A Second Look At C#

Subtype Polymorphism

```
Person x;
```

- Does this declare **`x`** to be a reference to an object of the **`Person`** class?
- Not exactly—the *type* **`Person`** may include references to objects of other classes
- C# has subtype polymorphism

Outline

- 15.2 Implementing interfaces
- 15.3 Extending classes
- 15.4 Extending and implementing
- 15.5 Multiple inheritance
- 15.6 Generics

Defining Interfaces

- A method *prototype* defines the method name and type—no method body
- C# interface is a collection of method prototypes —no method bodies allowed
- Defines *all* methods to be implemented

```
public interface Drawable {
  void show(int xPos, int yPos);
  void hide();
}
```

Implementing Interfaces

- A class declares that it implements an interface
- The class must then provide **`public`** method definitions matching *all* those in the interface prototypes

Examples

```
public interface Drawable {
  void show(int xPos, int yPos);
  void hide();
}
```

```
public class Icon : Drawable {
  public void show(int x, int y) {
     ... method body ...
  }
  public void hide() {
     ... method body ...
  }
  ...more methods and fields...
}
```

```
public class Square : Drawable, Scalable {
  ... all required methods of all interfaces implemented ...
}
```

Why Use Interfaces?

An interface must be implemented by many classes following an *identical prototype*:

```
public class Icon : Drawable …
public class MousePointer : Drawable …
public class Oval : Drawable …
```

- Interface name can be used as a reference type:

```
Drawable d;
d = new Icon("i1.gif");
d.show(0,0);
d = new Oval(20,30);
d.show(0,0);
```

- Provides form of *multiple* inheritance

Polymorphism With Interfaces

```
public class Window : Drawable …
public class MousePointer : Drawable …
public class Oval : Drawable …
```

```
static void flashoff(Drawable d, int k) {
  for (int i = 0; i < k; i++) {
    d.show(0,0);
    d.hide();
  }
}
```

- Class of object referred to by **`d`** is not known at compile time, could be any implementor.
- As a class that **`implements Drawable`**, it has **`show`** and **`hide`** methods defined

A More Complete Interface Example

- A **`Stack`** interface for a collection of Objects with *push* and *pop* methods
- An implementation using a **`List`**
- An implementation using an **`Array`**

```
using System;
using System.Collections.Generic;

interface Stack {
            void push( Object o );
            Object pop();
}
public class SimpleStack  {
   public static void Main(){
      Stack a = new ListStack();
       a.push("ListStack example");
       Console.WriteLine(a.pop());

       a = new ArrayStack();
       a.push("ArrayStack example");
       Console.WriteLine(a.pop());
   }
}
```

```
class ListStack : Stack {
  List <Object> data=new List<Object>();

   public void push ( Object o ) {
                              data.Add(o);     }
   public Object pop() {
      Object obj = data[data.Count - 1];
      data.RemoveAt(data.Count - 1);
      return obj;
   }
}

class ArrayStack : Stack {
    Object [] data=new Object[5];
    int top=-1;

 public void push ( Object o ){
                          data[++top]=o;  }
 public Object pop() { return data[top--]; }
}
```

Exercise 1

1. What is the program output?
2. Locate the interface and implementation.
3. Is polymorphism evident at **`a.pop()`?**
4. Is this inheritance?

Outline

- 15.2 Implementing interfaces
- 15.3 Extending classes
- 15.4 Extending and implementing
- 15.5 Multiple inheritance
- 15.6 Generics

