C# Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Online Course

```
1  class Lecture7
2  {
3              "Object-Oriented Programming (OOP)"
4  }
5
6  // Keywords:
7  class, new, private, public, get, set, value, const, readonly,
8  this, static, null, object, virtual, override, is, abstract, sealed,
9  interface, delegate, struct, enum, protected, internal,
10 namespace, using
```

Object & Class

- An object keeps its own states in fields (or attributes) and exposes its behaviors through associated methods.
- To create objects, we first collect fields together with accessory methods in a new class, which is the blueprint to create instances, aka runtime objects.
- A class also acts as a derived type.
  - From now, you are creating new types for your own purpose!

Example: Points

- We define the new class as follows:
  - assign a name with the first letter capitalized;
  - declare data and function members in the class body.

```
1 class Point
2 {
3     public double x, y;
4 }
```

```
1  class PointDemo
2  {
3      static void Main(string[] args)
4      {
5          Point p1 = new Point();
6          p1.x = 1;
7          p1.y = 2;
8
9          Point p2 = new Point();
10         p2.x = 3;
11         p2.y = 4;
12
13         Console.WriteLine("p1 = ({0, 2}, {1, 2})", p1.x, p1.y);
14         Console.WriteLine("p2 = ({0, 2}, {1, 2})", p2.x, p2.y);
15     }
16 }
```

- Could you draw the memory allocation when the program halts on Line 13?
- What if we remove public in Line 3?
  - Then these two members are private, by default.

Encapsulation

- Each member has an access modifier[1], say public and private:
  - public: accessible by any class.
  - private: accessible only within its own class.
- We pratically hide its data members.
- Then we expose the public methods which perform actions on these fields, for example,
  - getter: return its field;
  - setter: set a value to its field.
- We proceed to demonstrate how to encapsulate **Point** by get and set, two of contextual keywords[2] of C#.

---

[1] See the accessibility levels here: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels`.

[2] See `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/#contextual-keywords`.

Example: Point (Encapsulated)

```
1  class Point
2  {
3      // Data members
4      private double x, y;
5
6      // Function members
7      public double X
8      {
9          get { return x; }
10         set { x = value; }
11     }
12
13     public double Y
14     {
15         get { return y; }
16         set { y = value; }
17     }
18 }
```

```
1  class Point {
2
3      // Auto-Implemented Property
4      public double X { get; set; }
5      public double Y { get; set; }
6
7  }
```

- The object is read-only (immutable) if the object has no setters.

Immutability: constant[3] & readonly[4]

- You use the constant keyword to declare a constant field or a constant local, which is immutable.
- In a field declaration, readonly indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class.

[3] See https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/const.

[4] See https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly.

Constructors

- A constructor follows the new operator, acting like other methods.
- However, its name should be identical to the name of the class and it has no return type.
- A class may have several constructors if needed.
- Note that constructors belong to the class but not objects.
  - In other words, constructors cannot be invoked by any object.
- If you don't define any explicit constructor, C# assumes a default constructor for you.
  - Moreover, adding any explicit constructor disables the default constructor.

Parameterized Constructors

- You can initialize an object when the object is allocated.
- For example,

```
1  class Point
2  {
3      ...
4      public Point()
5      {
6          // Do something in common.
7      }
8
9      // Parameterized constructor
10     public Point(double a, double b)
11     {
12         X = a;
13         Y = b;
14     }
15     ...
16 }
```

Self Reference

- You can refer to any (instance) member of the current object within methods and constructors by using this.
- The most common reason for using this is because the field is shadowed by the local parameter.
- You can also use this to call another constructor in the same class, say this().

Example: Point (Revisited)

```
1  class Point
2  {
3      ...
4      public Point(double X, double Y)
5      {
6          this.X = X;
7          this.Y = Y;
8      }
9      ...
10 }
```

- However, we cannot use this in the static methods!

Instance Members

- Both data and function members are declared w/o static in this chapter!
- They are called instance members, which are available only after one object is created.
- Semantically, each object has its own fields associated with the accessory methods applying on.
- See the next page for another example.

Example: Distance Between Two Points

```
1  class Point
2  {
3      /* Ignore the previous part. */
4
5      public double GetDistanceFrom(Point that)
6      {
7          return Math.Sqrt(Math.Pow(this.X - that.X, 2)
8                           + Math.Pow(this.Y - that.Y, 2));
9      }
10 }
```

- In OOP design, it is important to clarify the responsibility among objects of various types, aka single responsibility principle.[5]
  - High cohesion, low coupling.
  - The Hollywood principle: don't call us, we'll call you.

[5] Also see https://en.wikipedia.org/wiki/SOLID_(object-oriented_design).

Static Members[6]

- The static members of one class are loaded once used.
  - For example, Main().
- These static members can be invoked directly by class name in absence of any instance.
  - For example, **Math**.PI.
- In particular, static methods perform algorithms.
  - For example, **Math**.Sqrt() and **Array**.Sort().
- Note that static methods cannot access to instance members directly. (Why?)

[6] See `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static`.

Example

```
1  class Point
2  {
3      /* Ignore the previous part. */
4      public static double Measure(Point first, Point second)
5      {
6          // You cannot use this in static context.
7          return Math.Sqrt(Math.Pow(first.X - second.X, 2)
8                           + Math.Pow(first.Y - second.Y, 2));
9      }
10 }
```

```
1  class PointDemo
2  {
3      static void Main(string[] args)
4      {
5          /* Ignore the previous part. */
6          Console.WriteLine(Point.Measure(p1, p2));
7      }
8  }
```

Another Example: Singleton Pattern

- The singleton pattern is one of design patterns.[7]
- For some situations, you need only one object of this type in the system.

```
1  class Singleton
2  {
3      // Do now allow to invoke the constructor by others.
4      Singleton() {}
5
6      // Will be ready as soon as the class is loaded.
7      static Singleton Instance = new Singleton();
8
9      // Only way to obtain this singleton by the outside world.
10     public static Singleton GetInstance()
11     {
12         return Instance;
13     }
14 }
```

[7] Design patterns are a collection of general reusable solution to a commonly occurring problem within a given context in software design.

Object Elimination: Garbage Collection[9]

- The garbage collector (GC) serves as an automatic memory manager.
- When the GC performs a collection, it releases the memory for objects that are no longer being used by the application.
  - Assign null to the reference variable if you don't need that object anymore.
- The further detail about the memory management in CLR can be found in the official website.[8]

[8] See `https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals`.

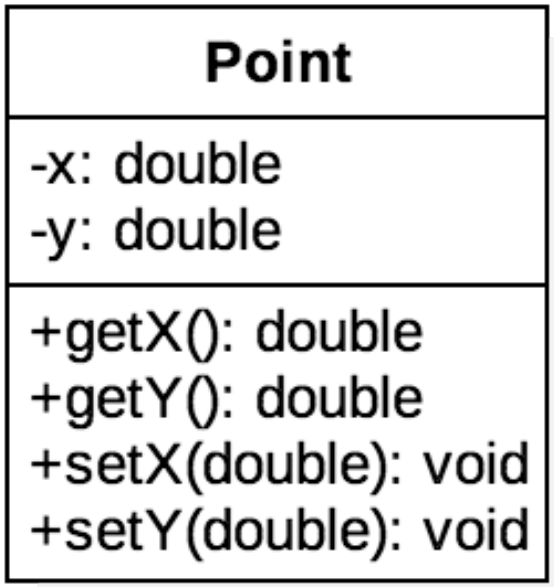
[9] See `https://docs.microsoft.com/en-us/dotnet/api/system.gc`.

