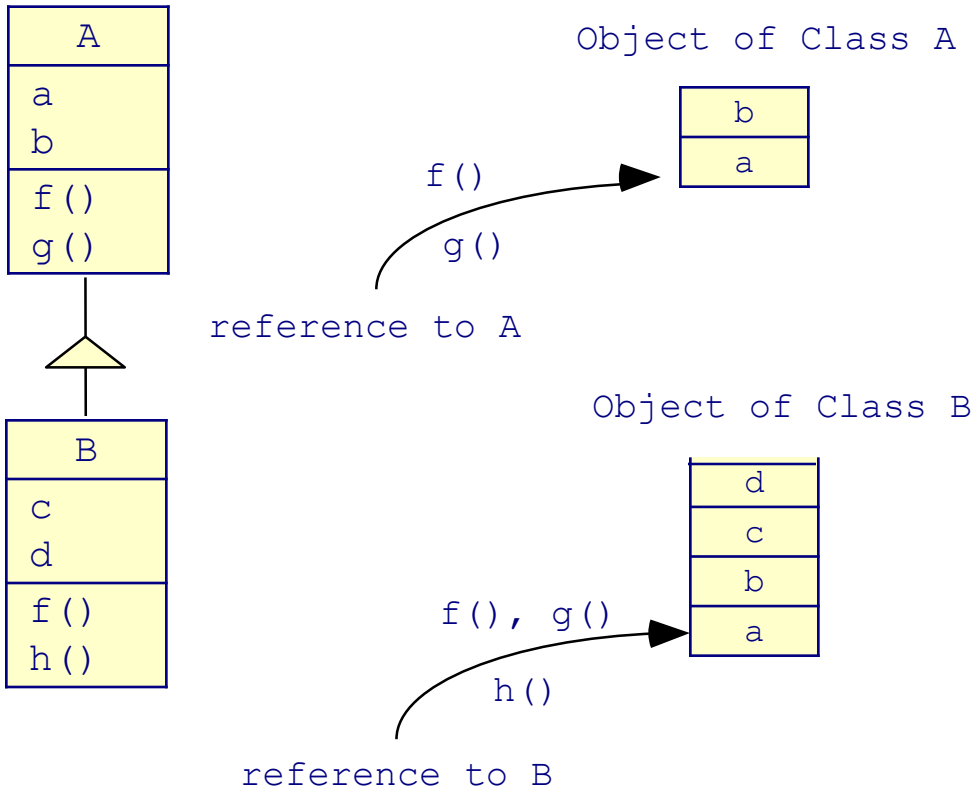
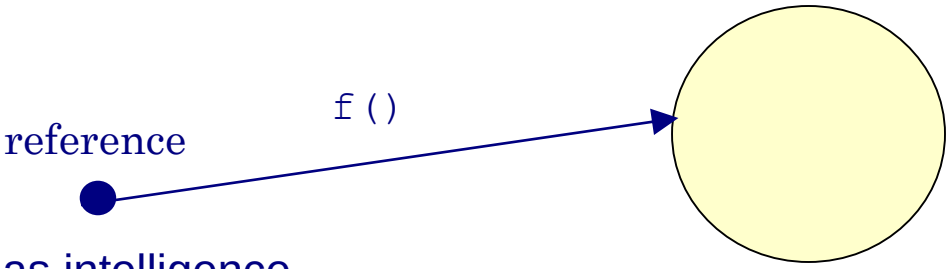
Inheritance



In C#, the definition of f() requires **virtual, override**, or **new** keyword

Polymorphism

- Comes from Smalltalk： Image of a Smalltalk object



- An object has intelligence
- A client can use the object if it has a reference to the object. The client does not need to know which class the object belongs to (a reference does not have type in Smalltalk)
- The client *asks an object to invoke a method by sending a message to the object* that the reference is pointing at (e.g., “please invoke f( )”)
- Upon a receipt of the message, *the object invokes the most appropriate method* for the request
  - If the object has no corresponding method, it throws an error

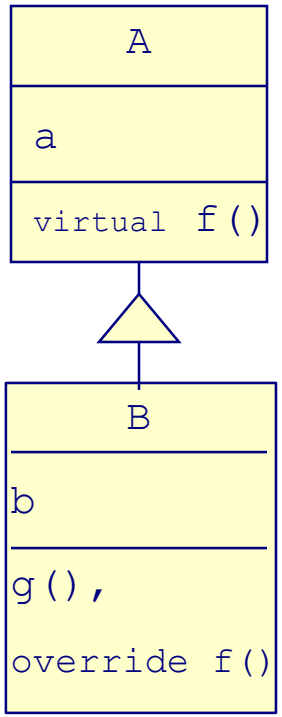
Polymorphism (cont)

- Declare “ap” to be a reference to class A and “bp” to be a reference to class B

```
A ap;
B bp;
```

- The C# compiler maintains the information about which fields and methods can be accessed through **ap** and **bp**

```
ap: a and f()
bp: a, b, f(), and g()
```

