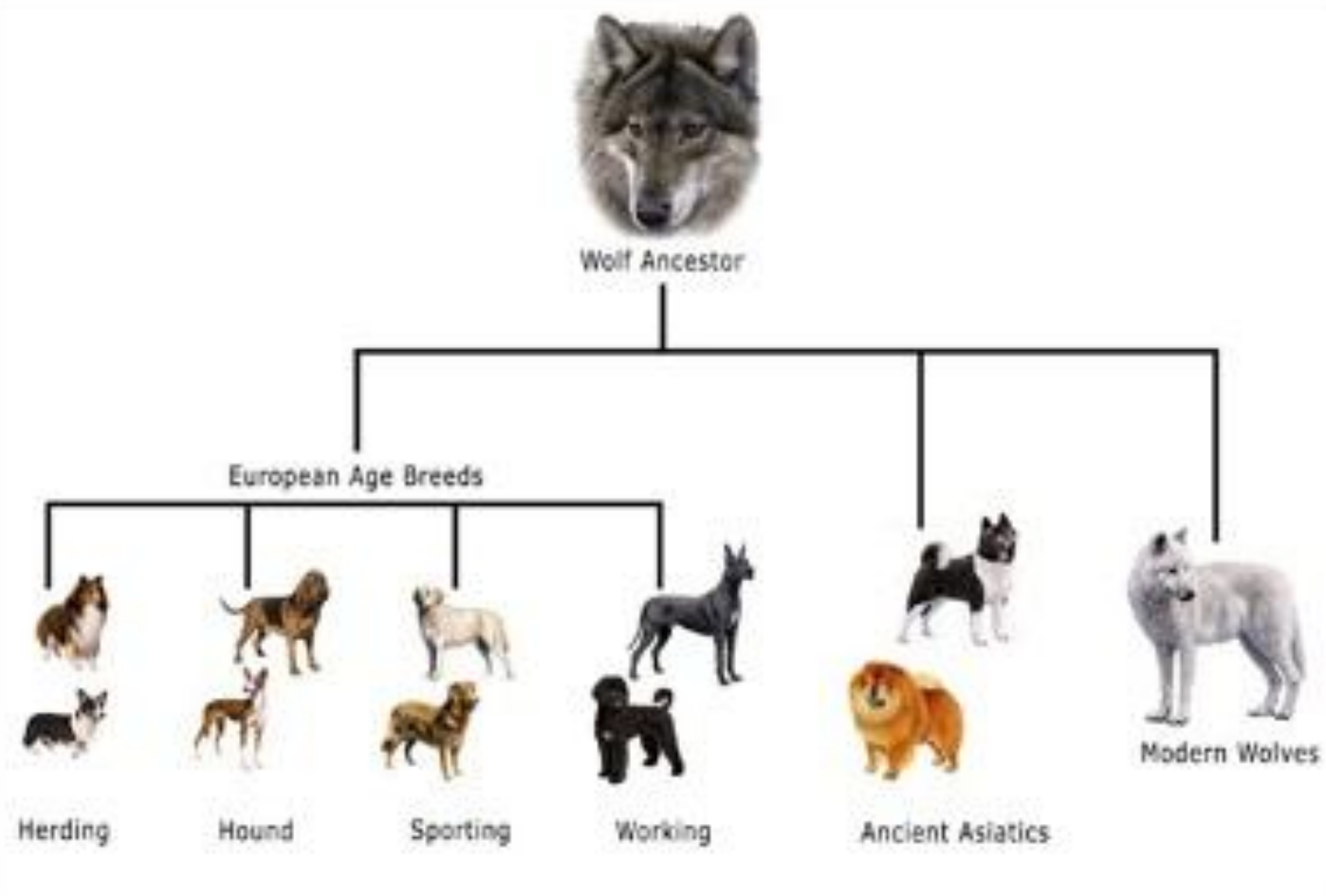
# OOP in C#

Școala
informală
de IT

# Agenda

- **Inheritance**
  - Inherit members from parent class
- **Abstraction**
  - Define and execute abstract actions
- **Encapsulation**
  - Hide the internals of a class
- **Polymorphism**
  - Access a class through its parent interface

# Inheritance



Școala informală de IT

# Inheritance

- Inheritance allows to create a derived (child) class based on a base (parent) class reusing its characteristics
  - Attributes (fields and properties) => state of the class
  - Operations (methods) => behavior of the class
- Derived (child) class can extend the base (parent) class
  - Adding new fields and methods
  - Redefining methods (modify existing behavior)

