**Why Interface?**

It starts with a dog. In particular, a pug.

The pug has various behaviors:

public class Pug

{

private String name;

public Pug(String n)

{

name = n;

}

public String getName()

{

return name;

}

public String bark()

{

return "Arf!";

}

public boolean hasCurlyTail()

{

return true;

}

}

And you have a Labrador, who also has a set of behaviors.

public class Lab

{

private String name;

public Lab(String n)

{

name = n;

}

public String getName()

{

return name;

}

public String bark()

{

return "Woof!";

}

public boolean hasCurlyTail()

{

return false;

}

}

We can make some pugs and labs:

Pug pug = new Pug("Spot");

Lab lab = new Lab("Fido");

And we can invoke their behaviors:

pug.bark() -> "Arf!"

lab.bark() -> "Woof!"

pug.hasCurlyTail() -> true

lab.hasCurlyTail() -> false

pug.getName() -> "Spot"

Let's say I run a dog kennel and I need to keep track of all the dogs I'm housing. I need to store my pugs and labradors in separate arrays:

public class Kennel

{

Pug[] pugs = new Pug[10];

Lab[] labs = new Lab[10];

public void addPug(Pug p)

{

...

}

public void addLab(Lab l)

{

...

}

public void printDogs()

{

// Display names of all the dogs

}

}

But this is clearly not optimal. If I want to house some poodles, too, I have to change my Kennel definition to add an array of Poodles. In fact, I need a separate array for each kind of dog.

**Insight:** both pugs and labradors (and poodles) are types of dogs and they have the same set of behaviors. That is, we can say (for the purposes of this example) that all dogs can bark, have a name, and may or may not have a curly tail. We can use an interface to define what all dogs can *do*, but leave it up to the specific types of dogs to implement those particular behaviors. The interface says "here are the things that all dogs can do" but doesn't say *how* each behavior is done.

public interface Dog

{

public String bark();

public String getName();

public boolean hasCurlyTail();

}

Then I slightly alter the Pug and Lab classes to **implement** the Dog behaviors. We can say that a Pug*is* a Dog and a Lab *is* a dog.

public class Pug implements Dog

{

// the rest is the same as before

}

public class Lab implements Dog

{

// the rest is the same as before

}

I can still instantiate Pugs and Labs as I previously did, but now I also get a new way to do it:

Dog d1 = new Pug("Spot");

Dog d2 = new Lab("Fido");

This says that d1 is not only a Dog, it's specifically a Pug. And d2 is also a Dog, specifically a Lab.

We can invoke the behaviors and they work as before:

d1.bark() -> "Arf!"

d2.bark() -> "Woof!"

d1.hasCurlyTail() -> true

d2.hasCurlyTail() -> false

d1.getName() -> "Spot"

Here's where all the extra work pays off. The Kennel class become much simpler. I need only one array and one addDog method. Both will work with any object that *is* a dog; that is, objects that implement the Dog interface.

public class Kennel

{

Dog[] dogs = new Dog[20];

public void addDog(Dog d)

{

...

}

public void printDogs()

{

// Display names of all the dogs

}

}

Here's how to use it:

Kennel k = new Kennel();

Dog d1 = new Pug("Spot");

Dog d2 = new Lab("Fido");

k.addDog(d1);

k.addDog(d2);

k.printDogs();

The last statement would display:

Spot

Fido

An interface gives you the ability to specify a set of behaviors that all classes that implement the interface will share in common. Consequently, we can define variables and collections (such as arrays) that don't have to know in advance what kind of specific object they will hold, only that they'll hold objects that implement the interface.

===========================

Question is- a class can directly implement the functions in an interface. eg:

Interface IPerson{

void jump(int);

}

class Person{

int name;

void jump(int height){

//Do something here

}

}

What specific difference does interface make?

class Person implements IPerson{

int name;

void jump(int height){

//Do something here

}

}

The answer is-

Having an IPerson interface allows you to have multiple implementers (Man, Woman, Employee etc.), but still treat them all through the interface in other classes.

So, in another class you simply state:

void myMethod(IPerson person, Integer howHigh)

{

person.jump(howHigh);

}

You don't have to have a separate method for each implementer.

To add to that: Interfaces also provide a degree of information hiding because they limit knowledge of the object to just what the interface defines. If a Man and Woman have other methods beyond what's in IPerson, they're completely invisible to anything referring to the objects using that interface.

===========================

The purpose of an interface is not to relocate or reuse code but more to ensure that some methods that operate on a wide range of types can use these types (which will be passed to them as arguments) in the exact same way even though they may not have common parent classes.

Interface is used to make the contract - the expected functionaity to exist in a certain class.

For example:

interface Transmittable {

public byte[] toBytes();

}

class Person implements Transmittable {

public byte[] toBytes() {

return this.name.getBytes()

}

}

class Animal implements Transmittable {

public byte[] toBytes() {

return this.typeOfAnimal.getBytes()

}

}

class NetworkTransmitter {

public void transmit(Transmittable object) {

byte data[] = object.toBytes();

//do something....

}

}

class TestExample {

public static void main(String args[]) {

NetworkTransmitter trobj = new NetworkTransmitter();

trobj.transmit(new Person());

trobj.transmit(new Animal());

}

}

Note that this is not like inheritance where object of one class inherits (or overrides) a method of the same name from the parent. The classes that implement a interface need not be descendant from the same parent class but other classes that want to ensure a contract exists, can call the same methods on objects of all classes that implement the interface. Interface ensures that this contract is available.