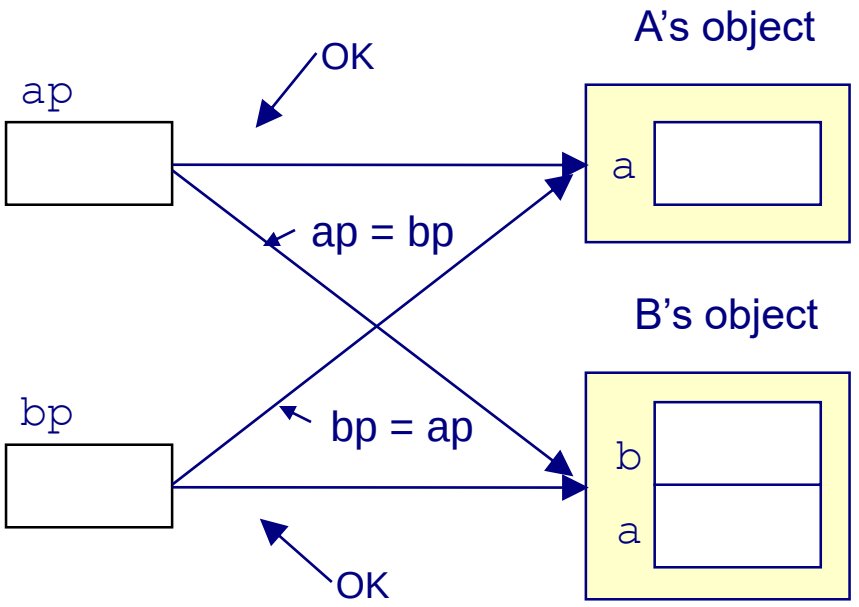


Polymorphism (cont)

- Create objects of class A and class B in the heap

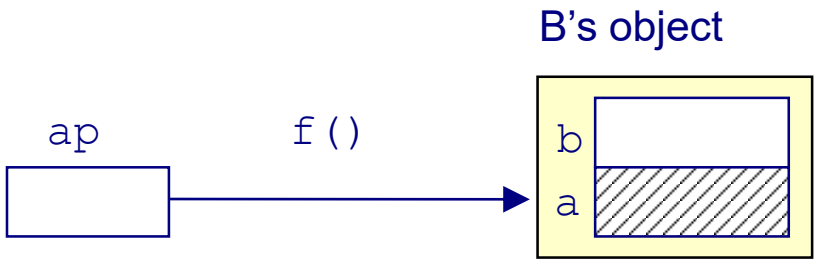
  ap = new A();

  bp = new B();



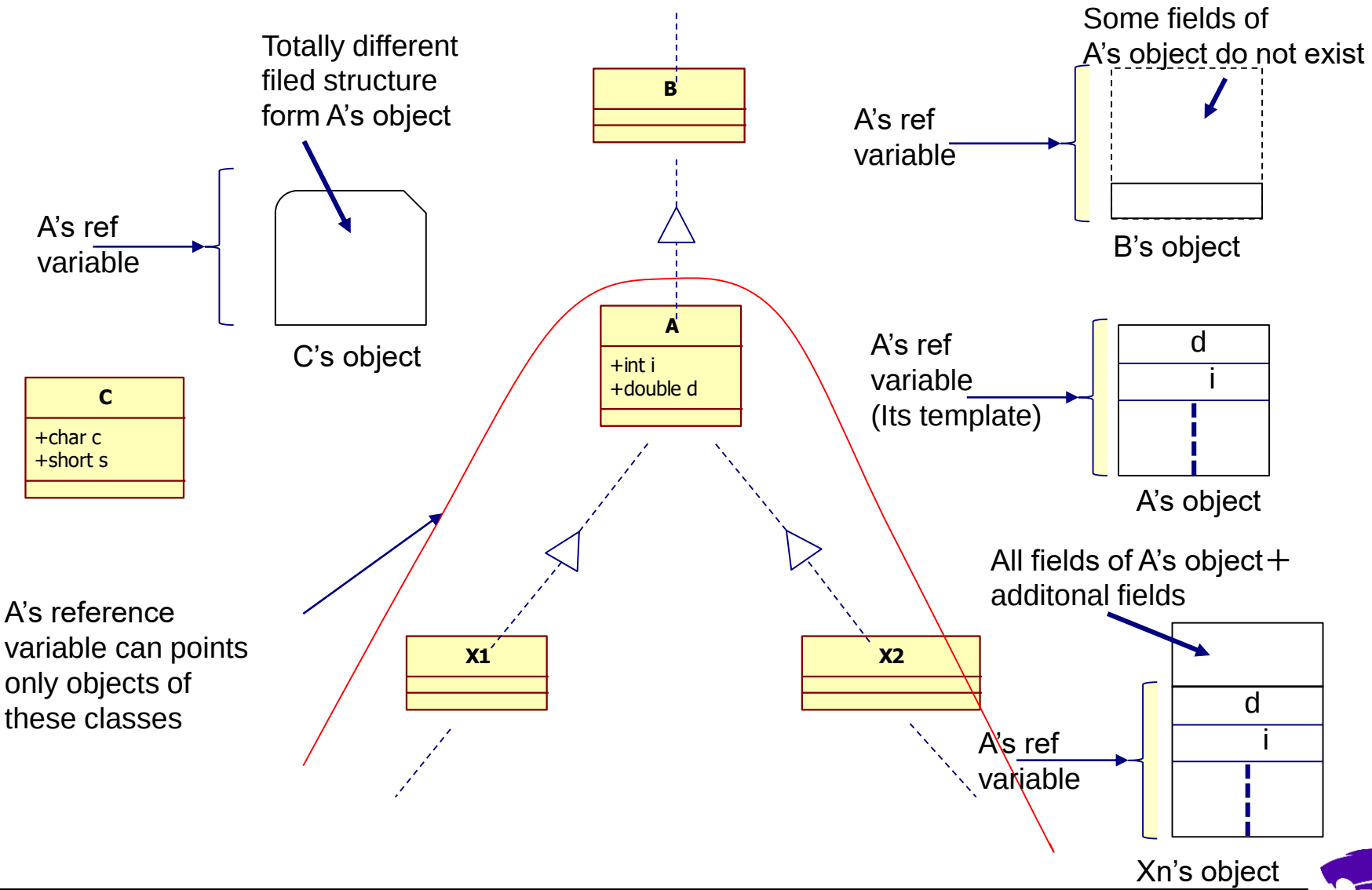
- Question: Which assignment is allowed ?