Unified Modeling Language[10]

- Unified Modeling Language (UML) is a tool for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
- Free software:
  - `http://staruml.io/` (available for all platforms)

[10] See `http://www.tutorialspoint.com/uml/` and `http://www.mitchellsoftwareengineering.com/IntroToUML.pdf`.

Example: Class Diagram for Point



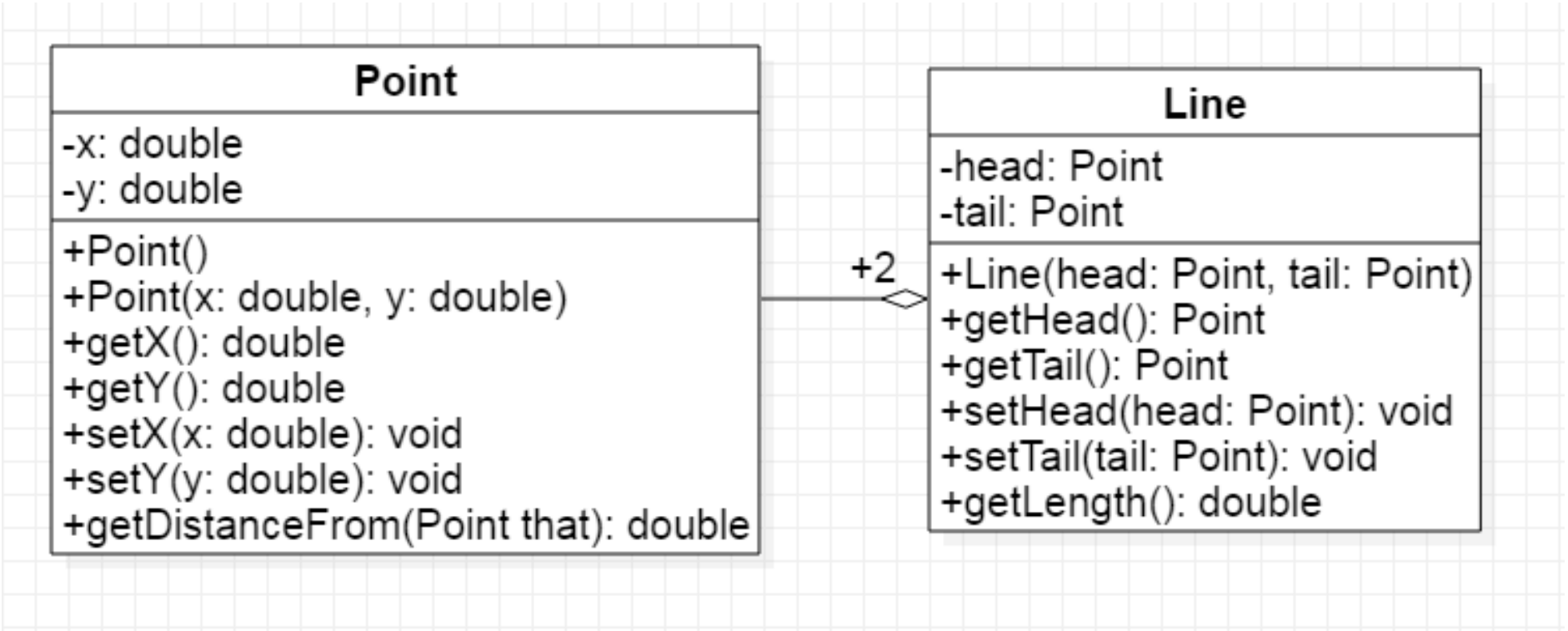
- $+$ refers to public.
- $-$ refers to private.

HAS-A Relationship

- Association is a weak relationship where all objects have their own lifetime and there is no ownership.
  - For example, teacher $\leftrightarrow$ student; doctor $\leftrightarrow$ patient.
- If A uses B, then it is an aggregation, stating that B exists independently from A.
  - For example, knight $\leftrightarrow$ sword; company $\leftrightarrow$ employee.
- If A owns B, then it is a composition, meaning that B has no meaning or purpose in the system without A.[11]
  - For example, house $\leftrightarrow$ room.

[11] We will see this later.

Example: Lines (Aggregation)



- $+2$: two **Point** objects used in one **Line** object.

```
1  class Line
2  {
3      public Point Head { get; set; }
4      public Point Tail { get; set; }
5
6      public Line(Point head, Point tail)
7      {
8          Head = head;
9          Tail = tail;
10     }
11
12     /* Ignore some methods. */
13
14     public double GetLength()
15     {
16         return Head.GetDistanceFrom(Tail);
17     }
18 }
```

Exercise: Circles

```
1  class Circle
2  {
3      public Point Center { get; set; }
4      public double Radius { get; set; }
5
6      public Circle(Point c, double r)
7      {
8          Center = c;
9          Radius = r;
10     }
11
12     public double GetArea()
13     {
14         return Radius * Radius * Math.PI;
15     }
16
17     public boolean IsOverlapped(Circle that)
18     {
19         return this.Radius + that.Radius >
20                this.Center.GetDistanceFrom(that.Center);
21     }
22 }
```

First IS-A Relationship: Class Inheritance

- We can define new classes by inheriting states and behaviors commonly used in predefined classes (aka prototypes).
- A class is a derived type of another, which is called the base type, by using the colon (:) operator.
- For example,

```
1 // Base class.
2 class A
3 {
4     public void DoAction() {} // A can run DoAction().
5 }
6 // Derived class.
7 class B : A {}                 // B can also run DoAction().
```

- Note that C# supports single inheritance only.

Example: Human & Dog

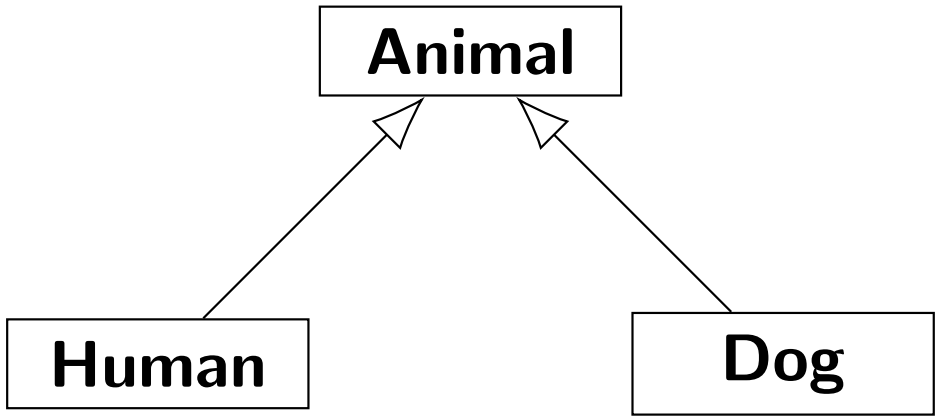
Photo credit: `https://www.sunnyskyz.com/uploads/2016/12/nlf37-dog.jpg`

Before Using Inheritance

```
1 class Human
2 {
3     public void Eat() {}
4     public void Exercise() {}
5     public void WriteCode() {}
6 }
```

```
1 class Dog
2 {
3     public void Eat() {}
4     public void Exercise() {}
5     public void Wag() {}
6 }
```

After Using Inheritance



- Move the common part between **Human** and **Dog** to another class, say **Animal**, as the base class.

```
1 class Animal
2 {
3     public void Eat() {}
4     public void Exercise() {}
5 }
```

