C# Polymorphism

Subtype Polymorphism

Example

```
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 related classes
- C# has subtype polymorphism

Interfaces

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

Example

```
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

Example

Example (Interface)

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

Example (Implementation)

```
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*
- Interface name can be used as a reference type
- Provides a form of *multiple* inheritance

Example

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

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

Polymorphism With Interfaces

Example

```
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

Example

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

Interface Example (pt. 1/2)

Example

```
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());
  }
}
```

Interface Example (pt. 2/2)

Example

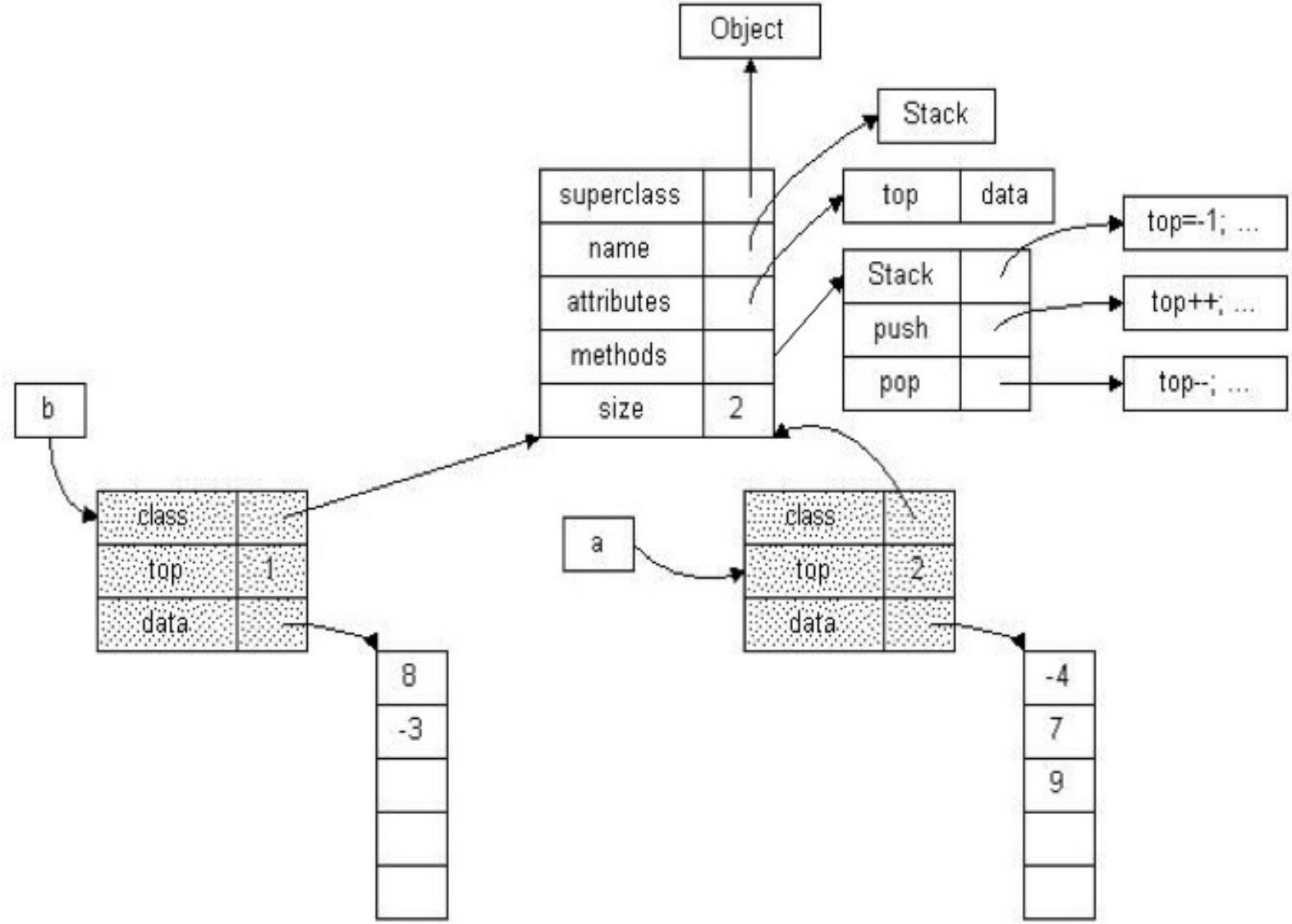
```
class ListStack : Stack {
  List <Object> data = new List<Object>();

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

Extending Classes

Object Representation



Tracing method execution

```
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.
6. The **push** method has full access to the **Stack** object referenced by **a**.