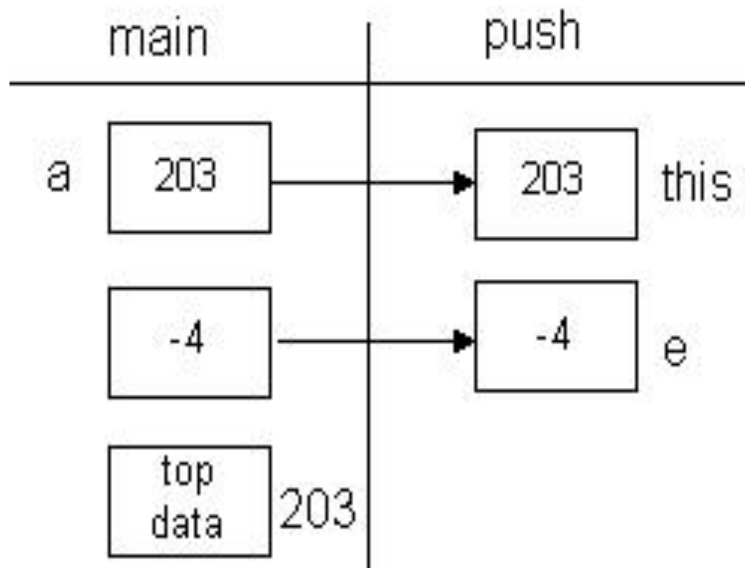
C# Parameter passing is by Value (mostly)

- C# passes parameters by value meaning that a copy of the parameters value is made and passed.
- When the parameter is an *object* its value is a reference to the object, an address.
- The object reference is copied and passed.