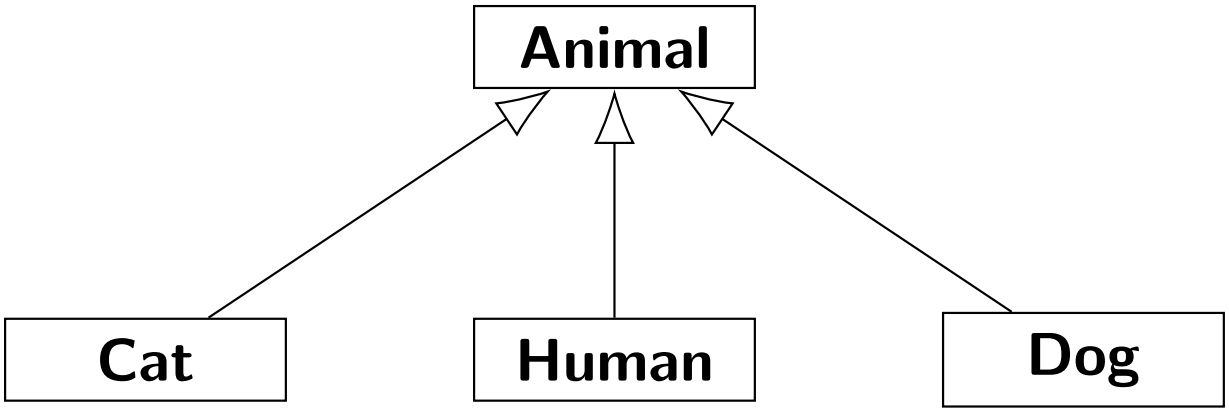
```
1 class Human : Animal
2 {
3     public void WriteCode() {}
4 }
```

```
1 class Dog : Animal
2 {
3     public void Wag() {}
4 }
```

Exercise: Add **Cat** to Animal Hierarchy[12]

https://cdn2.ettoday.net/images/2590/2590715.jpg

[12] See https://petsmao.nownews.com/20170124-10587.



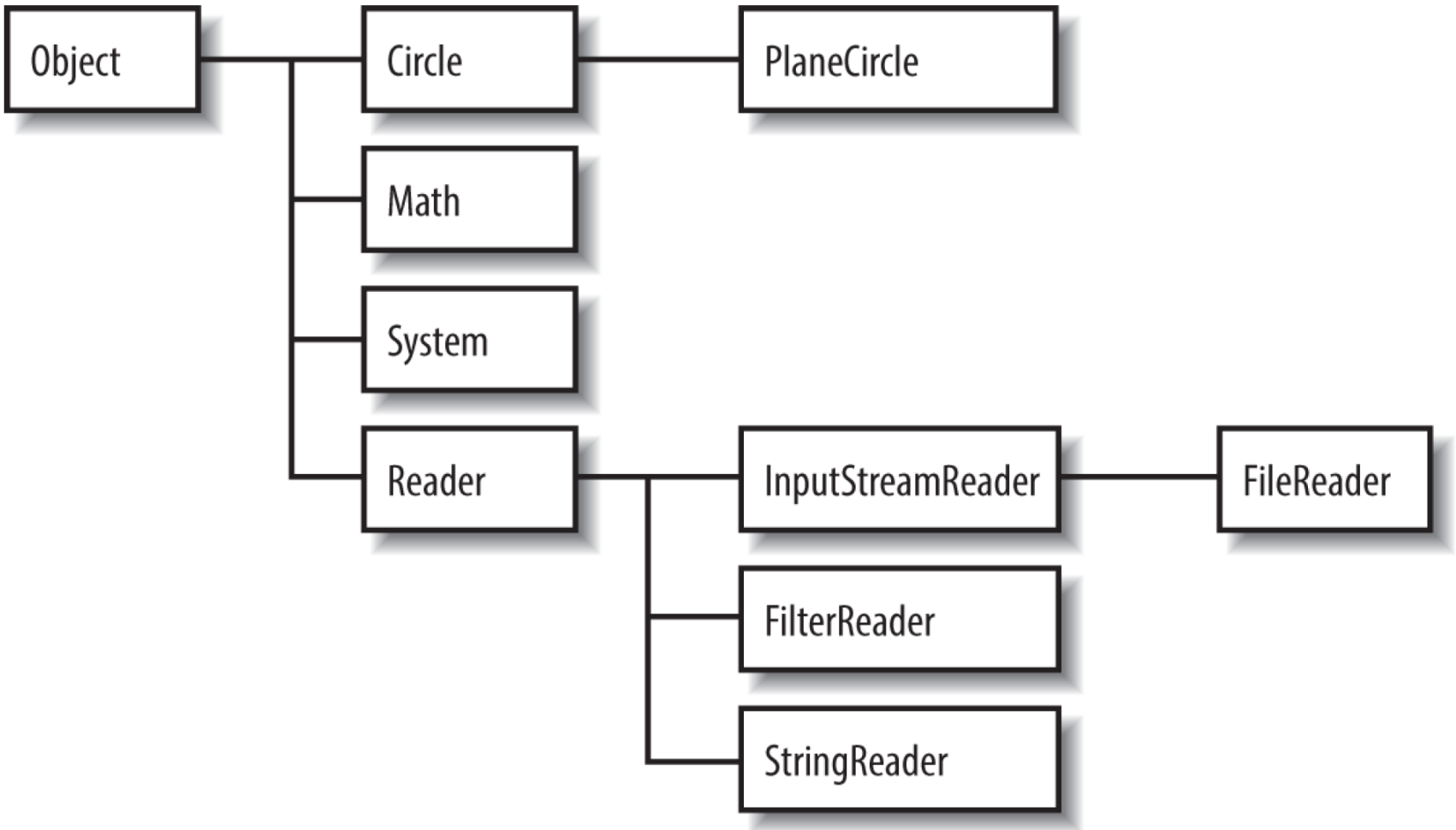

```
1 class Cat : Animal
2 {
3     public void Stepping() {}
4 }
```

- You could add more kinds of animals by extending **Animal**!
- Again, code reuse.

Constructor Chaining

- Once one constructor of the derived class is invoked, the constructor of its base class will be invoked.
- So you might think that there will be a chain of constructors invoked, all the way back to the constructor of the topmost class in C#, called **Object**.
  - The object keyword is the alias of **Object**.
- In this sense, we could say that every class is an immediate or a distant derived class of **Object**.

Illustration for Class Hierarchy



The base Operator

- Recall that this is used to refer to the object itself.
- You can use base to refer to (non-private) members of the base class.
- Note that base() can be used to invoke the constructor of its base class, just similar to this().