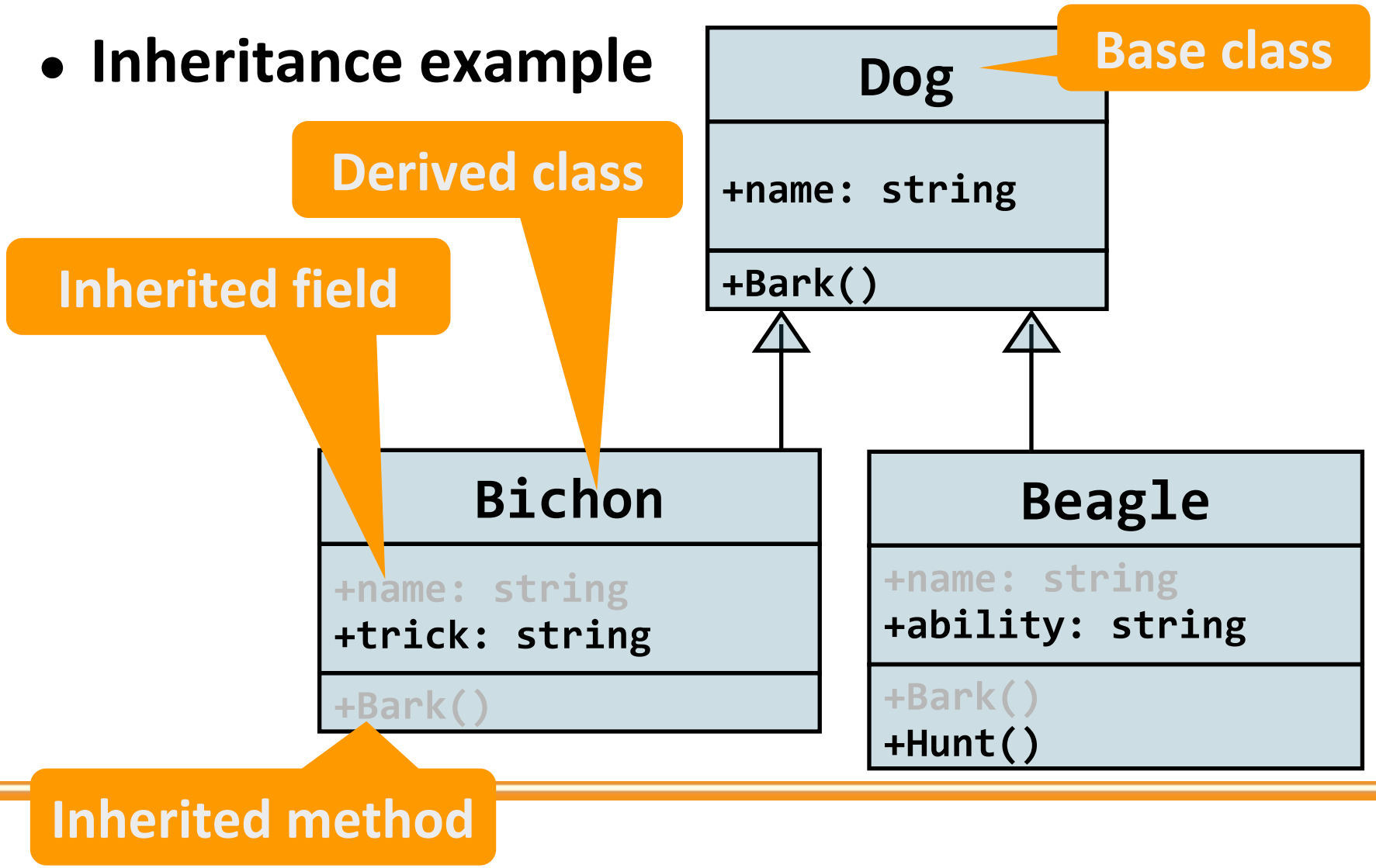
Școala informală de IT

# Inheritance

- **Inheritance advantages**
  - Extensibility (extends base class methods logic)
  - Reusability (uses the base class public members)
  - Provides abstraction (make data private), polymorphism
- **Inheritance disadvantages**
  - Tightly coupling (parent class change will affect all children)
  - A lot of data can be unused in the hierarchy
- **Use inheritance for building is-a relationships**
  - E.g. Dog is-a Animal (dogs are kind of animals)
- **To build has-a relationship use composition**
  - E.g. Dog has-a Address (dogs are not a kind of addresses)

# Inheritance

- Inheritance example



Școala informală de IT

# Inheritance

- Syntax. We must specify the name of the base class after the name of the derived

```
public class Dog
{...}
public class Beagle : Dog
{...}
```

- Calling base constructor. In the constructor of the derived class we use the keyword `base` to pass data to the constructor of the base class

```
public Beagle(string name, string ability) : base(name)
{...}
```

Școala informală de IT

# Inheritance

- **Simple inheritance example**

```
public class Dog
{
  public string Name { get; set; }

  public Dog(string name)
  {
    this.Name = name;
  }

  public void Bark()
  {
    Console.WriteLine("Dog barks...");
  }
}
```