C# Parameter passing is by Value

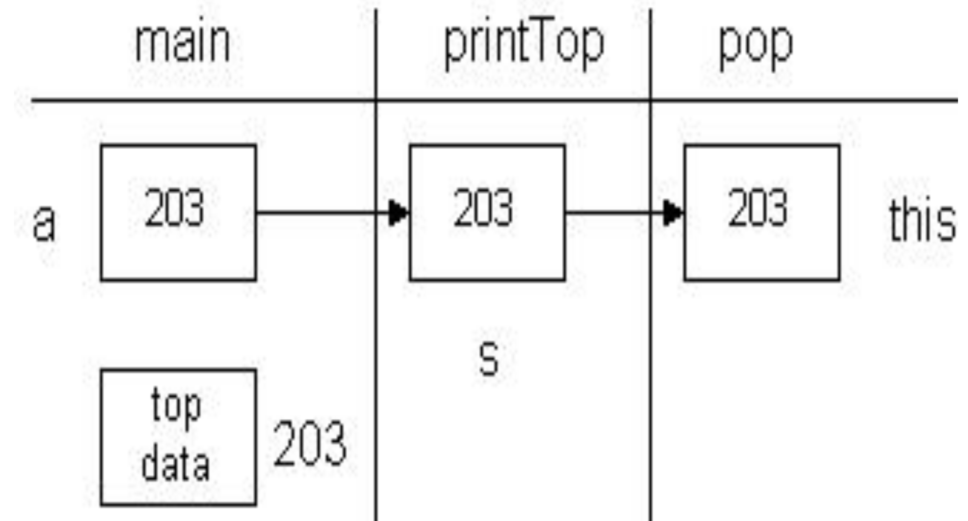
```
a.push(-4);
```

passes **-4** by value

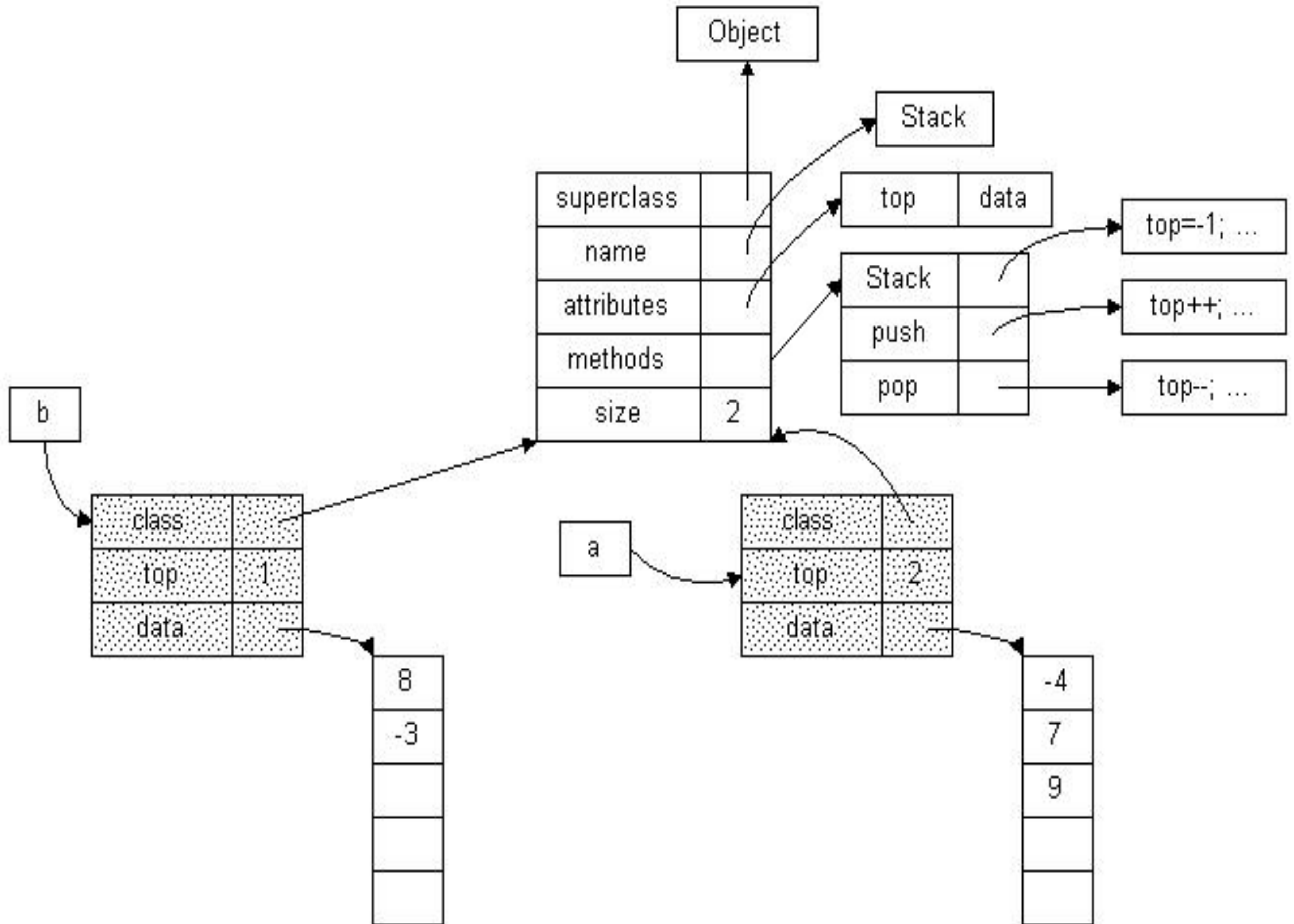



```
printTop( a );
void printTop( Stack s) {
  System.out.println( s.pop( ) );
}
```

