# Lecture 34 — Interfaces

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

# Sorting Programmer-Defined Types

Last time, we examined using the `Sort` methods of the language to do sorting automatically.

To sort a programmer-defined type, we can either do it manually, or we can tell the system how to sort that type automatically.

Let's use a class `Student` that has the following member variables:

- Student ID Number
- Last Name
- First Name
- Date of Birth
- User ID

# Sorting Programmer-Defined Types

If told to sort some students, you might ask, "sort them by what?"
Could be: Student ID Number, Last Name, Date of Birth...

You might also ask if you want it to be ascending, descending...

Let's say we want to sort by student ID number, ascending.

When you actually perform the sort, you will examine a particular student record, and compare it with others to see where it belongs.

So to tell the system how to sort a programmer-defined type:
Tell it how to compare two instances.

# Interface

The secret to indicating how to sort two objects is the interface.

An interface is an object oriented design concept to allow a common means for unrelated objects to interact with one another.

In this case, we want the sorting routine to interact with the `Student` class, even though the sort routine knows nothing about `Students`.

Think of an interface as a contract.

- An interface specifies some number of methods.
- To fulfill our side of the contract, we implement those methods.
- If we do our part, we gain some benefit.

# IComparable

The relevant interface in C# is `IComparable`.

`IComparable` contains one method signature:

```
public int CompareTo( object obj )
```

The first thing to do is to say that our class promises to implement the methods of the `IComparable` interface.

When we live up to our promise and implement `CompareTo` on our class, we get the benefit: automatic sorting of instances of that class.

# Comparing Two Instances

The contract says when `o1` is compared to `o2` (`o1.CompareTo( o2 )`):

- A negative result (e.g., -1) is returned if `o1` is "smaller than" `o2`
- Zero is returned if `o1` is equal to `o2`
- A positive result (e.g., 1) is returned if `o1` is "larger than" `o2`

The meaning of the comparison is defined by the method implementation, not the interface.

We are responsible for deciding what it means for an instance to be larger than or smaller than another.

# Comparing Two Instances

We have decided to sort by student ID.

Thus when `Student s1.CompareTo( s2 )` is run:

- -1 is returned if `s1`'s student ID is less than `s2`'s student ID
- Zero is returned if `s1`'s student ID is equal to `s2`'s student ID
- 1 is returned if `s1`'s student ID is greater than `s2`'s student ID

We reduce the problem of comparing two `Students` down to the problem of comparing two student IDs.

Comparing student IDs is easy: they're integers.

# Implementing IComparable

To indicate the promise to implement IComparable, we add after the name of the class: a colon and the name of the interface.

```
public class Student : IComparable
{
    public int CompareTo( object obj )
    {
        // TODO : Implement method
    }
}
```

By promising to implement that interface, it will be a compile time error if we do not add the CompareTo method.

We've decided that the CompareTo method should do a normal integer comparison of their student IDs.

# Implementing IComparable

Remember the `is` keyword from the discussion of `Equals`, and the use of explicit type casting.

```
public int CompareTo( object obj )
{
    if ( obj is Student )
    {
        Student other = (Student) obj;
        if ( studentID < other.studentID )
        {
            return -1;
        } else if ( studentID == other.studentID )
        {
            return 0;
        } else
        {
            return 1;
        }
    }
    else
    {
        throw new ArgumentException( "No comparison possible." );
    }
}
```

# Implementing IComparable

Given that studentID is an `int`, a system defined type for which `CompareTo` is implemented, we can simplify this:

```
public int CompareTo( object obj )
{
    if ( obj is Student )
    {
        Student other = (Student) obj;
        return studentID.CompareTo( other.studentID );
    }
    else
    {
        throw new ArgumentException( "No comparison possible." );
    }
}
```

# Changing the Implementation

If we decide in the future we would prefer to sort by userID, the change is easy to make.

Strings and all built-in numeric types are comparable as we expect.

If we wanted to sort by date of birth, it might be necessary to make the `Date` class implement comparable too.

# Implementing IComparable on User ID

Let's rewrite the comparison to check on User ID:

```
public int CompareTo( object obj )
{
    if ( obj is Student )
    {
        Student other = (Student) obj;
        return userID.CompareTo( other.userID );
    }
    else
    {
        throw new ArgumentException( "No comparison possible." );
    }
}
```

We delegate the comparison to the built in string comparison routine.

This still lives up to the contract of IComparable.

# Using Multiple Criteria

Sorting on student ID or user ID is “nice” because we expect that these are all unique: no two students will have the same student/user ID.

Last names, however, are not unique. Many are quite common, siblings, et cetera...

When we compare two identical last names we will have a tie.

To get a consistent sort order we will need a tiebreaker.

# Multiple Criteria IComparable

```
public int CompareTo( object obj )
{
    if ( obj is Student )
    {
        Student other = (Student) obj;
        int comparison = lastName.CompareTo( other.lastName );
        if (comparison == 0)
        {
            return firstName.CompareTo( other.firstName );
        }
        else
        {
            return comparison;
        }
    }
    else
    {
        throw new ArgumentException( "No comparison possible." );
    }
}
```

If two last names are equal, the check compares the first names.

We could check another criterion if the first names are also identical...

# Summing Up `IComparable`

As we saw for `IComparable`, an interface is a contract.
- If we accept the contract and implement the methods, we benefit.
- Here, the benefit is automatic sorting of the `Student` objects.

The system can sort without knowing how `Student` is implemented.
- It's enough to know that `Student` implements `IComparable`.

In short, the interface allows a piece of code (the sort routine) to interact with some object (the student), regardless of implementation.

Comparison is useful, but it's only an example of what interfaces are...

# Defining Interfaces

Formally, an interface defines a set of methods, properties, et cetera.

A class may implement as many interfaces as required.

An interface is defined as follows: *access-modifier* interface *identifier*

This is followed by a statement block that contains the definitions of methods, properties, etc...

Interfaces may not contain method implementation; only signatures.
They also may not contain fields.

# Interface Definition Example

```
public interface ITaxable
{
    public double GetTaxRate( );
    public double GetBasisAmount( );
    public void SetTotalAmount( double totalAmount );
}
```

It is a C# convention to add I (a capital letter i) to the beginning of an interface name. This is common practice, but not an enforced rule.

# Using Interfaces

It's important to note that it is not possible to instantiate an interface. An interface is only a specification of members (behaviour).

Instead, we must instantiate a class that implements that interface.

We may have several classes that implement `ITaxable`:

- `Invoice`
- `BillPayment`
- `Paycheque`
- ... and more.

We could instantiate any one of those.

# Using Code that Implements Interfaces

We can, for example, use the interface name as the type as a parameter of a function or method.

Let's use the `ITaxable` interface we saw before.

There may be another method elsewhere:

```
public void computeTotal( ITaxable t )
```

The interface is `ITaxable` and within this method we can call any method on `t` that is specified in the interface.

# Using Code that Implements Interfaces

```
public void computeTotal( ITaxable t )
{
    double tax = t.GetBasisAmount( ) * t.GetTaxRate( );
    double total = t.GetBasisAmount( ) + tax;
    t.SetTotalAmount( total );
}
```

This works even though the object referenced by `t` is not an instance of `ITaxable`, but of some class that implements `ITaxable`.

# Multiple Interfaces

It is permitted for one class to implement more than one interface.

To do so, the interfaces are separated by commas.

Example: `public class Invoice : ITaxable, IComparable`

Interfaces are a way to write code that is extensible and general.

Next time, we'll look at more advanced techniques.