Polymorphism (cont)



- This is allowed; however, only field **a** and method **f()** can be accessed through **ap**
- Since **f()** is declared to be **virtual** and re-declared (overridden) in class **B**, **B**'s **f()** is invoked (*Polymorphism*)
- if **f()** is not declared to be **virtual**, which function is called is determined at the compile time based on the type of the rerference
  - Since **ap** is of type "reference to class **A**", **A**'s **f()** is invoked

Objects of classes that A's reference variable can point to



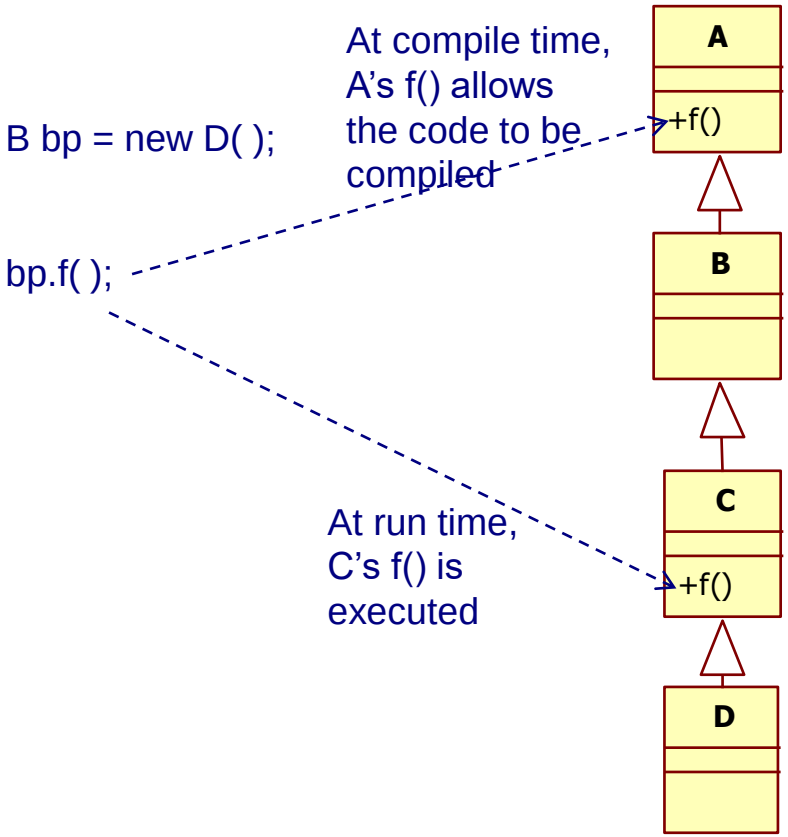
Polymorphism (cont)

- At Compile Time
  - For each assignment statement to an object pointer
    - the compiler checks whether the left-side pointer can point to the right side object
      - **ap** can point to both **A**'s object and **B**'s object
      - **bp** can point to only **B**'s object
  - For each method call
    - The compiler checks whether the method with the same signature can be found in any class in the super-class chain starting at **the class of the pointer's type**
      - If the method is not found, the compiler reports an error
    - It does not matter what class of object the pointer will point to at run-time

Polymorphism (cont)

- At run time
  - It calls the first method found in the super-class chain starting at **the class of the object** that the pointer is pointing to
  - It does not matter what type of pointer is pointing to the object
    1. Pointer's view: when **ap.f()** is executed
       - if **ap** is pointing at **A**'s object, **A**'s **f()** is invoked
       - if **ap** is pointing at **B**'s object, **B**'s **f()** is invoked
    2. Object's view: **B**'s object is asked to invoke **f()**
       - it always invokes **B**'s **f()** (it does not matter what pointer points at the object; whether a pointer can point to the object is resolved at compile time)
  - Note that at run time, this perfectly emulates the execution of Smalltalk
    - The method invocation *executes the most appropriate method for the object*

