**Interfaces:**

**A: They can make your code easier to maintain in the future by introducing a loose coupling between your front end and your back end implementation.**

You could write this to start with, as there are only going to be trolls:

// This is our back-end implementation of a troll

class Troll

{

void Walk(int distance)

{

//Implementation here

}

}

Front end:

function SpawnCreature()

{

Troll aTroll = new Troll();

aTroll.Walk(1);

}

Two weeks down the line, marketing decide you also need Orcs, as they read about them on twitter, so you would have to do something like:

class Orc

{

void Walk(int distance)

{

//Implementation (orcs are faster than trolls)

}

}

Front end:

void SpawnCreature(creatureType)

{

switch(creatureType)

{

case Orc:

Orc anOrc = new Orc();

anORc.Walk();

case Troll:

Troll aTroll = new Troll();

aTroll.Walk();

}

}

And you can see how this starts to get messy. You could use an interface here so that your front end would be written once and (here's the important bit) tested, and you can then plug in further back end items as required:

interface ICreature

{

void Walk(int distance)

}

public class Troll : ICreature

public class Orc : ICreature

//etc

Front end is then:

void SpawnCreature(creatureType)

{

ICreature creature;

switch(creatureType)

{

case Orc:

creature = new Orc();

case Troll:

creature = new Troll();

}

creature.Walk();

}

The front end now only cares about the interface ICreature - it's not bothered about the internal implementation of a troll or an orc, but only on the fact that they implement ICreature.

An important point to note when looking at this from this point of view is that you could also easily have used an abstract creature class, and from this perspective, this has the **same** effect.

And you could extract the creation out to a factory:

public class CreatureFactory {

public ICreature GetCreature(creatureType)

{

ICreature creature;

switch(creatureType)

{

case Orc:

creature = new Orc();

case Troll:

creature = new Troll();

}

return creature;

}

}

And our front end would then become:

CreatureFactory \_factory;

void SpawnCreature(creatureType)

{

ICreature creature = \_factory.GetCreature(creatureType);

creature.Walk();

}

The front end now does not even have to have a reference to the library where Troll and Orc are implemented (providing the factory is in a separate library) - it need know nothing about them whatsoever.

**B: Say you have functionality that only some creatures will have in your otherwise homogenous data structure**, e.g.

interface ICanTurnToStone

{

void TurnToStone();

}

public class Troll: ICreature, ICanTurnToStone

Front end could then be:

void SpawnCreatureInSunlight(creatureType)

{

ICreature creature = \_factory.GetCreature(creatureType);

creature.Walk();

if (creature is ICanTurnToStone)

{

(ICanTurnToStone)creature.TurnToStone();

}

}

**C: Usage for dependency injection**

Most dependency injection frameworks work when there is a very loose coupling between the front end code and the back end implementation. If we take our factory example above and have our factory implement an interface:

public interface ICreatureFactory {

ICreature GetCreature(string creatureType);

}

Our front end could then have this injected (e.g an MVC API controller) through the constructor (typically):

public class CreatureController : Controller {

private readonly ICreatureFactory \_factory;

public CreatureController(ICreatureFactory factory) {

\_factory = factory;

}

public HttpResponseMessage TurnToStone(string creatureType) {

ICreature creature = \_factory.GetCreature(creatureType);

creature.TurnToStone();

return Request.CreateResponse(HttpStatusCode.OK);

}

}

With our DI framework (e.g. Ninject or Autofac), we can set them up so that at runtime a instance of CreatureFactory will be created whenever an ICreatureFactory is needed in an constructor - this makes our code nice and simple.

It also means that when we write a unit test for our controller, we can provide a mocked ICreatureFactory (e.g. if the concrete implementation required DB access, we don't want our unit tests dependent on that) and easily test the code in our controller.

**D: There are other uses e.g. you have two projects A and B that for 'legacy' reasons are not well structured, and A has a reference to B.**

You then find functionality in B that needs to call a method already in A. You can't do it using concrete implementations as you get a circular reference.

You can have an interface declared in B that the class in A then implements. Your method in B can be passed an instance of a class that implements the interface with no problem, even though the concrete object is of a type in A.

An interface expresses a **behavior that a given class or structure may choose to support**.

Types ability to convert instances of itself to another type, provided by the [IConvertible](http://msdn2.microsoft.com/en-us/library/system.iconvertible.aspx) interface and types ability to serialize itself through the [ISerializable](http://msdn2.microsoft.com/en-us/library/system.runtime.serialization.iserializable(VS.80).aspx) interface.

When classes implement interface,objects of the **different classes can be treated as if their type were the type of the interface**.In addition the objects can be treated polymorphically because each class defines the common behaviours specified in the interface in its own different way.

**Example 2:** ISalary has property ‘property name’ and method ‘MethodName’

**ISalary anEmployee=new Employee().**

Code can invoke methods and property specified in the interface**ISalary**through the **anEmployee**reference like

anEmployee.MethodName();

anEmployee.PropertyName;

***Example 2:***

//Clearing Interface Confusion

public interface Ic

{

int Type { get; }

string Name { get; }

}

public class A : Ic

{

.

.

.

}

public class B : Ic

.

.

.

}

public Ic func(bool flag)

{

if (flag)

return new A();

return new B();

}

In this example func is like factory method — it can return different objects!

an interface is a "Type" description like an int, or class, but an interface can have multiple implementation of it's type.

Thus when you are returning the "interface" you are actually returning the implementation of said interface.

<https://medium.com/@ghanshyam.skipper/interfaces-in-c-6ea8d5037197>

Refer Above Article for more understanding.