No one has really explained in plain terms how interfaces are useful, so I'm going to give it a shot (and steal an idea from Shamim's answer a bit).

Lets take the idea of a pizza ordering service. You can have multiple types of pizzas and a common action for each pizza is preparing the order in the system. Each pizza **has to be prepared** but each pizza **is prepared differently**. For example, when a stuffed crust pizza is ordered the system probably has to verify certain ingredients are available at the restaurant and set those aside that aren't needed for deep dish pizzas.

When writing this in code, technically you could just do

public class Pizza()

{

public void Prepare(PizzaType tp)

{

switch (tp)

{

case PizzaType.StuffedCrust:

// prepare stuffed crust ingredients in system

break;

case PizzaType.DeepDish:

// prepare deep dish ingredients in system

break;

//.... etc.

}

}

}

However, deep dish pizzas (in C# terms) may require different properties to be set in the Prepare()method than stuffed crust, and thus you end up with a lot of optional properties, and the class doesn't scale well (what if you add new pizza types).

The proper way to solve this is to use interface. The interface declares that all Pizzas can be prepared, but each pizza can be prepared differently. So if you have the following interfaces:

public interface IPizza

{

void Prepare();

}

public class StuffedCrustPizza : IPizza

{

public void Prepare()

{

// Set settings in system for stuffed crust preparations

}

}

public class DeepDishPizza : IPizza

{

public void Prepare()

{

// Set settings in system for deep dish preparations

}

}

Now your order handling code does not need to know exactly what types of pizzas were ordered in order to handle the ingredients. It just has:

public PreparePizzas(IList<IPizza> pizzas)

{

foreach (IPizza pizza in pizzas)

pizza.Prepare();

}

Even though each type of pizza is prepared differently, this part of the code doesn't have to care what type of pizza we are dealing with, it just knows that it's being called for pizzas and therefore each call to Prepare will automatically prepare each pizza correctly based on its type, even if the collection has multiple types of pizzas.

103

For me, the point to these only became clear when you stop looking at them as things to make your code easier/faster to write - this is not their purpose. They have a number of uses:

(This is going to lose the pizza analogy, as it's not very easy to visualise a use of this)

Say you are making a simple game on screen and It will have creatures with which you interact.

**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;

switch(creatureType)

{

case Orc:

creature = new Orc();

case Troll:

creature = new Troll();

}

creature.Walk();

if (creature is ICanTurnToStone)

{

(ICanTurnToStone)creature.TurnToStone();

}

}

**C: Usage for dependency injection**

Most dependency injection frameworks are easier to work with 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.