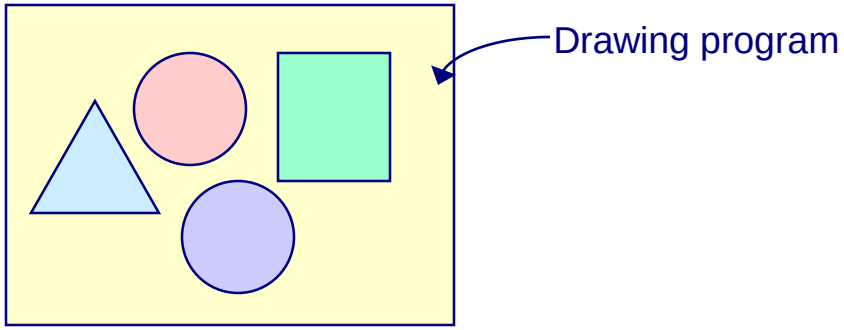
Polymorphism (cont)



Why does the compiler check A's f() for "bp.f();" to be compiled ?

- At run time, bp can points to only an object of B and its sub-classes
- Therefore, even in the worst case in which f() is not defined in B or any of its sub-classes, bp.f() can be executed at run time by invoking A's f() (avoiding a run-time error)

Example of Polymorphism 1



- The program maintains information necessary to re-draw circles, rectangles, and triangles.

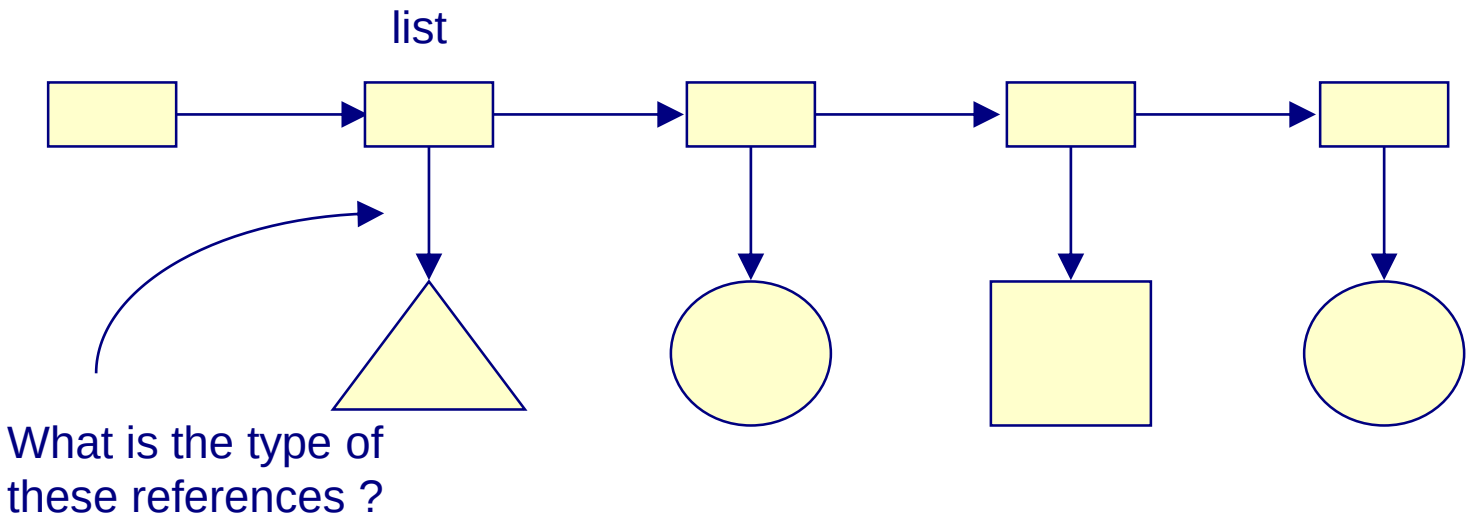
Circle
point diameter
`draw()`

rectangle
point1 point2
`draw()`

triangle
point1 point2 point3
`draw()`

Example of Polymorphism 1 (cont)

- Each time the user draws a figure, the program stores the corresponding object, say in a list
- When redrawing is required, the program traverses the list and sends each object a message asking to **`draw()`**

