Basic OOP in C#

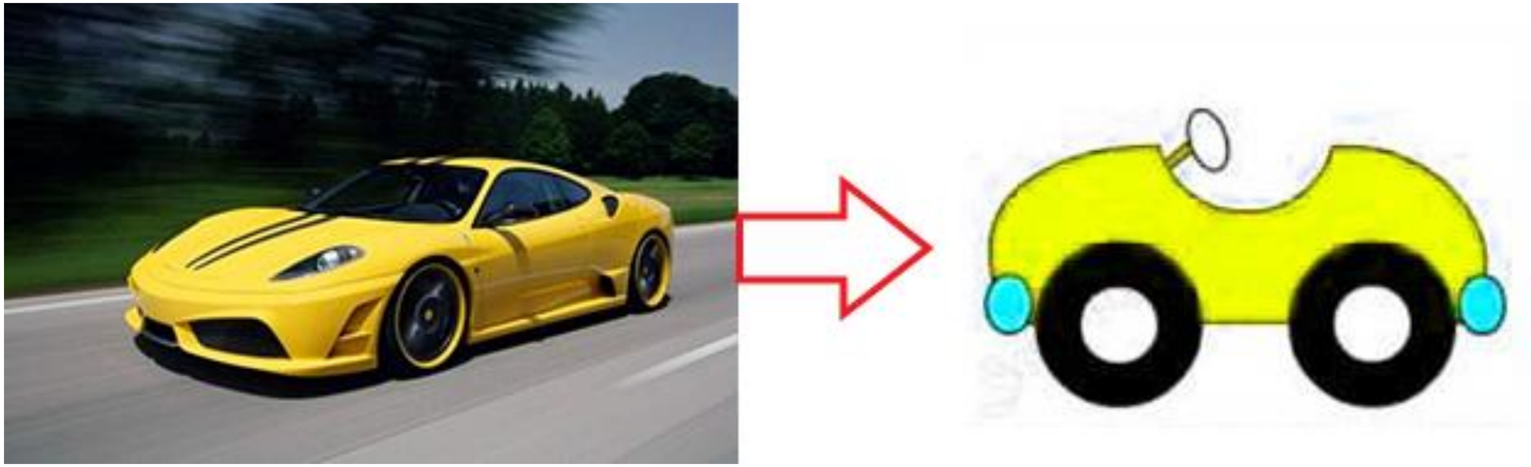
Agenda

- ❑ Abstraction
- ❑ Encapsulation
- ❑ Inheritance
- ❑ Polymorphism
- ❑ Abstract Class & Interface

Abstraction

The art of being wise is the art of knowing what to overlook

Abstraction



take a simpler view of a complex concept.

Reduce complexity by **focusing on the essentials** relative to perspective of viewer

Abstraction
Sample

- ❑ Object: Sport Car
  - ✓ Data: Information
    - Wheel: 4 wheels
    - Main color: Yellow
    - Rear port: 2 ports
    - With upper window: Yes
    - Seat: 2 seats
    - Cylinder volume:2.1L
  - ✓ Action
    - Engine start
    - Speed up, Slow down
    - Turn left, turn right
    - Stop

Abstraction
What is an object?

- Represent an entity in the "real" world

Mary's Car

Petter's Car

- □ Possesses operation (behavior) and attributes (data – state)
  - ★ Data ➔ contain information describe

    the state of objects
  - ★ Operation/Behavior ➔ Method inside class

    Consists of things that the object know how to do
- □ Unique – Identifiable

How are you?
I'm fine!

- □ Is a "black box" which receives messages

Abstraction
What is a class?

- □ Abstract description of a set of objects
- ❑ Class defines methods, variables for a kind of object
- □ We actually write code for a class, not object
- □ Use class as a **blue-print** to create (instantiate) an object

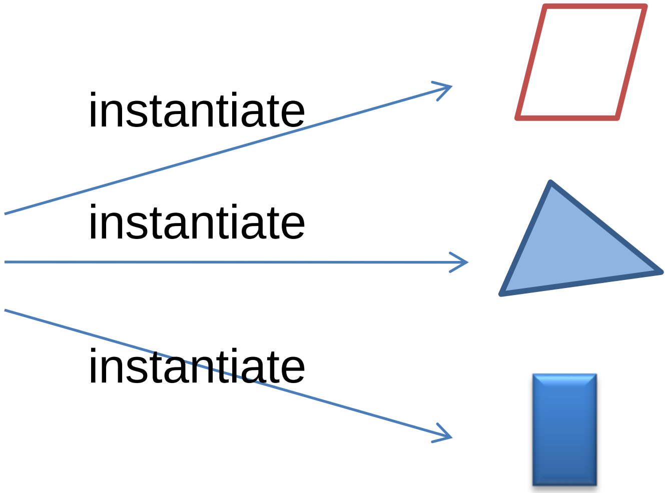
Polygon

Attributes:

- vertices
- border color
- fill color

Operations:

- draw
- erase
- move



instantiate

instantiate

instantiate

Abstraction
Class vs. Struct

- ❑ A class or struct defines the template for an object
- ❑ A class represents a reference type
- ❑ A struct represent a value type
- ❑ Reference and value imply memory strategies

When to use struct?	When to use class?
• Instances of the type are small • The struct is commonly embedded in onther type • The struct logically represent a single value • It is rarely “boxed” • Structs can have performance benefits in computational intensive applications	• Defines a reference type • Can optionally be declared as: ✓ Static ✓ Abstract ✓ sealed

Abstraction
Nested/Inner class

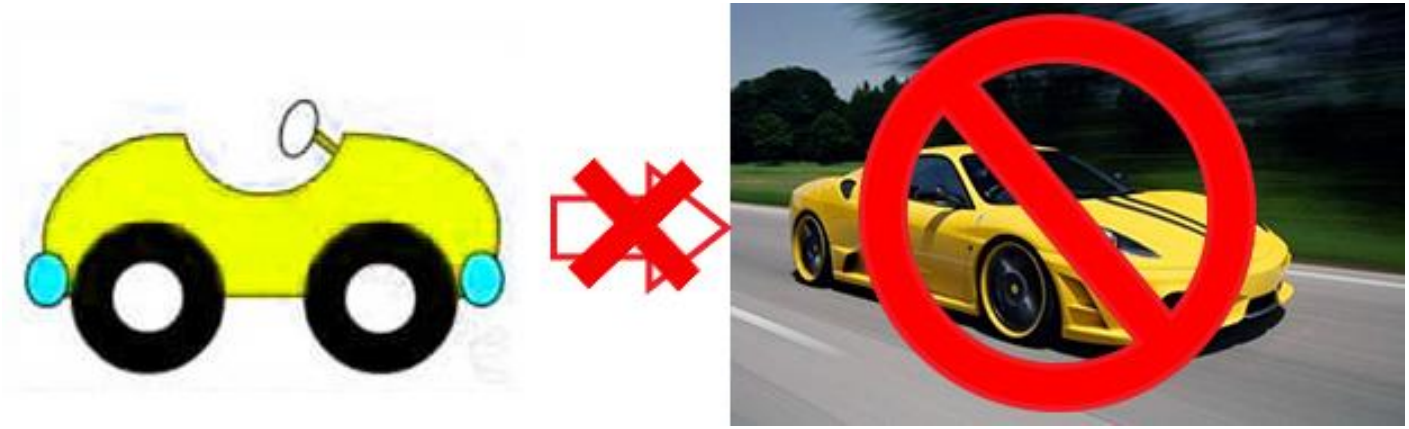
```
class OuterClass{
   private int i;
   public class NestedClass{ // public for outside access
                             // not encouraged
      void methodA(){
         i = 5;              // OK, event i is outer private
      }
      void methodB(){
         int i = 3;          // hide/shadowing the outer i
                             // the outer i member is unchanged
      }
      void methodC(){NestedClass oIC = new NestedClass();}
   }
}
OuterClass oOC = new OuterClass();
OuterClass.NestedClass oIC = new OuterClass.NestedClass();
```

Abstraction
Partial class

```
partial class PartialClass{private int i;}
class PartialClass{}          // error, keyword partial missed
class partial PartialClass{}  // error, invalid keyword order
partial class PartialClass{   // maybe in another cs file but
                              // same namespace is required
   public int Increase(){i++;}
}
…
partial class PartialClass{
   partial class NestedPartialClass{}
}
partial class PartialClass{
   partial class NestedPartialClass{} // OK, nested partial class
}
```

Encapsulation

- **Allow to show only** the important methods as interface
- **Hide** detail information
  - Hide the data item
  - Hide the implementation
  - Access to data item only through member methods



Encapsulation
Class, Member and Method

```
class Car{                                  // Object model
    int NumberWheels;                       // Data => Member
    string MainColor;
    int NumberRearPorts;
    bool isWithUpperWindow;
    int NumberSeats;
    float CylinderVolume;
    void EngineStart(){…}                   // Action => Method
    void SpeedUp(){…}
    void SlowDown(){…}
    void TurnLeft(){…}
    void TurnRight(){…}
    void Stop(){…}
}
```