*(continued)*

Școala informală de IT

# Inheritance

- **Simple inheritance example**

```
public class Beagle : Dog
{
  public string Breed { get; set; }

  public Beagle(string name, string breed) : base(name)
  {
    this.Breed = breed;
  }

  public void Hunt()
  {
    Console.WriteLine("Beagle is a good hunter.");
  }
}
```

Școala informală de IT

# Inheritance

- Constructors in details
  - Constructors of the child class are calling constructors of the parent class

```
public class Dog
{
}

public class Beagle : Dog
{
}
```

```
public class Dog
{
  public Dog()
  {}
}

public class Beagle : Dog
{
  public Beagle() : base()
  {}
}
```

Școala
informală
de IT

# Interfaces

Școala
informală
de IT

# Interfaces

- Interfaces define a set of operations (behavior): methods, properties and events
- Interfaces don't contain state (fields, consts)
- Can be used to define abstract data types
- Can not be instantiated
- Members do not have access modifier (by default is public)
- Methods do not have body (content)
- A class or struct can implement an interface by providing implementation for all its methods

# Interfaces Samples

```
interface IPet
{
  string Trick { get; set; }     // automatic property
  void Play();                   // method
}

interface IHunt
{
  void Hunt(string pray);
}

class Animal {
   public int Age { set; get; }
}
```

Școala informală de IT

# Interfaces Implementation

```
class Dog : Animal, IPet, IHunt {
   string Trick { get; set; }

   public Dog()
   {
      base.Trick = "fetching...";
   }

   public void Play()                                    // IPet method
   {
      Console.WriteLine("playing " + Trick);
   }

   public void Hunt(string pray)                    // IHunt method
   {
      Console.WriteLine("hunting " + pray);
   }
}
```

# Workshop Interfaces

Școala informală de IT

# Inheritance

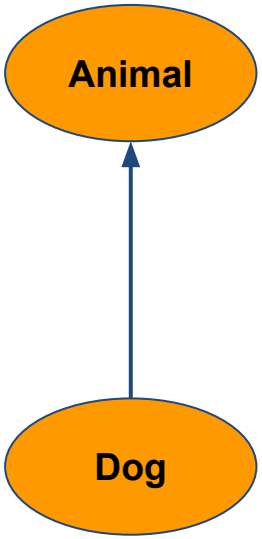
- Structures cannot be inherited
- In C# there is no multiple inheritance, but multiple interfaces can be implemented

```
public class Dog : Animal, IPet, IHunter
```

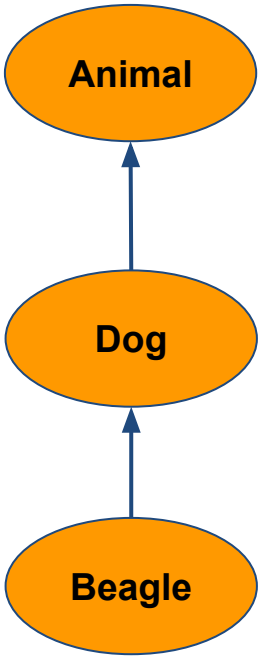
- Static constructors are not inherited
- Inheritance is transitive relation
  - If C is derived from B, and B is derived from A, then C inherits A as well
- Classes marked `sealed` cannot be extended