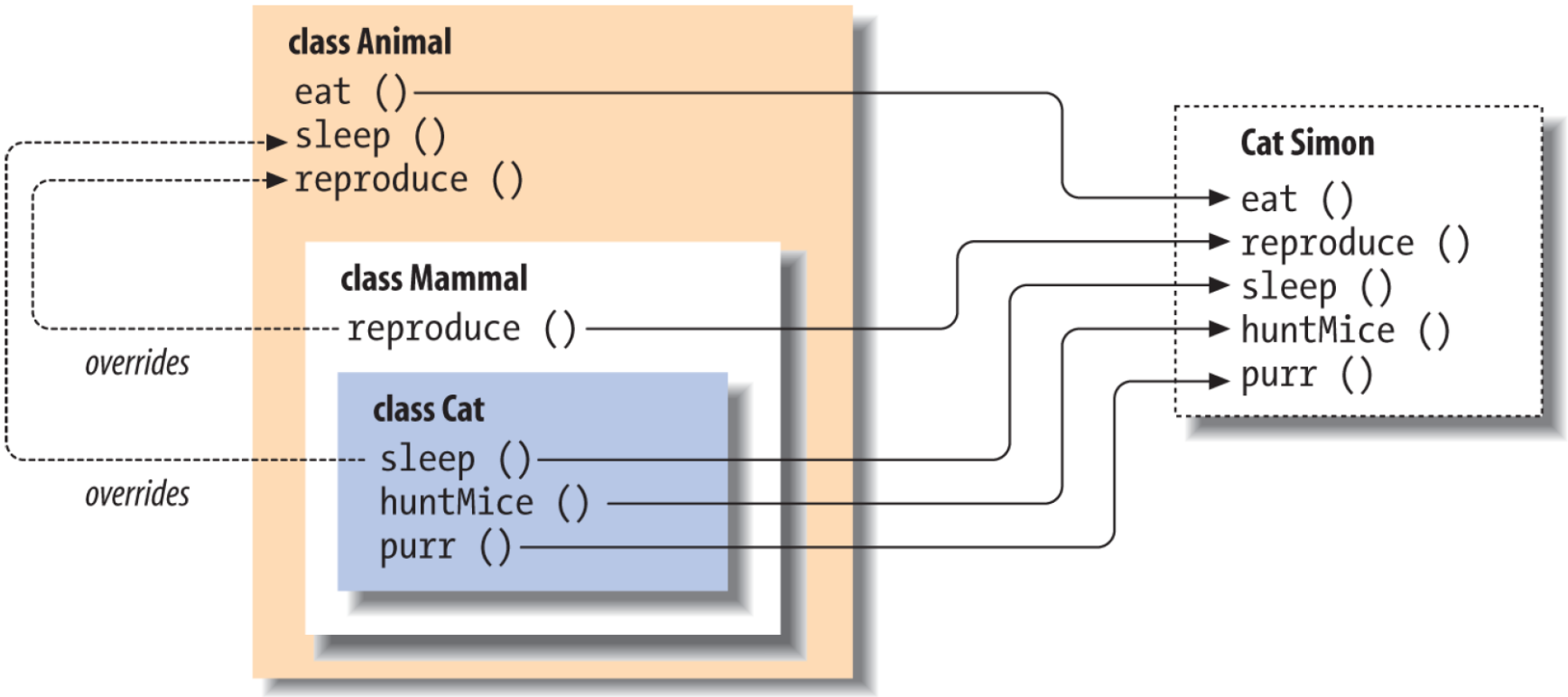
Method Overriding[13]

- A derived class is used to, if necessary, re-implement the virtual methods inherited from its base class.
- Note that you cannot override a non-virtual or static method.

```
1  class A
2  {
3      public virtual void DoAction() { /* Impl. */ }
4  }
5
6  class B : A
7  {
8      public override void DoAction()
9      {
10         /* New impl. w/o changing API. */
11     }
12 }
```

[13] See https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual and https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override.

Example


class Animal
eat ()
sleep ()
reproduce ()
class Mammal
reproduce ()
class Cat
sleep ()
huntMice ()
purr ()
overrides
overrides
Cat Simon
eat ()
reproduce ()
sleep ()
huntMice ()
purr ()

Example: Overriding ToString()

- The method ToString() defined in **Object** is deliberately designed to be invoked by **Console**.WriteLine()!
- By default, it returns the type name but useless for us.
- It could be overridden so that it returns an informative string.

```
1 class Point
2 {
3 ...
4     public override string ToString()
5     {
6         return String.Format("({0}, {1})", x, y);
7     }
8 ...
9 }
```

Subtype Polymorphism[15]

- The word polymorphism literally means "many forms."
- One of OOP design rules is to separate the interface from implementations and program to abstraction, not to implementation.[14]
- Subtype polymorphism fulfills this rule.
- How to make a "single" interface for different implementations?
  - Use the base class of those types as the placeholder.

[14] GoF (1995). The original statement is "program to interface, not to implementation."

[15] See `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism`.

Example: Dependency Reduction (Decoupling)

```
1  class HighSchoolStudent
2  {
3      public void DoHomework() {}
4  }
5
6  class CollegeStudent
7  {
8      public void WriteFinalReports() {}
9  }
```

- Now let these two students go study.

```
1  class PolymorphismDemo
2  {
3      static void Main(string[] args)
4      {
5          HighSchoolStudent Emma = new HighSchoolStudent();
6          GoStudy(Emma);
7
8          CollegeStudent Richard = new CollegeStudent();
9          GoStudy(Richard);
10     }
11
12     static void GoStudy(HighSchoolStudent student)
13     {
14         student.DoHomework();
15     }
16
17     static void GoStudy(CollegeStudent student)
18     {
19         student.WriteFinalReports();
20     }
21
22     // What if the 3rd kind of students comes into the system?
23 }
```

Using Inheritance & Subtype Polymorphism

```
1  class Student
2  {
3      public virtual void DoMyJob() { /* Do not know the detail yet. */}
4  }
5
6  class HighSchoolStudent : Student
7  {
8      void DoHomework() {}
9      public override void DoMyJob() { DoHomework(); }
10 }
11
12 class CollegeStudent : Student
13 {
14     void WriteFinalReports() {}
15     public override void DoMyJob() { WriteFinalReports(); }
16 }
```

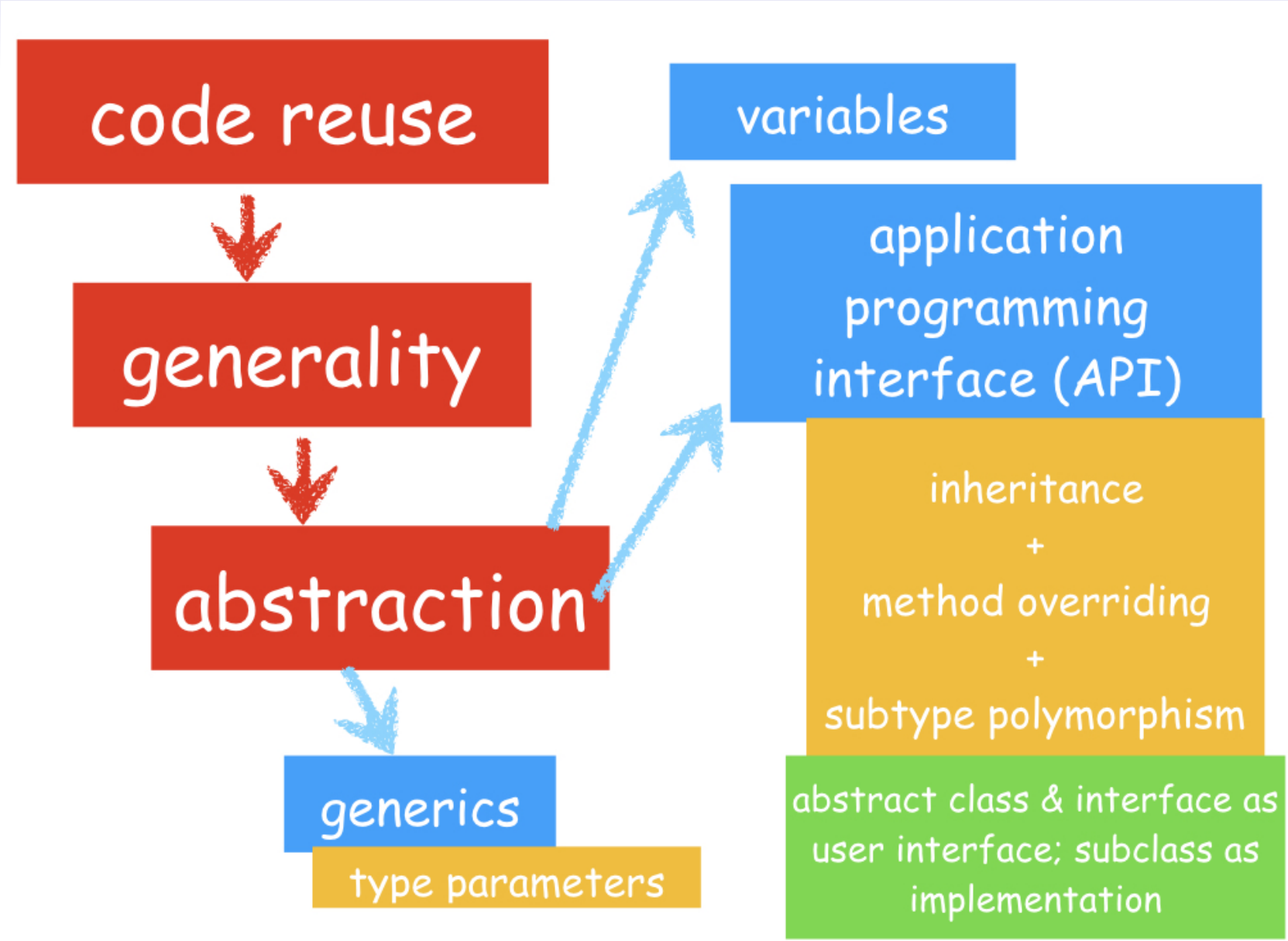
```
1  class PolymorphismDemo
2  {
3      static void Main(string[] args)
4      {
5          Student Emma = new HighSchoolStudent();
6          GoStudy(Emma);
7
8          Student Richard = new CollegeStudent();
9          GoStudy(Richard);
10     }
11
12     // We can handle all kinds of students in this way!!!
13     public static void GoStudy(Student student)
14     {
15         student.DoMyJob();
16     }
17 }
```

