Object Oriented Programming (OOP) with C#

Alim Ul Karim

alim@developers-organism.com

Microsoft Technical Community(MSTC)

Community Speaker of **Microsoft Bangladesh**

/DevelopersOrganism
developers-organism.com
info@developers-organism.com

Prerequisites

- **Basic C# or java programming skills**
- **Functions**
- **Namespace**
- **Enum**
- Unified Modeling Language: UML
- Property
- Field

Outline

- We are going cover core concepts of OOP
  - Class
    - Constructor
    - Multiple constructor
    - Using, destructor
  - Static: class, member, function
  - Passing reference parameter in function.
  - Generic
  - Object

Outline

- We are going cover core concepts of OOP
  - Inheritance
    - Relationship type **IS-A**
    - Relationship type **HAS-A** (Modular Approach)
    - Extend with Extension methods
  - Encapsulation
    - Getter/Setter
    - C# getter setter
  - Polymorphism
    - Access Modifiers
    - Modifiers
- Strategy Design Pattern (modify HAS-A relationship)

Why? Story History

Object Oriented Programming(OOP)

I will try give real life examples so that you can remember.

Class Object Inheritance Encapsulation Polymorphism

No OOP

Not in good shape

Right OOP

Good shape

Overdoing OOP

Very bad shape

Drawn by Alim Ul Karim

Class

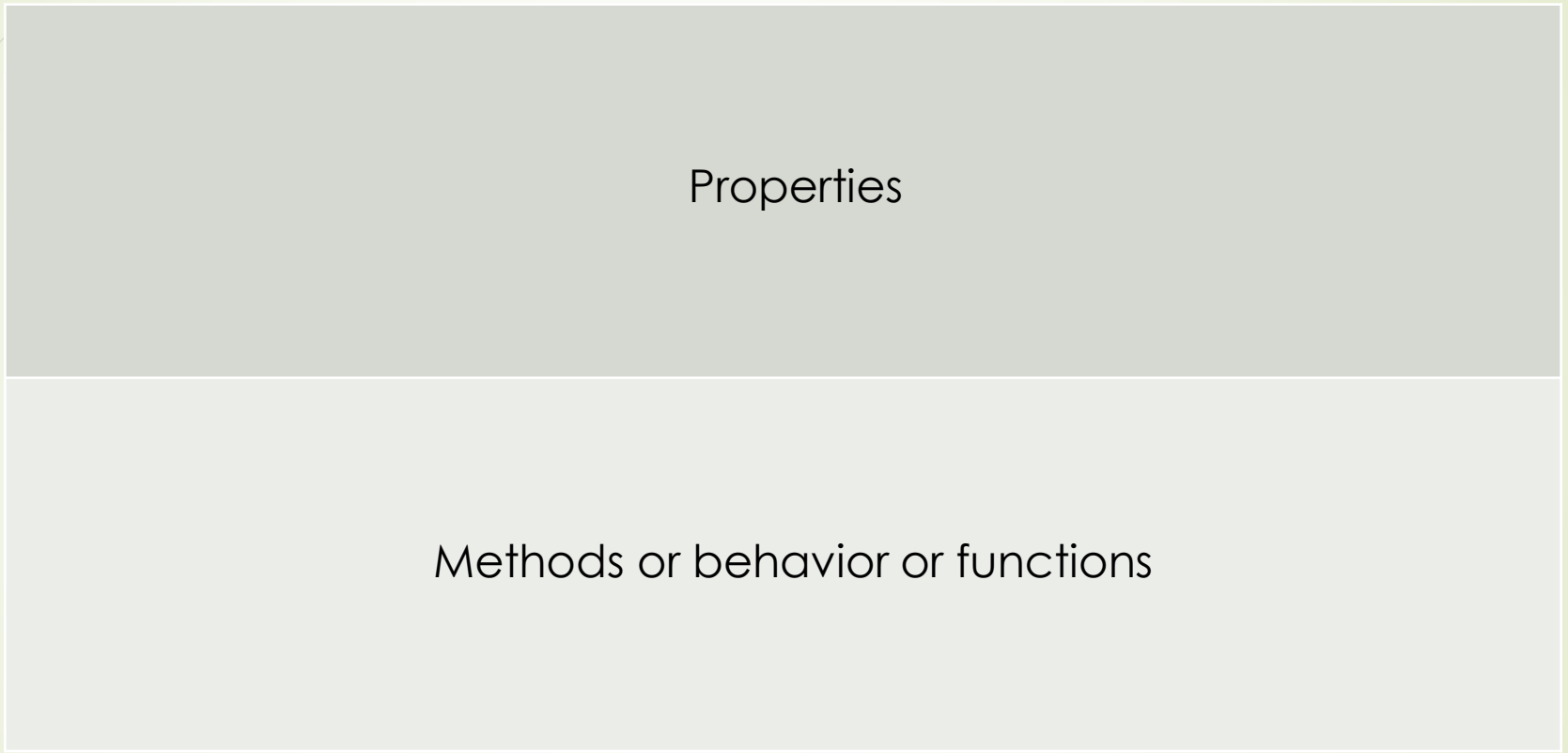
Why do we even need class?

1. Custom data-structure or custom data-type
2. More than one value, however struct is another option `int i = 500`
3. Organize your code

Class

1. The concept of **`Class`** came from biology.
2. Purpose of the **`Class`** is to organize code.
3. **`Class`** itself is an abstract representation of an **`object`** like blue print.
4. In simple words a **`Class`** is a collection of `methods/functions` with `properties` or `attributes` of an **`object`** .

UML : Unified Modeling Language



Class

Person

Attributes / Properties

FirstName: **String**
LastName: **String**
DateOfBirth: **DateTime**
Address: **String**
Gender: **bool**