passes **a** by value



Object construction & representation



Tracing method execution

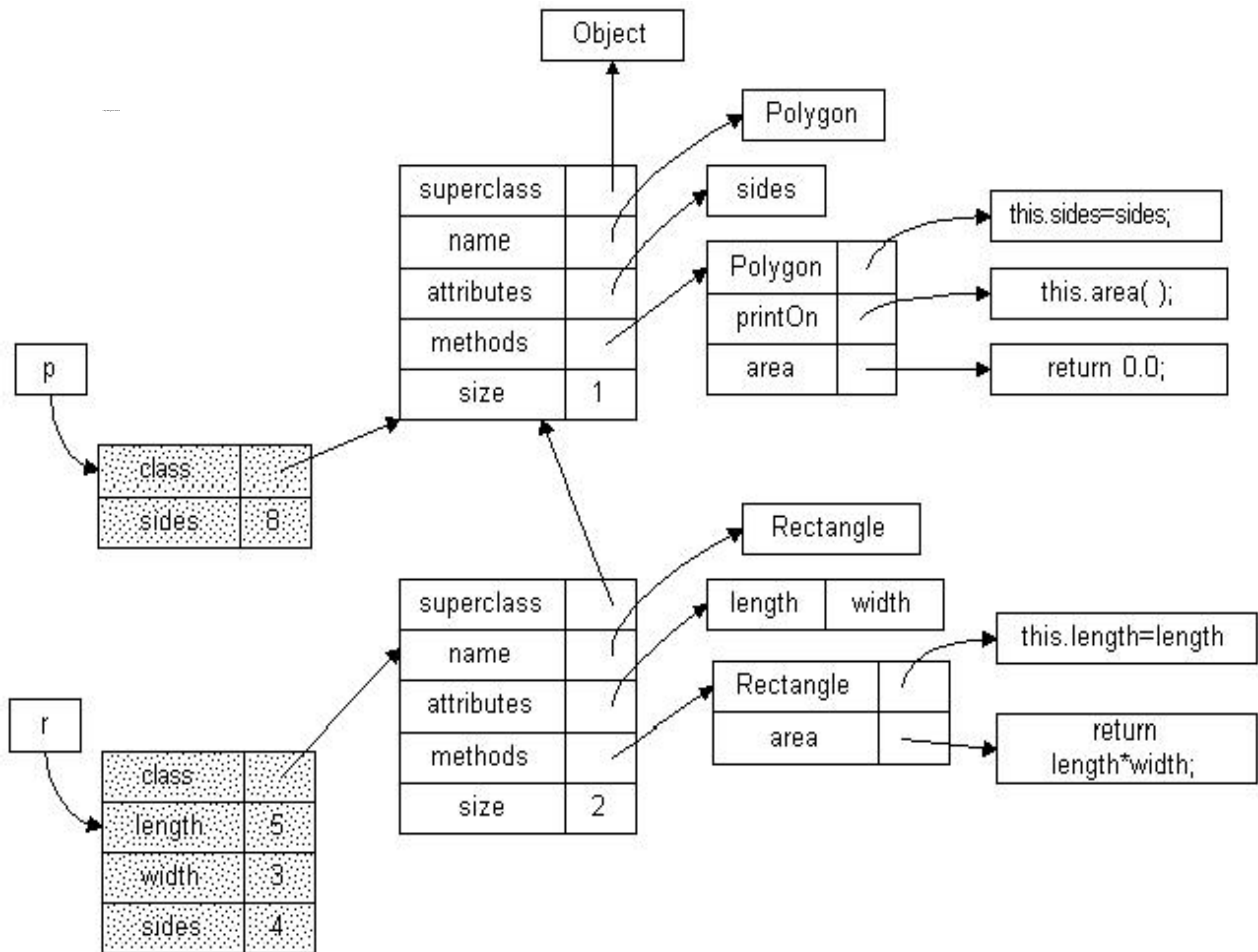
Start: `a.push(-4);`

1. follow reference through *a* to object,
2. follow reference through *class* attribute of object to *class definition* for *Stack,*
3. follow reference through *methods* attribute of class definition to *push* method,
4. follow reference through *push* attribute of *methods* to executable code for *push* method.
5. The object referenced by *a* is the implicit parameter *this* to the *push* method. The *push* method has full access to the *Stack* object referenced by *a*.

```
public class Shapes {
  public static void Main()
  {  Polygon p = new Polygon(8);
     Rectangle r=new Rectangle(5.0, 3.0);
     r.area( );
     p.printOn( );
     r.printOn( );
   }
}
```

```
class Polygon  {
  private int sides;
  public Polygon(int sides) {
     this.sides = sides;
  }
  public double area( ) { return 0.0; }
  public void printOn( ) {
   System.Console.Write( this.getClass( )
                        + " " + this.area( ) );
  }
}
```

```
class Rectangle : Polygon {
  private double length, width;
  public Rectangle(double length, double width) {
     super(4);
     this.length = length;
     this.width = width;
  }
  public double area( ) { return length * width; }
}
```



Object
Polygon
superclass
name
attributes
methods
size
1
sides
Polygon
printOn
area
this.sides=sides;
this.area( );
return 0.0;
p
class
sides
8
Rectangle
superclass
name
attributes
methods
size
2
length
width
Rectangle
area
this.length=length
return
length*width;
r
class
length
5
width
3
sides
4