Object Oriented Programming

Encapsulation
Instantiate, Constructor

```
// Instantiate - Create an object/"instance"
// from its model class with "default constructor"
Car aCar = new Car();
aCar = null;                // now, aCar is no more an object
class Car{
   // Parameterized constructor
   Car(int NumberWheels,    string MainColor,
       int NumberRearPorts, bool   isWithUpperWindow,
       int NumberSeats,     float  CylinderVolume){
       this.NumberWheels      = NumberWheels;
       this.MainColor         = MainColor;
       this.NumberRearPorts   = NumberRearPorts;
       this.isWithUpperWindow = isWithUpperWindow;
       this.NumberSeats       = NumberSeats;
       this.CylinderVolume    = CylinderVolume;
   }
   …
}
// New instantiation with parameterized constructor
Car aCar = new Car(2, "Orange", 2. true, 2, 2.1);
// all members of aCar are now called instance variables
// then, all methods are instance methods
```

Encapsulation
Access Modifiers

- Used for accessibility of data member and member methods
- Access Modifiers: Private, Public, Protected, Friendly (Java)

```
class Car{
    private int NumberWheels; // private
    string MainColor;    // no access modifier means private
    …
    public Car(…){…}     // public constructor
    public void EngineStart(){…}
    public void SpeedUp(){…}
    public void SlowDown(){…}
    public void TurnLeft(){…}
    public void TurnRight(){…}
    public void Stop(){…}
}
Car aCar = new Car(…);
aCar.EngineStart();     // OK, method is public
aCar.NumberWheels = 6; // error: member is private
```

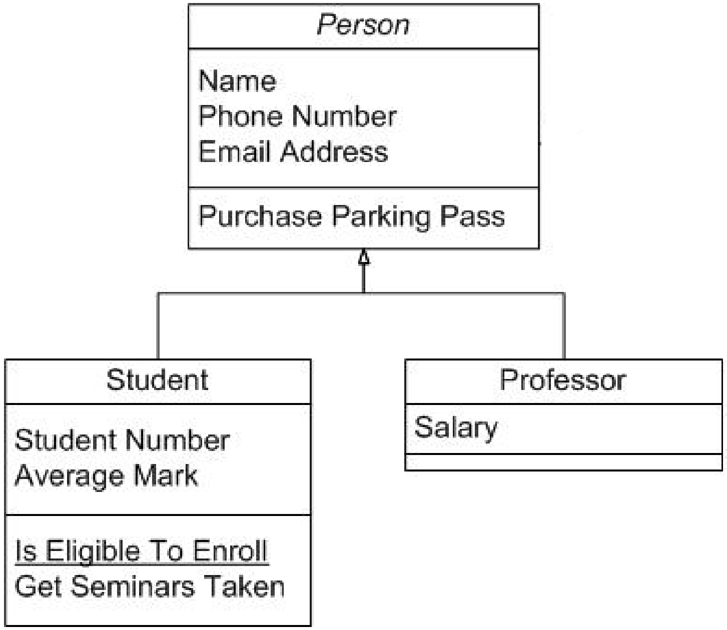
Encapsulation
Information Hiding

- ❑ Implemented by “Private” access specifies
- □ Hidden methods, member data can only be accessed by member methods
- □ Benefit:
  - ★ Your brains doesn’t have to deal with it unless you’re specifically concerned with it
  - ★ When change occurs, the effects are localized

