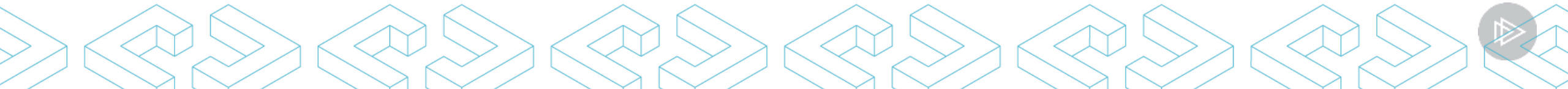
C#: Using Interfaces

Eric J Fisher
AUTHOR, PLURALSIGHT

@EricJamesFisher

What Is an Interface?

An interface contains only the signatures of methods, properties, events, and indexers. Interfaces are typically used for purposes of code abstraction.



An Example Interface

An interface looks a lot like a class, except it only contains signatures

IRetailer.cs

```
public interface IRetail
{
  string Name { get; set; }
  void Sale(int amount);
  void Return(int amount);
}
```

Consider This Code

Each retailer's implementation is slightly different, resulting in a giant case switch

Program.cs

```
void Sale(int amount, string retailer)
{
  switch(retailer)
  {
    case "Bob's Tools":
      Console.WriteLine("Your sale of {0} has been processed.",amount);
      break;
    case "Bikes R Us":
      Console.WriteLine("Thank you for your purchase!");
      break;
```

One big switch case statement isn't too terrible, but what happens when we add a method to handle returns?

Consider this Code – it is Getting Repetitive

Every method requiring a giant case switch statement is A LOT of duplication

Program.cs

```
void Sale(int amount, string retailer)
{
  switch(retailer)
  {
    case "Bob's Tools":
      Console.WriteLine("Bob's Tools th
      break;
    case "Bikes R Us":
      Console.WriteLine("Thank you for
      break;
    case "Tool Shack":
      Console.WriteLine("Let's build do
      break;
```

Program.cs (continued)

```
void Return(int amount, string ret
{
  switch(retailer)
  {
    case "Bob's Tools":
      Console.WriteLine("We're sor
      break;
    case "Bikes R Us":
      Console.WriteLine("Thank you
      break;
    case "Tool Shack":
      Console.WriteLine("We'll do
      break;
```

Consider Abstracting Our Code

Instead of each method containing the implementation for every company, let's have every company have it's own version of each method.

Program.cs

```
void Sale(int amount, string retailer)
```

```
case Company A: Implementation
```

```
case Company B: Implementation
```

```
case Company C: Implementation
```

CompanyA.cs

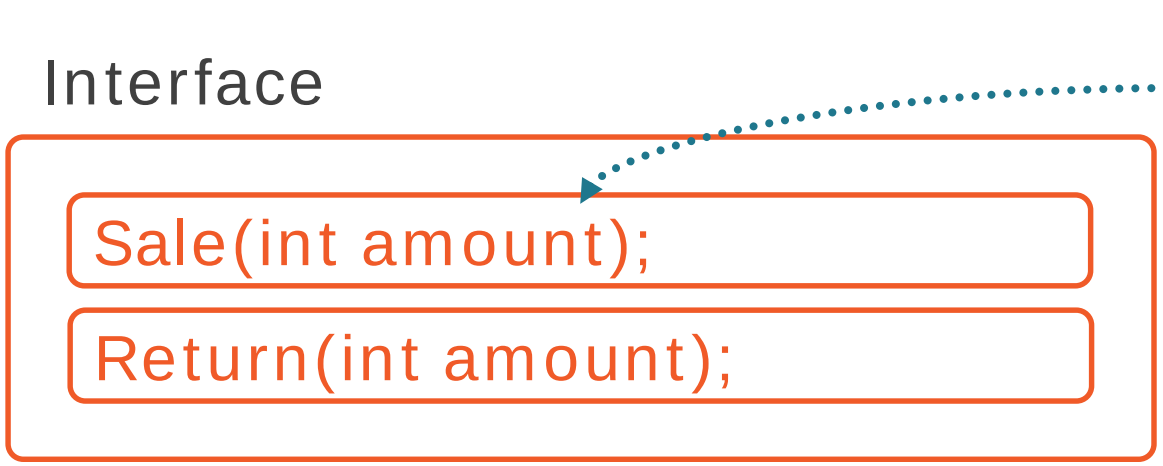
```
public class CompanyA
```

```
void Sale(int amount)
```

```
void Return(int amount)
```

> To do this we will use an Interface to act as a blueprint for our company classes to follow.

Our Interface Will Contain Sale and Return



Sale and Return no longer need the retailer parameter.

Our interface acts as a blueprint listing what methods, properties, etc our class will contain.

Company Classes Inherit and Implement the Interface

Interface

Sale(int amount);

Return(int amount);

Company A : Interface

Sale(int amount) {...}

Return(int amount) {...}

Company B : Interface

Sale(int amount) {...}

Return(int amount) {...}

Our Main Sale Method Will Call Our Interface

Program.cs

```
Sale(int amount) { IRetail.Sale(amount); }
```

Our Pragram.cs's Sale method will call our Interface's Sale Method

Interface

```
Sale(int amount);
```

Which will run the company specific implementation based on the class used to instantiate the interface

Company A

```
Sale(int amount) {...}
```

Company B

```
Sale(int amount) {...}
```

But how do we do this in actual code?