Shapes Example

Example

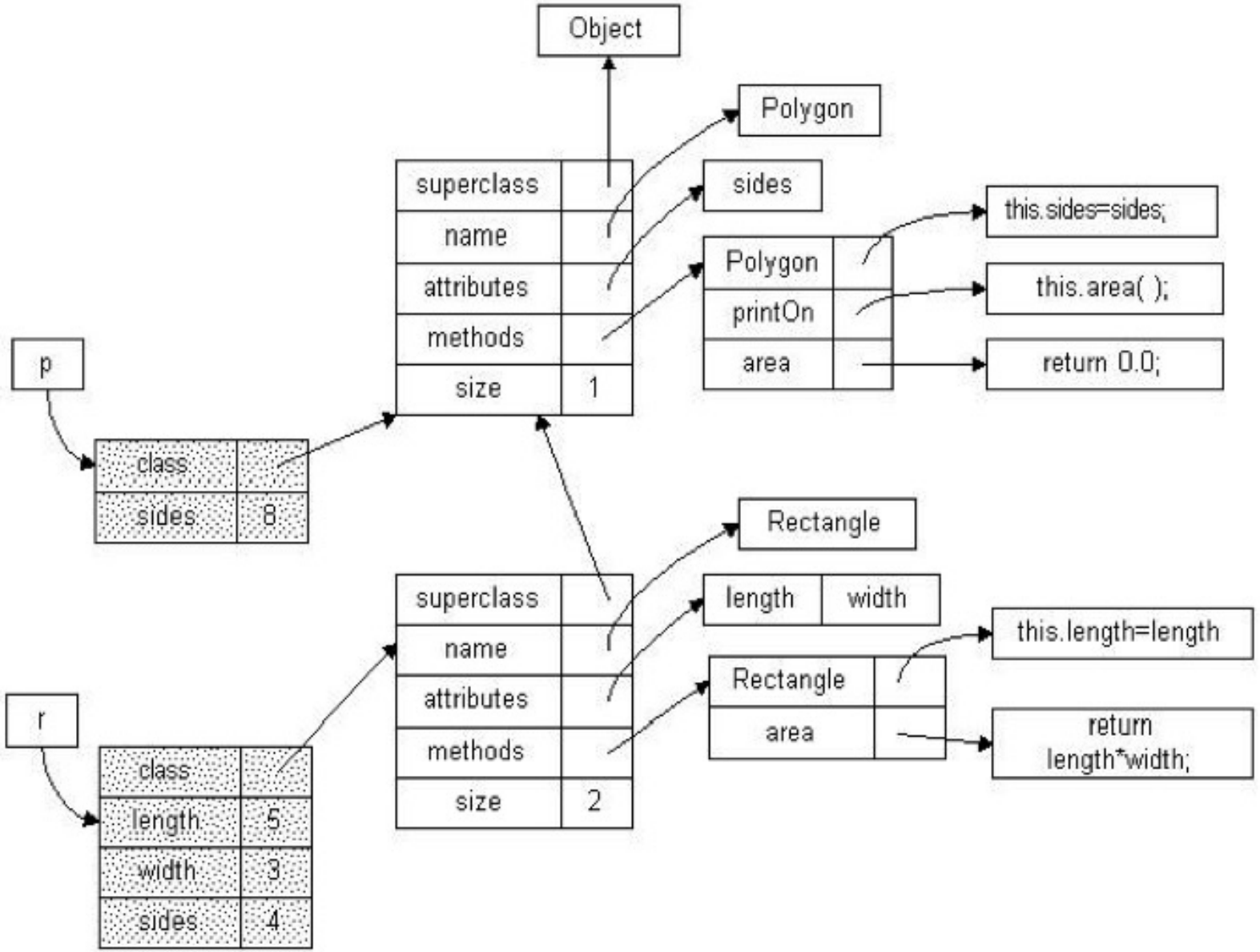
```
class Polygon  {
  private int sides;
  public Polygon(int sides) {
    this.sides = sides;
  }
  public double area( ) { return 0.0; }
  public void printOn( ) {
    System.Console.Write( this.GetType( ) + " " + this.area( ) );
  }
}

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

Shapes Example

Example

```
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( );
  }
}
```

Object Representation



More Polymorphism

- Class inheritance is source of polymorphism
- One class can be derived from another, using the colon character (:)
- For example: a class **PeekableStack** that is just like **Stack**, but also has a **peek** method

Example

```
public class PeekableStack : Stack {
  /**
   * Examine the top element on the stack, without popping it.
   * @return the top string from the 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

PeekableStack

- 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.