Inheritance

- □ Is the ability to compose new abstraction from existing one

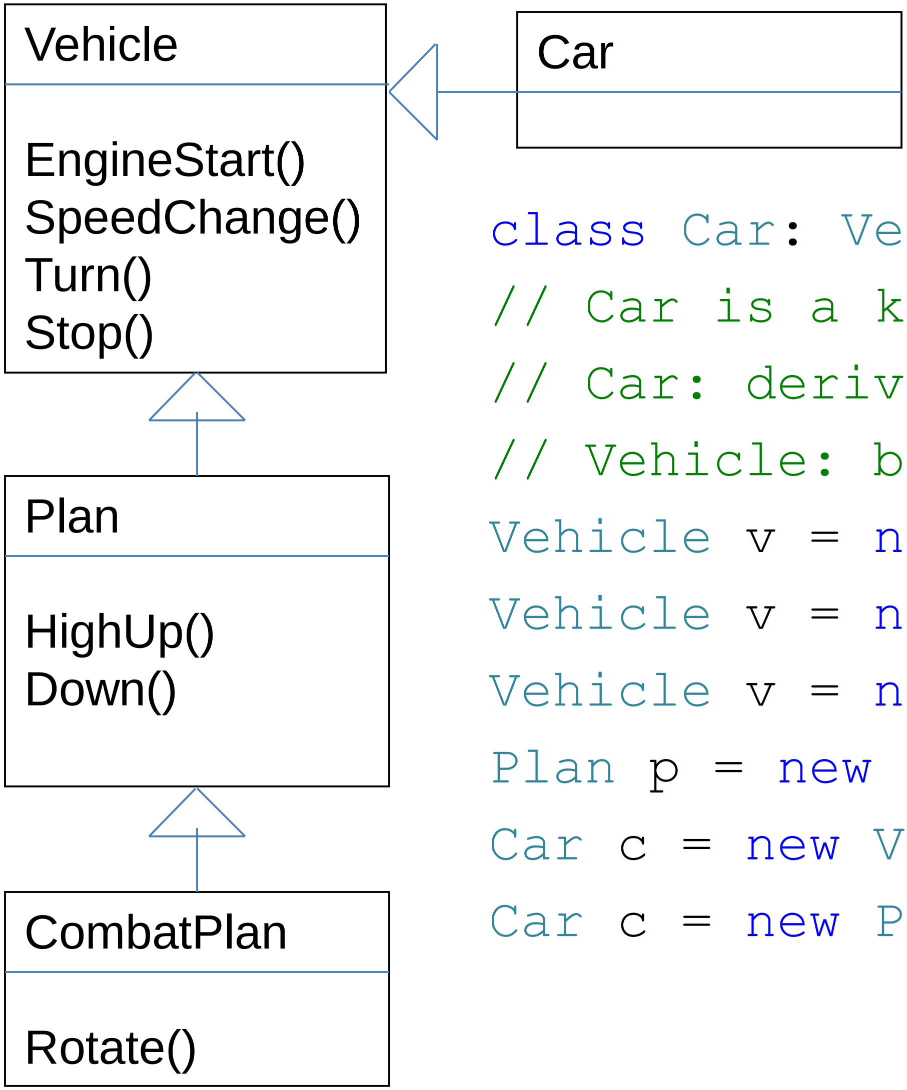
- □ Promotes Re-usability



- □ Base class – Super class: Class provide implementation

- □ Derived class – Subclass: The class inheriting the implementation

Inheritance – Extension



```
class Car: Vehicle {…}
// Car is a kind of Vehicle
// Car: derived/sub class
// Vehicle: base/super class
Vehicle v = new Car();           // OK
Vehicle v = new Plan();          // OK too
Vehicle v = new CombatPlan(); // OK
Plan p = new CombatPlan();       // OK
Car c = new Vehicle();           // error
Car c = new Plan();              // error
```

Inheritance Advantages

- ❑ Development model **closer to real life** object model with hierarchical relationships
- ❑ **Reusability** – reuse public methods of base class
- □ **Extensibility** – Extend the base class
- □ **Data hiding** – base class keeps some data private

  ➔ derive class cannot change it

Protected Accessibility:

```
class Car{
    protected int NumberWheels;
    protected string MainColor;
    protected int NumberRearPorts;
    protected bool isWithUpperWindow;
    protected int NumberSeats;
    protected float CylinderVolume;
    …
}
```

Inheritance
this, base, sealed class

```
class A{
   protected int i;
   protected int aMethod(int i){
      this.i = i; // refer to current object
      return i;
   }
}
class B:A{
   public int aMethod(int i){
     return base.aMethod(i); // refer to parent
   }
}
```

```
sealed class A{} // Inheritance forbidden
class B:A{}      // error

static class C{}
class D:C{}  // error: static implies sealed
```

Polymorphism

❑ Polymorphism = multiple forms/many shape

□ By Definition:

1. The ability of objects to have different operations from the **same interface**.
2. The ability of different objects to respond in their own unique way to the same message

□ Implemented by:

★ Overloading
- function overloading
- operator overloading

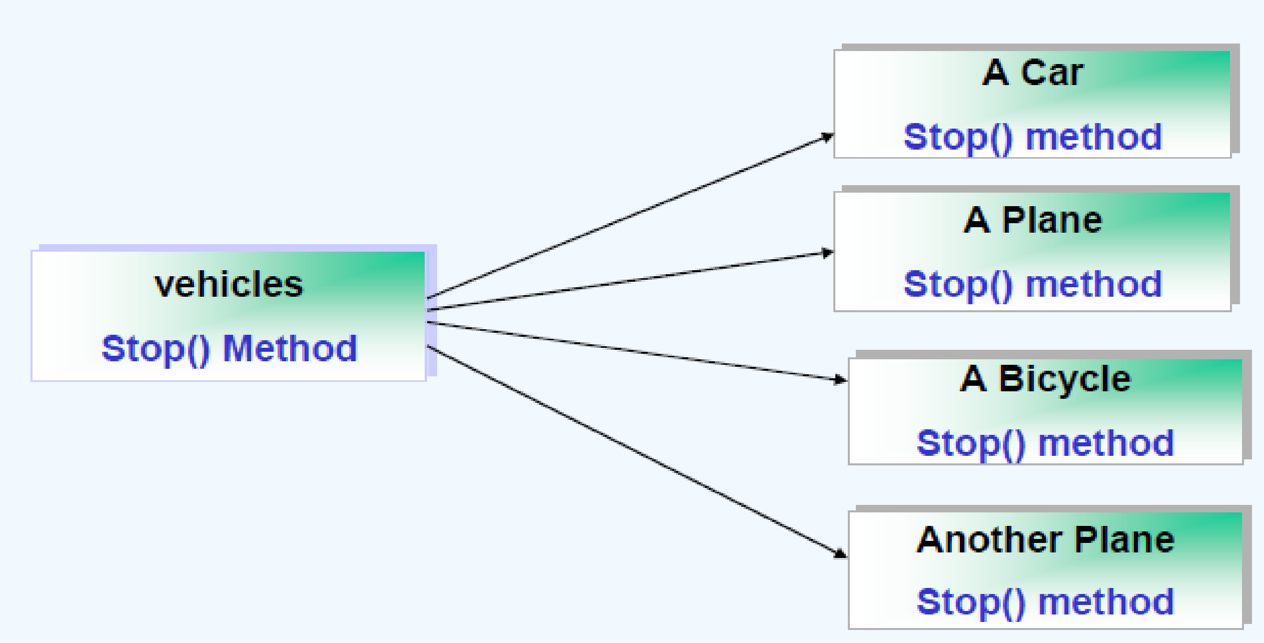
★ Override

Polymorphism
Overloading

- □ *To assign an* ***operator, identifier*** *or* ***literal more than one meaning****, depending upon the* ***data types associated*** *with it at any given time during program execution.*
- □ Two or more method with the **same name, different signature**
- □ Example:
  - ★ void display (int iX)
  - ★ void display (float fY)
  - ★ void display (char[] chC)
  - ★ void display (char[] chC, int offset, int numchar)

Polymorphism
Overriding

- ❑ Process of defining/re-defining the methods in the derived class
- ◻ Methods have same name/same signature



Polymorphism
Override - Feature Hiding

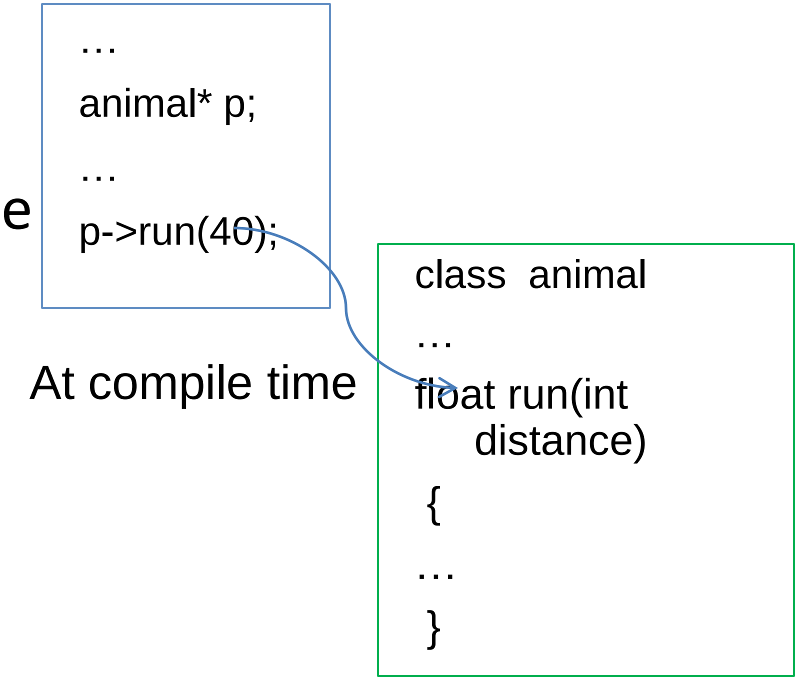
```
class A{
   public int DefaultTempeture(){
      return 25;
   }
}
class B:A{
   // object of type A cannot use the
   // following method,
   // the "new" keyword is optional
   public new int DefaultTempeture(){
      return 28;
   }
}
A a = new B();
int x = a.DefaultTempeture(); // x = 25
```