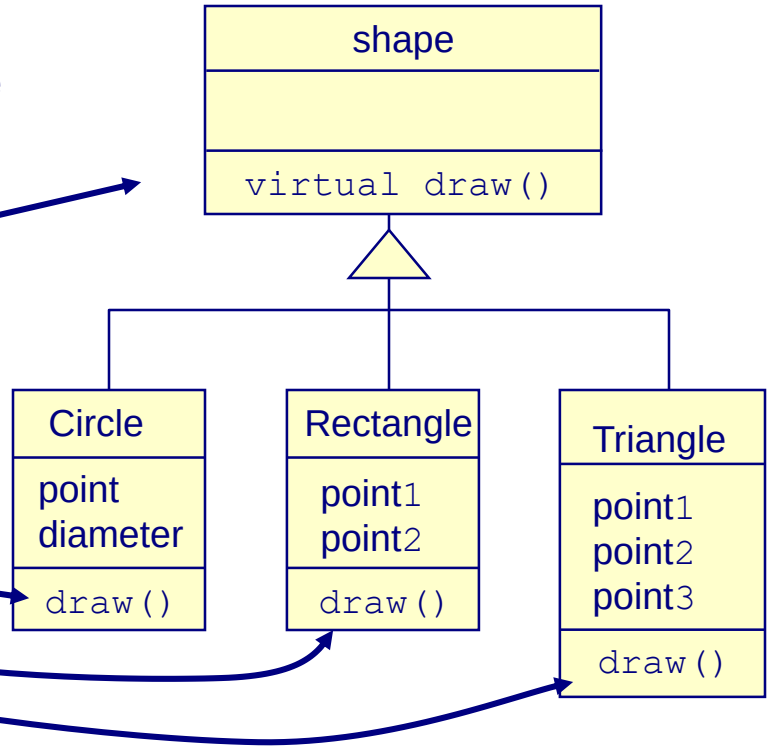


Example of Polymorphism 1 (cont)

The type of the references in the list should be “Shape” (or downcast the type to “Shape”

Define draw() in “Shape” so that the references to “Shape” can see draw( )

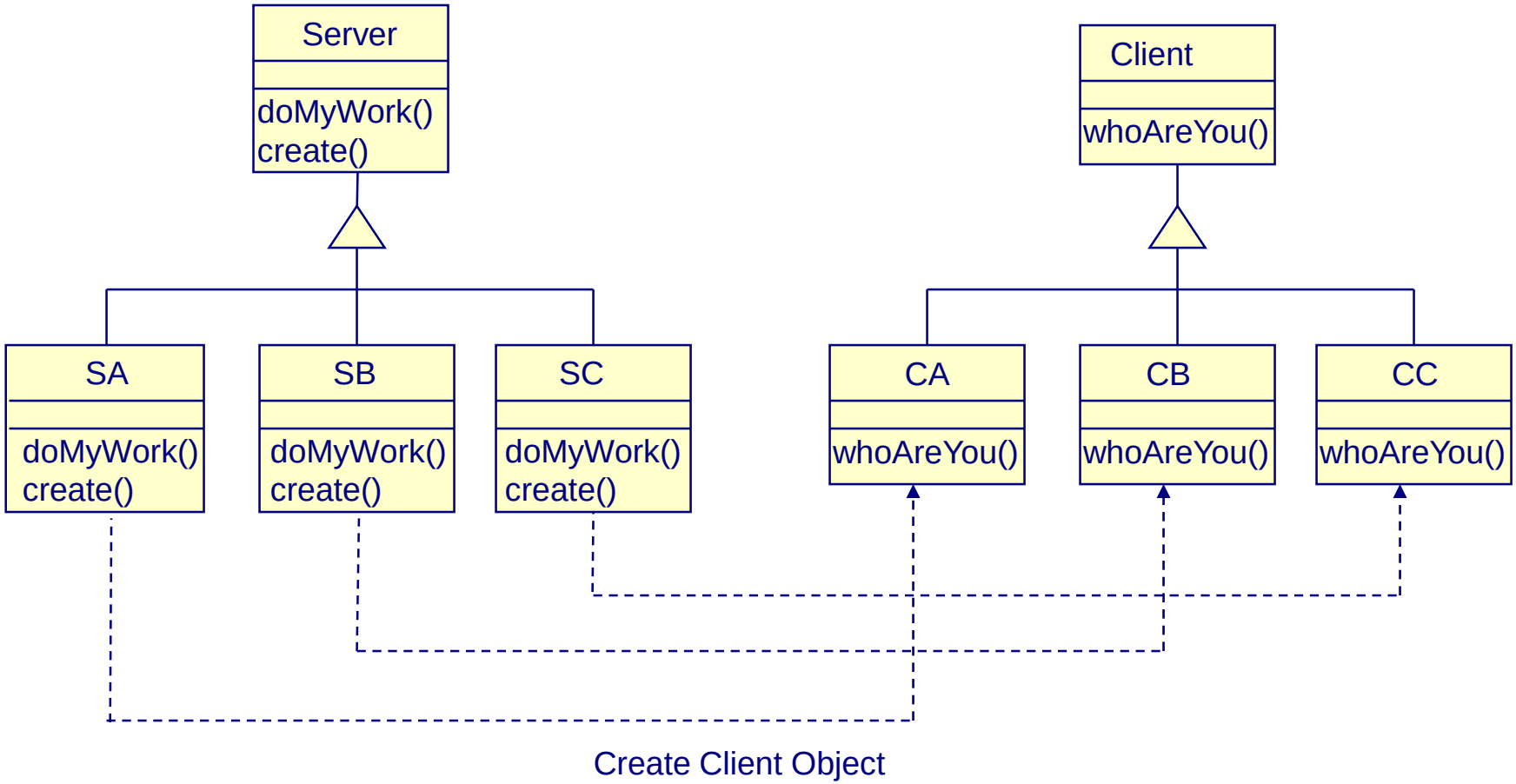
The draw() methods of Circle, Rectangle, and Triangle are executed



