was a pilot (Abstract class) and a C-141 pilot (concrete class). At one of my assignments, I was given an additional duty: Safety Officer. Now I was still a pilot and a C-141 pilot, but I also performed Safety Officer duties (I implemented ISafetyOfficer, so to speak). A pilot wasn't required to be a safety officer, other people could have done it as well.

All USAF pilots have to follow certain Air Force-wide regulations, and all C-141 (or F-16, or T-38) pilots 'are' USAF pilots. Anyone can be a safety officer. So, to summarize:

* Pilot: abstract class
* C-141 Pilot: concrete class
* Safety Officer: interface

There are a couple of other differences -

Interfaces can't have any concrete implementations. Abstract base classes can. This allows you to provide concrete implementations there. This can allow an abstract base class to actually provide a more rigorous contract, wheras an interface really only describes how a class is used. (The abstract base class can have non-virtual members defining the behavior, which gives more control to the base class author.)

More than one interface can be implemented on a class. A class can only derive from a single abstract base class. This allows for polymorphic hierarchy using interfaces, but not abstract base classes. This also allows for a pseudo-multi-inheritance using interfaces.

Can you write object oriented code in C? Especially with regard to polymorphism.

I would advise against preprocessor (ab)use to try and make C syntax more like that of another more object-oriented language. At the most basic level, you just use plain structs as objects and pass them around by pointers:

struct monkey

{

float age;

bool is\_male;

int happiness;

};

void monkey\_dance(struct monkey \*monkey)

{

/\* do a little dance \*/

}

To get things like inheritance and polymorphism, you have to work a little harder. You can do manual inheritance by having the first member of a structure be an instance of the superclass, and then you can cast around pointers to base and derived classes freely:

struct base

{

/\* base class members \*/

};

struct derived

{

struct base super;

/\* derived class members \*/

};

struct derived d;

struct base \*base\_ptr = (struct base \*)&d; // upcast

struct derived derived\_ptr = (struct derived \*)base\_ptr; // downcast

To get polymorphism (i.e. virtual functions), you use function pointers, and optionally function pointer tables, also known as virtual tables or vtables:

struct base;

struct base\_vtable

{

void (\*dance)(struct base \*);

void (\*jump)(struct base \*, int how\_high);

};

struct base

{

struct base\_vtable \*vtable;

/\* base members \*/

};

void base\_dance(struct base \*b)

{

b->vtable->dance(b);

}

void base\_jump(struct base \*b, int how\_high)

{

b->vtable->jump(b, how\_high);

}

struct derived1

{

struct base super;

/\* derived1 members \*/

};

void derived1\_dance(struct derived1 \*d)

{

/\* implementation of derived1's dance function \*/

}

And that's how you do polymorphism in C. It ain't pretty, but it does the job. There are some sticky issues involving pointer casts between base and derived classes, which are safe as long as the base class is the first member of the derived class. Multiple inheritance is much harder - in that case, in order to case between base classes other than the first, you need to manually adjust your pointers based on the proper offsets, which is really tricky and error-prone.

Another (tricky) thing you can do is change the dynamic type of an object at runtime! You just reassign it a new vtable pointer. You can even selectively change some of the virtual functions while keeping others, creating new hybrid types. Just be careful to create a new vtable instead of modifying the global vtable, otherwise you'll accidentally affect all objects of a given type.

What does it mean program to an interface?

There are some wonderful answers on here to this questions that get into all sorts of great detail about interfaces and loosely coupling code, inversion of control and so on. There are some fairly heady discussions, so I'd like to take the opportunity to break things down a bit for understanding why an interface is useful.

When I first started getting exposed to interfaces, I too was confused about their relevance. I didn't understand why you needed them. If we're using a language like Java or C#, we already have inheritance and I viewed interfaces as a *weaker* form of inheritance and thought, "why bother?" In a sense I was right, you can think of interfaces as sort of a weak form of inheritance, but beyond that I finally understood their use as a language construct by thinking of them as means of classifying common traits or behaviors that were exhibited by potentially many non-related classes of objects.

For example -- say you have a SIM game and have the following classes:

class HouseFly inherits Insect {

void FlyAroundYourHead();

void LandOnThings();

}

class Telemarketer inherits Person {

void CallDuringDinner();

void ContinueTalkingWhenYouSayNo();

}

Clearly, these two objects have nothing in common in terms of direct inheritance. But, you could say they are both annoying.