More Polymorphism

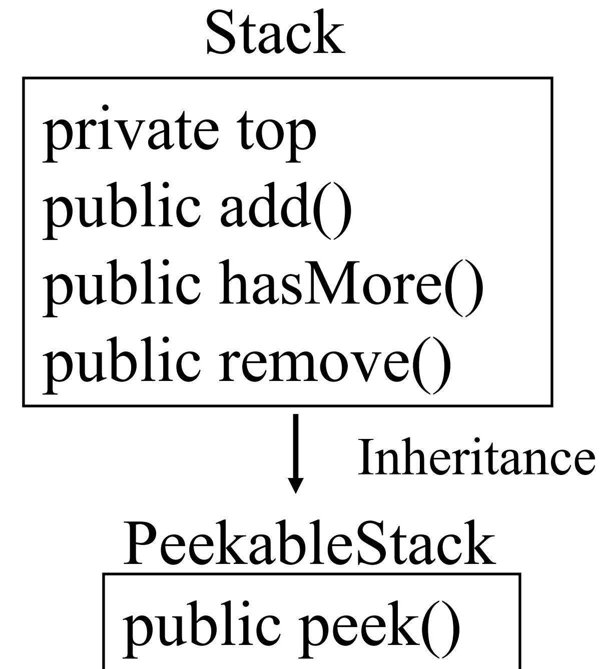
- Class inheritance is source of polymorphism
- One class can be derived from another, using the keyword **`:`**
- For example: a class **`PeekableStack`** that is just like **`Stack`**, but also has a method **`peek`** to examine the top element without removing it

```
/**
 * A PeekableStack is an object that does everything a
 * Stack can do, and can also peek at the top element
 * of the stack without popping it off.
 */
public class PeekableStack : Stack {

  /**
   * Examine the top element on the stack, without
   * popping it off.  This should be called only if
   * the stack is not empty.
   * @return the top String from the stack
   */
  public String peek() {
    String s = remove();
    add(s);
    return s;
  }
}
```

Stack and PeekableStack

```
public class Stack : Worklist {
     private Node top = null;
     public void add(String data) {
       top = new Node(data,top);
     }
     public boolean hasMore() {
       return (top!=null);
     }
     public String remove() {
       Node n = top;
       top = n.getLink();
       return n.getData();
     }
   }
public class PeekableStack : Stack {
    public String peek() {
       String s = remove();
       add(s);
       return s;
     }
   }
```



Inheritance

- Because **`PeekableStack`** extends **`Stack`**, it inherits all its methods and fields
- (Nothing like this happens with interfaces—when a class implements an interface, all it gets is an obligation to implement methods)
- Through inheritance and interfaces, polymorphism

```
Stack s1 = new PeekableStack();
PeekableStack s2 = new PeekableStack();
s1.add("drive");
s2.add("cart");
System.Console.WriteLine(s2.peek());
```

- Note that **`s1.peek()`** is not legal here, even though **`s1`** is a reference to a **`PeekableStack`**.

- It is the static type of the reference, not the object's class, that determines the operations C# will permit.

Question

- Our **peek** was inefficient:

```
public String peek() {
  String s = remove();
  add(s);
  return s;
}
```

- Why not just do this?

```
public String peek() {
  return top.getData();
}
```

Answer

- The **`top`** field of **`Stack`** is **`private`**
- **`PeekableStack`** cannot access it
- For more efficient **`peek`**, **`Stack`** must make **`top`** visible in classes that extend it
- **`protected`** instead of **`private`**
- A common design challenge for object-oriented languages: designing for reuse by inheritance

Inheritance Chains

- A derived class can have more classes derived from it
- All classes but one are derived from some class
- If you do not give an **`inheritance`** clause, C# supplies one: **`: Object`**
- **`Object`** is the ultimate base class in C#

The Class **`Object`**

- All classes are derived, directly or indirectly, from the predefined class **`Object`** (except **`Object`** itself)
- All classes inherit methods from **`Object`**:
  - **`ToString`**, for converting to a **`String`**
  - **`equals`**, for comparing with other objects
  - **`hashcode`**, for computing an **`int`** hash code
  - etc.