Factory Method Pattern: Example of Design Pattern 1

- In general, instantiation of an object is done by ”new Constructor()”.
- The Factory Method Pattern provides a method that instantiates an object
  - Such a method is called a “factory method”
  - An execution of “new Constructor()” is carried out in a factory method

MMFactory Method Pattern : Example



Iterator Pattern : Example of Design Pattern 2

- Consider the following situation of implementing a List
  - In an early stage of analysis/design, we thought that we would often need to access elements in the List by their indices and therefore decided to use an array structure
  - Later, however, we discovered that we would insert and delete elements in the List more often and therefore decided to change the implementation of the List from the array to a linked list
- Consider how we traverse and access each element in an array and a linked-list in C
  - Their programs are totally different
- That is, we need to rewrite the code in the left to the one in the right when we switch the implementation from the array to the list

Traverse and access the elements in array A

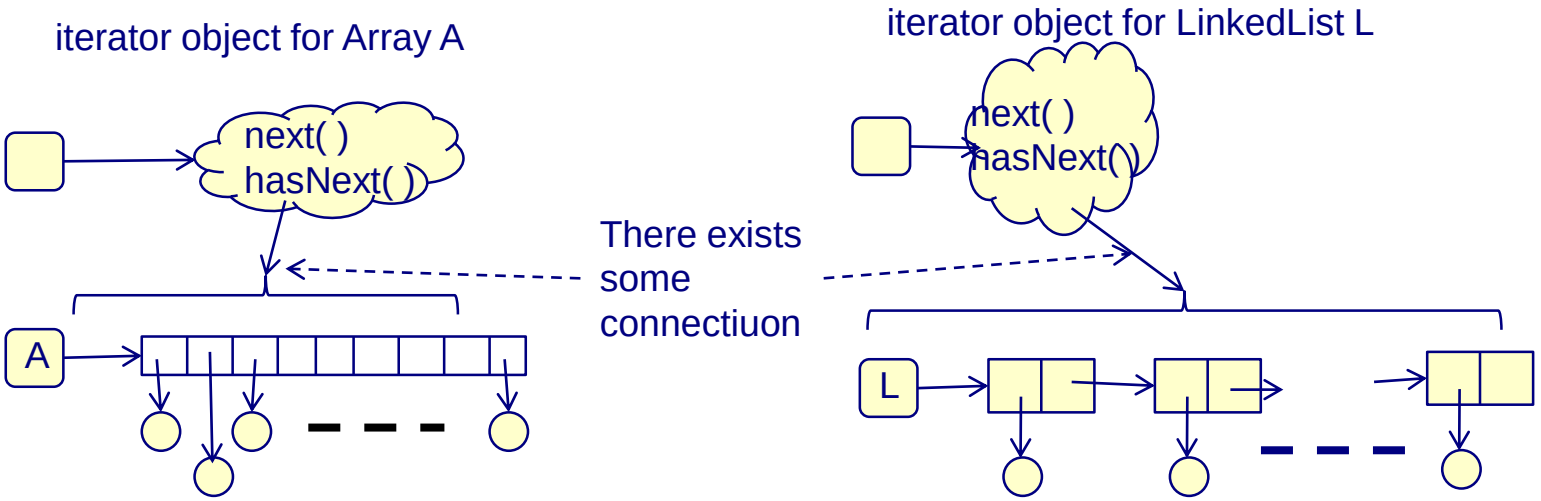
```
for (i = 0; i < A.length; i++)
  …  A[i] …
```

Traverse and access the elements in list L

```
ptr=L;
while (ptr != NULL) {
 …  ptr.obj …
  ptr = ptr.next;
}
```

Iterator Pattern (cont)