Let's say our game needs to have some sort of random *thing* that annoys the game player when they eat dinner. This could be a HouseFly or a Telemarketer or both -- but how do you allow for both with a single function? And how do you ask each different type of object to "do their annoying thing" in the same way?

The key to realize is that both a Telemarketer and HouseFly share a common loosely interpreted behavior even though they are nothing alike in terms of modeling them. So, let's make an interface that both can implement:

interface IPest {

void BeAnnoying();

}

class HouseFly inherits Insect implements IPest {

void FlyAroundYourHead();

void LandOnThings();

void BeAnnoying() {

FlyAroundYourHead();

LandOnThings();

}

}

class Telemarketer inherits Person implements IPest {

void CallDuringDinner();

void ContinueTalkingWhenYouSayNo();

void BeAnnoying() {

CallDuringDinner();

ContinueTalkingWhenYouSayNo();

}

}

We now have two classes that can each be annoying in their own way. And they do not need to derive from the same base class and share common inherent characteristics -- they simply need to satisfy the contract of *IPest* -- that contract is simple. You just have to *BeAnnoying*. In this regard, we can model the following:

class DiningRoom {

DiningRoom(Person[] diningPeople, IPest[] pests) { ... }

void ServeDinner() {

when diningPeople are eating,

foreach pest in pests

pest.BeAnnoying();

}

}

Here we have a dining room that accepts a number of diners and a number of pests -- note the use of the interface. This means that in our little world, a memeber of the *pests* array could actually be a Telemarketer object or a HouseFly object.

The *ServeDinner* method is called when dinner is served and our people in the dining room are supposed to eat. In our little game, that's when our pests do their work -- each pest is instructed to be annoying by way of the IPest interface. In this way, we can easily have both Telemarketers and HouseFlys be annoying in each of their own ways -- we care only that we have something in the DiningRoom object that is a pest, we don't really care what it is and they could have nothing in common with other.

This very contrived pseudo-code example (that dragged on a lot longer than I anticipated) is simply meant to illustrate the kind of thing that finally turned the light on for me in terms of when we might use an interface. I apologize in advance for the silliness of the example, but hope that it helps in your understanding. And, to be sure, the other posted answers you've received here really cover the gamut of the use of interfaces today in design patterns and development methodologies.

The specific example I used to give to students is that they should write

List myList = new ArrayList(); // programming to the List interface

instead of

ArrayList myList = new ArrayList(); // this is bad

These look exactly the same in a short program, but if you go on to use myList 100 times in your program you can start to see a difference. The first declaration ensures that you only call methods onmyList that are defined by the List interface (so no ArrayList specific methods). If you've programmed to the interface this way, later on you can decide that you really need

List myList = new TreeList();

and you only have to change your code in that one spot. You already know that the rest of your code doesn't do anything that will be broken by changing the *implementation* because you programmed to the*interface*.

As far as I can tell, in spite of the countless millions or billions spent on OOP education, languages, and tools, OOP has not improved developer productivity or software reliability, nor has it reduced development costs. Few people use OOP in any rigorous sense (few people adhere to or understand principles such as LSP); there seems to be little uniformity or consistency to the approaches that people take to modelling problem domains. All too often, the class is used simply for its syntactic sugar; it puts the functions for a record type into their own little namespace.

I've written a large amount of code for a wide variety of applications. Although there have been places where true substitutable subtyping played a valuable role in the application, these have been pretty exceptional. In general, though much lip service is given to talk of "re-use" the reality is that unless a piece of code does *exactly* what you want it to do, there's very little cost-effective "re-use". It's extremely hard to design classes to be extensible *in the right way*, and so the cost of extension is normally so great that "re-use" simply isn't worthwhile.

In many regards, this doesn't surprise me. The real world isn't "OO", and the idea implicit in OO--that we can model things with some class taxonomy--seems to me very fundamentally flawed (I can sit on a table, a tree stump, a car bonnet, someone's lap--but not one of those is-a chair). Even if we move to more abstract domains, OO modelling is often difficult, counterintuitive, and ultimately unhelpful (consider the classic examples of circles/ellipses or squares/rectangles).

So what am I missing here? Where's the value of OOP, and why has all the time and money failed to make software any better?

While this is true and has been observed by other people (take Stepanov, inventor of the STL), the rest is nonsense. OOP may be flawed and it certainly is no silver bullet but it makes large-scale applications much simpler because it's a great way to reduce dependencies. Of course, this is only true for “good” OOP design. Sloppy design won't give any advantage. But good, decoupled design can be modelled very well using OOP and not well using other techniques.