Walk():void
Talk() :void
Eat():void

Behaviors / Methods or Functions

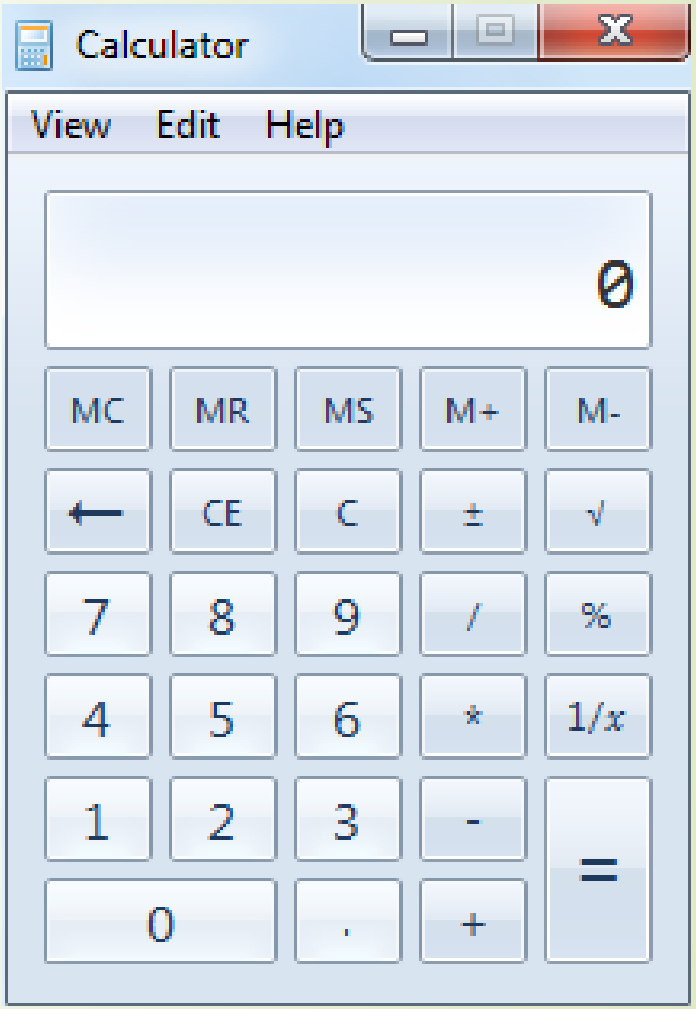
Class

Calculator

BasicCalculation

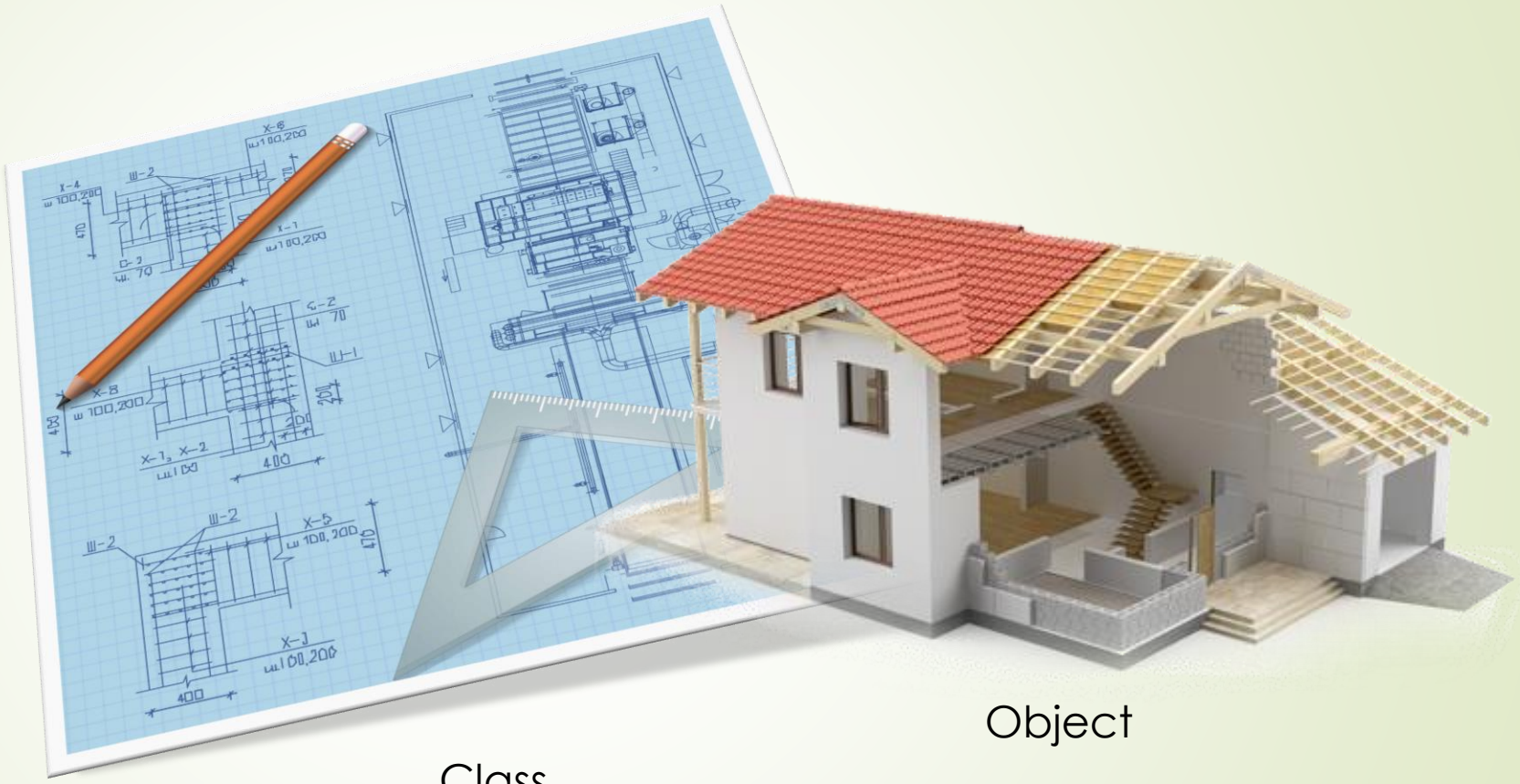
ScientificCalculation

ProgrammingCalculation



Object

```
Class Person {
    ...
    ...
    ...
}
```



Class

Object

```
Person person = new Person();
```

person: object

Person: class

Object

You can get an object from an class after **instantiation** or **creation** of that class.

```
class Person {
    ...
    ...
    ...
}
```

Person

```
Person person = new Person();
person.FirstName = "Alice";
person.LastName = "Romanson";
String firstName = person.FirstName;
```

→ Creation of a object from person class

→ Sets value

→ Gets value

internal

```
class Person {
    public Person() {
        ...
    }

    public string FirstName;
    string LastName;
    ...
    public void Walk() {
        Console.WriteLine(FirstName + " is walking.");
    }
    ...
}

Person person = new Person();
```

given

private

Member of the class

Person

given

```
public class Person {
        public Person() {
            ...
        }

        public string FirstName;
        string LastName;
        ...
        public void Walk() {
            Console.WriteLine(FirstName + " is walking.");
        }
        ...
}

Person person = new Person();
```

Person

Lets see some coding!

Class Constructor

The first method that is invoked when a class is created.

A class can have zero or multiple constructors (VS snippets “ctor”).

Class Destructor

A class can have zero or one destructor.

Usages to clean up memory.

Static Modifier

Access members without **instantiating** the class. Loads before everything shared across the application.

Static

class

```
public static class ClassName {
```

- All members **must** be static or else compilation error.
- Can't be **inherited** or **created**
- Can't have any **constructor** or **destructor**.

Member

- Public static members can be accessible by **class name only** from anywhere **without** **instantiating** the class.
- Private static or only static members can only be accessible within the class from static functions only.
- a non-static member can call private static.

Public static members can be accessible by **class name only** from anywhere **without instantiating** the class.

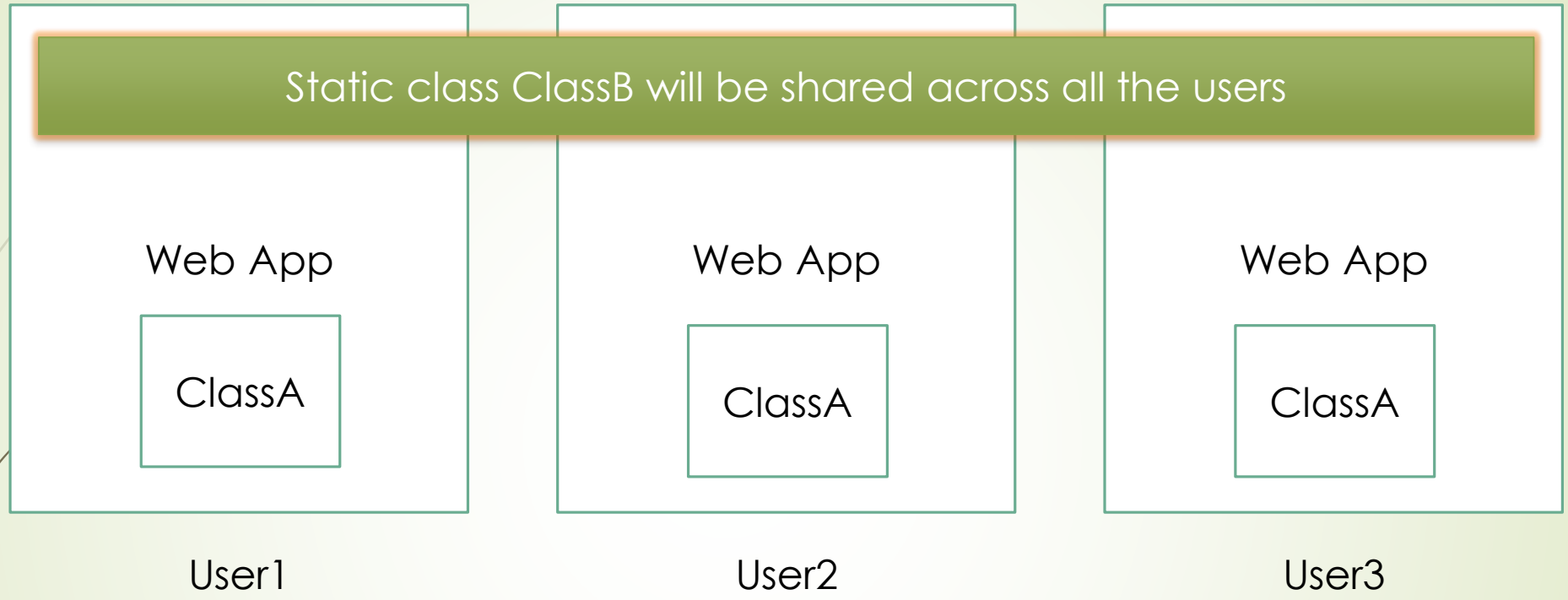
```
class ClassA {
      //public static method
      public static void Print(string x){
            Console.WriteLine(x)
      }
      //private non-static method
      void foo(){
            Console.WriteLine("Called foo.");
            ClassA.Print("Called foo."); ✓ //call public static members by it's class name without instantiating
      }
}
class ClassB {
      //constructor
      public ClassB (){
         ClassA.Print("Called foo."); ✓ //call public static members by it's class name without instantiating
      }
      void Anything(){
         ClassA.Print("Called foo."); ✓ //call public static members by it's class name without instantiating
      }
      static void Anything(){
        ClassA.Print("Called foo."); ✓ //call public static members by it's class name without instantiating
      }
}
```

Private static or only static members can only be accessible within the class from static functions only.

```
class ClassA {
      //private method static method
      static void Print(string x){
            Console.WriteLine(x)
      }
      //non-static method
      void foo(){
            Console.WriteLine("Called foo.");
            ClassA.Print("Called foo."); ✓  //Possible
            this.Print("Called foo."); X   //calling a non-public static member is not possible from non static member.
      }
      static void foo(){
            Console.WriteLine("Called foo.");
            this.Print("Called foo.");  X  //calling a non-public static member is not possible from non static member.
            ClassA.Print("Called foo."); ✓  //calling a static member is only possible inside class static member.
      }

}

class ClassB {
      //constructor
      public ClassB (){
          ClassA.Print("Called foo."); X  //calling a non-public static member is not possible outside the class.
      }
      void Anything(){
          ClassA.Print("Called foo."); X  //calling a non-public static member is not possible outside the class.
      }
      static void Anything(){
          ClassA.Print("Called foo."); X  //calling a non-public static member is not possible outside the class.
      }
}
```

Let's visualize static objects

Let's say you have an web app


Static class ClassB will be shared across all the users
Web App
ClassA
User1
Web App
ClassA
User2
Web App
ClassA
User3

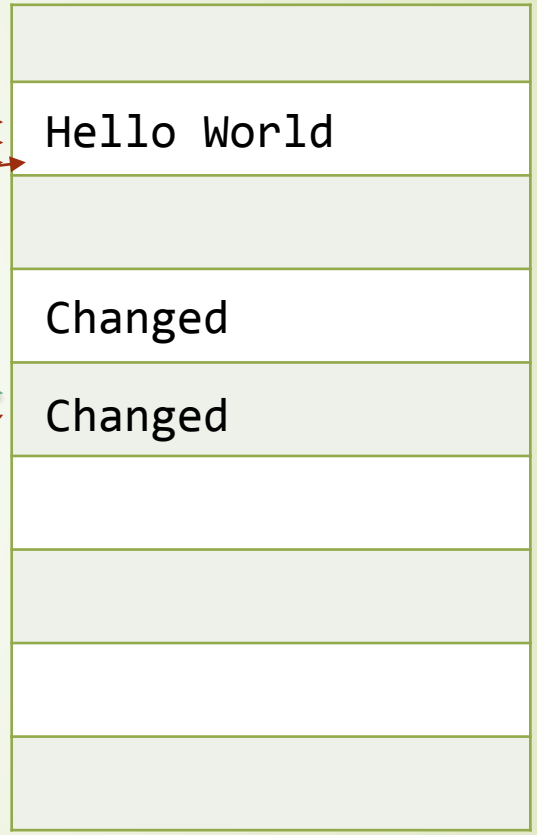
Passing reference Parameter

Passing data as reference and non reference in function.

```
class ClassA {
     static void main(string[] args) {
          string x = “Hello World”;
          Console.WriteLine(x); // prints Hello World
          Change(x);
          Console.WriteLine(x); // prints Hello World
          ChangeRef(ref x);
          Console.WriteLine(x); // prints Changed
          Console.ReadLine();

     }

     //non reference type method
     public static void Change(string x){
          x = “Changed”;
     }

     //reference type method
     public static void ChangeRef(ref string x){
          x = “Changed”;
     }
}
```

