SIT771 – Lecture 8

Interface and delegate

DEAKIN UNIVERSITY

Further reading

- Paul Deitel and Harvey Deitel (2018). Visual C# how to Program (6th ed). Pearson. Ebook on Deakin Library – Chapter 12 and Chapter 21.

Outline

In this lecture...

- Inheritance revisited
  - What we know
  - Case study
- Interface
  - Generics
  - Abstract class vs. interface
- Delegate
- Lambda expression

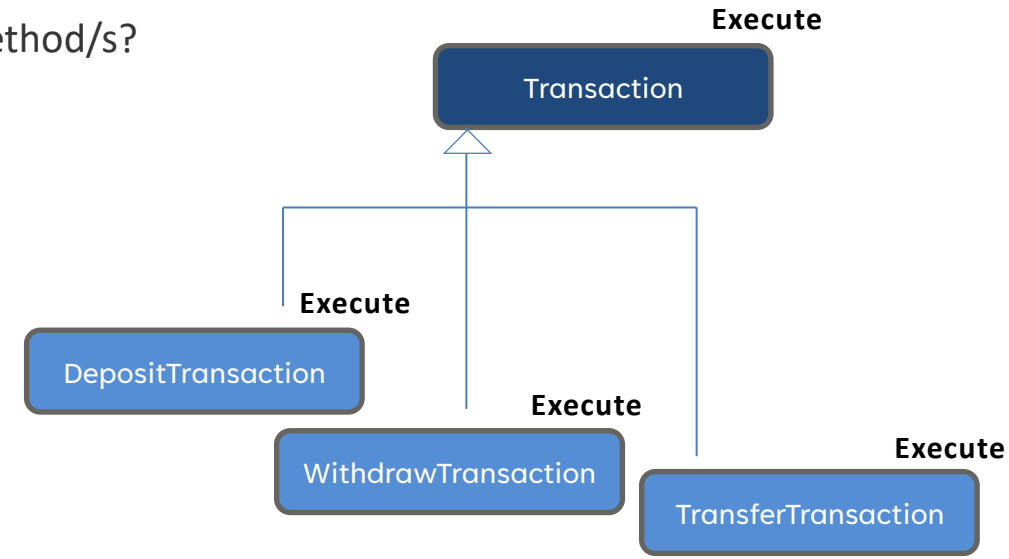
INHERITANCE REVISITED

What we know

From previous weeks...

- **Inheritance** gives us the ability to work with families of **related** types, i.e., parent classes and child classes.
  - Child classes inherit properties and methods from parent classes
  - Child classes can change the behavior of inherited methods if needed
  - Thus, the family of related classes will have the same methods **with/without** the same implementation.

- How about **unrelated** types?
  - How can we make sure some **unrelated classes** will all have the same method/s?



Case study

Sorting

- You have a Robot that...
  - perceives the room through image and video processing. You want the Robot to be able to sort some sets of **unrelated** objects that it perceives on the basis of **a specific property of each object** type. This will result in separate sorted sets of same-type objects. The Robot will then have to put the objects in each sorted set in a stack in order. You want to focus only on the sorting of these objects at this stage.
- For this, you know that…
  - these object types are <u>not</u> in an inheritance hierarchy.
  - each set of same-type objects (class) needs to have a sorting mechanism of its own (you need a **binding contract!**)

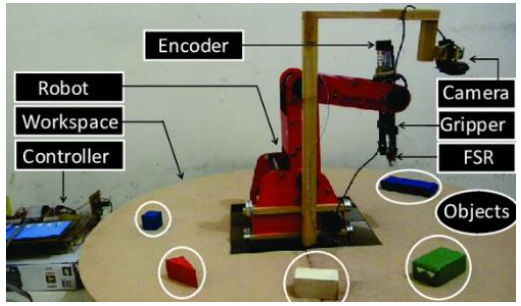


Image source: "Automating industrial tasks through mechatronic systems – A review of robotics in industrial perspective", Iqbal et al., 2016.

Case study

Sorting (cont.)

