Liskov Substitution Principle

Principles

Produced by

Eamonn de Leastar

edeleastar@wit.ie

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

SOLID Principles

- *The **S**ingle Responsibility Principle*
  - A class should have one, and only one, reason to change.
- *The **O**pen Closed Principle*
  - You should be able to extend a classes behavior, without modifying it.
- *The **L**iskov Substitution Principle*
  - Derived classes must be substitutable for their base classes.
- *The **I**nterface Segregation Principle*
  - Make fine grained interfaces that are client specific.
- *The **D**ependency Inversion Principle*
  - Depend on abstractions, not on concretions.

Source Material (2)

Initial	Stands for (acronym)	Concept
S	SRP	**Single responsibility principle** the notion that an object should have only a single responsibility.
O	OCP	**Open/closed principle** the notion that "software entities … should be open for extension, but closed for modification".
L	LSP	**Liskov substitution principle** the notion that "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program". See also design by contract.
I	ISP	**Interface segregation principle** the notion that "many client specific interfaces are better than one general purpose interface."[5]
D	DIP	**Dependency inversion principle** the notion that one should "Depend upon Abstractions. Do not depend upon concretions."[5] Dependency injection is one method of following this principle.

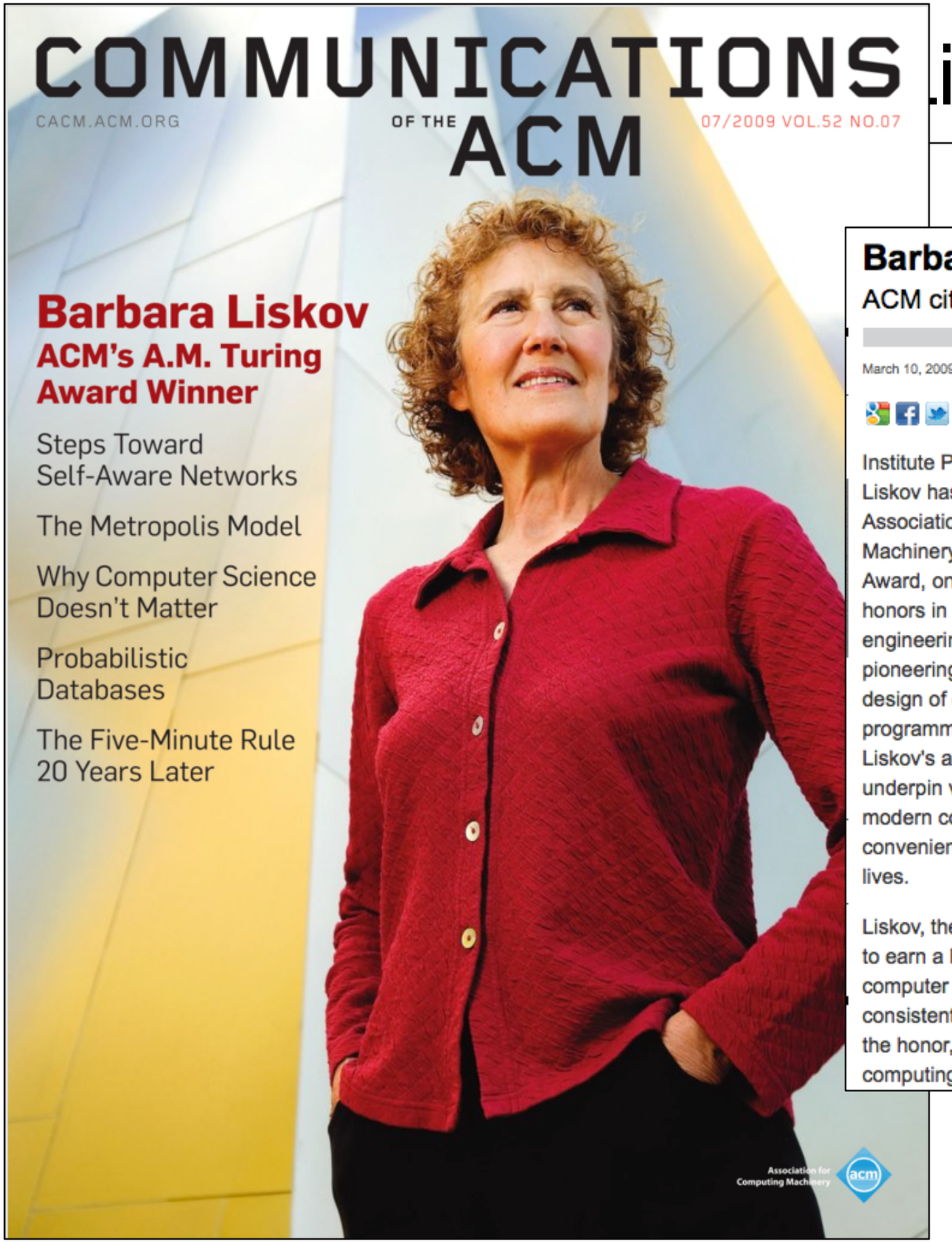
http://en.wikipedia.org/wiki/Solid_(object-oriented_design)