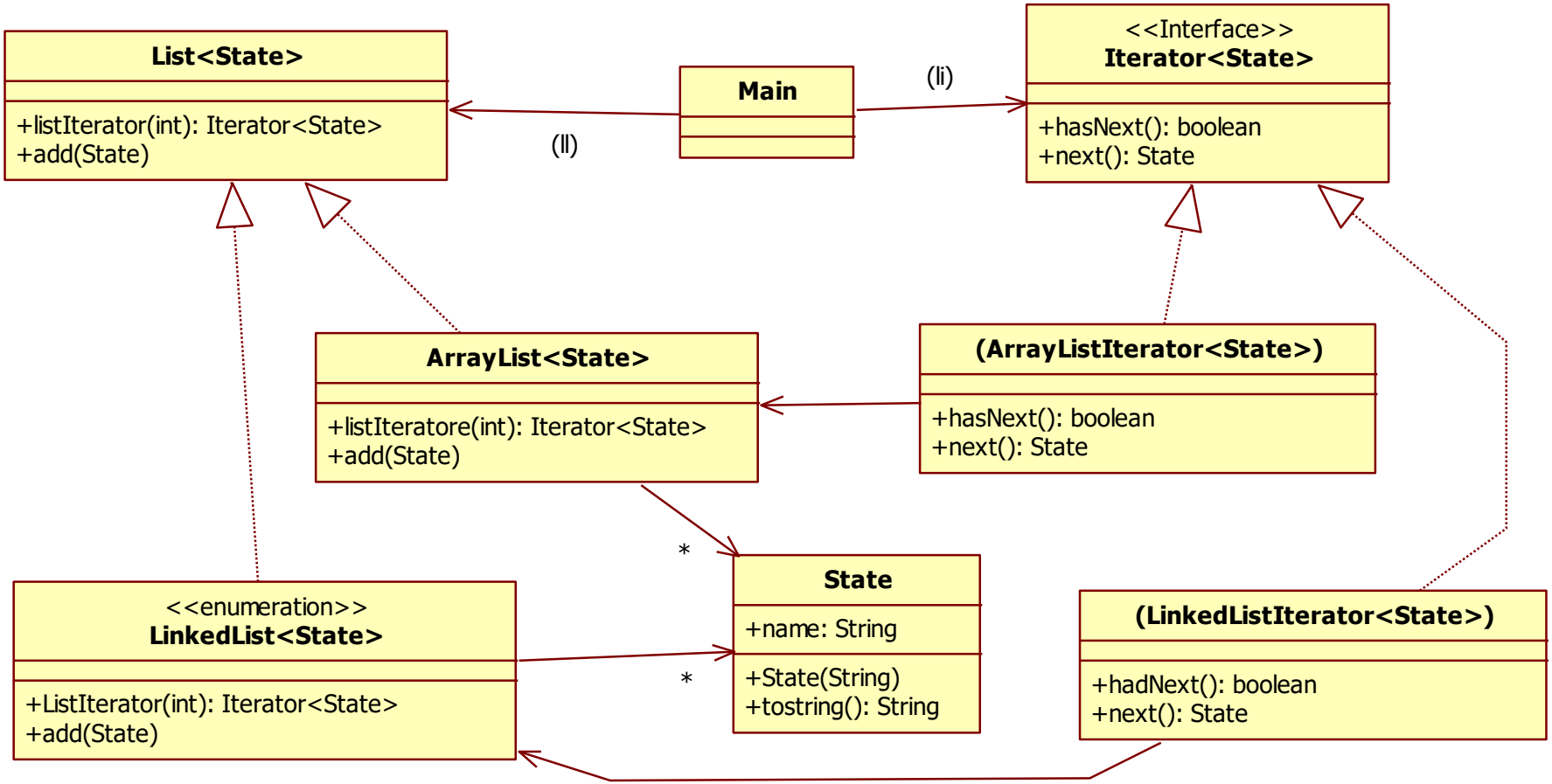
- Instead of directly traversing a collection object, such as Array A or Linked List L, we create an object of a special class (called an “Iterator” or “Enumerator” class) that implements methods that traverse the collection object and access the iterator object to traverse the collection.
- Note that the implementation of the Iterator class for Array is completely different from the implementation of the Iterator class for LinkedList. However, they both have the same interface.



Iterator Pattern (cont)

- Using Iterator classes, we do not need to change code fragments that traverse a collection object even if we change the collection from Array to LinkedList or visa versa
- Furthermore, by applying the Factory Method Pattern when creating an Iterator object, we can eliminate “new IteratorClass()” in the code and can always create an appropriate Iterator object. The factory methods are defined in the collection classes.
  - Collection Classes: Host
  - Iterator Classes: Supporter

Iterator Pattern used in JFC



Iterator Pattern (cont)

- In this program, even if we switch the implementation of the List<State> from ArrayList<State> to LinkedList<State>, the code to traverse and access the collection elements are the same

```
Iterator<State> li = ll.listIterator(0);
while (li.hasNext())
   System.out.println(li.next().toString());
```

- Note that the types of "ll" and "li" are "List<State>" and "Iterator<State>", respectively. They are both interfaces (not classes)
  - We apply the rule "a reference to a super-class (interface) can point to a sub-class object"