- You know...
  - how to sort primitive data types, e.g., strings, integers, doubles, and thus lists of any single variable that is of these primitive types. For instance, you know *naturally* how to sort a list of integer values whether it be in an increasing or decreasing order. But..., the core questions are:
    - How do you sort such a list?
    - What is the **basic operation** you do to accomplish such sorting? **Compare!** 🤔

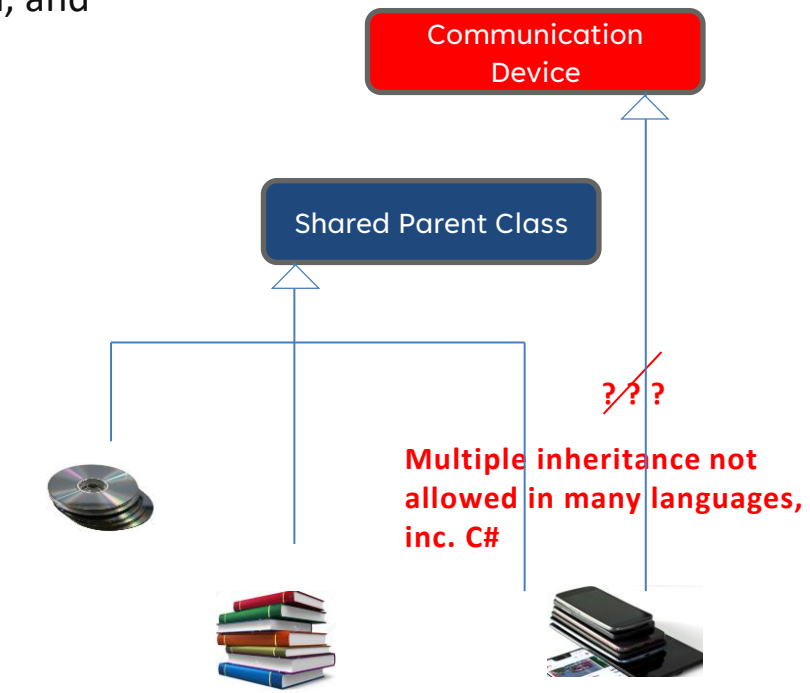
- Now, you have...
  - lists of objects that are not of primitive data types, they are phones, books, and disks. The next core questions are:
    - How will you sort lists of such non-primitive objects?
    - Or, more specifically, how will you compare these objects?
    - Importantly, how will you enforce the binding contract?

🤔

Case study

Solution 1 – A shared parent class

- This process goes as...
  - You will implement sortable classes. You make sure, and you will not forget, to have the following steps taken: i) Create a shared **sortable parent class** for all classes that you want your Robot to be able to sort, ii) Implement the inheritance structure for the newly created parent class and all the other classes you want your Robot to be able to sort the objects of, and iii) Implement a sorting method for the parent class with two pieces of hope:
    - The child classes will be happy and use the same comparison/sorting logic.
    - Or, the child classes will override the parent's comparison/sorting logic.



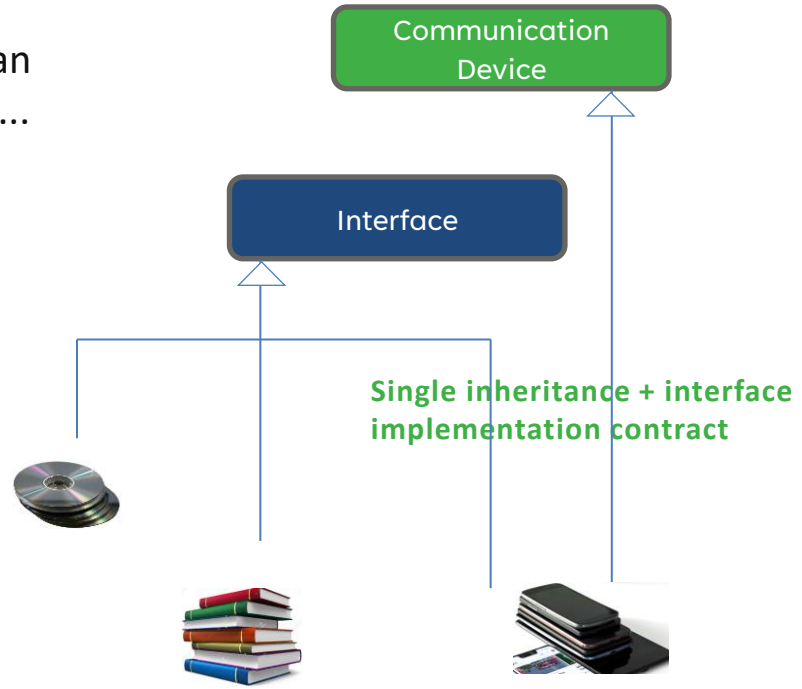
Any problems with this solution?