There are much better, more universal models ([Haskell's type model](http://stackoverflow.com/questions/16770/haskells-algebraic-data-types) comes to mind) but these are also often more complicated and/or difficult to implement efficiently. OOP is a good trade-off between extremes.

--

Reuse shouldn't be a goal of OOP - or any other paradigm for that matter.

Reuse is a side-effect of an good design and proper level of abstraction. Code achieves reuse by doing something useful, but not doing so much as to make it inflexible. It does not matter whether the code is OO or not - we reuse what works and is not trivial to do ourselves. That's pragmatism.

The thought of OO as a new way to get to reuse through inheritance is fundamentally flawed. As you note the LSP violations abound. Instead, OO is properly thought of as a method of managing the complexity of a problem domain. The goal is maintainability of a system over time. The primary tool for achieving this is the separation of public interface from a private implementation. This allows us to have rules like "This should only be modified using ..." enforced by the compiler, rather than code review.

Using this, I'm sure you will agree, allows us to create and maintain hugely complex systems. There is lots of value in that, and it is not easy to do in other paradigms

**--**

Relative to straight procedural programming, the first fundamental tenet of OOP is the notion of information hiding and encapsulation. This idea leads to the notion of the ***class*** that seperates the interface from implementation. These are hugely important concepts and the basis for putting a framework in place to think about program design in a different way and better (I think) way. You can't really argue against those properties - there is no trade-off made and it is always a cleaner way to modulize things.

Other aspects of OOP including inheritance and polymorphism are important too, but as others have alluded to, those are commonly over used. ie: Sometimes people use inheritance and/or polymorphism because they can, not because they should have. They are powerful concepts and very useful, but need to be used wisely and are not automatic winning advantages of OOP.

Relative to re-use. I agree re-use is over sold for OOP. It is a possible side effect of well defined objects, typically of more primitive/generic classes and is a direct result of the encapsulation and information hiding concepts. It is potentially easier to be re-used because the interfaces of well defined classes are just simply clearer and somewhat self documenting.

**Difference between abstract function and virtual function**

An abstract function can have no 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.

# [Are Getters and Setters evil](http://stackoverflow.com/questions/565095/are-getters-and-setters-evil)

There is also the point of view that most of the time, using setters still breaks encapsulation by allowing you to set values that are meaningless. As a very obvious example, if you have a score counter on the game that only ever goes up, instead of

// Game

private int score;

public void setScore(int score) { this.score = score; }

public int getScore() { return score; }

// Usage

game.setScore(game.getScore() + ENEMY\_DESTROYED\_SCORE);

it should be

// Game

private int score;

public int getScore() { return score; }

public void addScore(int delta) { score += delta; }

// Usage

game.addScore(ENEMY\_DESTROYED\_SCORE);

This is perhaps a bit of a facile example. What I'm trying to say is that discussing getter/setters vs public fields often obscures bigger problems with objects manipulating each others' internal state in an intimate manner and hence being too closely coupled.

The idea is to make methods that directly do things you want to do. An example would be how to set enemies' "alive" status. You might be tempted to have a setAlive(boolean alive) method. Instead you should have:

private boolean alive = true;

public boolean isAlive() { return alive; }

public void kill() { alive = false; }

The reason for this is that if you change the implementation that things no longer have an "alive" boolean but rather a "hit points" value, you can change that around without breaking the contract of the two methods you wrote earlier:

private int hp; // Set in constructor.

public boolean isAlive() { return hp > 0; } // Same method signature.

public void kill() { hp = 0; } // Same method signature.

public void damage(int damage) { hp -= damage; }

# [When should you use 'friend' in C++?](http://stackoverflow.com/questions/17434/when-should-you-use-friend-in-c)

The 'friend' specifier allows the designated class access to protected data or functionality within the class making the friend statement. For example in the below code anyone may ask a child for their name, but only the mother and the child may change the name.

You can take this simple example further by considering a more complex class such as a Window. Quite likely a Window will have many function/data elements that should not be publicly accessible, but ARE needed by a related class such as a WindowManager.

class Child

{

friend class Mother;

public:

string name( void );

protected:

void setName( string newName );

};

As an extra note, the C++ FAQ mentions that friend *enhances* encapsulation. friend grants *selective access* to members, just like protected does. Any fine-grained control is better than granting public access. Other languages define selective access mechanisms too, consider C#'s internal. Most negative criticism around the use of friend is related to tighter coupling, which is generally seen as a bad thing. However, in some cases, tighter coupling is precisely what you want and friend gives you that power.