- This example illustrates the mechanism between ToString() and WriteLine().

Why OOP?[16]

- OOP is the solid foundation of modern (large-scale) software design.
- In particular, great reuse mechanism and abstraction are realized by these three concepts:
  - encapsulation isolates the internals (private members) from the externals, fulfilling the abstraction and providing the sufficient accessibility (public methods);
  - inheritance provides method overriding w/o changing the method signature;
  - polymorphism exploits the base class as a placeholder to manipulate the implementations (subtype objects).
- We use PIE as the shorthand for these three concepts.

[16] See `https://en.wikipedia.org/wiki/Programming_paradigm`


code reuse
generality
abstraction
generics
type parameters
variables
application programming interface (API)
inheritance
+
method overriding
+
subtype polymorphism
abstract class & interface as user interface; subclass as implementation

- This leads to the production of frameworks[17], which actually do most of the job, leaving the (application) programmer only with the job of customizing with business logic rules and providing hooks into it.
- This greatly reduces programming time and makes feasible the creation of larger and larger systems.
- In analog, we often manipulate objects in an abstract level; we don't need to know the details when we use them.
  - For example, using computers and cellphones, driving a car, and so on.

[17] See `https://docs.microsoft.com/en-us/dotnet/core/` and `https://docs.microsoft.com/en-us/aspnet/`.

Another Example

```
1  class Animal
2  {
3      public virtual void Speak() {}
4  }
5
6  class Dog : Animal
7  {
8      public override void Speak() { Console.WriteLine("Woof! Woof!"); }
9  }
10
11 class Cat : Animal
12 {
13     public override void Speak() { Console.WriteLine("Meow~"); }
14 }
15
16 class Bird : Animal
17 {
18     public override void Speak() { Console.WriteLine("Tweet!"); }
19 }
```

```
1  class PolymorphismDemo2
2  {
3      static void Main(string[] args)
4      {
5          Animal[] animals = { new Dog(), new Cat(), new Bird() };
6
7          foreach (Animal animal in animals) {
8              animal.Speak();
9          }
10     }
11 }
```

- Again, **Animal** is a placeholder for the three derived types.

Liskov Substitution Principle[18]

- For convenience, let **U** be a subtype of **T**.
- We manipulate objects (right-hand side) via references (left-hand side)!
- Liskov states that **T**-type objects may be replaced with **U**-type objects without altering any of the desirable properties of **T** (correctness, task performed, etc.).

[18] See `https://en.wikipedia.org/wiki/Liskov_substitution_principle`.

Casting

- Upcasting[19] is to cast the **U** object/variable to the **T** variable.

```
1        U u1 = new U(); // Trivial.
2        T t1 = u1;      // OK.
3        T t2 = new U(); // OK.
```

- Downcasting[20] is to cast the **T** variable to a **U** variable.

```
1        U u2 = (U) t2;  // OK, but dangerous. Why?
2        U u3 = new T(); // Error! Why?
```

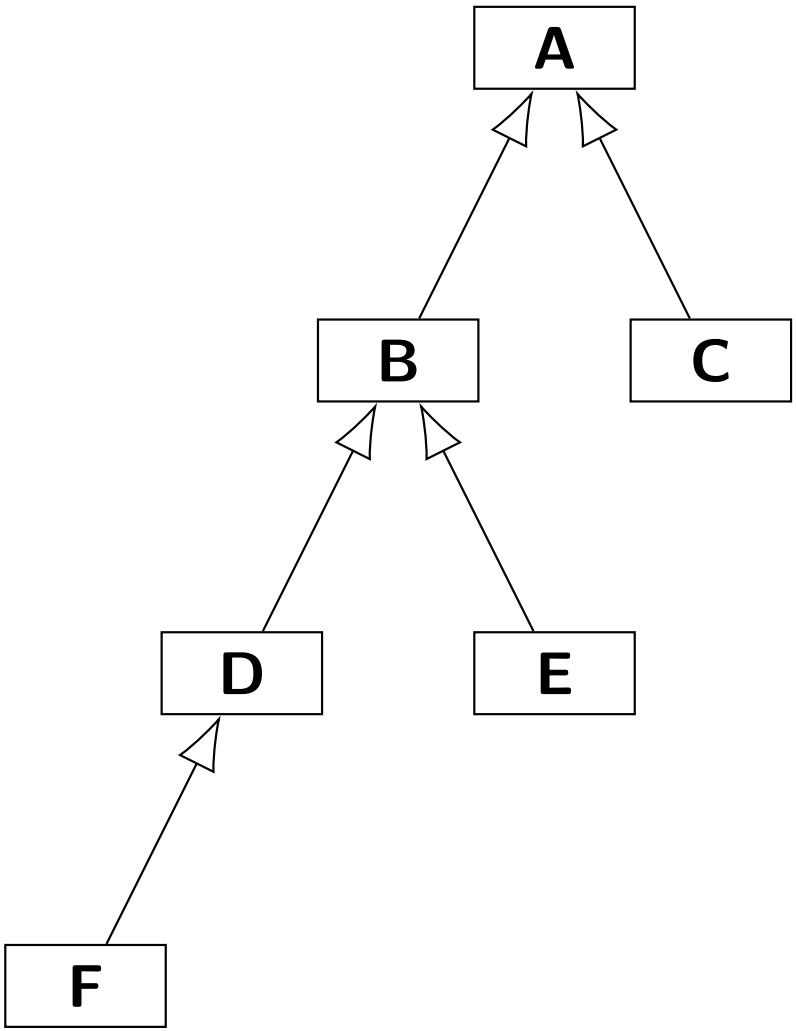
[19] A widening conversion; back compatibility.

[20] A narrow conversion; forward advance.