INTERFACE

The what

Interface

- This process goes as...
  - You will define an **interface** or use an existing interface (such as IComparable). You only make sure, and you will not forget, to have the classes you would like your Robot to be able to sort the objects of, to implement the interface and thus, its methods. Interfaces define a set of features that a class can implement.
  - *Terminology Note:* In OOP, a class **inherits** from "another class". A class **implements** an "interface". When a class implements an interface, the binding contract is set, where...
    - The interface will **not** have any implementation for the comparison/sorting logic.
    - The **child classes will have to implement** their comparison/sorting logic.

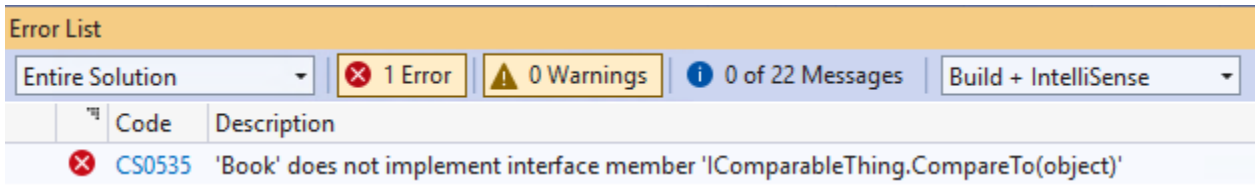


The how

Implementation in C#

- This involves a few steps…
  - The class implements (:) the interface **IComparable**
  - The class implements the **CompareTo** method of the interface for the comparison behavior
  - The **CompareTo** method will be used for sorting of objects

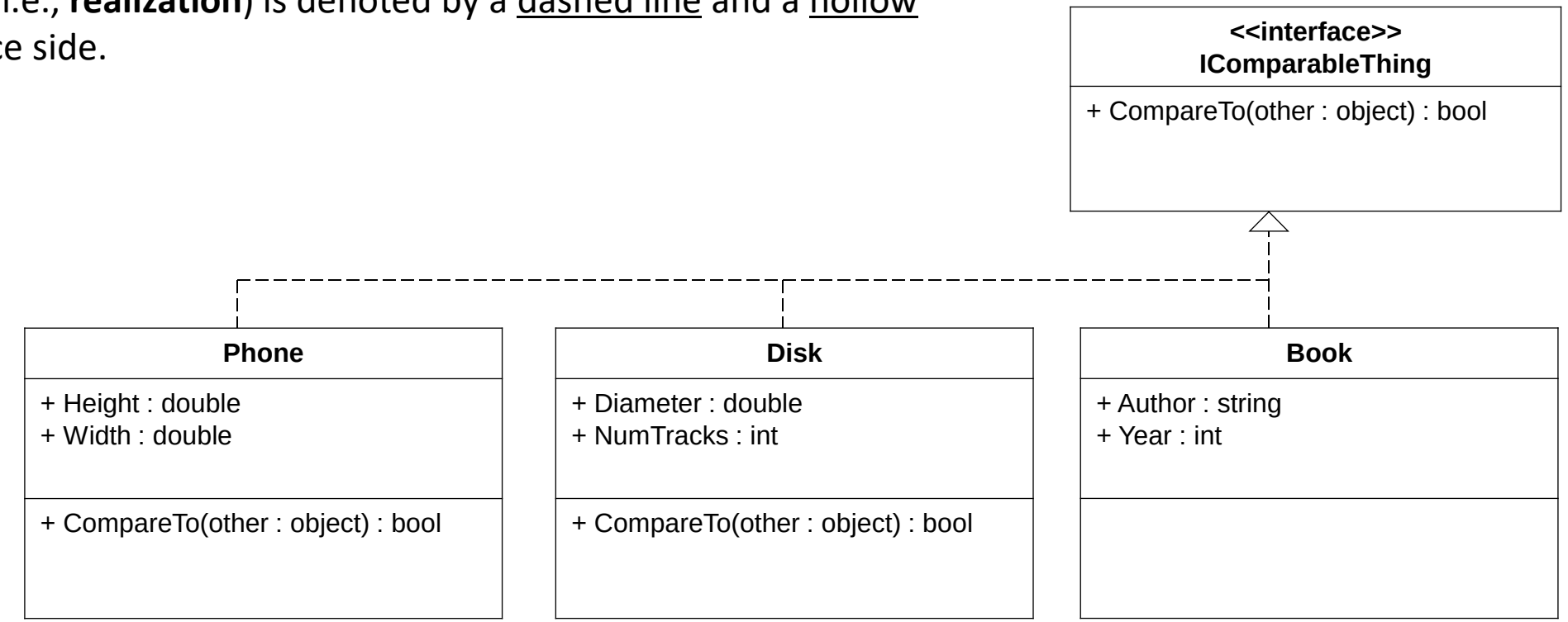
```
130  public interface IComparableThing
131  {
132      bool CompareTo(Object other);
133  }
134
135  public class Phone : IComparableThing
136  {
137      public double Height;
138      public double Width;
139      public bool CompareTo(Object other) { return this.Height > ((Phone)other).Height ? true : false; }
140  }
141
142  public class Disk : IComparableThing
143  {
144      public double Diameter;
145      public int NumTracks;
146      public bool CompareTo(Object other) { return this.Diameter > ((Disk)other).Diameter ? true : false; }
147  }
148
149  public class Book : IComparableThing
150  {
151      public string Author;
152      public int Year;
153  }
```