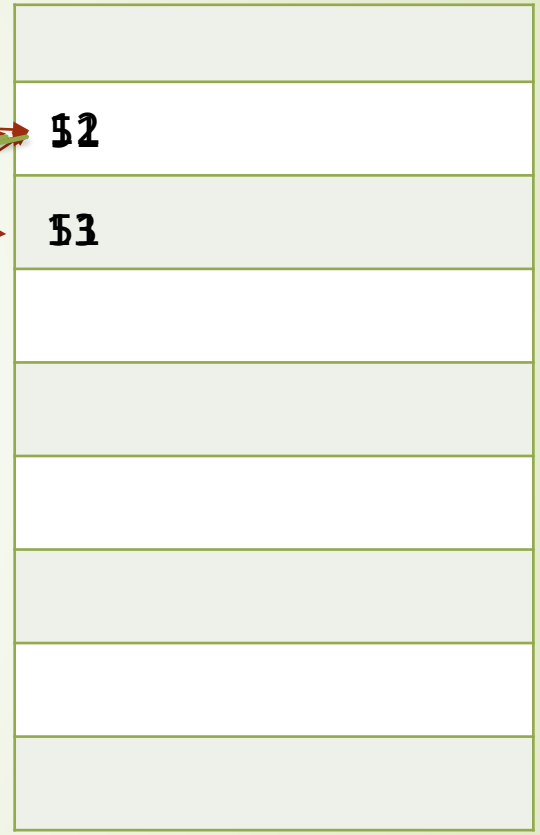
String is a immutable type which means it is fixed type.

```
class ClassA {
     static void main(string[] args) {
          string x = "Hello World";
          Console.WriteLine(x); // prints Hello World
          Change(x);
          Console.WriteLine(x); // prints Hello World
          ChangeRef(ref x);
          Console.WriteLine(x); // prints Changed
          Console.ReadLine();

     }

     //non reference type method
     public static void Change(string x){
          x = "Changed";
     }

     //reference type method
     public static void ChangeRef(ref string x){
          x = "Changed";
     }
}
```



RAM

```
class ClassA {
     static void main(string[] args) {
          int x = 51;
          Console.WriteLine(x); // prints 51
          Change(x);
          Console.WriteLine(x); // prints 51
          ChangeRef(ref x);
          Console.WriteLine(x); // prints 12
          Console.ReadLine();

     }

     //non reference type method
     public static void Change(int x){
          x = 13;
     }

     //reference type method
     public static void ChangeRef(ref int x){
          x = 12;
     }
}
```



RAM

Passing class as datatype

When a class datatype is send to parameter , it sends as a reference of that type.

var vs. dynamic

Interview Question

var: is strongly typed, it's like I am too lazy to figure out the type and compiler will put the exact type in the compile time.

dynamic: is strongly typed dynamic datatype, which is very confusing at first.

Generic Class

class ClassIdentifier<T,K,L,….>

```
class ClassA {
     public void Add(object x){
          x += 1;
     }
}
```

```
class ClassB<T> {
     public void Add(T x){
          x += 1;
     }
}
```

Why use generics, we could use object instead?

It impacts on performance.

C# Class Pattern

```
[access modifier] [modifier] class[<Generic>] identifier {
    // [optional] zero or multiple constructors
    // [optional] data or field or properties or attributes
    // [optional] logic or behaviors or functions or methods
    // [optional] can have zero or one destructor
}
```

[access modifier]	Description	[modifier]	Description
public	No restriction	static	Can't be created as well as inherited
internal or none	Only accessible from same project or assembly	sealed	Can't be inherited
		partial	Same class merge on compile time if within same namespace.
		sealed partial	Can't inherit + merge with same name class on compile time.

Class & Object Conclusion

Whenever you create a class you create a new type of data type like String or List object.

Inheritance

It is like getting something from the **previous generation**.

Person

FirstName: **string**
LastName: **string**
DateOfBirth: **date**
Address: **string**
Gender: **bool**

Walk():void
Talk() :void
Eat():boolean

Student

FirstName: **string**
LastName: **string**
DateOfBirth: **date**
Address: **string**
Gender: **bool**
CGPA: float
Completed Courses: byte
Joined Date: date
…

Display():void
….

Teacher

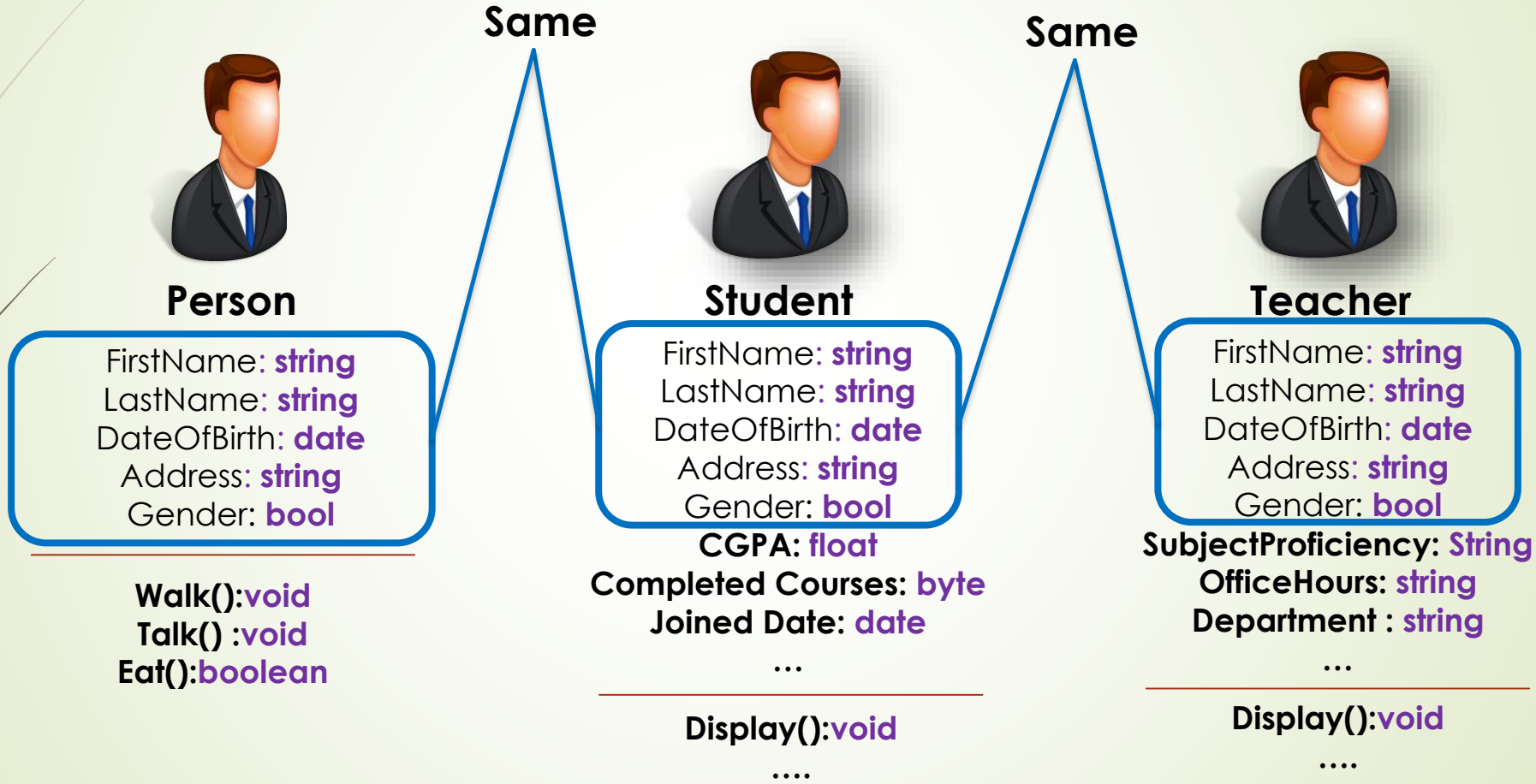
FirstName: **string**
LastName: **string**
DateOfBirth: **date**
Address: **string**
Gender: **bool**
SubjectProficiency: String
OfficeHours: string
Department : string
…

Display():void
….

Lets see some coding!