Polymorphism
Override - Feature Override

```
class A{
   // virtual prevent for override
   public virtual int DefaultTempeture(){
      return 25;
   }
}
class B:A{
   // override: for predefined virtual only
   // override: used for "deferred loading"
   public override int DefaultTempeture(){
      return 28;
   }
}
A a = new B();
int x = a.DefaultTempeture(); // x = 28
```

Polymorphism
Binding

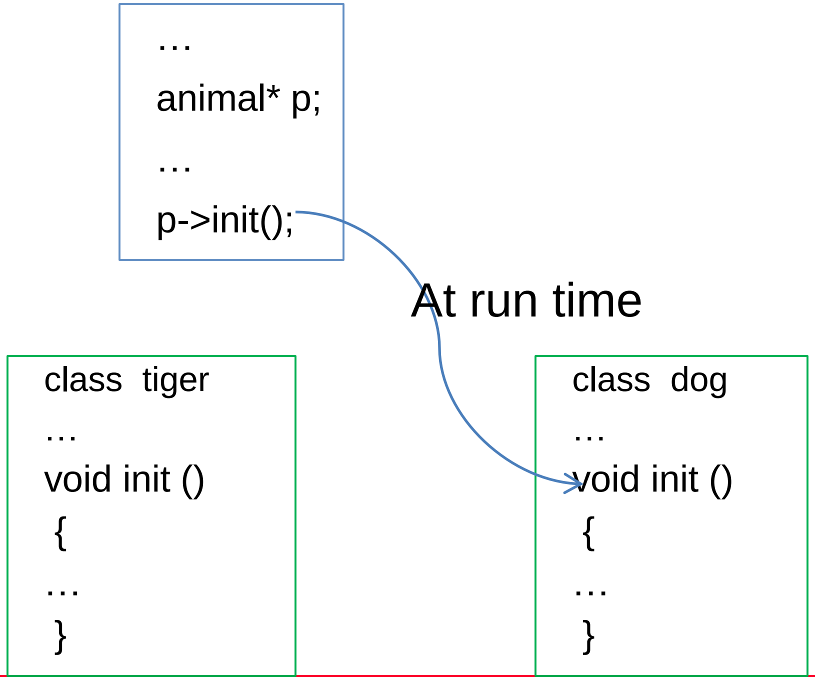
- Process of connecting a method to a method body
  - Static binding:
    - Binding is performed before program runs – At Compile time



```
…
animal* p;
…
p->run(40);
```

At compile time

```
class  animal
…
float run(int
     distance)
 {
 …
 }
```



```
…
animal* p;
…
p->init();
```

At run time

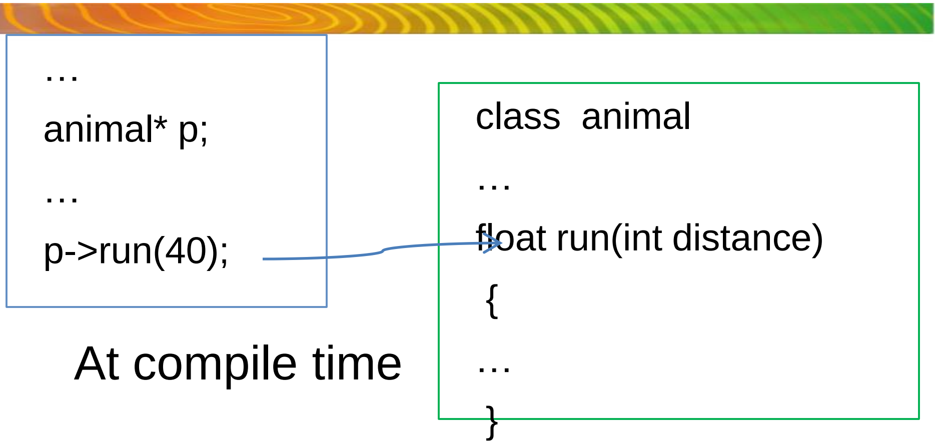
```
class  tiger
…
void init ()
 {
 …
 }
```

```
class  dog
…
void init ()
 {
 …
 }
```