LSP

- Methods that use references to base class types must be able to use objects or derived types without knowing it

> *What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.*
>
> Barbara Liskov, "Data Abstraction and Hierarchy," *SIGPLAN Notices,* 23,5 (May, 1988).

Liskov



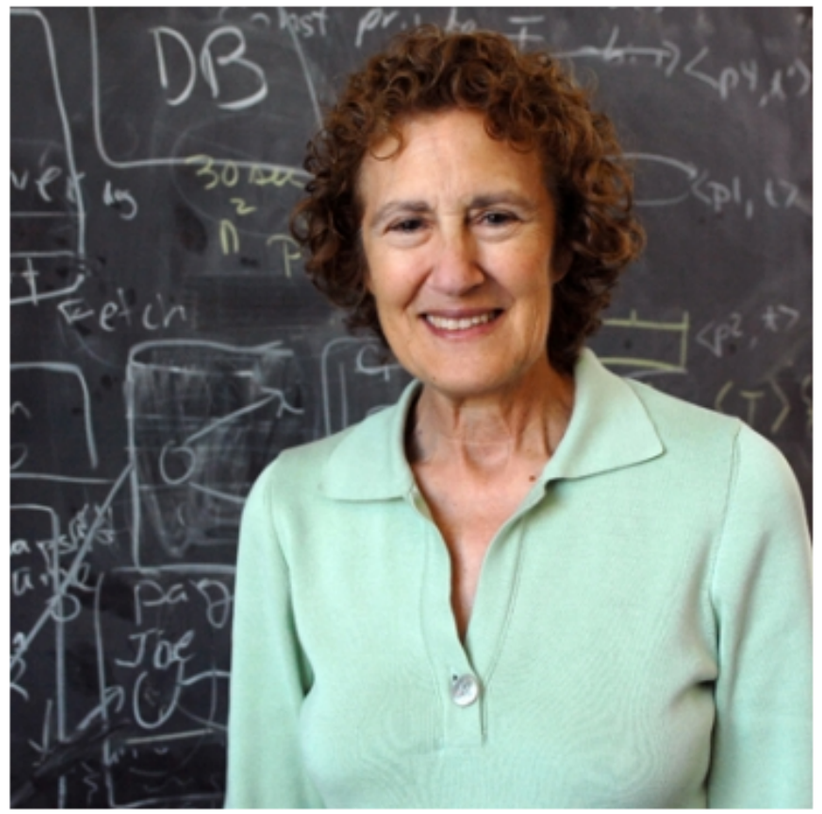
Barbara Liskov wins Turing Award

ACM cites 'foundational innovations' in programming language design

March 10, 2009

Share

Institute Professor Barbara Liskov has won the Association for Computing Machinery's A.M. Turing Award, one of the highest honors in science and engineering, for her pioneering work in the design of computer programming languages. Liskov's achievements underpin virtually every modern computing-related convenience in people's daily lives.



Barbara Liskov
Photo / Donna Coveney

Liskov, the first U.S. woman to earn a PhD from a computer science department, was recognized for helping make software more reliable, consistent and resistant to errors and hacking. She is only the second woman to receive the honor, which carries a $250,000 purse and is often described as the "Nobel Prize in computing."