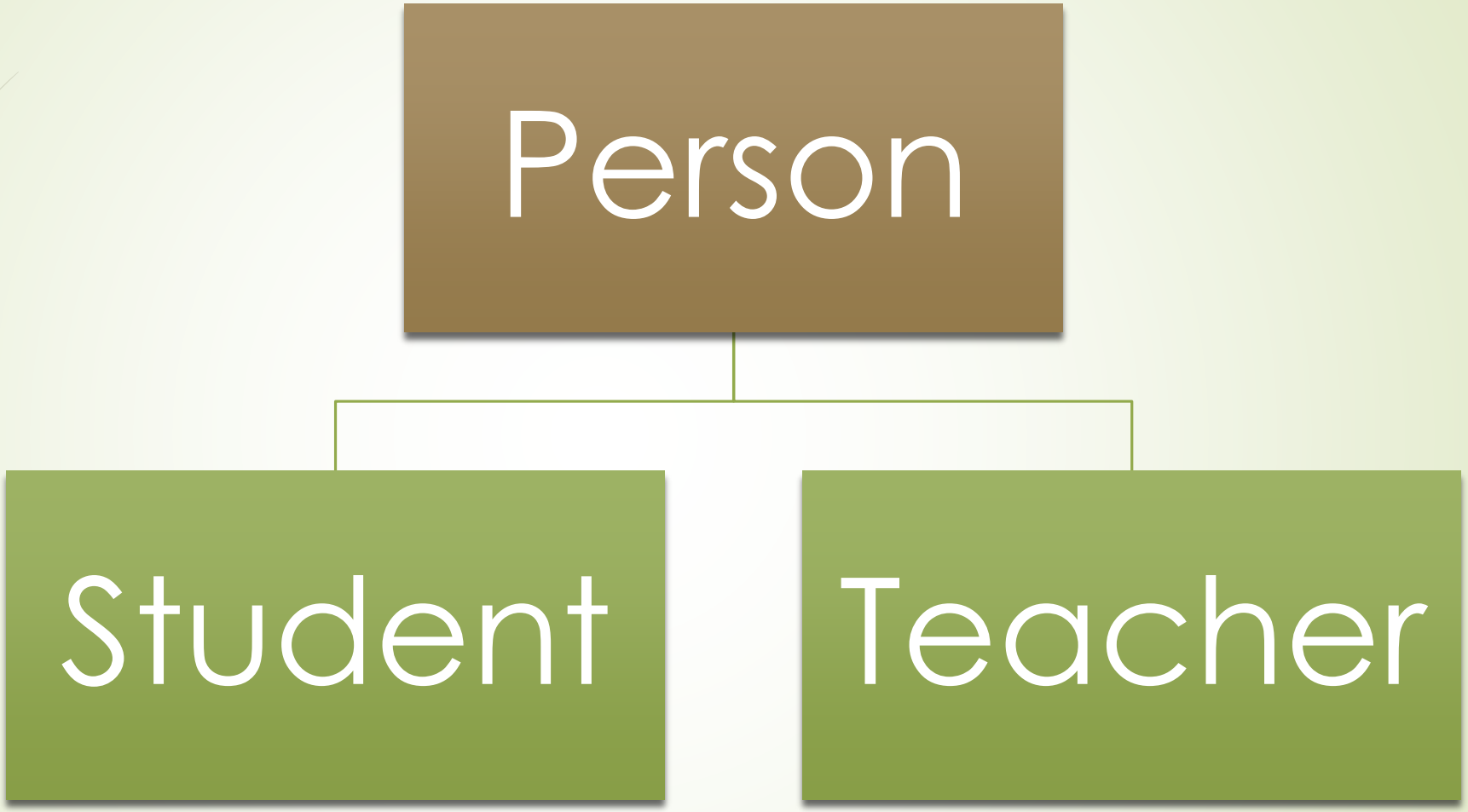
Inheritance

It is like getting something from the **previous generation**.



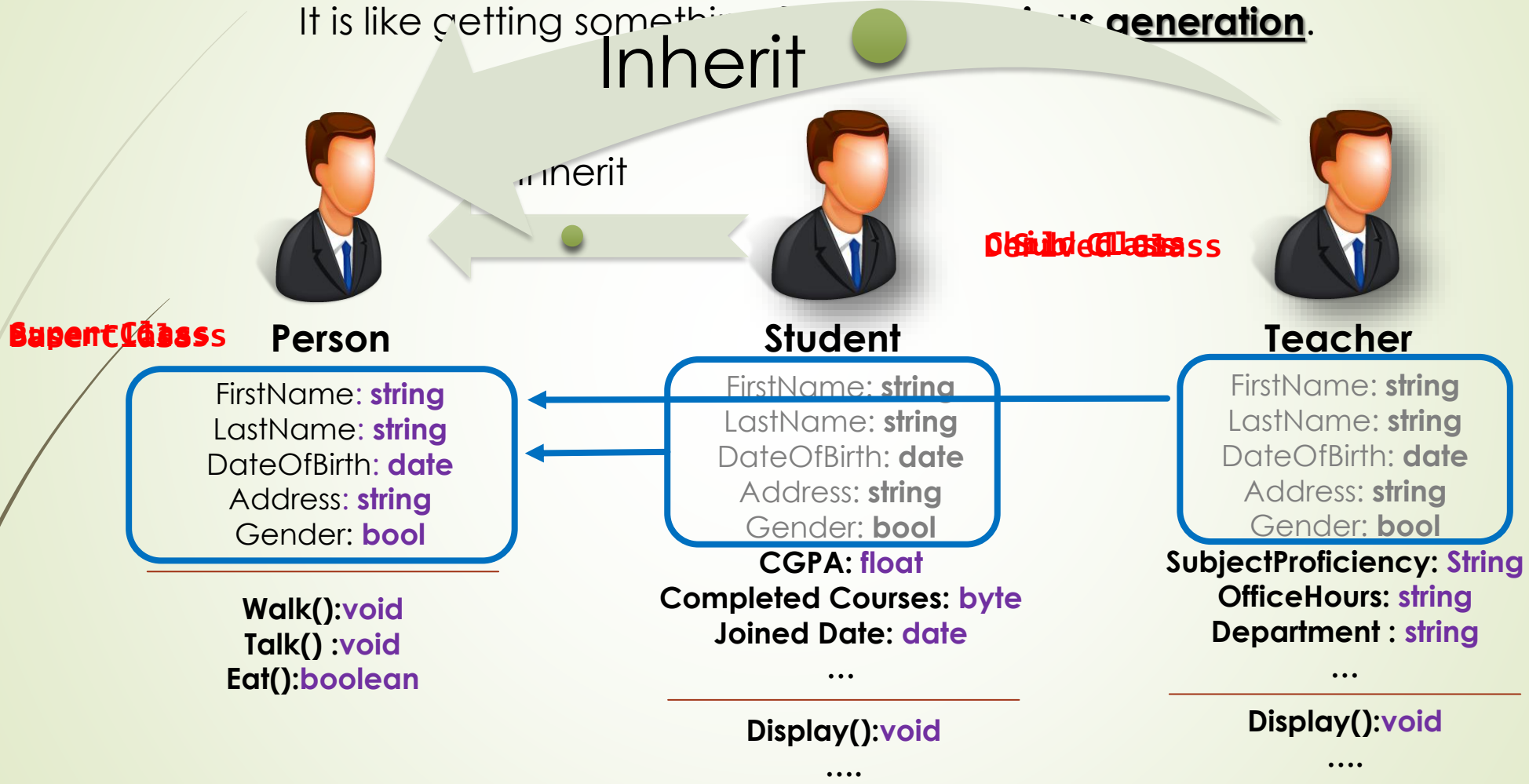
DRY

Don’t Repeat yourself.


Person
Student
Teacher
IS-A
Person
IS-A
Person

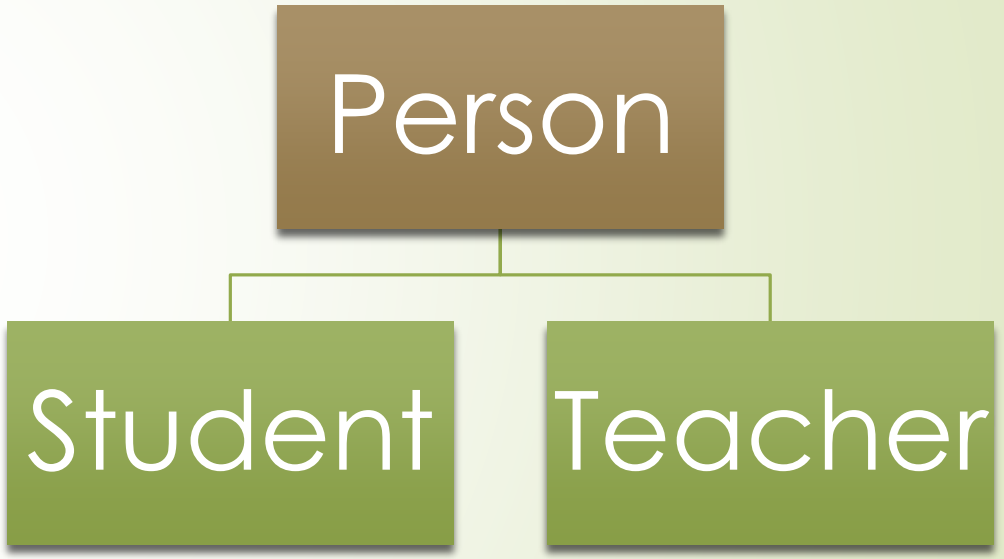
Inheritance

It is like getting someth... ...us **generation**.