Overriding Inherited Definitions

- Sometimes you want to redefine an inherited method
- No special construct for this: a new method definition automatically overrides an inherited definition of the same name and type

Overriding Example

```
System.Console.Write(new Stack());
```

- The inherited **`ToString`** just combines the class name and hash code (in hexadecimal)
- So the code above prints something like:
  ```
  Stack@b3d
  ```
- A custom **`ToString`** method in **`Stack`** can override this with a nicer string:

```
public override String ToString() {
  return "Stack with top at " + top;
}
```

Inheritance Hierarchies

- Inheritance forms a hierarchy, a tree rooted at `Object`
- Sometimes inheritance is one useful class extending another
- In other cases, it is a way of factoring out common code from different classes into a shared base class

```java
public class Label {
  private int x,y;
  private int width;
  private int height;
  private String text;
  public void move
      (int newX, int newY)
  {
    x = newX;
    y = newY;
  }
  public String getText()
  {
    return text;
  }
}
```

```java
public class Icon {
  private int x,y;
  private int width;
  private int height;
  private Gif image;
  public void move
      (int newX, int newY)
  {
    x = newX;
    y = newY;
  }
  public Gif getImage()
  {
    return image;
  }
}
```

Two classes with a lot in common—but neither is a simple extension of the other.

```
public class Graphic {
  protected int x,y;
  protected int width,height;
  public void move(int newX, int newY) {
    x = newX;
    y = newY;
  }
}
```

```
public class Label
    : Graphic {
  private String text;
  public String getText()
  {
    return text;
  }
}
```

```
public class Icon
    : Graphic {
  private Gif image;
  public Gif getImage()
  {
    return image;
  }
}
```

Common code and data factored out to a common base class.

A Design Problem

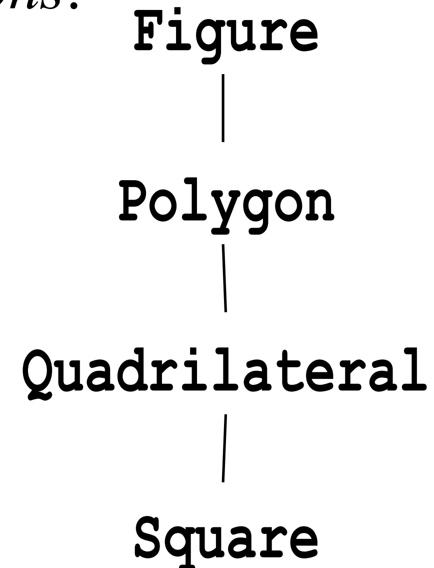
- When you write the same statements repeatedly, you think: that should be a method
- When you write the same methods repeatedly, you think: that should be a common base class
- The real trick is to see the need for a shared base class early in the design, before writing a lot of code that needs to be reorganized

Subtypes and Inheritance

- A derived class is a subtype
- From Chapter Six:

  A subtype is a subset of the values, but it can support a superset of the operations.

- When designing class hierarchies, think about inheritance of functionality
- Not all intuitively reasonable hierarchies work well for inheriting functionality



Outline

- 15.2 Implementing interfaces
- 15.3 Extending classes
- 15.4 Extending and implementing
- 15.5 Multiple inheritance
- 15.6 Generics

Extending And Implementing

- Classes can use **`inheritance`** and **`interface`** together
- For every class, the C# language system keeps track of several properties, including:

A: the interfaces it implements
B: the methods it is obliged to define
C: the methods that are defined for it
D: the fields that are defined for it

Simple Cases For A Class

- A method definition affects C only
- A field definition affects D only
- An **`implements`** part affects A and B
  - All the interfaces are added to A
  - All the methods in them are added to B

A: the interfaces it implements
B: the methods it is obliged to define
C: the methods that are defined for it
D: the fields that are defined for it