--

At work we use friends extensively for out testing code. It means we can provide proper encapsulation and information hiding for the main application code. But also we can have seperate test code that uses friends to inspect internal state and data for testing.

Suffice to say I wouldn't use the friend keyword as an essential component of your design.

Encapsulation is not broken here because the class itself dictates who can access its private members. Encapsulation would only be broken if this could be caused from outside the class, e.g. if youroperator << would proclaim “I'm a friend of class foo.”

friend replaces use of public, not use of private!

The friend keyword has a number of good uses. Here are the two uses immediately visible to me:

## Friend Definition

Friend definition allows to define a function in class-scope, but the function will not be defined as a member function, but as a free function of the enclosing namespace, and won't be visible normally except for argument dependent lookup. That makes it especially useful for operator overloading:

namespace utils {

class f {

private:

typedef int int\_type;

int\_type value;

public:

// let's assume it doesn't only need .value, but some

// internal stuff.

friend f operator+(f const& a, f const& b) {

// name resolution finds names in class-scope.

// int\_type is visible here.

return f(a.value + b.value);

}

int getValue() const { return value; }

};

}

int main() {

utils::f a, b;

std::cout << (a + b).getValue(); // valid

}

## Private CRTP Base Class

Sometimes, you find the need that a policy needs access to the derived class:

// possible policy used for flexible-class.

template<typename Derived>

struct Policy {

void doSomething() {

// casting this to Derived\* requires us to see that we are a

// base-class of Derived.

some\_type const& t = static\_cast<Derived\*>(this)->getSomething();

}

};

// note, derived privately

template<template<typename> class SomePolicy>

struct FlexibleClass : private SomePolicy<FlexibleClass> {

// we derive privately, so the base-class wouldn't notice that,

// (even though it's the base itself!), so we need a friend declaration

// to make the base a friend of us.

friend class SomePolicy<FlexibleClass>;

void doStuff() {

// calls doSomething of the policy

this->doSomething();

}

// will return useful information

some\_type getSomething();

};

You will find a non-contrived example for that in [this](http://stackoverflow.com/questions/356294/is-partial-class-template-specialization-the-answer-to-this-design-problem#356576) answer. Another code using that is in [this](http://stackoverflow.com/questions/286402/initializing-struct-using-an-array#287353) answer. The CRTP base casts its this pointer, to be able to access data-fields of the derived class using data-member-pointers.

# [Functional programming vs Object Oriented programming](http://stackoverflow.com/questions/2078978/functional-programming-vs-object-oriented-programming)

When you anticipate a different kind of software evolution:

* Object-oriented languages are good when you have a fixed set of operations on things, and as your code evolves, you primarily add new things. This can be accomplished by adding new classes which implement existing methods, and the existing classes are left alone.
* Functional languages are good when you have a fixed set of things, and as your code evolves, you primarily add new operations on existing things. This can be accomplished by adding new functions which compute with existing data types, and the existing functions are left alone.

When evolution goes the wrong way, you have problems:

* Adding a new operation to an object-oriented program may require editing many class definitions to add a new method.
* Adding a new kind of thing to a functional program may require editing many function definitions to add a new case.

This problem has been well known for many years; in 1998, [Phil Wadler dubbed it the "expression problem"](http://www.daimi.au.dk/~madst/tool/papers/expression.txt). Although some researchers think that the expression problem can be addressed with such language features as mixins, a widely accepted solution has yet to hit the mainstream.

What are the typical problem definitions where functional programming is a better choice?

Functional languages excel at manipulating symbolic data in tree form. A favorite example is compilers, where source and intermediate languages change seldom (mostly the same things), but compiler writers are always adding new translations and code improvements or optimizations (new operations on things). Compilation and translation more generally are "killer apps" for functional languages.

--

In a purely procedural style, data tends to be highly decoupled from the functions that operate on it.

In an object oriented style, data tends to carry with it a collection of functions.

In a functional style, data and functions tend toward having more in common with each other (as in Lisp and Scheme) while offering more flexibility in terms of how functions are actually used. Algorithms tend also to be defined in terms of recursion and composition rather than loops and iteration.