Inheritance

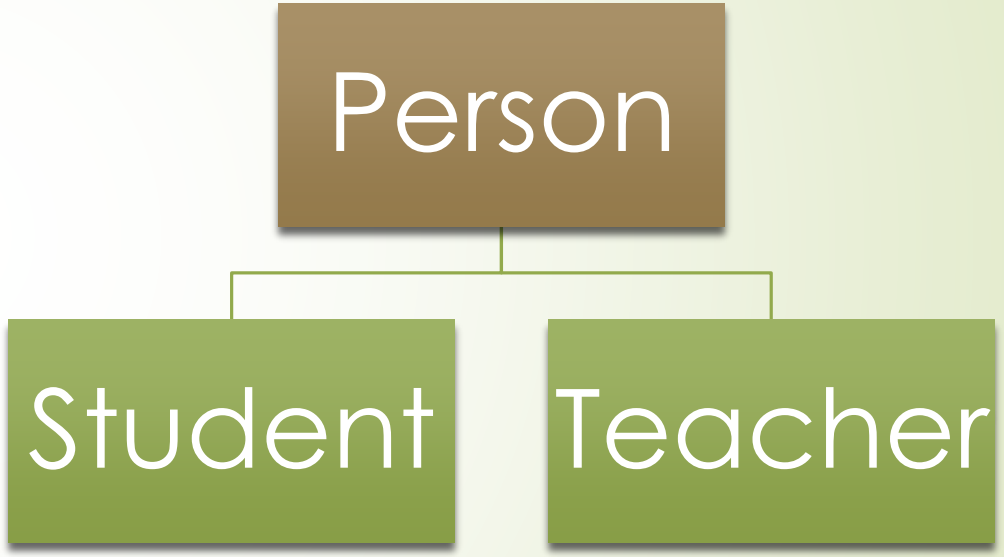

```
Class Person {
    ...
    ...
}

Class Student inherits from Person {
    ...
    ...
}

Class Teacher inherits from Person {
    ...
    ...
}
```




```
Student student = new Student();
student.FirstName = “Alice”;
student.LastName = “Romanson”;
```

Inheritance


```
Class Person {
    ...
    ...
}

Class Student:Person {
    ...
    ...
}

Class Teacher:Person {
    ...
    ...
}
```




```
Teacher teacher = new Teacher();
teacher.FirstName = “Alice”;
teacher.LastName = “Romanson”;
```

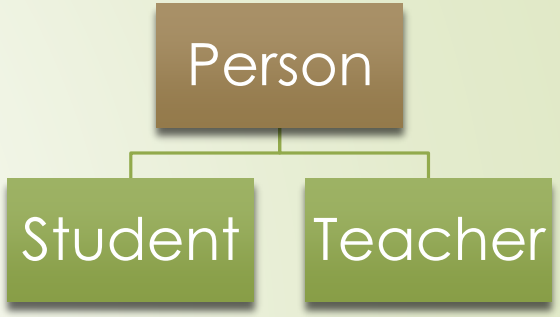
Inheritance


```
Class Person {
      ...
      ...
}

Class Student:Person {
      ...
      ...
}

Class Teacher:Person {
      ...
      ...
}
```

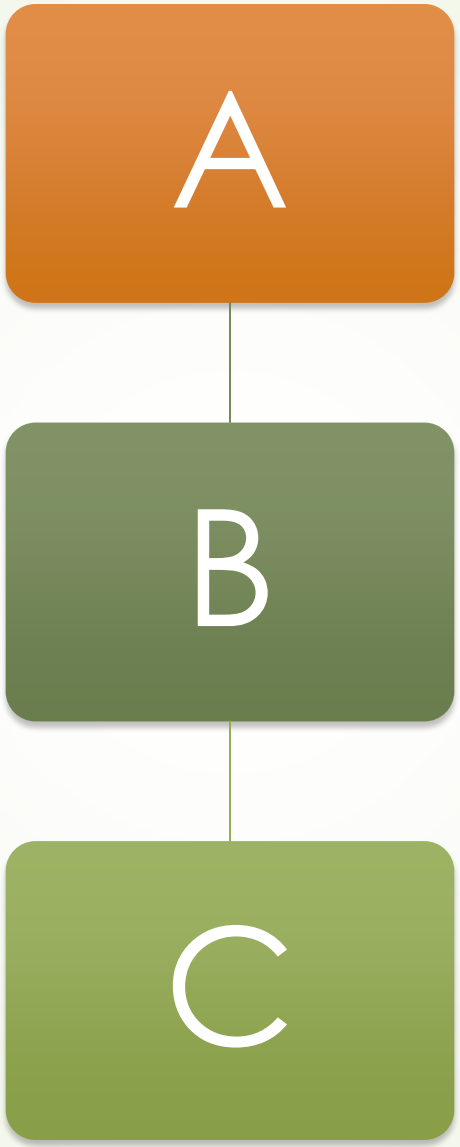
```
Student student = new Student();
student.FirstName = "Alice";
student.LastName = "Romanson";

SuperClass type = new SubClass();

Person student1 = new Student();
Person teacher1 = new Teacher();
```

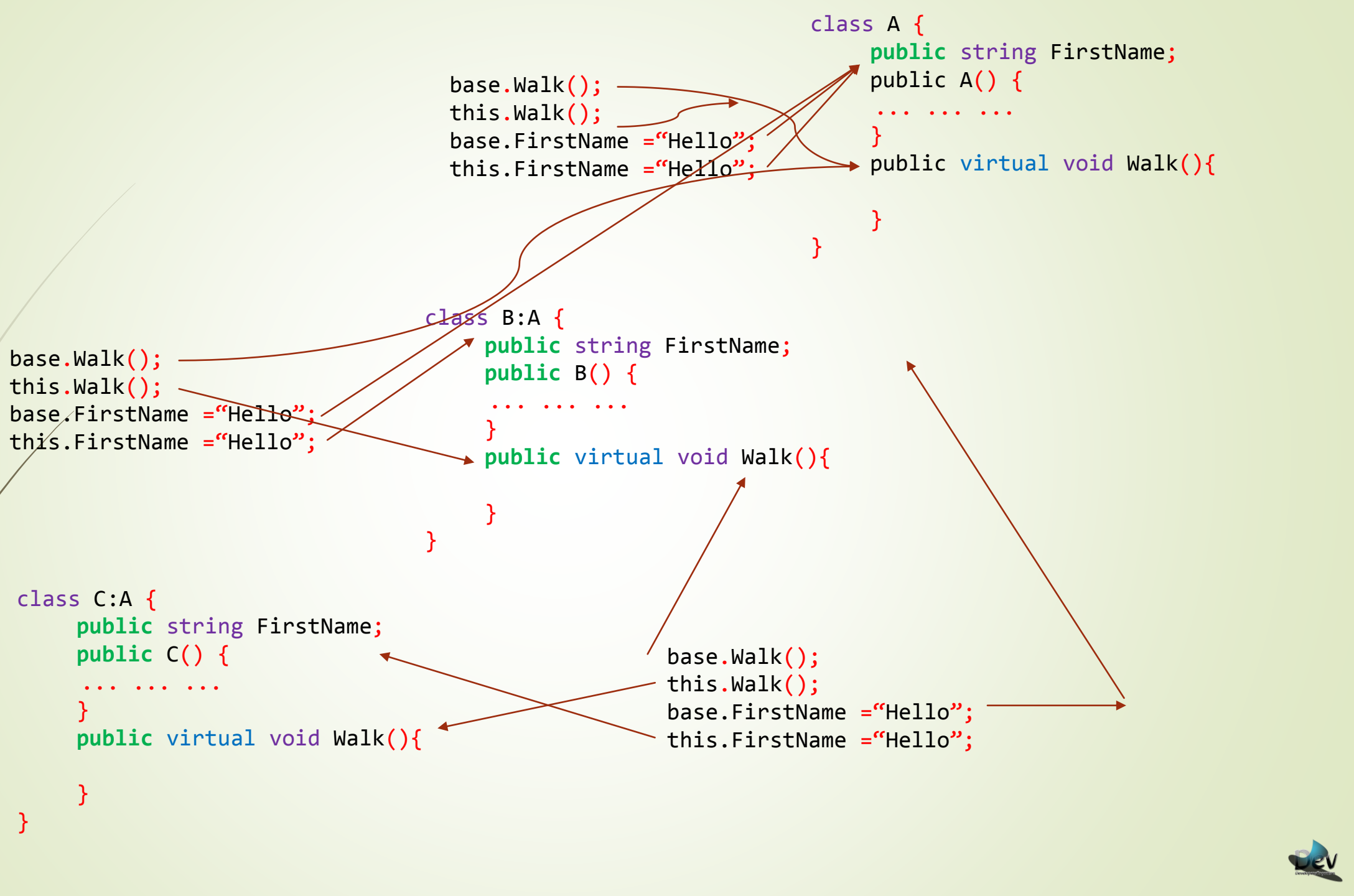
```
Type type ≠ new Different Type();

Teacher teacher2 ≠ new Student();
```

```
SubClass type ≠ new SuperClass();

Student student2 ≠ new Person();

Teacher teacher2 ≠ new Person();
```

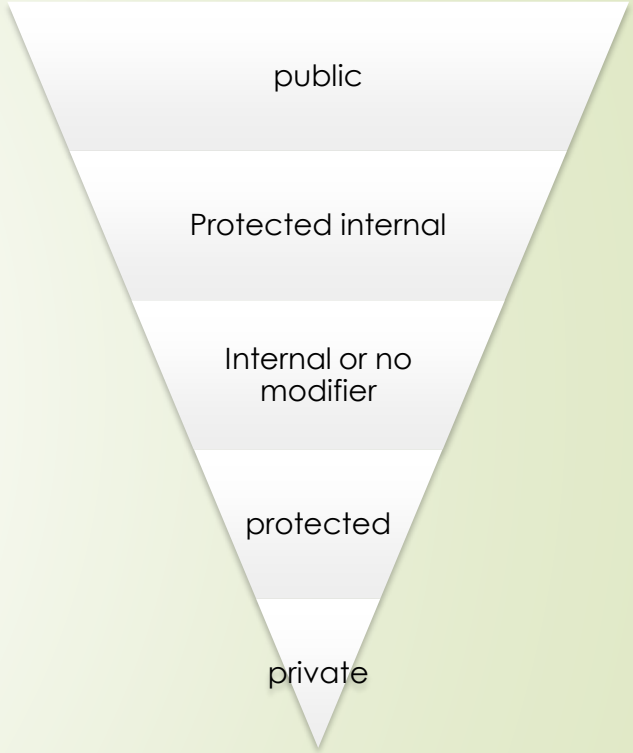


Calling base class constructor


A
B
C


class A {
public string FirstName;
public A() {
... ...
}
public virtual void Walk(){
}
}
base.Walk();
this.Walk();
base.FirstName =“Hello”;
this.FirstName =“Hello”;
class B:A {
public string FirstName;
public B() {
... ...
}
public virtual void Walk(){
}
}
base.Walk();
this.Walk();
base.FirstName =“Hello”;
this.FirstName =“Hello”;
class C:A {
public string FirstName;
public C() {
... ...
}
public virtual void Walk(){
}
}
base.Walk();
this.Walk();
base.FirstName =“Hello”;
this.FirstName =“Hello”;

Access Modifiers (MSDN)	
public	No restriction
protected internal	Can be accessed from same assembly , however other assemblies can only be accessed within class if that class is derived from related class.
internal or no access modifier for class	Can be accessed from same assembly only.
protected	The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.
private	The type or member can be accessed only by code in the same class or struct.