# Inheritance

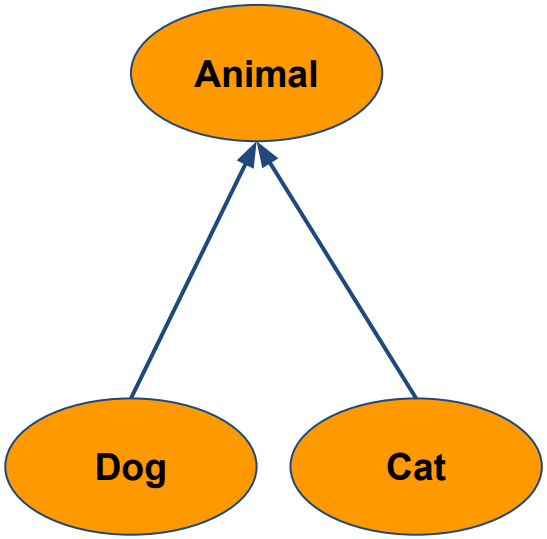
## Simple inheritance




## Multi-level inheritance



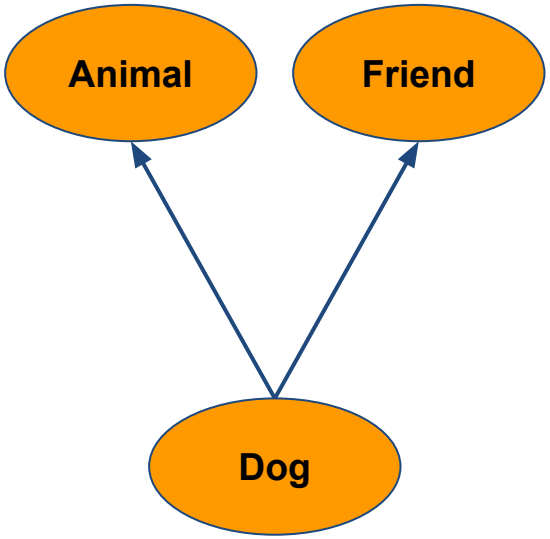

## Hierarchical inheritance



Școala informală de IT

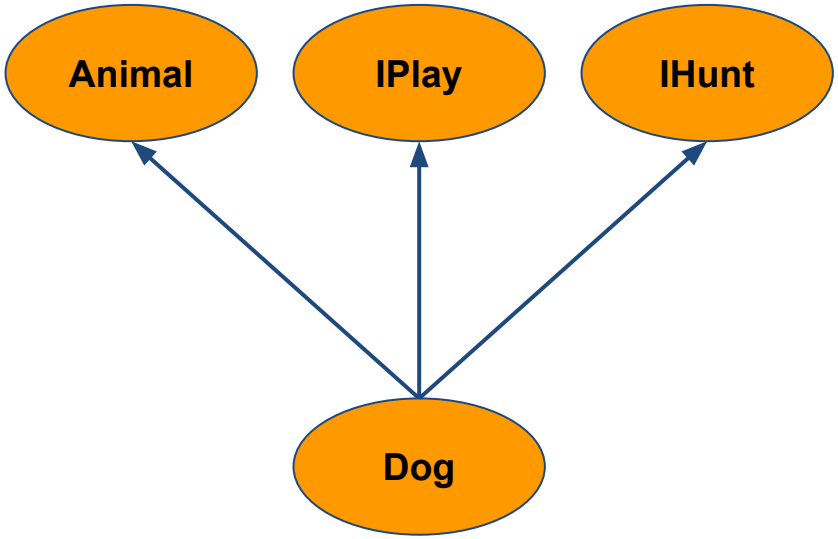
# Multiple Inheritance

## Multiple inheritance*




*Not allowed in C# !!

## Multiple inheritance**




**Only one class and/or multiple interfaces

Școala
informală
de IT

# Inheritance

- A derived class extends its base class
  - It can add new members but cannot remove derived ones
- Declaring new members with the same name or signature hides the inherited ones (warning alert)

Școala informală de IT

# When Inheritance?

- Use inheritance when:
  - Inheritance hierarchy represents an "is-a" relationship and not a "has-a" relationship (which stands for composition).
  - You can reuse code from the base classes.
  - You want to make global changes to derived classes by changing a base class.
  - You need to apply the same class and methods to different data types (abstraction).

Școala informală de IT

# When Interfaces?

- Use interfaces when:
  - Many possibly *unrelated* object types are required to provide certain functionality.
  - You want a more flexible design (you can implement *multiple* interfaces).
  - You do not have to inherit implementation from a base class.
  - You cannot use class inheritance (structures cannot inherit from classes, but they can implement interfaces).