Solution: is

- Upcasting is wanted and always allowed. (Why?)
- However, downcasting is not always true even when you use cast operators.
  - In fact, type checking at compile time becomes unsound if any cast operator is used. (Why?)
- Even worse, a **T**-type variable can point to all siblings of **U**-type.
  - Recall that a **T**-type variable works as a placeholder.
- Run-time type information (RTTI) is needed to resolve the error: **InvalidCastException**.
- We can use is to check if the referenced object is of the target type at runtime.

Example



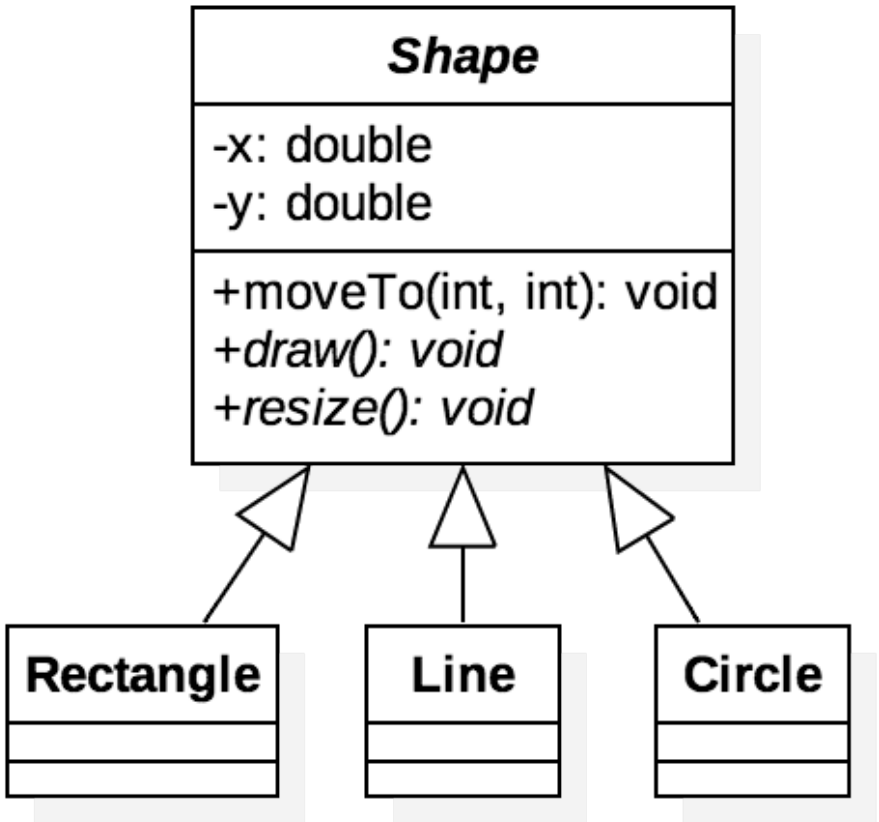
- The class inheritance can be represented by a digraph (directed graph).
- For example, **D** is a subtype of **A** and **B**, which are both reachable from **D** on the digraph.

```
1  class A {}
2  class B : A {}
3  class C : A {}
4  class D : B {}
5  class E : B {}
6  class F : D {}
7
8  class RTTI_Demo
9  {
10     public static void Main(string[] args)
11     {
12         Object o = new D();
13
14         Console.WriteLine(o is A); // Output true.
15         Console.WriteLine(o is B); // Output true.
16         Console.WriteLine(o is C); // Output false.
17         Console.WriteLine(o is D); // Output true.
18         Console.WriteLine(o is E); // Output false.
19         Console.WriteLine(o is F); // Output false.
20     }
21 }
```

Abstract Methods/Classes

- A method without implementation can be declared abstract.
  - The method has no brace but ends by a semicolon.
- If a class has one or more abstract methods, then the class itself must be declared abstract.
- Typically, one abstract class sits at the top of one class hierarchy, acting as an placeholder.
- No abstract class cannot be instantiated directly.
- When inheriting an abstract class, the editor could help you recall every abstract methods.

Example



- In UML, abstract methods and classes are presented in italic.
- The method *Draw*() and *Resize*() can be implemented when the specific shape is known.

The sealed Keyword[21]

- When applied to a class, the sealed modifier prevents other classes from inheriting from it.

```
1 class A {}
2 sealed class B: A {}
3 class C: B {} // You cannot extend C by inheriting from B.
```

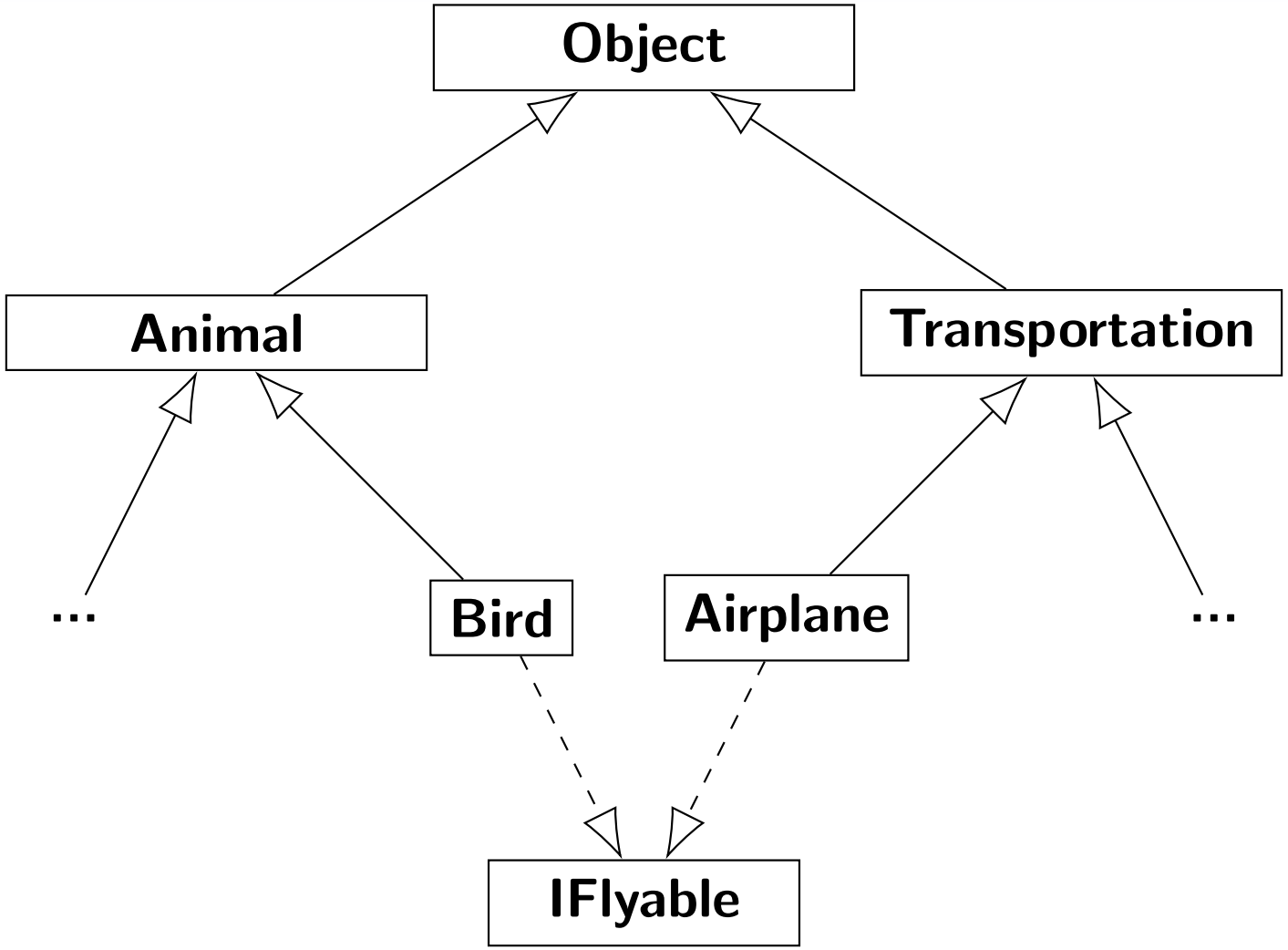
[21] See https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed.