Lets see some coding!

Extension Methods

Lets see some coding!

Overdoing Inheritance is dangerous

No OOP

Not in good shape

Right OOP

Good shape

Overdoing OOP

Very bad shape

Drawn by Alim Ul Karim

Modular Approach

Refer to **HAS-A** relationship.

Here
1 Processor is Required
1 Ram is required
Power supply required
However you can add
RAMS as much as it supports
GPU
TvCard etc....

Inheritance

Sometimes you want to break things into more than one part.

Bike

Inheritance

Sometimes you want to break things into more than one part.

Bike

Bike ~~inherits~~
Seat,handle,body,
tire,paddle,chain

Inheritance

Sometimes you want to break things into more than one part.

Bike

```
Bike inherits
Seat,handle,body,
tire,paddle,chain
```

chain

paddle — **Inherits** chain

Tire — **Inherits** Paddle

Handle — **Inherits** Tire

Seat — **Inherits** Handle

Body — **Inherits** Seat

Bike — **Inherits** Body

Look for the **interface** implementation

What is an interface?

An interface is like contracts that you must implement in your code.

Let's see some examples in code!

IDisposable Example

Adding new methods to an existing interface may break your code.

The best solution is make another interface and inherit that.

Encapsulation

Encapsulation

Encapsulation

Encapsulation


```
Class Person {

    private string firstName;

    ...
    ...
    // getter method for First Name attribute
    public string getFirstName() {
        return firstName;
    }

    // setter method for First Name attribute
    public void setFirstName(string name) {
        if(name != null){
            firstName = name;
        }
    }

}
```

getter

setter

Encapsulation


```
//C#
class Person {

    public string FirstName {get; set;}

}
```

```
// getter method for First Name attribute
public string getFirstName() {
     return firstName;
}

// setter method for First Name attribute
public void setFirstName(string name) {
     firstName = name;
}
```

Encapsulation


```
Class Person {
    private string firstName;

    public string FirstName {
        get{
            return firstName;
        }
        set{
            if(value != null){
                firstName = value;
            }
        }
    }
}
```

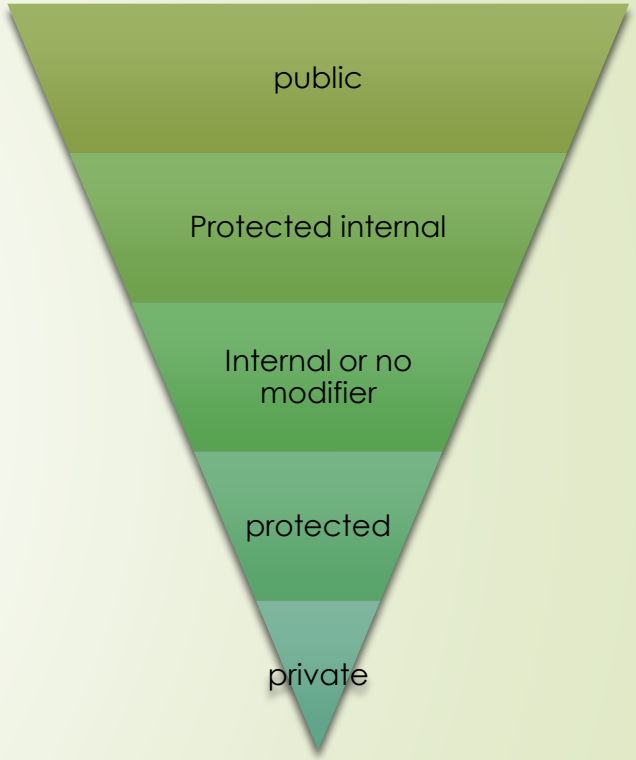
```
// getter method for First Name attribute
public string getFirstName() {
    return firstName;
}

// setter method for First Name attribute
public void setFirstName(string name) {
    if(name != null){
        firstName = name;
    }
}
```

Polymorphism

Simplicity: **same thing** in **different kinds**.

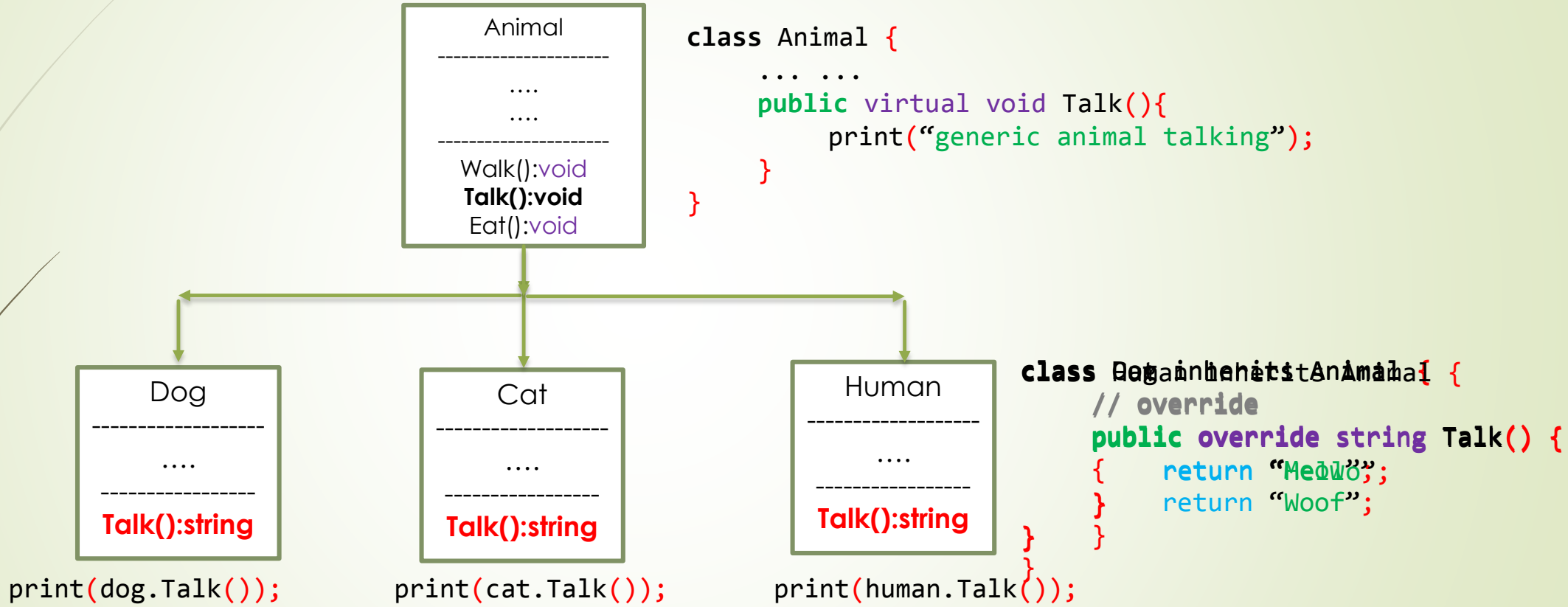
1. Inherit from a class from another or base class
2. Override a method from base class which has a **virtual** keyword on it.
3. Keep in mind access modifiers

Access Modifiers (MSDN)	
public	No restriction
protected internal	Can be accessed from same assembly , however other assemblies can only be accessed within class if that class is derived from related class.
internal or no access modifier for class only	Can be accessed from same assembly only.
protected	The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.
private	The type or member can be accessed only by code in the same class or struct.