Școala informală de IT

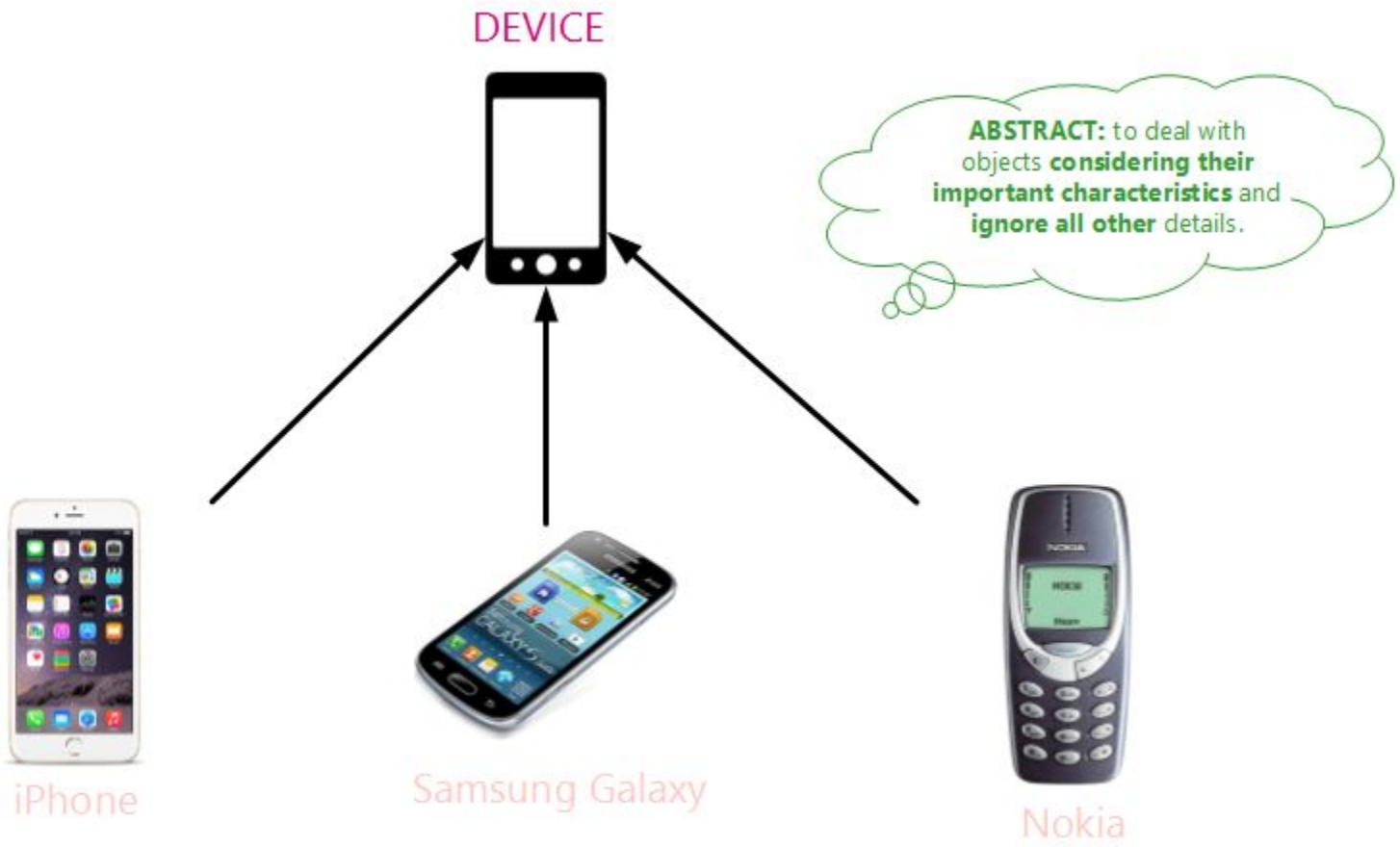
# Inheritance

- **Inheritance terminology**

| | | |
|---|---|---|
| derived class / child class | **inherits** | base class / parent class |
| class | **implements** | interface |
| derived interface | **extends** | base interface |

# Workshop Inheritance

# Abstraction



Școala
informală
de IT

# Abstraction

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones, relevant to the given project (with an eye to future reuse in similar projects)
- Abstraction = managing complexity

Școala
informală
de IT

# Abstraction

- Abstraction is something we do every day
  - Looking at an object, we see those things about it that have meaning to us
  - We abstract the properties of the object, and keep only what we need
  - dog  get "name", "age" but can get "horsePower"
- Allows us to represent a complex reality in terms of a simplified model
- Abstraction highlights the properties of an entity that we need and hides the others