Example

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

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 explicit inheritance class, C# supplies a default one: **: Object**
- **Object** is the ultimate base class in C#
- A class can make a field visible only in classes that extend it using the **protected** keyword, instead of **private**

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

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

- 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

Example

```
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

Example without Refactoring

Example

```
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; }
}
```

Example

```
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; }
}
```

Example with Refactoring

Example (Base class)

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

Example

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

Example

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

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 trick is to anticipate the need for a shared base class early in your code design, before writing a lot of code that needs to be reorganized

Subtypes and Inheritance

- A derived class is a subtype
- When designing class hierarchies, think about inheritance of functionality
- Not all intuitively reasonable hierarchies work well for inheriting functionality

Extending and Implementing

Extending And Implementing

- Classes can use inheritance and interface together
- For every class, the C# language system keeps track of several properties, including:
  1. the interfaces it implements
  2. the methods it is obliged to define
  3. the methods that are defined for it
  4. the fields that are defined for it

Simple Case For a Class

- A method definition affects 3 only
- A field definition affects 4 only
- An **implements** part affects 1 and 2
- All the interfaces are added to 1
- All the methods in them are added to 2
  1. the interfaces it implements
  2. the methods it is obliged to define
  3. the methods that are defined for it
  4. 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 1
  - All methods the base class is obliged to define are added to 2
  - All methods of the base class are added to 3
  - All fields of the base class are added to 4

1. the interfaces it implements
2. the methods it is obliged to define
3. the methods that are defined for it
4. the fields that are defined for it

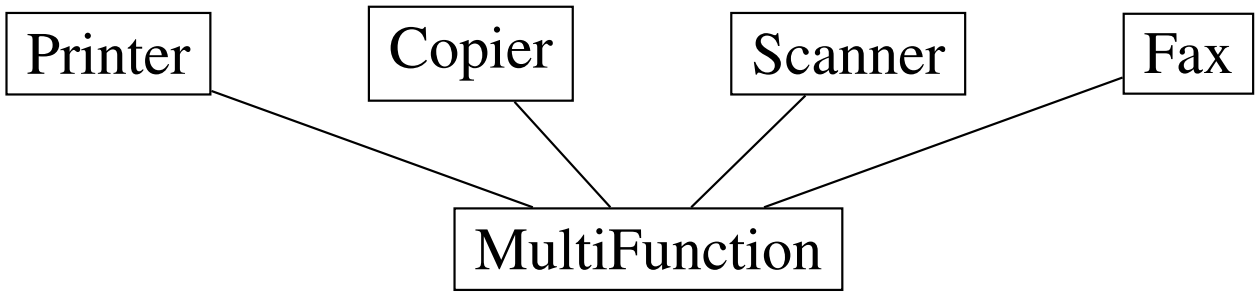
abstract

- Note that 3 is a superset of 2: 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

Multiple Inheritance

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

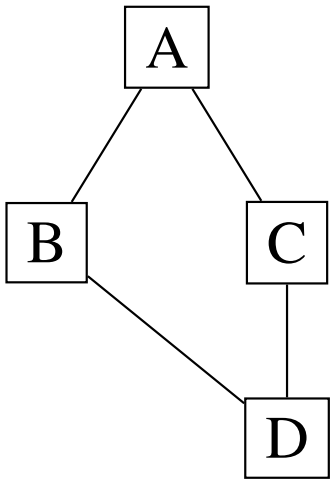


Collision Problem

- 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

Example

```
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();
  }
  // class code here
}
```

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

Example

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

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

Example

```
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;
  }
}
```

Weaknesses

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

Example

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

Weaknesses

- Primitive types must first be stored in an object before being stacked
- Again, laborious and inefficient
- **Int** is a defined wrapper class

Example

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

True Generics

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

Example

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

Using Generic Classes

Example

```
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();
```

Example

```
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--]; }
}
```