===========================

http://programmers.stackexchange.com/questions/145437/why-use-an-interface-when-the-class-can-directly-implement-the-functions

**What is an Interface**

First and foremost, interfaces in VB.NET are a means to get around the lack of multiple inheritance in VB.NET, meaning you cannot inherit from multiple classes but you can implement multiple interfaces. OOP tries to resemble how objects are defined in the real life, and interfaces are a very logical way of grouping objects in terms of behavior.

An interface is a contract between itself and any class that implements it. This contract states that any class that implements the interface will implement the interface's properties, methods and/or events. An interface contains no implementation, only the signatures of the functionality the interface provides. An interface can contain signatures of methods, properties, indexers & events.

You can think of an interface as an abstract class with the implementation stripped out. An interface doesn't actually do anything, like a class or abstract class, it merely defines what a class that implements it will do. An interface can also inherit/implement other interfaces.

**Why use interfaces**

So if an interface implements no functionality then why should we use them? Using interface based design concept provides loose coupling, component-based programming, easier maintainability, makes your code base more scalable and makes code reuse much more accessible because implementation is separated from the interface. Interfaces add a *plug and play* like architecture into your applications. Interfaces help define a contract (agreement or blueprint, however you chose to define it), between your application and other objects. This indicates what sort of methods, properties and events are exposed by an object.

For example let's take a vehicle. All vehicles have similar items, but are different enough that we could design an interface that holds all the common items of a vehicle. Some vehicles have 2 wheels, some have 4 wheels and can even have 1 wheel, though these are differences they have something in common, they're all movable, they all have some sort of engine, they all have doors, but each of these items may vary. So we can create an interface of a vehicle that has these properties, then we inherit from that interface to implement it.

While wheels, doors and engines are different they all rely on the same interface (I sure hope this is making sense). Interfaces allow us to create nice layouts for what a class is going to implement. Because of the guarantee the interface gives us, when many components use the same interface it allows us to easily interchange one component for another which is using the same interface. Dynamic programs begin to form easily from this. An interface is a contract that defines the signature of some piece of functionality.

So here's a simple example of an interface and implementing it. From the above example we're created a IVehicle interface that looks like this

Namespace InterfaceExample

Public Interface IVehicle

Property Doors() As Integer

Property Wheels() As Integer

Property VehicleColor() As Color

Property TopSpeed() As Integer

Property Cylinders() As Integer

ReadOnly Property CurrentSpeed() As Integer

Function DisplayTopSpeed() As String

Sub Accelerate([step] As Integer)

End Interface

End Namespace

Now we have our vehicle blueprint, and all classes that implement it must implement the items in our interface, whether it be a motorcycle, car, or truck class we know that all will contain the same functionality. Now for a sample implementation, in this example we'll create a motorcycle class that implements our IVehicle class. This class will contains everything we have defined in our interface

Namespace InterfaceExample

Public Class Motorcycle

Implements IVehicle

Private \_currentSpeed As Integer = 0

Public Property Doors() As Integer

Get

Return m\_Doors

End Get

Set

m\_Doors = Value

End Set

End Property

Private m\_Doors As Integer

Public Property Wheels() As Integer

Get

Return m\_Wheels

End Get

Set

m\_Wheels = Value

End Set

End Property

Private m\_Wheels As Integer

Public Property VehicleColor() As Color

Get

Return m\_VehicleColor

End Get

Set

m\_VehicleColor = Value

End Set

End Property

Private m\_VehicleColor As Color

Public Property TopSpeed() As Integer

Get

Return m\_TopSpeed

End Get

Set

m\_TopSpeed = Value

End Set

End Property

Private m\_TopSpeed As Integer

Public Property HorsePower() As Integer

Get

Return m\_HorsePower

End Get

Set

m\_HorsePower = Value

End Set

End Property

Private m\_HorsePower As Integer

Public Property Cylinders() As Integer

Get

Return m\_Cylinders

End Get

Set

m\_Cylinders = Value

End Set

End Property

Private m\_Cylinders As Integer

Public ReadOnly Property CurrentSpeed() As Integer

Get

Return \_currentSpeed

End Get

End Property

Public Sub New(doors As Integer, wheels As Integer, color As Color, topSpeed As Integer, horsePower As Integer, cylinders As Integer, \_

currentSpeed As Integer)

Me.Doors = doors

Me.Wheels = wheels

Me.VehicleColor = color

Me.TopSpeed = topSpeed

Me.HorsePower = horsePower

Me.Cylinders = cylinders

Me.\_currentSpeed = currentSpeed

End Sub

Public Function DisplayTopSpeed() As String

Return "Top speed is: " + Me.TopSpeed

End Function

Public Sub Accelerate([step] As Integer)

Me.\_currentSpeed += [step]

End Sub

End Class

End Namespace

Now in the same application we could interchange our Motorcycle class with a Truck class or a Car class and they will all have the same base functionality, that of a IVehicle.

So as you can see interface based development can make a developers life much easier, and our applications much cleaner, maintainable and extensible.

<https://dzone.com/articles/c-interfaces-what-are-they-and>