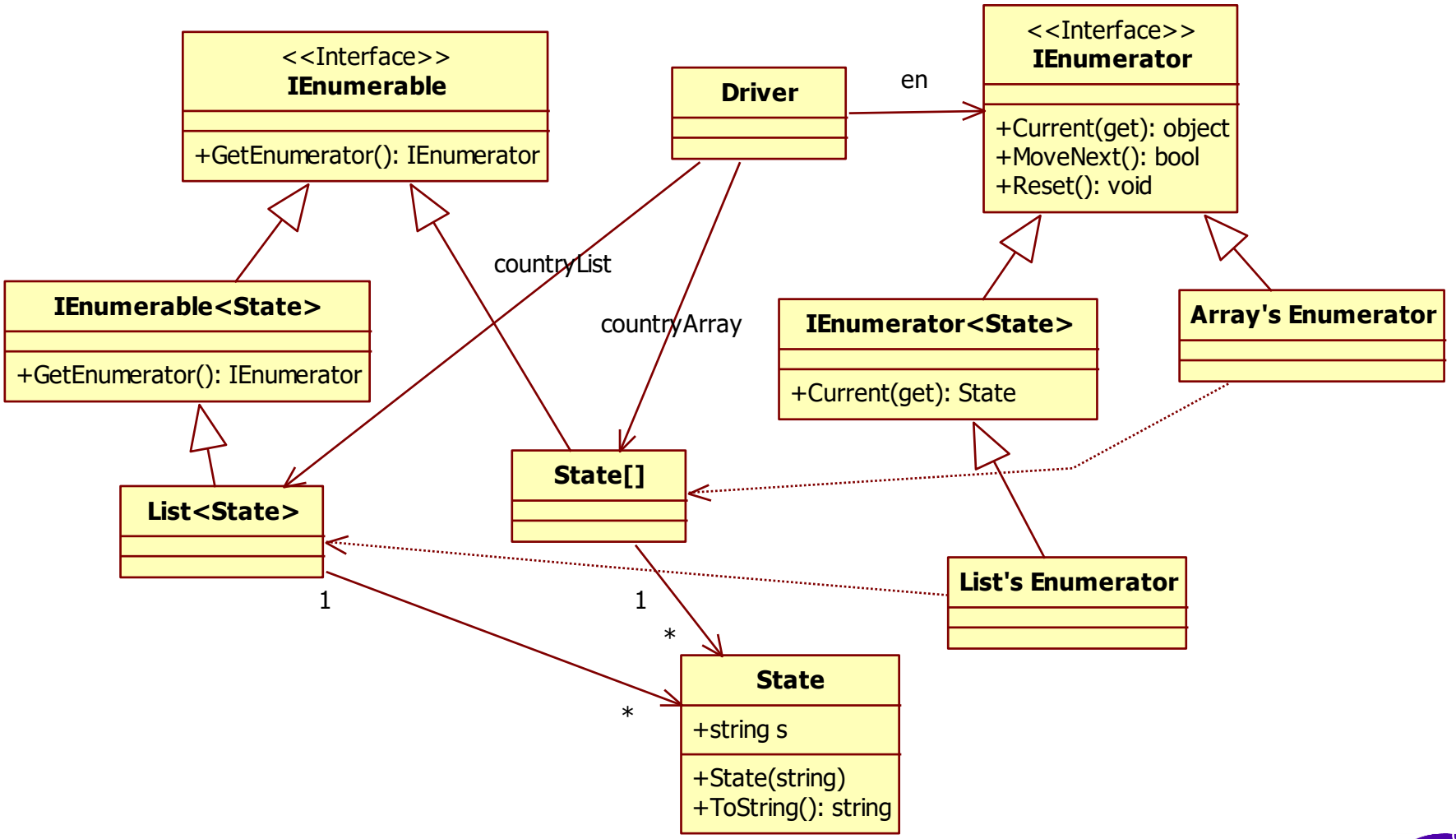
Iterator Pattern (cont)

- This is a program that illustrates "programming in an abstract world"
  - The only link to the concrete world is the value of "ll", which is set to either an "ArrayList<State>" object or an "LinkedList<State>" object
    - If "ll" is set to an "ArrayList<State>" object, all objects become "Array" type
    - If "ll" is set to an "LinkedList<State>" object, all objects become "List" type

Iterator Pattern (cont)

- In particular, note that the class name of an object that "li" points to does not appear in main() at all
  - Such an object is not created by "new ClassName()", and the programmer does not even know its class name at all
  - Then, how is such an object created ?
- Method "listIterator(0)" instantiates an appropriate object
  - If "ll" points to
    - an "ArrayList<State>" object, an iterator object for ArrayList<State> is created (by "ArrayList<State>"'s "listIterator(0)")
    - an "LinkedList<State>" object, an iterator object for LinkedList<State> is created (by "LinkedList<State>"'s "listIterator(0)")
  - Such a method is called a "Factory Method" (refer to the Factory Method Pattern)
- Methods "hasNext()" and "next()" are not defined in "List<State>". They are defined in the independent interface "Iterator<State>".
  - Multiple "Iterator<State>" objects may be created for one "List<State>" object
  - Each "Iterator<State>" object may point to a different element in the "List<State>" object

Iterator Pattern used in .NET



The foreach construct in .NET (cont)

- foreach in .NETforeach

```
public static void Main()
{
  IEnumerable st= new States();

  foreach (string state in st)
    … loop body …
}
```

- is transformed into the following code by the compiler

```
public static void Main()
{
  IEnumerable st = new States();

  IEnumerator en = st.GetEnumerator();
  while (en.MoveNext())
  {
    string state = (string)en.Current;
    … loop body …
  }
}
```