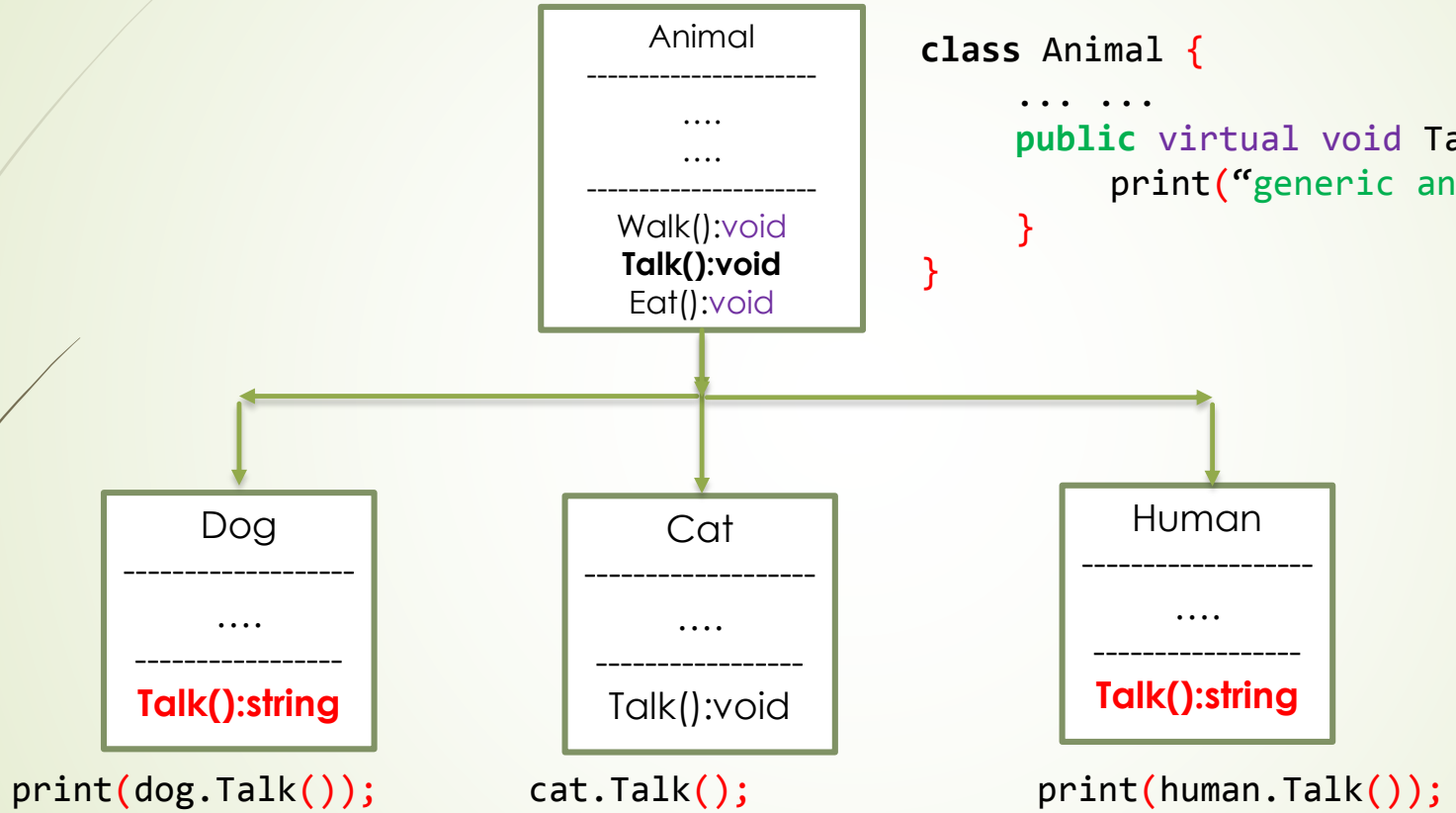
Polymorphism

Simplicity: **same thing** in **different kinds**.



Polymorphism

Simplicity: **same thing** in **different kinds**.




```
class Animal {
     ... ...
     public virtual void Talk(){
          print("generic animal talking");
     }
}
```

```
print(dog.Talk());
```

```
cat.Talk();
// display
// generic animal talking
```

```
print(human.Talk());
```

Overriding ! Lets see some coding!

Solve previous teacher and student class problems and show method overriding with ToString() method.

Method Overloading

Ad-hoc polymorphism.

Calling any base class method

Modifiers (MSDN)	
static	Declares a member that belongs to the type itself instead of to a specific object.
virtual	Declares a method or an accessor whose implementation can be changed by an overriding member in a derived class.
partial	Defines partial classes, structs and methods throughout the same assembly.
sealed	Specifies that a class cannot be inherited.
override	Provides a new implementation of a virtual member inherited from a base class.

http://msdn.microsoft.com/en-us/library/6tcf2h8w.aspx

```
class Animal {
     ... ...
      public void Talk(){
          print(“generic animal talking”);
     }
}
```

Enforce Polymorphism

Abstract class or interface.

```
abstract class Animal {
     ... ...
     public abstract void Talk(); // there must be no implementation
}
```

Lets see some coding!

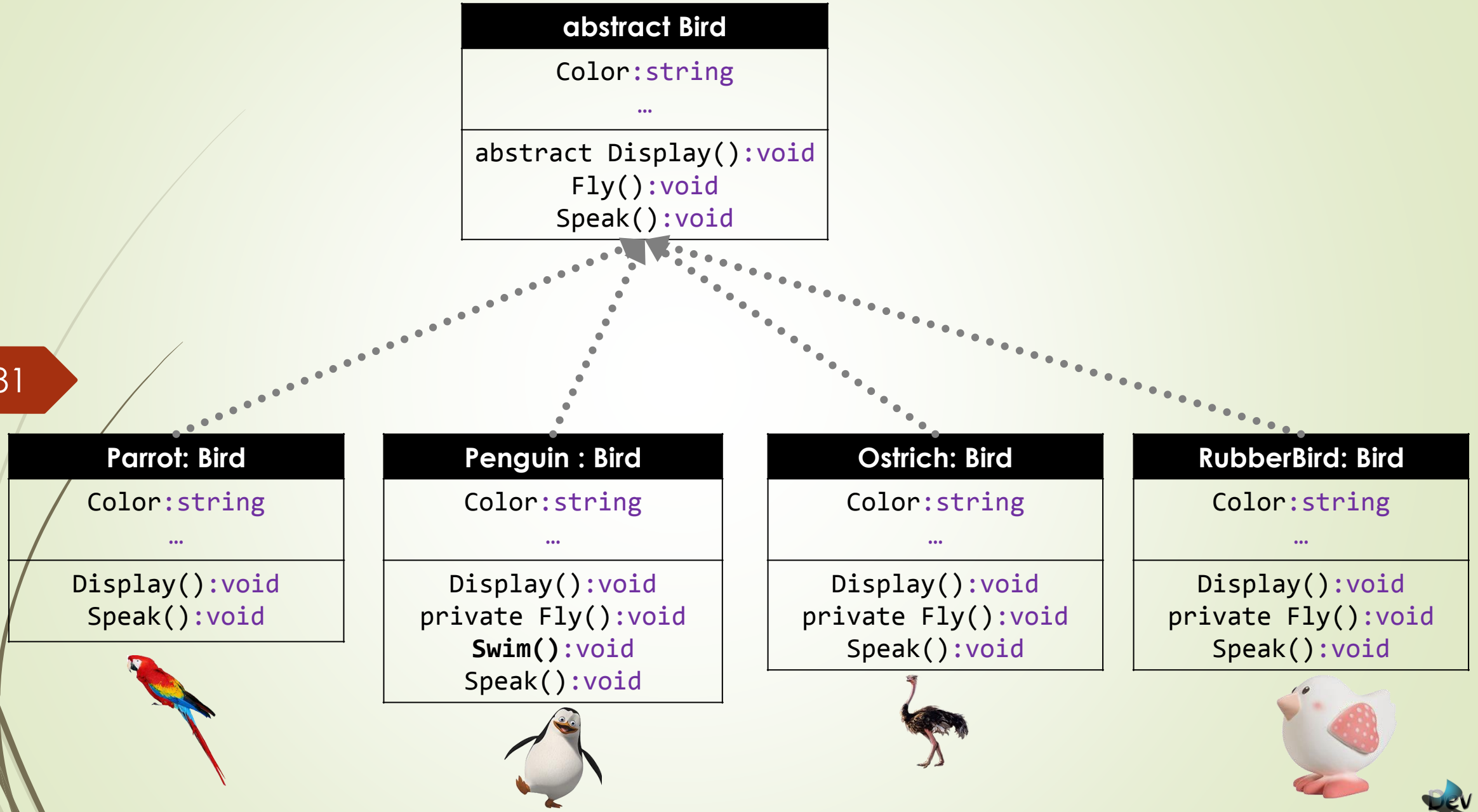
Abstract Class Vs. Interface

Abstract Class	Interface
• Share common implemented code. • (i.e. shares already implemented functions)	• Only share contracts without implementation, in addition with public properties or fields.
• Single Inheritance Support Only.	• A class can inherit multiple interfaces.
• Members can have different access modifiers.	• Properties or members are all public.
• May contain constructors, destructors, properties, functions or function contracts.	• May only contain properties and function contracts.

Strategy Pattern

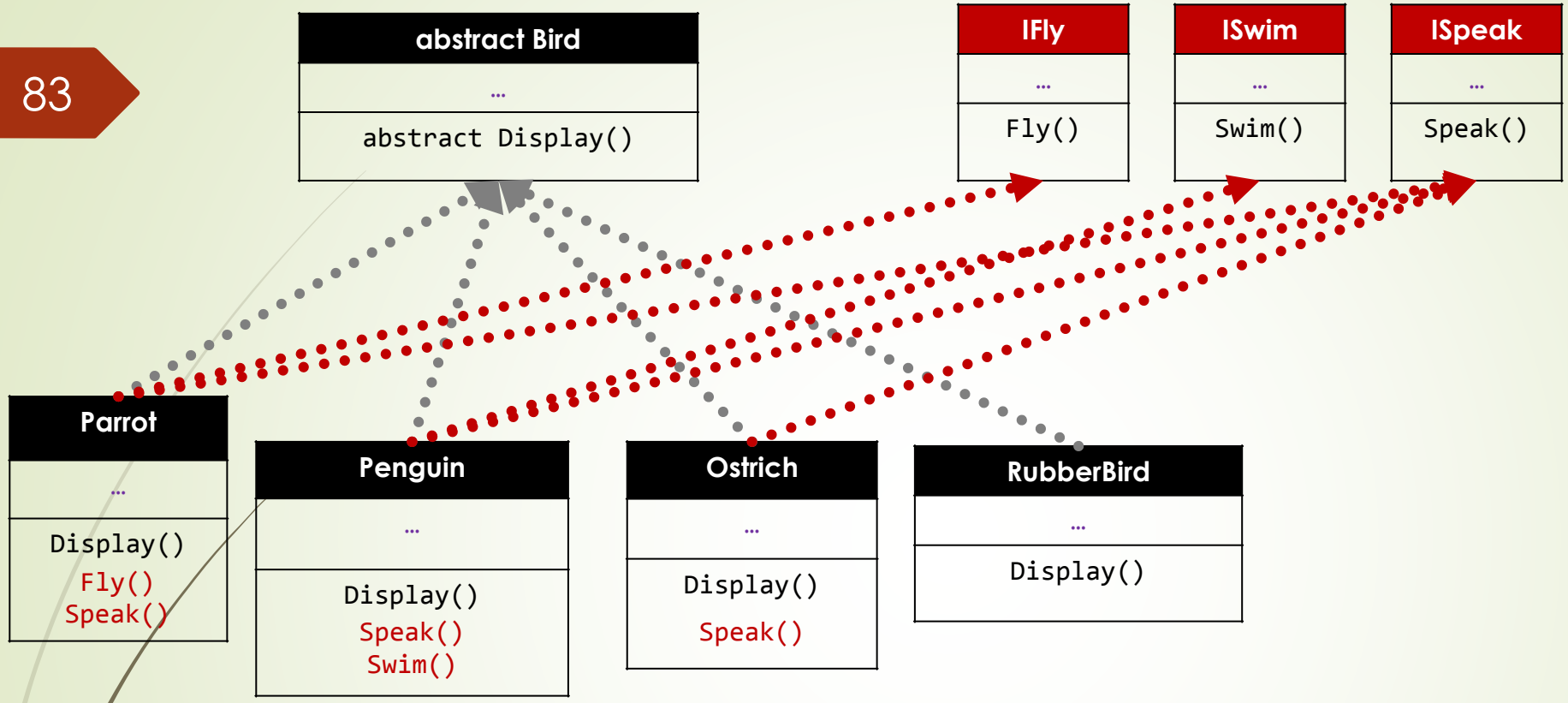
Why we need this?