Tricky Case For A Class

- An `extends` part affects all four:
  - All interfaces of the base class are added to A
  - All methods the base class is obliged to define are added to B
  - All methods of the base class are added to C
  - All fields of the base class are added to D

A: the interfaces it implements
B: the methods it is obliged to define
C: the methods that are defined for it
D: the fields that are defined for it

Previous Example

```
public class Stack : Worklist {…}

public class PeekableStack : Stack {…}
```

- **`PeekableStack`** has:
  - A: **`Worklist`** interface, inherited
  - B: obligations for **`add`**, **`hasMore`**, and **`remove`**, inherited
  - C: methods **`add`**, **`hasMore`**, and **`remove`**, inherited, plus its own method **`peek`**
  - D: field **`top`**, inherited

A Peek At **`abstract`**

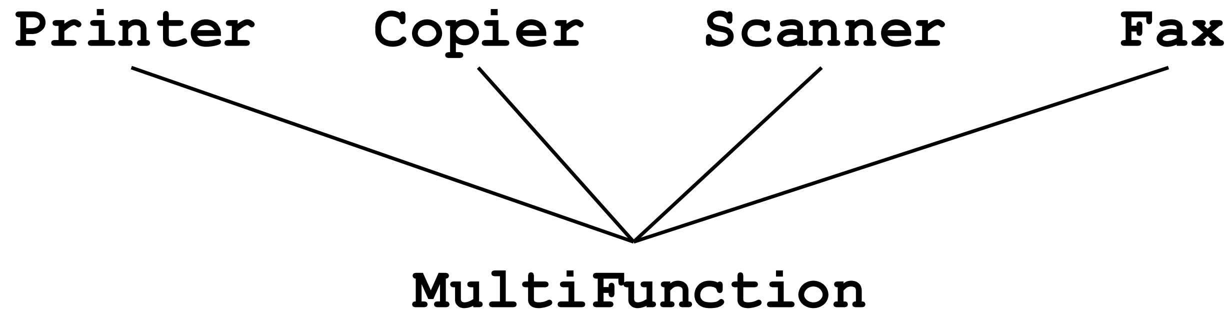
- Note that C is a superset of B: the class has definitions of all required methods
- C# ordinarily requires this
- Classes can get out of this by being declared **`abstract`**
- An **`abstract`** class is used only as a base class; no objects of that class are created

Outline

- 15.2 Implementing interfaces
- 15.3 Extending classes
- 15.4 Extending and implementing
- 15.5 Multiple inheritance
- 15.6 Generics

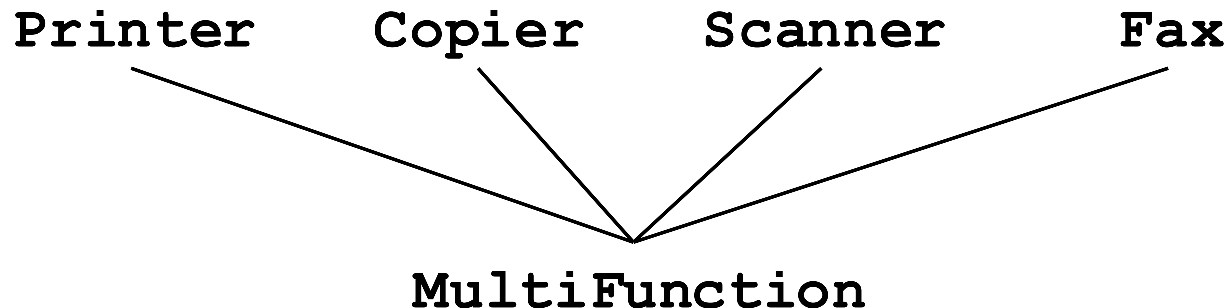
Multiple Inheritance

- In some languages (such as C++) a class can have more than one base class
- Seems simple at first: just inherit fields and methods from all the base classes
- For example: a multifunction printer



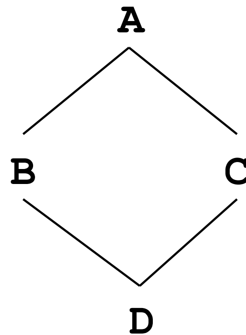
Collision Problem

- The different base classes are unrelated, and may not have been designed to be combined
- **`Scanner`** and **`Fax`** might both have a method named **`transmit`**
- When **`MultiFunction.transmit`** is called, what should happen?