Another IS-A Relationship: Interface Inheritance

- Objects of different types are supposed to work together without a proper vertical relationship.
- For example, consider **Bird** inherited from **Animal** and **Airplane** inherited from **Transportation**.
- Both **Bird** and **Airplane** are able to fly in the sky, say by calling the method Fly().
- In semantics, the method Fly() could not be defined in their base classes. (Why?)

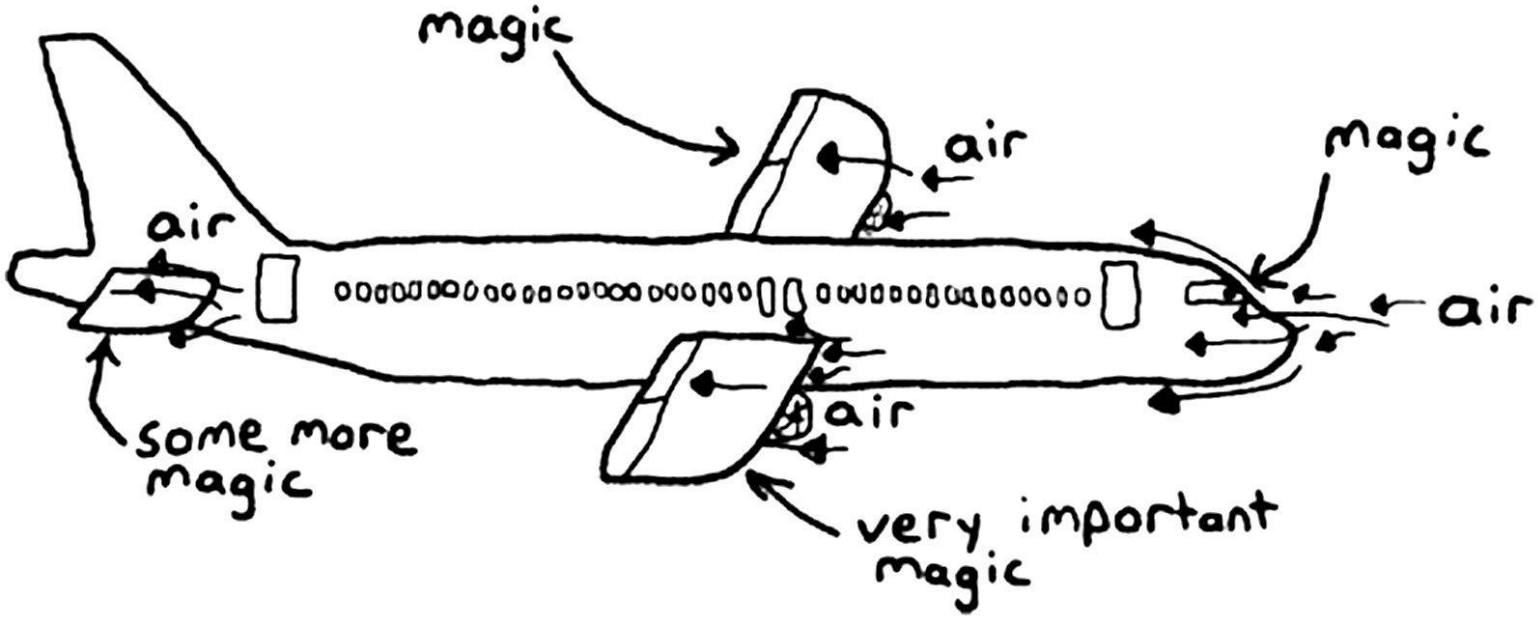
- We wish those flyable objects go flying by calling one API, just like the way of **Student**.
- Recall that **Object** is the base class of everything.
- So, how about using **Object** as the placeholder?
  - Not really. (Why?)
- Clearly, we need a horizontal relationship: interface.

```
1 interface IFlyable
2 {
3     void Fly(); // Implicitly public and abstract.
4 }
```


Object
Animal
Transportation
...
Bird
Airplane
...
IFlyable

```
1  class Animal { }
2  class Bird : Animal, IFlyable
3  {
4      void FlyByFlappingWings()
5      {
6          Console.WriteLine("Flapping wings!");
7      }
8
9      public void Fly() { FlyByFlappingWings(); }
10 }
11
12 class Transportation { }
13 class Airplane : Transportation, IFlyable
14 {
15     void FlyByCastingMagic()
16     {
17         Console.WriteLine("#$%@$^@!#$!");
18     }
19
20     public void Fly() { FlyByCastingMagic(); }
21 }
```

how planes fly



https://i.imgur.com/y2bmNpz.jpg

```
1  class InterfaceDemo
2  {
3      static void Main(string[] args)
4      {
5          Bird owl = new Bird();
6          GoFly(owl);
7
8          Airplane a380 = new Airplane();
9          GoFly(a380);
10     }
11
12     public static void GoFly(IFlyable flyer)
13     {
14         flyer.Fly();
15     }
16 }
```

A Deep Dive into Interfaces

- An interface is a contract between the object and the client.
- As shown, an interface is a reference type, just like classes.
- Unlike classes, interfaces are used to define methods without implementation so that they cannot be instantiated (directly).
- Also, interfaces are stateless.
- A class could implement one or multiple interfaces!

- Note that an interface can extend another interfaces!
  - Like a collection of contracts, in some sense.
- For example, **IRunnable**, **IList**, and **IComparable**[22].
- Beginning with C# 8.0, we have new features as follows:
  - may define a default implementation for members;
  - may also define static members in order to provide a single implementation for common functionality.

[22] See `https://docs.microsoft.com/zh-tw/dotnet/api/system.icomparable?view=netcore-3.1`.

Timing for Interfaces & Abstract Classes

- Consider using abstract classes if you want to:
  - share code among several closely related classes, and
  - declare non-constant or non-static fields.
- Consider using interfaces for any of situations as follows:
  - unrelated classes would implement your interface;
  - specify the behavior of a particular data type, but not concerned about who implements its behavior;
  - take advantage of multiple inheritance.