- Over the years I have seen lot of hard core developers to overuse or overdo inheritance.
- Means they tend to think everything should have a IS-A relationship and derived from superclass.
- In result it becomes quit inflexible and hard to modify.
- So lets see an example of Bird simulator to address some problems.


abstract Bird
Color:string
…
abstract Display():void
Fly():void
Speak():void
Parrot: Bird
Color:string
…
Display():void
Speak():void
Penguin : Bird
Color:string
…
Display():void
private Fly():void
Swim():void
Speak():void
Ostrich: Bird
Color:string
…
Display():void
private Fly():void
Speak():void
RubberBird: Bird
Color:string
…
Display():void
private Fly():void
Speak():void

Observation

- It was a good design though.
- However, there are lot of code duplication on FLY, SPEAK behaviors.
- We don't want that we want our code to be **DRY(as much as possible)**.
- And think if you have 100's methods like this then you have to rewrite those 100's methods.
- So if you remember from previous slides we could use Interface to achieve HAS-A relationship.
- Let's do it then.

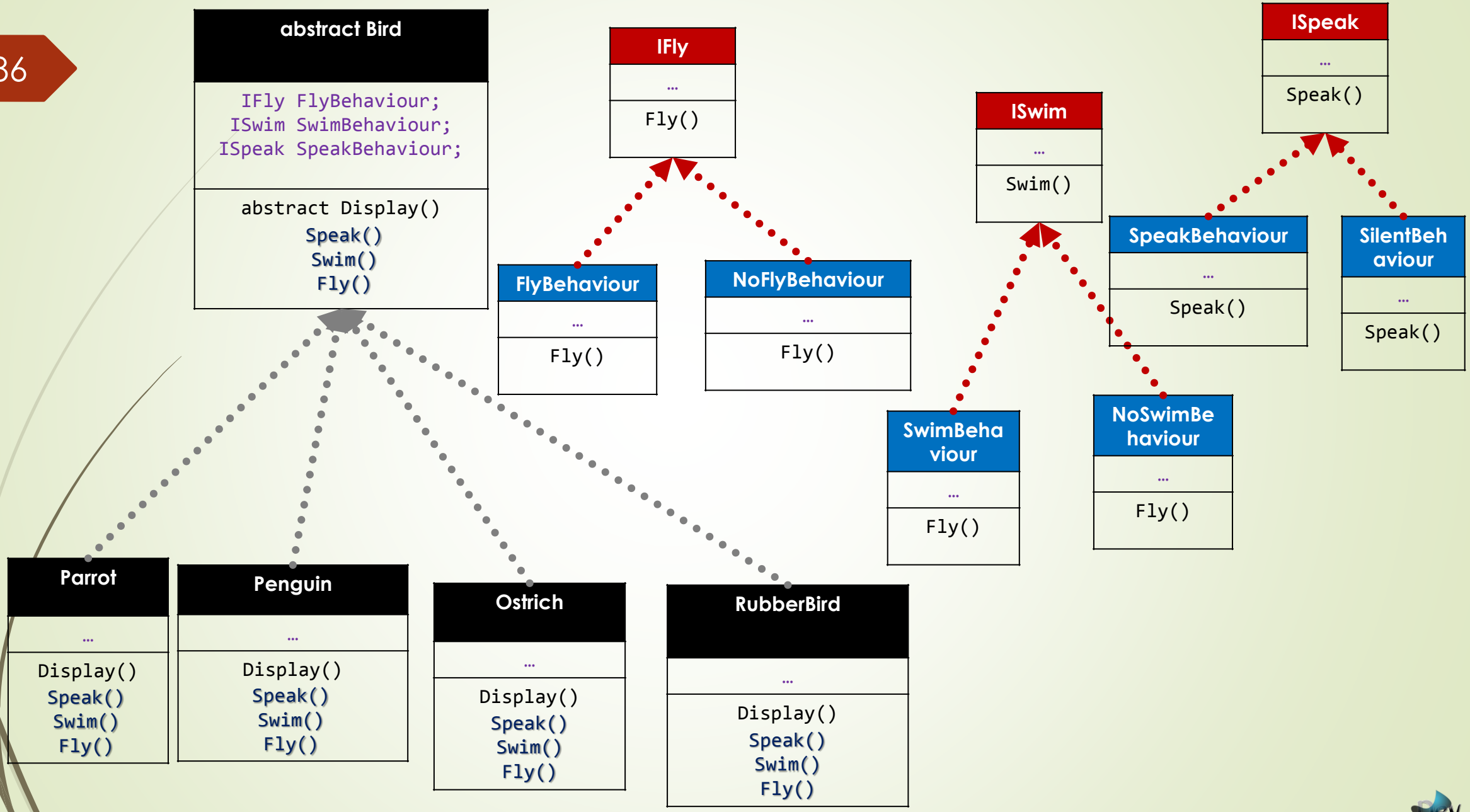

abstract Bird
...
abstract Display()
IFly
...
Fly()
ISwim
...
Swim()
ISpeak
...
Speak()
Parrot
...
Display()
Fly()
Speak()
Penguin
...
Display()
Speak()
Swim()
Ostrich
...
Display()
Speak()
RubberBird
...
Display()

Observation

- It is also a good design.
- However, if we have 20 classes of birds we have to write each FLY or SPEAK or SWIM methods which would be inflexible.
- Again we are stuck with the same problem where if 100's methods in Bird class with interfaces we have to rewrite those 100's methods.
- So we can't reuse our code that's big problem.
- So lets do something about it.

Abstract Class Vs. Interface

Abstract Class	Interface
• Share common implemented code. • (i.e. shares already implemented functions)	• Only share contracts without implementation, in addition with public properties or fields.
• Single Inheritance Support Only.	• A class can inherit multiple interfaces.
• Members can have different access modifiers.	• Properties or members are all public.
• May contain constructors, destructors, properties, functions or function contracts.	• May only contain properties and function contracts.


abstract Bird
IFly FlyBehaviour;
ISwim SwimBehaviour;
ISpeak SpeakBehaviour;
abstract Display()
Speak()
Swim()
Fly()
IFly
...
Fly()
ISwim
...
Swim()
ISpeak
...
Speak()
FlyBehaviour
...
Fly()
NoFlyBehaviour
...
Fly()
SwimBehaviour
...
Fly()
SpeakBehaviour
...
Speak()
SilentBehaviour
...
Speak()
NoSwimBehaviour
...
Fly()
Parrot
...
Display()
Speak()
Swim()
Fly()
Penguin
...
Display()
Speak()
Swim()
Fly()
Ostrich
...
Display()
Speak()
Swim()
Fly()
RubberBird
...
Display()
Speak()
Swim()
Fly()

Observation

- That's much better we can reuse our codes, whenever we need our behavior we can attach it.
- It will extend the design at point.
- However, there is no sliver bullet for every problem scenarios. Strategy pattern is good for some or many problem scenarios but it's not a sliver bullet which solves all the problems.

Before we finalize, I would like to summarize

with Clean Code!

Clean Code

- DRY : Don't Repeat Yourself
  - Encapsulate what varies
- Modular Approach
- Naming Convention

Naming Convention: Rule of thumb

Upper camel case or Pascal casing

1. Don't *use all upper case*
2. Don't any underscore if not explicitly set.
3. Don't use Hungarian notation or any other type identification in identifiers

```
string strExample;
int iCounter;
```

4. Avoid using abbreviations

Naming Convention: Namespace

Upper camel case or Pascal casing

1. *Organize* namespaces with a clearly defined structure
2. Don't use underscore or like Com.website….

Naming Convention: Class

Upper camel case or Pascal casing

1. Better if use noun or phrases.
2. Declare fields or properties at the top and methods at bottom.

Naming Convention: private variables

Prefix **underscore + *lower camel case***

- private string _firstName; ✓
- private string _first_Name; X
- string _firstName; ✓

Naming Convention: class public variables

Prefix ***lower camel case***

- public string firstName; ✓
- public string _first_Name; X

Naming Convention: Fields

Use ***upper camel case or pascal case***

- public string FirstName {get; set;} ✓
- public string firstName {get; set;} X
- public string _firstName {get; set;} X

Naming Convention: Constants or ReadOnly

Use upper ***camel case or Pascal Case***

- public static const string FirstName = “Alim”; ✓
- public static const string First_Name = “Alim”; X
- public static const string FIRST_NAME = “Alim”; X

Naming Convention: method param

Prefix ***lower camel case***

- void Method(string paramWord) ✓

Naming Convention: Method or function

Upper camel case or Pascal casing

1. One method should do one task. Not more than one.
    a. SaveAndSend() X
    b. Save()
    b. Send()

Naming Convention: Interface

Upper camel case or Pascal casing

1. Use 'I' as prefix
2. Use noun or adjective
    a. **ISendable**
    b. **ICollection**
    c. **IIteration**

Write Comments

But not too many, remember less is more.

Lines of code

Try to keep each class consistent and more than 400 lines of code.

Conclusion

Happy coding ☺

Questions ?

Thank you for watching

Instructor

Alim Ul Karim

alim@developers-organism.com

Like + Share + Subscribe

Link it with your blog

Dislike + Comments

Find us

/DevelopersOrganism

developers-organism.com

info@developers-organism.com