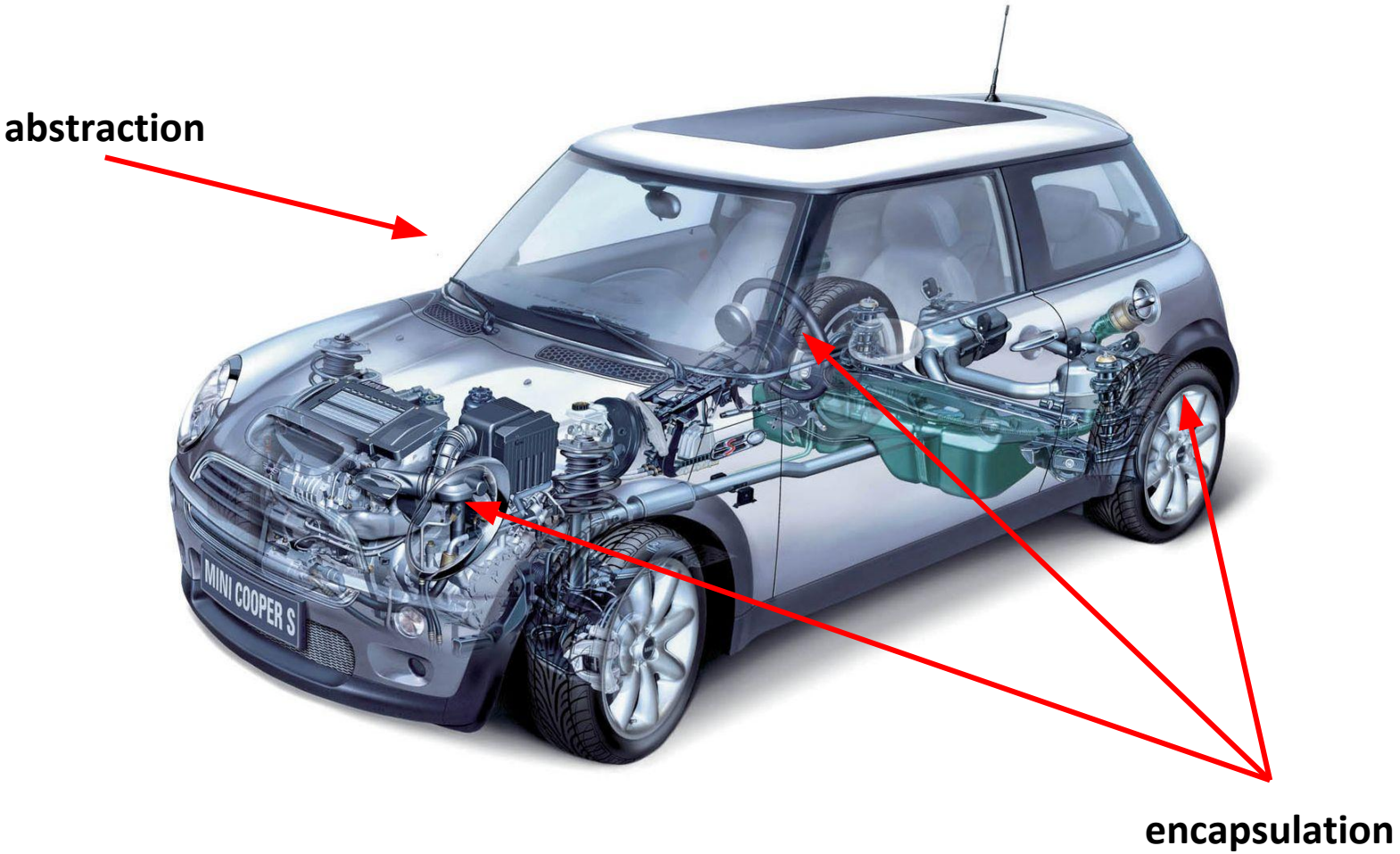
# Abstraction

- **In .NET abstraction is achieved in several ways:**

  - **Interfaces**
  - **Inheritance**
  - **Abstract classes (see polymorphism)**

Școala
informală
de IT

# Encapsulation



Școala
informală
de IT

# Encapsulation

- Encapsulation hides the implementation details, reducing complexity → easier maintenance
- Class announces some operations (methods) and attributes available for class users by its public interface

Școala informală de IT

# Encapsulation

- Access modifiers in C#
  - `public`: access is not restricted
  - `private`: access is restricted to the containing type
  - `protected`: access is limited to the containing type and types derived from it
  - `internal`: access is limited to the current assembly
  - `protected internal`: access is limited to the current assembly or types derived from the containing class

Școala informală de IT

# Encapsulation Best Practices

- Fields should be hidden using private accessor
  - Accessed through properties in read-only or read-write mode
- Constructors are almost always declared public (except when we want to control the number of instances)
- Interface methods are always public
  - Not explicitly declared with public
- Non-interface methods are declared private / protected

Școala informală de IT

# Encapsulation

- **Simple inheritance example**

```
public class Dog
{
   public string Breed { get; private set; }
   public string Name { get; set; }

   private void Profile()
   {
      Console.WriteLine("Dog profile is...");
   }
   protected Trick()
   {
      Console.WriteLine("See some tricks...");
   }
   public Bark()
   {
      Console.WriteLine("Dog is barking");
   }
}
```

Școala informală de IT

# Encapsulation

- **Simple inheritance example**

```
class Beagle : Dog
{
   public void Play()
   {
      base.Trick();        // can be invoked here in child and in base class
      base.Profile();      // cannot be invoked here, it's private!
   }
}
```

```
static void Main()
{
   Beagle puppy = new Beagle();
   puppy.Name = "Spot";
   puppy.Bark();
   puppy.Trick();              // is protected and can not be invoked here
   puppy.Breed = "Beagle";     // readonly property, cannot be set
}
```