Creating an Interface

An interface is written a lot like a class, except it contains signatures, not implementation

IRetailer.cs

```
public interface IRetail
{
  string Name { get; set; }
  void Sale(int amount);
  void Return(int amount);
}
```

To declare an interface use the keyword "interface" before the name of your interface

Standard naming convention for interfaces in C# is PascalCase with the name preceded by an "I"

Unlike classes interface's properties, methods, etc are always public, so you don't need access modifiers

Implementing Our Interface

To use a class to implement our interface we need to first inherit the interface

BobsTools.cs

```
public class BobsTools : IRetail
{

}
```

To inherit our interface we need to add a : followed by the name of our interface to the end of our class declaration

Implement All Parts of the Interface

Your code won't compile until all parts of the interface are implemented by the inheriting class

BobsTools.cs

```
public class BobsTools : IRetail
{
  public Name { get; set; } = "Bob's Tools";
  public void Sale(int amount)
  {
    Console.WriteLine("Bob's Tools thanks you for your purchase! Make it happ
  }

  public void Return(int amount)
  {
    Console.WriteLine("We're sorry you weren't happy with your purchase! Hope
  }
}
```

> *This means we need to implement the Name property as well as the Sale and Return methods*

Delaying the Implementation of Interface Methods

BobsTools.cs

```
public class BobsTools : IRetail
{
  public Name { get; set; } = "Bob's Tools"
  public void Sale(int amount)
  {
    Console.WriteLine("Bob's Tools thanks you for your purchase! Make it happ
  }

  public void Return(int amount)
  {
    throw new NotImplementedException();
  }
}
```

Sometimes you'll have methods in an interface you're not ready to implement yet, common practice is to throw a "NotImplementedException" in such cases

Using Our Interface

Now we want to use our interface to eliminate code duplication

Program.cs

```
private IRetail retailer = new BobsTools();

void Sale(int amount)
{
  retailer.Sale(amount);
}

void Return(int amount)
{
  retailer.Return(amount);
}
```

First instantiate the interface using an implementing class.

Next, use the interface to call methods

This will result in our Sale and Return methods calling BobsTools version of the Sale and Return method.

Enabling Multiple Implementations

To support multiple retailers we'll need to instantiate our interface conditionally

Program.cs

```
private IRetail retailer { get; set; }

static SetRetailer(string name)
{
  switch(name)
  {
    case "Bob's Tools"
      retailer = new BobsTools();
      break;
    case "Bikes R Us"
      retailer = new BikesRUs();
      break;
…
```

For other retailers, we would just need to instantiate their implementation of the interface instead. Only now we only need to figure this out once, instead of in every single method.

Inheriting Multiple Interfaces

While C# does not allow inheriting from multiple classes, it does allow inheriting from multiple interfaces!

Retail Interface

- void Sale(int amount)
- void Return(int amount)

Rental Interface

- void Rental(int amount)
- void Return(int id)

But what happens when the interfaces we're inheriting from result in conflicts?

In this example our Return methods would conflict, while ideally we'd adjust the interfaces to not conflict, when that's not realistic we have two options

Common Implementation

If we keep just one implementation of Return both IRetail and IRental will use it

BobsTools.cs

```
public class BobsTools : IRetail, IRental
{
  public void Return(int amount)
  {
    Console.WriteLine("Your refund of {0} has been processed. We hope you'll
  }
}
```

If we call IRetail.Return or IRental.Return both would execute our Return method

```
((IRental)company).Return(amount);
```

```
((IRetail)company).Return(amount);
```

Unfortunately, we want IRetail.Return and IRental.Return to have different results

Explicitly Implementing Our Interfaces

To explicitly implement an interface use Interface.Method as the method name

BobsTools.cs

```
public class BobsTools : IRetail, IRental
{
  public void IRental.Return(int id)
  {
    Console.WriteLine("We have verified the return of unit {0}, thank you fo
  }

  public void IRetail.Return(int amount)
  {
    Console.WriteLine("Your refund of {0} has been processed. We hope you'll
  }
}
```

Now we can call each method using their respective interfaces!

Explicitly Using Our Interface

To explicitly use our interface we just instantiate or cast to that interface

Program.cs

```
private object company = new BobsTools();

void ReturnSale(int amount)
{
  IRetail retail = company;
  retail.Sale(amount);
}

void ReturnRental(int amount)
{
  ((IRental)company).Return(amount);
}
```

Example of instantiating our interface

Example of casting our interface

Summary

Interfaces contain only the signatures of methods, properties, events, and indexers

Interfaces can be used for code abstraction (which can make code cleaner, easier to maintain, and easier to test)

Any class or struct inheriting an interface must implement all members of that interface.

When necessary you can declare and use Interfaces explicitly by using the Interface.Method naming convention.