# [Object Oriented vs Relational Databases](http://stackoverflow.com/questions/800/object-oriented-vs-relational-databases)

**Why we abandoned Object Databases**

A while back, I was part of a Solaris/C++ project that used the Objectivity object database. We eventually switched the project over to Sybase. This was about 10 years ago, so I'm sure a lot has changed since then, but a few of the observations still apply.

The application was a carrier-class telecom system.

* The **basic functionality was nice**. You reference an object and it magically pops into memory. If your programming problem boils down to having a large graph of objects (as opposed to tabular data) it is a definite win.
* There were some **stability-related "early adopter" problems**. Seeing as the company is still in business it's safe to assume they are fixed.
* We wanted to allow the **client to define their own schema**. This was a huge problem, since you give the object database a specification and it spits out a header file which defines your object. This is definitely not conducive to post-compiling schema definition, which is a definite strength of SQL databases.
* The customer perceived it as being a **new, experimental technology**. If you're familiar with the telecom world, you know that's not a recommendation.
* SQL databases have **tons of tools for working with schemas**, database administration, UI generation, backup, etc. We were having to write every little piece of that ourselves.
* I had a bad experience working with some of the early-adopter objectologists in that group. If you're familiar with the DailyWTF, these were the True,False,FileNotFound guys.

While attempting to address the client customization part, our lead objectologist came up with some data classes called Rows and Columns. That was when it became apparent that for our particular application we were just a whole lot better off going with a relational database.

Anyways, that was my experience with Object DBs, I would love to hear some other (hopefully happier!) experiences with them.

# [Inheritance vs. Aggregation](http://stackoverflow.com/questions/269496/inheritance-vs-aggregation)

There are two schools of thought on how to best extend, enhance, and reuse code in an object-oriented system:

1. Inheritance: extend the functionality of a class by creating a subclass. Override superclass members in the subclasses to provide new functionality. Make methods abstract/virtual to force subclasses to "fill-in-the-blanks" when the superclass wants a particular interface but is agnostic about its implementation.
2. Aggregation: create new functionality by taking other classes and combining them into a new class. Attach an common interface to this new class for interoperability with other code.

What are the benefits, costs, and consequences of each? Are there other alternatives?

It's not a matter of which is the best, but of when to use what.

In the 'normal' cases a simple question is enough to find out if we need inheritance or aggregation.

* If The new class **is** more or less as the original class. Use inheritance. The new class is now a subclass of the original class.
* If the new class must **have** the original class. Use aggregation. The new class has now the original class as a member.

However, there is a big gray area. So we need several other tricks.

* If we have used inheritance (or we plan to use it) but we only use part of the interface, or we are forced to override a lot of functionality to keep the correlation logical. Then we have a big nasty smell that indicates that we had to use aggregation.
* If we have used aggregation (or we plan to use it) but we find out we need to copy almost all of the functionality. Then we have a smell that points in the direction of inheritance.

To cut it short. We should use aggregation if part of the interface is not used or has to be changed to avoid an illogical situation. We only need to use inheritance, if we need almost all of the functionality without major changes. And when in doubt, use Aggregation.

An other possibility for, the case that we have an class that needs part of the functionality of the original class, is to split the original class in a root class and a sub class. And let the new class inherit from the root class. But you should take care with this, not to create an illogical separation.

Lets add an example. We have a class 'Dog' with methods: 'Eat', 'Walk', 'Bark', 'Play'.

class Dog

Eat;

Walk;

Bark;

Play;

end;

We now need a class 'Cat', that needs 'Eat', 'Walk', 'Purr', and 'Play'. So first try to extend it from a Dog.

class Cat is Dog

Purr;

end;

Looks, alright, but wait. This cat can Bark (Cat lovers will kill me for that). And a barking cat violates the principles of the universe. So we need to override the Bark method so that it does nothing.

class Cat is Dog

Purr;

Bark = null;

end;

Ok, this works, but it smells bad. So lets try an aggregation:

class Cat

has Dog;

Eat = Dog.Eat;

Walk = Dog.Walk;

Play = Dog.Play;

Purr;

end;

Ok, this is nice. This cat does not bark anymore, not even silent. But still it has an internal dog that wants out. So lets try solution number three:

class Pet

Eat;

Walk;

Play;

end;

class Dog is Pet

Bark;

end;

class Cat is Pet

Purr;

end;

This is much cleaner. No internal dogs. And cats and dogs are at the same level. We can even introduce other pets to extend the model. Unless it is a fish, or something that does not walk. In that case we again need to refactor. But that is something for an other time.