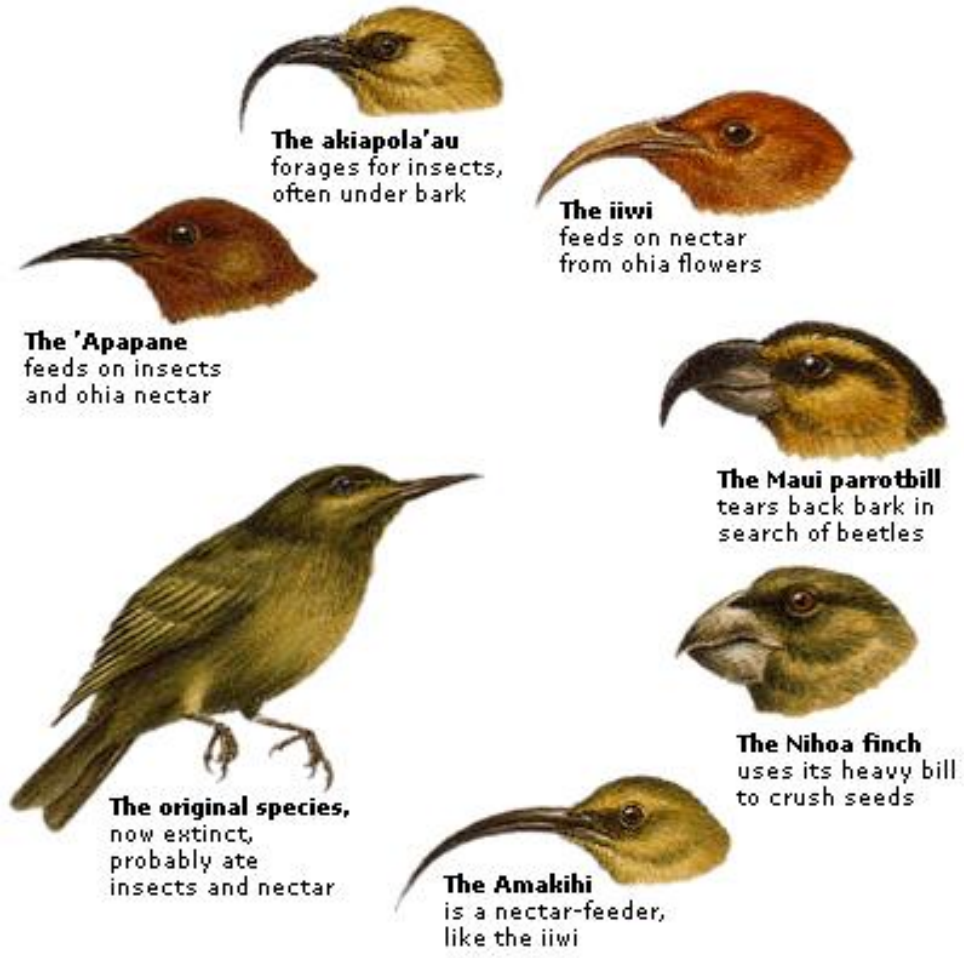
Școala informală de IT

# Encapsulation Benefits

- Ensures that structural changes remain local:
  - Changing the class internals does not affect any code outside of the class
  - Changing methods implementation does not reflect the clients using them
- Encapsulation allows adding some validation logic when accessing sensitive data
  - E.g. validation on modifying a property value

# Workshop Encapsulation

# Polymorphism



Școala informală de IT

# What is Polymorphism

- What is polymorphism?
  - Ability to take more than one shape
  - Classes from same "family" may have different behaviors but share same interface (methods with different signatures or methods with different body)

# Polymorphism

- Types of polymorphism are:
  - method overloading (inside a class signatures are different)
  - method overriding (inherited signature is the same)
    - using abstract
    - using virtual
  - method hiding (inherited signature can be changed)

Școala
informală
de IT

# Static Polymorphism

- This is compile-time polymorphism
- By method overloading we can add more behaviours to the same class method using different signature (= number, type and order of parameters)
- Methods cannot differentiate only by return type (return type is not part of the signature!)

Școala informală de IT

# Dynamic Polymorphism

- This is `runtime` polymorphism
- Using `override` we can modify a method or property, providing a new implementation of a member inherited from a base class
- You cannot override a non-virtual or static method
- The overridden base method must be `virtual` or `abstract` to `override`
- using `new` modifier base method is hidden by current (similar to virtual overriding but signature can be changed)

Școala informală de IT

# Using Virtual Methods

- Virtual method has a default implementation in parent class that can be changed in child class

```
class Animal
{
  public virtual void Play() {...}
}
```

- Virtual methods can be overridden using the keyword override in the derived class

```
class Dog : Animal
{
  public override void Play() {...}
}
```

Școala informală de IT

# Virtual Method Sample

```
class Animal {
   public string name;
   public virtual void Play()
   {
      Console.WriteLine("rolling...");
   }
}

class Dog : Animal {
   private string trick;
   public override void Play()
   {
      WriteLine(name + " playing " + trick);
   }
}
```

Școala informală de IT

# Virtual Method Sample

```
void static Main()
{
    Animal pet;
    pet = new Animal();
    pet.name = "Rex";
    pet.Play();

    Dog puppy;
    puppy = new Dog();
    puppy.name = "Spot";
    puppy.trick = "fetch";
    puppy.Play();
}
```

Școala informală de IT

# Using Abstract Classes

- Abstract classes are hybrid (partially implemented) or fully implemented (at least one method or property is abstract) and cannot be instantiated
- Not implemented methods are declared abstract and are left empty to be implemented by derived classes

```
abstract class Animal
{
    public abstract void Play();
}
```

Școala informală de IT

# Using Abstract Classes

- Abstract methods must be overridden using the keyword override in the derived class

```
class Dog : Animal
{
    public override void Play() {...}
}
```

- Child classes should implement abstract methods or declare them abstract as well (they will be implemented in the child of the current)