Simple Violation of LSP

- drawShapes() references a base type shape
- It violates LSP because it must know of every derived type of Shape

```
void drawShapes (Shape shape)
{
  if (shape instanceof Square)
  {
    drawSquare ((Square)shape);
  }
  else if (shape instance of Circle)
  {
    drawCircle ((Circle) shape);
  }
}
```

- It must be modified whenever new derivatives of Shape are presented.

Adhering to LSP

```
class Shape
{
  void draw()
  {//…}
}

class Circle extends Shape
{
  private double itsRadius;
  private Point  itsCenter;
  public void draw()
  { //… }
}

class Square extends Shape
{
  private double itsSide;
  private Point itsTopLeft;
  public void draw()
  { //… }
}
```

```
void drawShape (Shape s)
{
  s.draw();
}
```

- drawShape now adheres to LSP

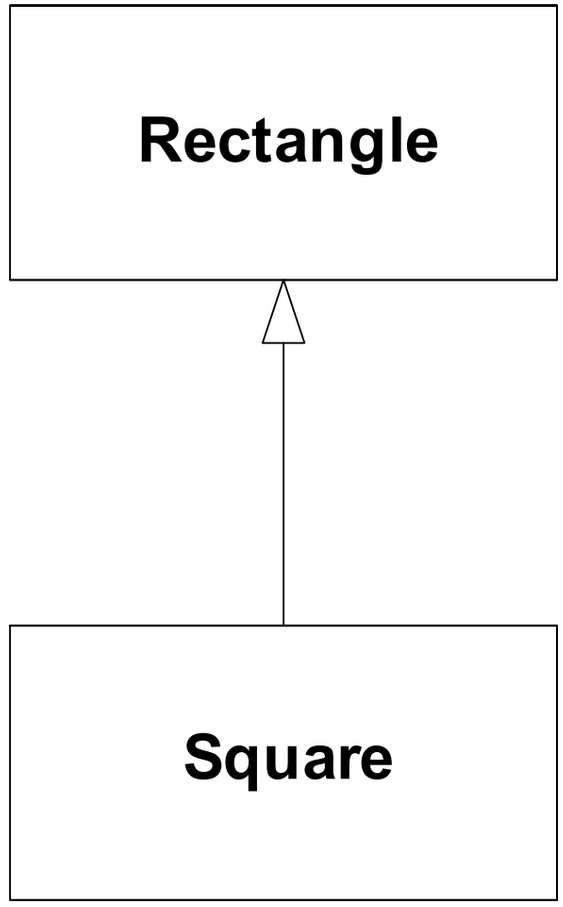
Rectangle

```
class Rectangle
{
  private int width;
  private int height;

  public void setWidth (int width)
  {...}
  public void setHeight (int height)
  {...}
  public int getWidth ()
  {...}
  public int getHeight ()
  {...}
}
```

- Rectangle class is released for general use

Square

- A Square class is required
- Square is introduced as a subclass of Rectangle
- At one level, this use of inheritance can be considered appropriate:
  - A Square is a rectangle whose width and height are equal
- However, both width & height not needed (just one).
- Potential inefficiency if many rectangles created (e.g. CAD application)



Rectangle Width & Height

- Both setWidth() and setHeight() should not vary independently
- Client could easily call one and not the other – thus compromising the Rectangle
- Potential solution is to implement setWidth() and setHeight() in Square class
- These methods then make sure width & height are adjusted

```
class Square extends Rectangle
{
  public void setWidth (int width)
  {
    super.setWidth(width);
    super.setHeight(width);
  }
  public void setHeight (int height)
  {
    super.setWidth(height);
    super.setHeight(height);
  }
}
```

Polymorphism

```
void f (Rectangle r)
{
  r.setWidth(5);
}
```

- Polymorphism ensures that:
  - If the f() method is passed a Rectangle, then its width will be adjusted
  - If passed a Square, then both height and width will be changed
- Assume model is consistent & correct
- However….