Diamond Problem

- A class may inherit from the same base class through more than one path



- If **A** defines a field **x**, then **B** has one and so does **C**
- Does **D** get two of them?

Solvable, But…

- A language that supports multiple inheritance must have mechanisms for handling these problems
- Not all that tricky
- The question is, is the additional power worth the additional language complexity?
- C#' s designers did not think so

Living Without Multiple Inheritance

- One benefit of multiple inheritance is that a class can have several unrelated types (like **`Copier`** and **`Fax`**)
- This can be done in C# by using interfaces: a class can implement any number of interfaces
- Another benefit is inheriting implementation from multiple base classes
- This is harder to accomplish with C#

Forwarding

```
public class MultiFunction {
  private Printer myPrinter;
  private Copier myCopier;
  private Scanner myScanner;
  private Fax myFax;

  public void copy() {
    myCopier.copy();
  }
  public void transmitScanned() {
    myScanner.transmit();
  }
  public void sendFax() {
    myFax.transmit();
  }
  …
}
```

Outline

- 15.1 Implementing interfaces
- 15.2 Extending classes
- 15.3 Extending and implementing
- 15.4 Multiple inheritance
- 15.5 Generics

Generic Classes/Interfaces

- Previous **`Stack`** example: a stack of strings
- Can be reused for stacks of other types
- In F# we used type variables for this:

```
type mylist =
  | NIL
  | CONS of 'a * mylist
```

- C#, Ada and C++ have something similar

Living Without Generics

- We can make a stack whose element type is **`Object`**
- The type **`Object`** includes all references, so this will allow any objects to be placed in the stack

```
public class GenericNode {
  private Object data;
  private GenericNode link;
  public GenericNode(Object theData,
       GenericNode theLink) {
    data = theData;
    link = theLink;
  }
  public Object getData() {
    return data;
  }
  public GenericNode getLink() {
    return link;
  }
}
```

Similarly, we could define **`GenericStack`** (and a **`GenericWorklist`** interface) using **`Object`** in place of **`String`**

Weaknesses

- To recover the type of the stacked object, we will have to use an explicit *type cast:*

```
GenericStack s1 = new GenericStack();
s1.add("hello");
String s = (String) s1.remove();
```

- This is a pain to write, and also inefficient
- C# checks at runtime that the type cast is legal—the object really is a **`String`**

Weaknesses

- Primitive types must first be stored in an object before being stacked:

```
GenericStack s2 = new GenericStack();
s2.add(new Int(1));
int i = (Int) s2.remove().N;
```

- Again, laborious and inefficient
- **`Int`** is a defined wrapper class

True Generics

- Generics in C#: parameterized polymorphic classes (and interfaces)
- Uses a notation like C++ templates

```
public class Stack<T> : Worklist<T> {
  private Node<T> top = null;
  public void add(T data) {
    top = new Node<T>(data,top);
  }
  public boolean hasMore() {
    return (top!=null);
  }
  public T remove() {
    Node<T> n = top;
    top = n.getLink();
    return n.getData();
  }
}
```

Using Generic Classes

```
Stack<String> s1 = new Stack<String>();
Stack<int> s2 = new Stack<int>();
s1.add("hello");
String s = s1.remove();
s2.add(1);
int i = s2.remove();
```

Generic Stack

```
public class SimpleStack  {
   public static void Main()  {
      Stack<double> a = new Stack<double>();
      a.push(-4.3);
      a.push(9.7);
      System.Console.Write(a.pop());

      Stack<String> b = new Stack<String>();
      b.push("Hello");
      System.Console.Write(b.pop());
   }
}

class Stack<T> {
   private T[] data = new T[5];
   private int top = -1;
   public void push(T t) { data[++top] = t; }
   public T pop() { return data[top--]; }
}
```