Școala informală de IT

# Abstract Method Sample

```
abstract class Animal {
   public string name;
   public abstract void Play();
}

class Dog : Animal {
   public string trick;
   public override void Play()
   {
      WriteLine(name + " playing " + trick);
   }
}

class Cat : Animal {
   public string toy;
   public override void Play()
   {
      WriteLine(name + " playing with " + toy);
   }
}
                                                    // continued
```

Școala informală de IT

# Abstract Method Sample

```
static void Main()
{
    Dog puppy;
    pet = new Animal();      // instance of abstract class not allowed

    puppy = new Dog();
    puppy.name = "Rex";
    puppy.trick = "fetch";
    puppy.Play();            // will print "Rex playing fetch"

    Cat kitten;
    kitten = new Cat();
    kitten.name = "Tom";
    kitten.toy = "laser";
    kitten.Play();           // will print "Tom playing with laser"
}
```

Școala informală de IT

# Why Polymorphism

- Why handle an object of given type as object of its base type?
  - To invoke abstract operations
  - To mix different related types in the same collection
    - E.g. `List<object>` can hold anything
  - To pass more specific object to a method that expects a parameter of a more generic type
  - To declare a more generic field which will be initialized and "specialized" later

Școala informală de IT

# Why Polymorphism

- **Why do we need polymorphism?**
  - **To handle objects without knowing exact behaviour, but using a generic parent class interface (at runtime the code determines which exact type it is and calls the associated code)**
  - **Simplicity (makes the code easier)**
  - **Extensibility (other subclasses could be added later to the family of types, and objects of those new subclasses would also work with the existing code)**

Școala informală de IT

# Overloading vs Overriding

| Method Overloading | Method Overriding |
|---|---|
| Method Overloading lets you have 2 methods with same name and different signature | Method Overriding lets you have 2 methods with same name and same signature |
| Overloading is called as compile time polymorphism or early binding | Overriding is called as run time polymorphism or late binding or dynamic polymorphism |
| Overloading can be achieved:<br>-By changing the number of parameters used.<br>-By changing the order of parameters.<br>-By using different data types for the parameters. | Overriding can be achieved:<br>-Creating the method in a derived class with same name, same parameters and same return type as in base class is called as method overriding |
| Method overloading can be overloaded in same class or in the child class. | Method overriding is only possible in derived class not within the same class where the method is declared |

Școala informală de IT

# Overriding vs Shadowing

| Shadowing | Overriding |
|---|---|
| Shadowing provides a new implementation for the base class method without overriding it. | Overriding allows us to re-write a base class function with a different definition. |
| Using the “new” keyword we can do the shadowing or method hiding. | C# uses the virtual/abstract and override keyword for method overriding. |
| Shadowing redefines an entire method or function. | Overriding redefines only the implementation of a method or function. |
| Showing is used to protect against subsequent base class modification. | Overriding does polymorphism by defining a different implementation. |
| We can change the access modifier. | We cannot change the access modifier. The access modifier must be the same as in the base class method or function. |
| There is no control of a base class on shadowing. In other words, a base class element cannot enforce or stop shadowing. | The base class has some control over the overriding. Using the keyword abstract, the base class forces the child (derived) class to implement the function or method. |
| Shadowing an element (function method or property) can be inherited further in a child (derived) class. The shadowed element is still hidden. | The same as shadowing, overriding an element is inherited further in a derived class and the overridden element is still overridden. |
| In shadowing, the signature of an element could be different. | In overriding, the signature of the element must be the same. |
| In shadowing, the base class cannot access the newly created child (derived) class method. This is because the base class has the same name of the element. | In concept, the base class can be accessed using the child object’s overridden method. |

Școala informală de IT

# Interface vs Abstract class

| | Interface | Abstract class |
|---|---|---|
| **Multiple inheritance** | Yes. A class may inherit **several** interfaces | No. A class may inherit **only one** abstract class |
| **Default implementation** | No. An interface cannot provide any code, just the method signature | Yes. An abstract class can provide complete, hybrid or just the signatures which have to be overridden |
| **Access Modfiers** | No. An interface cannot have access modifiers for the members, everything is assumed as public | Yes. An abstract class can contain access modifiers for the members |
| **Core vs. Peripheral** | Interfaces are used to define the peripheral abilities of a class. In other words both Dog and Cat can inherit from a IPet interface | An abstract class defines the core identity of a class and there it is used for objects of the same type |
| **Homogeneity** | If various implementations only share method signatures then it is better to use interfaces | If various implementations are of the same kind and use common behaviour or status then abstract class is better to use |
| **Speed** | Requires more time to find the actual method in the corresponding classes | Faster |
| **Adding functionality (versioning)** | If we add a new method to an interface then we have to track down all the implementations of the interface and define implementation for the new method | If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly |
| **Fields and Constants** | No. Fields cannot be defined in interfaces | Yes. An abstract class can have fields defined |

# Workshop Polymorphism