More Subtle Problem

```
void g (Rectangle r)
{
  r.setWidth(5);
  r.setHeight(4);
  assert (r.getWidth() * r.getHeight()) == 20;
}
```

- If r is a Rectangle instance
  - g() methods works as expected
- If r is a Square
  - g() assertion is triggered
- g() assumes that width and height of a Rectangle can be varied independently
- Substitution of a Square violates this assumption
- Square violates LSP

Validating the Model

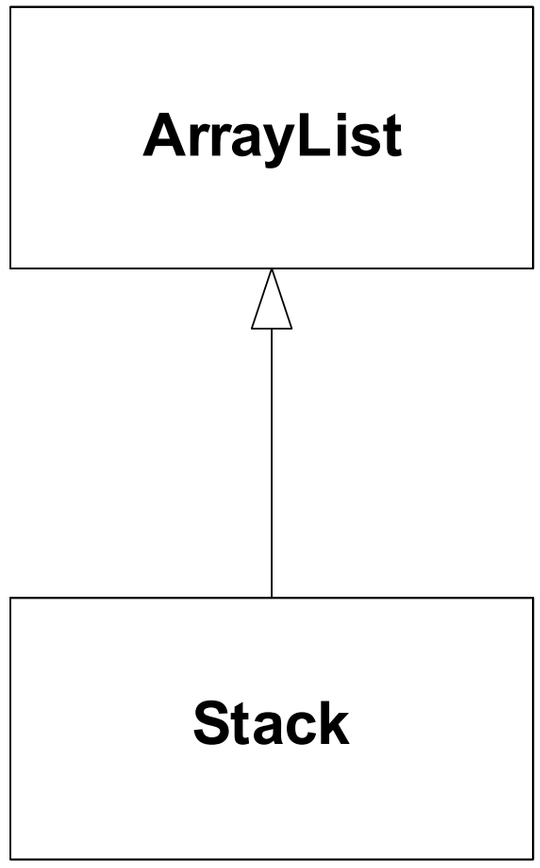
- A model, viewed in isolation, cannot be meaningfully validated
- The validity of a model can only be expressed in terms of its clients:
  - Examining the final version of the Square and Rectangle classes in isolation, we found that they were self consistent and valid.
  - When we examined from the viewpoint of g() (which made reasonable assumptions) the model broke down.
- Thus, when considering whether a design is appropriate or not, it must must examined in terms of the reasonable assumptions that will be made by the users of that design

Behavioural Problems

- A square might be a rectangle, but a Square object is *not* a Rectangle object.
  - the *behaviour* of a Square object is not consistent with the behaviour of a Rectangle object.
- The LSP makes clear that inheritance relationship pertains to *behaviour*
- Not intrinsic private behaviour, but extrinsic public behaviour; behaviour that clients depend upon.
- g() depended on the fact that Rectangles behave such that their height and width vary independently of one another.
- That independence of the two variables is an extrinsic public behaviour that other methods are also likely to depend upon.

Fragile Base Class Problem

```
class Stack extends ArrayList
{
  private int stack_pointer = 0;

  public void push( Object article )
  {
    add( stack_pointer++, article );
  }

  public Object pop()
  {
    return remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < articles.length; ++i )
      push( articles[i] );
  }
}
```

