**Abstract Class**

**FullTimeEmployee**

**PartTimeEmployee**

**Base Employee**

**If we use base employee as a concrete class instead of abstract class then the user could create an instance of the base class that is useless to them. Use an abstract base employee and they cannot create an instance of that "Template only" class.**

**Abstract is inheritance just like base but it prevents the base class from being used by itself.**

An abstract class means that, no object of this class can be instantiated, but can make derivations of this.

The [abstract](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx) keyword enables you to create classes and [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) members that are incomplete and must be implemented in a derived class.

An abstract class can contain either abstract methods or non abstract methods. Abstract members do not have any implementation in the abstract class, but the same has to be provided in its derived class.

An abstract class cannot be a sealed class. I.e. the following declaration is incorrect.

An **abstract class** is a partially defined class that cannot be instantiated.

An **abstract class** is a partially defined class that cannot be instantiated. It (usually) includes some implementation, but leaves some functions as pure virtual- declared only by their signature. Pure virtual functions are not defined in the class that declares them, so they must be implemented by a subclass (unless it too is an abstract class). Only a subclass which defines all of the pure virtual functions can be instantiated. The purpose of an abstract class is to define some common behavior that can be inherited by multiple subclasses, without implementing the entire class. In C#, the abstract keyword designates both an abstract class and a pure virtual method.

In practical terms, the difference between the two is that an interface defines only pure virtual functions, while an abstract class may also include concrete functions, members, or any other aspect of a class.

An **interface** is like a contract. It says that a class which implements the interface agrees to implement all of the functions declared (as signatures only; no function definition) by that interface. The class may do so in any way it chooses, and provide any other functionality, as long as it implements each one of the declared functions. An interface is useful when you want to be able to use some common functionality of otherwise unrelated classes- they share no implementation details, only the function signatures. In C#, function declarations within an interface are implicitly pure virtual.

**Structure vs. Class**

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.
* The real payoff is that a Struct is a valuetype. That means that if you assign a struct to another variable, the other variable gets its own copy of the data.

The **virtual** keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

The **protected** keyword is a member access modifier. A protected member is accessible within its class and by derived class instances. For a comparison of **protected** with the other access modifiers, see [Accessibility Levels](https://msdn.microsoft.com/en-us/library/ba0a1yw2.aspx).

**Polymorphism**

Polymorphism allows you to invoke derived class methods through a base class reference during runtime.

* Casting: The act of changing the type of a reference which points to an object. Either moving up or down the object hierarchy or to an implemented interface
* Converting: Creating a new object from the original source object of a different type and accessing it through a reference to that type.

**What is a Delegate.**

A Delegate is a class.

A delegate is a Type Safe Function Pointer.

Type Safe? The delegate and the function it points to has to have exact same Signature.

It is a reference type.

The syntax of a delegate is similar to class but uses the delegate key word.

**Multicast Delegate**

A multicast delegate is a delegate the has references to more than on function. When we invoke a multicast delegate, all the functions the delegate is pointing to are invoked.

Represents a multicast delegate; that is, a delegate that can have more than one element in its invocation list.

**When do we use multicast delegates?**

**HybridDictionary** attempts to optimize Hashtable. It implements a linked list and hash table data structure, switching over to the second from the first when the number of elements increases past a certain threshold.

The HybridDictionary type is found in the System.Collections.Specialized namespace.