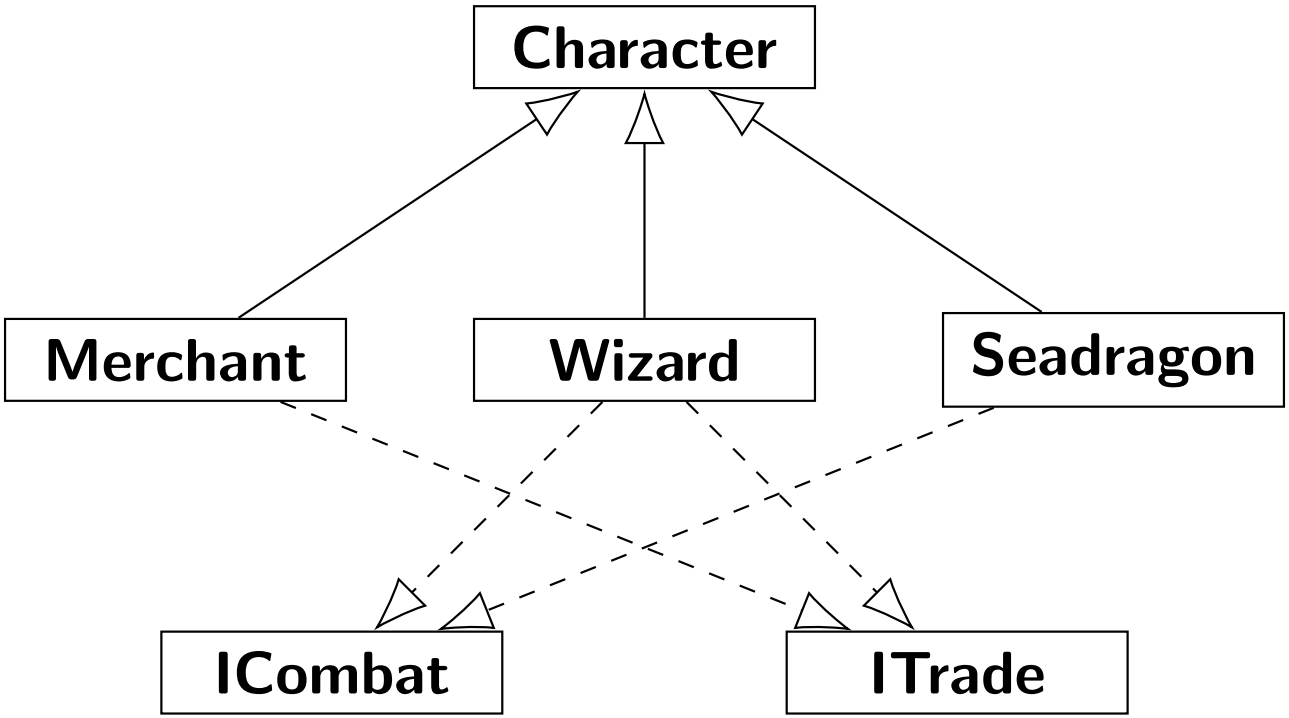
Exercise: RPG

https://i.pinimg.com/originals/86/23/1e/86231eb0d17be0192765c38dd4afea89.jpg

https://pm1.narvii.com/7075/794ae8b788ed9ae4c442e371ad5206911fbb2cber1-1950-1550v2_hq.jpg

https://s-media-cache-ak0.pinimg.com/originals/97/5b/81/975b8177e951f45c7f372de2dc08da6e.jpg

- First, **Wizard**, **SeaDragon**, and **Merchant** are three of **Character**s.
- In particular, **Wizard** fights with **SeaDragon** by invoking Attack().
- **Wizard** buys and sells stuffs with **Merchant** by invoking BuyAndSell().
- However, **SeaDragon** cannot buy and sell stuffs; **Merchant** cannot attack others.


Character
Merchant
Wizard
Seadragon
ICombat
ITrade

```
1  abstract class Character {}
2
3  interface ICombat
4  {
5      void Attack(ICombat enemy);
6  }
7
8  interface ITrade
9  {
10     void BuyAndSell(ITrade counterpart);
11 }
```

```
1 class Wizard : Character, ICombat, ITrade
2 {
3     public void Attack(ICombat enemy) {}
4     public void BuyAndSell(ITrade counterpart) {}
5 }
```

```
1 class SeaDragon : Character, ICombat
2 {
3     public void Attack(ICombat enemy) {}
4 }
```

```
1 class Merchant : Character, ITrade
2 {
3     public void BuyAndSell(ITrade counterpart) {}
4 }
```

Delegation[24]

- A delegate is a type that safely encapsulates a method, similar to a function pointer in C and C++.
- The type of a delegate is defined by the name of the delegate.
- A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an anonymous function.[23]
- Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method.

[23] See `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions`.

[24] See `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates`.

Example

```
1  class Program
2  {
3      delegate double Calculator(double x, double y);
4
5      static double Add(double x, double y) { return x + y; }
6      static double Mul(double x, double y) => x * y;
7      // The above statement is presented in lambda expressions!
8
9      static void Main(string[] args)
10     {
11         Calculator calculator;
12
13         calculator = Add;
14         Console.WriteLine(calculator(10, 20)); // Output 30.
15         calculator = Mul;
16         Console.WriteLine(calculator(10, 20)); // Output 200.
17     }
18 }
```

Another Example: Callback

```
1  class Program
2  {
3      delegate void Del(string message);
4
5      static void MethodWithCallback(int param1, int param2,
6                                     Del callback)
7      {
8          callback(String.Format("Sum = {0}", param1 + param2));
9      }
10
11     static void Main(string[] args)
12     {
13         MethodWithCallback(1, 2, x => Console.WriteLine(x));
14     }
15 }
```

Delegation vs. Inheritance

- Class inheritance is a powerful way to achieve code reuse.
- However, class inheritance violates encapsulation!
- This is because a derived class depends on the implementation details of its base class for its proper function.
- To solve this issue, we favor delegation over inheritance.[25]

[25] GoF (1995); Also see Item 18 in Bloch (2018).

Enumeration Types[26]

- An enum type is a distinct value type that declares a set of named constants.
- Each enum type has a corresponding integral type called the underlying type of the enum type, say int, without explicitly declaration.
- This mechanism enhances type safety and makes the source code more readable!
- For example,

```
1 enum Season
2 {
3     Spring, Summer, Autumn, Winter
4 }
```

[26] See https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum.

Structure Types[27]

- A structure type (or struct type) is a value type that can encapsulate data and related functionality.
- Typically, you use structure types to design small data-centric types that provide little or no behavior.
- Because structure types have value semantics, we recommend you to define immutable structure types.

[27] See `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct`.

Example

```
1  public readonly struct Coords
2  {
3      public Coords(double x, double y)
4      {
5          X = x;
6          Y = y;
7      }
8
9      public double X { get; }
10     public double Y { get; }
11
12     public override string ToString() => $"({X}, {Y})";
13 }
```

Tuple Types[28]

- Available in C# 7.0 and later, the tuples feature provides concise syntax to group multiple data elements.

```
...
        (double, int) t1 = (4.5, 3);
        Console.WriteLine("Tuple with elements {0} and {1}.",
                          t1.Item1, t1.Item2);

        (double Sum, int Count) t2 = (4.5, 3);
        Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
...
```

- You may add the dollar sign ($) before the string to indicate the members in the tuple.

[28] See https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples.

Nested Types[29]

- A type defined within a class, struct, or interface is called a nested type.

```
1  class Program
2  {
3      public static Main(string[] args)
4      {
5          House home = new House();
6          // Room bedreoom = new Room(); // You cannot do this.
7      }
8  }
9
10 class House
11 {
12     class Room { } // Another HAS-A relationship: composition.
13 }
```

[29] See https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/nested-types.

Special Issue: Using Namespaces[31]

- Namespaces are heavily used within C# programs in two ways.
- Firstly, the .NET classes use namespaces to organize its many classes.[30]
- Secondly, declaring your own namespaces can help control the scope of class and method names in larger programming projects.
- The using directives facilitate the use of namespaces and types defined in other namespaces.

[30] The counterpart of Java is the "package."

[31] See `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/namespaces`.

Example

```
1  using Toolbox;
2
3  namespace NamespaceDemo
4  {
5      using B = NestedNamespace.A;
6
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             B b = new B();
12             C c = new C();
13         }
14     }
15
16     namespace NestedNamespace
17     {
18         class A {}
19     }
20 }
```

```
1  namespace Toolbox
2  {
3      public class C { }
4  }
```