The how

Implementation in UML

- A class diagram uses the stereotype **<<interface>>** to represent an interface implemented by a class. In a class diagram, note that the relationship between a class and an interface (i.e., **realization**) is denoted by a dashed line and a hollow triangle on the interface side.



Generics

Unspecified types in C#

- Add the concept of **parameterized types** to .Net
  - Design classes and methods **without** any **specific type**
  - The type information will be added by the users of the generics
  - This will make the method or a class to work with any data type
  - For more info, see:

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/

T can be any data type

```
155    public interface IComparableThing<T>
156    {
157        bool CompareTo(T other);
158    }
159
160    public class Laptop : IComparableThing<Laptop>
161    {
162        public double CPUSpeed;
163        public bool CompareTo(Laptop other) { return this.CPUSpeed > other.CPUSpeed ? true : false; }
164    }
```

Abstract class vs. interface

Abstract class

- An abstract class can contain both properties and instance fields
- An abstract class can contain both abstract method signatures and non-abstract methods with implementation
- No object instances can be created from an abstract class
- One class can inherit directly from only one abstract class in C#

Interface

- An interface can only contain properties but no instance fields
- No method in an interface has implementation
- No object instances can be created from an interface
- One class can implement more than one interface

Quiz

Consider the code…

- Which statement is incorrect:
  - A) The **Processor** class implements the **IProcessor** interface correctly.
  - ✓ B) The **Processor** class must implement **ExtraMethod** to satisfy the **IProcessor** interface.
  - C) The **proc** variable in **Main** can only call the **Process** method.
  - D) **ExtraMethod** is not accessible through the **IProcessor** interface.

```
166  interface IProcessor
167  {
168      void Process();
169  }
170
171  class Processor : IProcessor
172  {
173      public void Process() { }
174
175      public void ExtraMethod() { }
176  }
177
178  class Program
179  {
180      static void Main()
181      {
182          IProcessor proc = new Processor();
183          proc.Process();
184      }
185  }
```

DELEGATE

The why

Telling a method what to do

- What if we want to have different ways to compare two objects, e.g., two books can be compared based on…
  - price
  - the number of pages
  - publication date
  - etc.

- What if we wanted the **caller code** to pass some behavior to the code that is **called**…
  1. Wrap behavior as a method reference
  2. Pass the method reference when calling a method as an argument
  3. **Callback**: Call the behavior that is implemented within the caller code

The what

In C#...