  - Dynamic binding:
    - Binding is performed at the time of execution – Run time

Polymorphism
Static Binding

- ❑ Also called early binding



```
…
animal* p;
…
p->run(40);
```

```
class  animal
…
float run(int distance)
{
…
}
```

At compile time

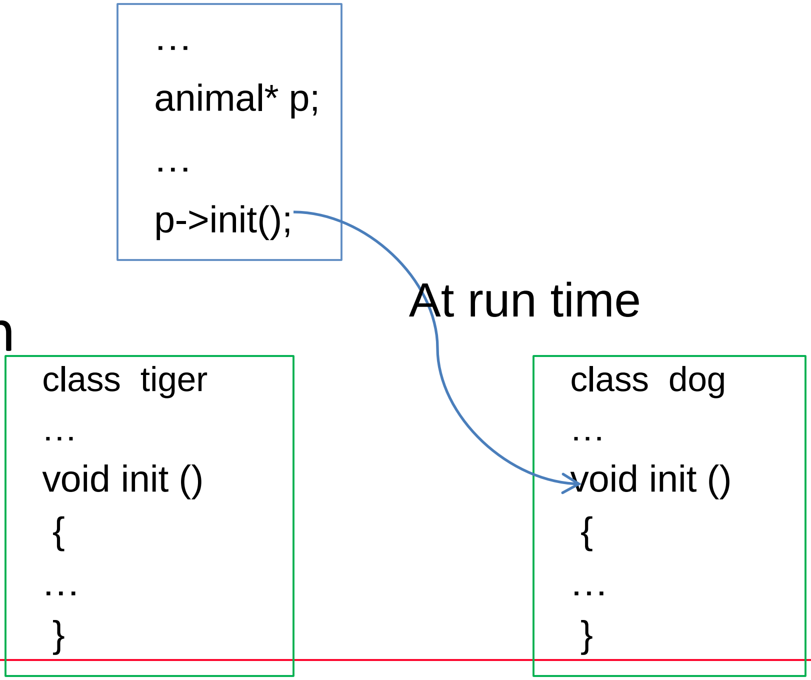
- □ Function overloading implement static binding
- □ Compiler decides the overloaded function to be invoked by looking at **signature**
- □ Binding get earlier:
  - ★ Efficiency goes up – Run-time efficiency – compiler optimize code
  - ★ Safety goes up
  - ★ Flexibility goes down

Polymorphism
Dynamic Binding

- Which method called depends upon the type of object

- The type of object cannot be resolved at the time of compilation.
  - ★ Dynamic binding resolves the method to be used at run-time

- Method overriding :
  - ★ Use dynamic binding
  - ★ Flexible – high level of problem abstraction



```
…
animal* p;
…
p->init();
```

At run time

```
class  tiger
…
void init ()
{
…
}
```

```
class  dog
…
void init ()
{
…
}
```

Abstract Class & Interface

Abstract Class 1/2

- ❑ Class that contains one or more abstract methods
- □ Abstract method: Method that has only declaration but **no implementation**.
- □ Classes that extend abstract class make them concrete by implementing those abstract methods

