What is the difference between a struct and a class?

In .NET, there are two categories of types, *reference types* and *value types*.

Structs are *value types* and classes are *reference types*.

The general different is that a *reference type* lives on the heap, and a *value type* lives inline, that is, wherever it is your variable or field is defined.

A variable containing a *value type* contains the entire *value type* value. For a struct, that means that the variable contains the entire struct, with all its fields.

A variable containing a *reference type* contains a pointer, or a *reference* to somewhere else in memory where the actual value resides.

This has one benefit, to begin with:

* *value type*s always contains a value
* *reference types* can contain a *null*-reference, meaning that they don't refer to anything at all at the moment

Internally, *reference type*s are implemented as pointers, and knowing that, and knowing how variable assignment works, there are other behavioral patterns:

* Copying the contents of a *value type* variable into another variable, copies the entire contents into the new variable, making the two distinct. In other words, after the copy, changes to one won't affect the other
* copying the contents of a *reference type* variable into another variable, copies the reference, which means you now have two references to the same *somewhere else* storage of the actual data. In other words, after the copy, changing the data in one reference will appear to affect the other as well, but only because you're really just looking at the same data both places

When you declare variables or fields, here's how the two types differ:

* variable: *value type* lives on the stack, *reference type* lives on the stack as a pointer to somewhere in heap memory where the actual memory lives
* Class/struct-field: *value type* lives inside the class, *reference type* lives inside the class as a pointer to somewhere in heap memory where the actual memory lives.

What does static do?

In C# what is the difference between an interface and a class?

**Class**: We use Classes as a template to put the properties and functionalities or behaviors in one building block for some group of objects and after that we use that template to create the objects we need.  
  
A class can contain declarations of the following members:  
  
Constructors, Destructors, Constants, Fields, Methods, Properties,Indexers, Operators, Events, Delegates, Classes, Interfaces, Structs

**Interface**: An **interface** contains only the signatures of **methods, delegates or events**. The implementation of the methods is done in the class that implements the interface. A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface.  
  
An interface can inherit from one or more base interfaces. When a base type list contains a base class and interfaces, the base class must come first in the list.

Exactly how did you create a COM server?

How did your threads communicate?

What did the threads do?

What happened when the thread completed?

How would you write a program to solve this problem?

Tell me a good bug story.

Difference between Abstract Class and Interface?

In **abstract** class is a class that cannot be instantiated but that can contain code.   
An **interface** only contains method definitions but does not contain any code. With an interface, you need to implement all the methods defined in the interface.

***If you have logic that will be the same for all the derived classes, it is best to go for an abstract class instead of an interface.***

You can implement multiple interfaces but only inherit from one class.

A good way to distinguish between a case for the one or the other for me has always been the following:  
  
1. Are there many classes that can be "grouped together" and described by one ***noun***? If so, have an abstract class by the name of this noun, and inherit the classes from it. (A key decider is that these classes share functionality, and you would never instantiate just an **Animal**... you would always instantiate a certain kind of **Animal**: an implementation of your **Animal** base class)  
*Example*: **Cat** and **Dog** can both inherit from abstract class **Animal**, and this abstract base class will implement a method **void Breathe()** which all animals will thus do in exactly the same fashion. (I might make this method virtual so that I can override it for certain animals, like **Fish**, which does not breath the same as most animals).  
  
2. What kinds of ***verbs*** can be applied to my class, that might in general also be applied to others? Create an interface for each of these verbs.  
*Example*: All animals can be fed, so I will create an interface called **IFeedable** and have Animal implement that. Only **Dog** and **Horse** are nice enough though to implement **ILikeable** - I will not implement this on the base class, since this does not apply to **Cat**.  
  
As said by someone else's reply: the main difference is where you want your implementation. By creating an interface, you can move your implementation to any class that implements your interface.  
By creating an abstract class, you can share implementation for all derived classes in one central place, and avoid lots of bad things like code duplication.