- A **delegate** is a type that represents references to methods with a specific parameter list and a return type. Essentially, a delegate allows you to encapsulate a method into a delegate object, which can then be invoked (called). Thus, a delegate allow for…
  - Storage and working with methods like variables (i.e., passing methods as arguments to other methods). This is another way in which **encapsulation** is implemented.
  - The implementation of **callback**
  - (Recommended!) For more examples, see https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates

The how

Implementation in C#...

- This involves…
  - A delegate signature 1
  - Del. signature matching method in caller code 2
  - A delegate object and passing it to code that is called 3
  - Calling back the delegated behavior from within code that is called 4

```
43          CallerClass callerClassObject = new CallerClass();
44          callerClassObject.MainMethod();
45          Console.ReadLine();
46      }
```

```
96  1 public delegate void BehaviorDelegate(string input);
97
98    public class CallerClass
99    {
100 2      private void DelegateMacthingBehavior(string message)
101       {
102           Console.WriteLine(message);
103       }
104
105       public void MainMethod()
106       {
107 3         BehaviorDelegate myDelegate = DelegateMacthingBehavior;
108         CalledClass calledClassObject = new CalledClass();
109         calledClassObject.CalledMethod(myDelegate);
110       }
111   }
112
113   public class CalledClass
114   {
115       public void CalledMethod(BehaviorDelegate del)
116       {
117           Console.WriteLine("What I am doing now is: Doing my job!");
118 4         del("What I am doing now is: Callback!");
119       }
120   }
121 }
122
```

```
C:\Users\bahadorreza\source\repos\SIT771\SIT771\bin\Debug\SIT771.exe
What I am doing now is: Doing my job!
What I am doing now is: Callback!
```

The how

Another example

- Sorting lists using delegates
  - C# Lists have an overloaded method Sort() that can be used to sort objects
    - Sort()
    - Sort(**Comparison<T>** comparison)
    - …

```
58    delegate int Comparison(Object first, Object second);
59
60    delegate int Comparison<T>(T first, T second);
```

```
71    public class StockManager
72    {
73        public List<StockItem> Items;
74
75        private int CompareCosts(StockItem item_1, StockItem item_2)
76        {
77            return item_1.cost.CompareTo(item_2.cost);
78        }
79
80        public void SortByCost()
81        {
82            Comparison<StockItem> compare = CompareCosts;
83            Items.Sort(compare);
84        }
```

delegate-matching method

delegating the sorting behaviour

How do we show delegates and their relationships with classes in UML?

Discuss on CloudDeakin!

Can we have multiple methods delegated through one delegate reference?

Discuss on CloudDeakin!

LAMBDA EXPRESSION

Anonymous methods

In-line methods with no name

- No overhead of declaring an explicit method in the class
- Can be declared in two ways...
  1. (input-parameters) => expression
  2. (input-parameters) => { <sequence-of-statements> }

```
71  public class StockManager
72  {
73      public List<StockItem> Items;
74
75      //private int CompareCosts(StockItem item_1, StockItem item_2)
76      //{
77      //    return item_1.cost.CompareTo(item_2.cost);
78      //}
79
80      public void SortByCost()
81      {
82          //Comparison<StockItem> compare = CompareCosts;
83          Comparison<StockItem> compare;
84          compare = (StockItem item_1, StockItem item_2) => item_1.cost.CompareTo(item_2.cost);
85          Items.Sort(compare);
86      }
```

lambda declaration operator

Closure

Accessing non-local variables

- As lambdas are coded within other methods, they gain a special feature known as **closure**.
- Closure is an enclosed code block that can access data from within its enclosing or referencing outside environment.

accessible within the scope of the closure/lambda block although not declared within the code-block of the lambda statements

```
88      public void SortByCost_2()
89      {
90          bool compareName = true;
91          Comparison<StockItem> compare;
92
93          compare = (StockItem item_1, StockItem item_2) =>
94          {
95              if (compareName)
96                  return item_1.name.CompareTo(item_2.name);
97              else
98                  return item_1.cost.CompareTo(item_2.cost);
99          };
100         Items.Sort(compare);
101     }
```

Epilogue

I’M NOT A GREAT PROGRAMMER; I’M JUST A GOOD PROGRAMMER WITH GREAT HABITS...

KENT BECK