Abstract Class & Interface

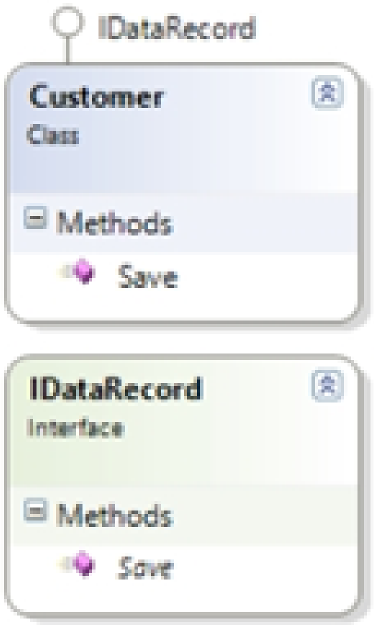
Abstract Class 2/2

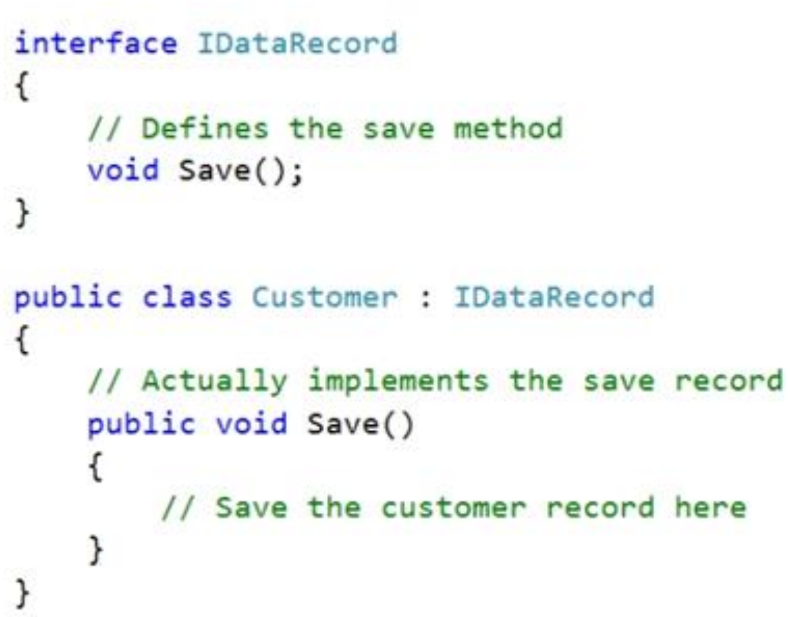
```
abstract class A{       // having at least one abstract method
   // abstract method: no implementation
   public abstract int DefaultTempeture();
   // ordinal method: with implementation
   public int TempIncStep(){
      return 1;
   }
}
class B:A{}
class C:B{}
   public override int DefaultTempeture(){
      return 25;
   }
}
A a = new A(); // error: no infor about DefaultTempeture behavior
B a = new B(); // error: no infor about DefaultTempeture behavior
C a = new C(); // OK
A a = new C(); // OK
B a = new C(); // OK
```

Abstract Class & Interface

Interface 1/3

An interface defines a set of related functionality that can belong to one or more classes or structs.





```
interface IDataRecord
{
    // Defines the save method
    void Save();
}

public class Customer : IDataRecord
{
    // Actually implements the save record
    public void Save()
    {
        // Save the customer record here
    }
}
```

Abstract Class & Interface

Interface 2/3

- ❑ An interface defines a contract
  - ✓An interface is a type
  - ✓Includes methods, properties, indexers, events
  - ✓Any class or struct implementing an interface must support all parts of the contract
- ❑ Interfaces provide no implementation
  - ✓When a class or struct implements an interface it must provide the implementation
- ❑ Interfaces provide polymorphism
  - ✓Many classes and structs may implement a particular interface

Abstract Class & Interface

Interface 3/3

```
interface IA{                // Interface is a special "abstract" class
   int i;                    // Error: no member is allowed
   int NewTempeture{get;}    // OK
   int DefaultTempeture();   // no abstract keyword, no access modifier,
                             // public access is fixed
   int TempIncStep(){        // Error: No ordinal method allowed
      return 1;
   }
}
abstract class A:IA{}        // abstract class can prevent the
                             // implementation of an interface
class B:IA{                  // non-abstract class, when declared to use
                             // an interface, must implement all methods
                             // declared in the interface
   public int DefaultTempeture(){return 1;}
}
class C:A{
   public int DefaultTempeture(){return 2;}
}
IA a = new B(); IA b = new C(); // Interface is a type
```

Abstract Class & Interface
Multi Inheritance

- Classes and structs can inherit from multiple interfaces
- Interfaces can inherit from multiple interfaces

```
class A1{void a1(){}}
class A2{void a2(){}}
class A:A1, A2{}
   // Error: "class multi inheritance" forbidden

interface IA1{void IA();}
interface IA2{void IA();}
class A:IA1, IA2{
   // OK interface "multi interface" implementation
   void IA1.IA(){} // All explicit implementation
   void IA2.IA(){}
}
class B:IA1, IA2{
   void IA(){}     // one implementation for all interface
}
```

Abstract Class vs. Interface

- **Abstract:**
  - Single inheritance
  - Fast performance
  - Security problem in distributed application
  - When base class change lead to the changing derived class.
- **Interface:**
  - Multiple inheritance.
  - Slow performance but flexible
  - Good for separate interface & implementation (eg : plug-in programming)

- More : http://liveonmyown.wordpress.com/2007/09/11/abstract-class-vs-interface/

Lesson Summary

Object-oriented systems describe entities as objects.

Objects are part of a general concept called classes

Abstraction

Encapsulation

Inheritance

Polymorphism

Object Oriented

Abstract class, Concrete class, Base class, Derive class, Attribute, Method, Instance, Instantiation,

Q&A