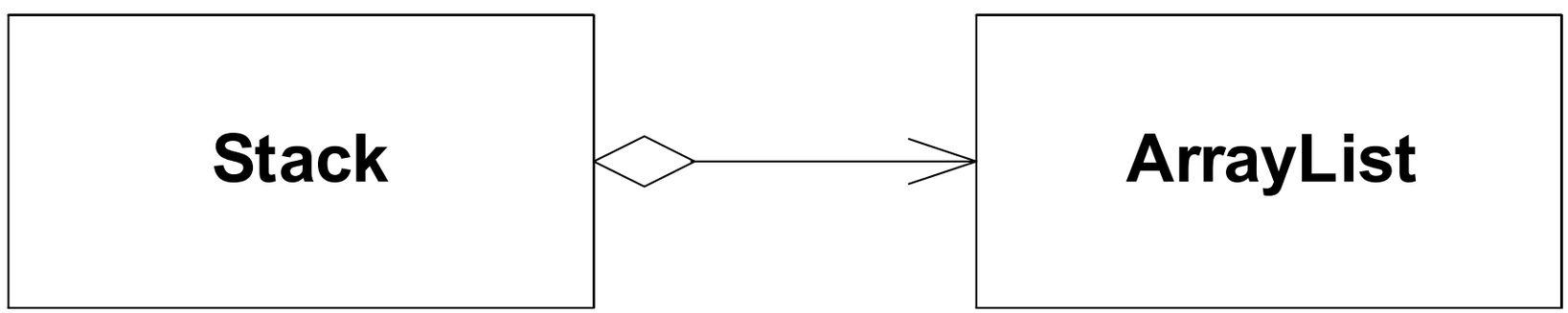


Clearing the Stack

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

- This code and uses the ArrayList's clear() method to pop everything off the stack
- The code successfully executes, but since the base class doesn't know anything about the stack pointer, the Stack object is now in an undefined state.
- The next call to push() puts the new item at index 2 (the stack_pointer's current value), so the stack effectively has three elements on it—the bottom two are garbage.

Use Composition instead of Inheritance



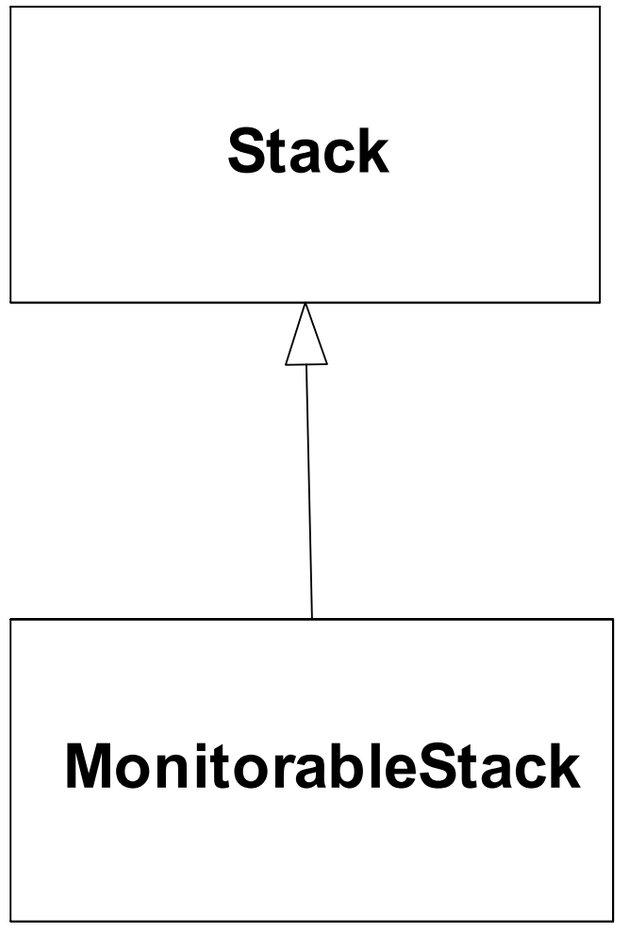
Composed Solution

```
class Stack
{
  private int stack_pointer = 0;
  private ArrayList the_data = new ArrayList();

  public void push( Object article )
  {
    the_data.add( stack_pointer++, article );
  }

  public Object pop()
  {
    return the_data.remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < o.length; ++i )
      push( articles[i] );
  }
}
```

Monitorable Stack

```
class Monitorable_stack extends Stack
{
  private int high_water_mark = 0;
  private int current_size;

  public void push( Object article )
  {
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    super.push(article);
  }
  public Object pop()
  {
    --current_size;
    return super.pop();
  }
  public int maximum_size_so_far()
  {
    return high_water_mark;
  }
}
```

- Tracks the maximum stack size over a certain time period.



push_many Implementation

```
void f(Stack s)
{
  //...
  s.push_many (someObjectArray);
  //...
}
```

- Which class implements push_many nethod?
- If f() is passed a MonitorableStack, does a call to push_many update high_water_mark?
- Polymorphism ensures that MonitrableStack's push method is called, and hande high_water_mark is appropriately updated.
- This is because Stack.push_many() calls the push() method, which is overridden by MonitorableStack

Revised Stack

- A profiler is run against an implementation using Stack
- It notices the Stack isn't as fast as it could be and is heavily used.
- Stack is rewritten so it doesn't use an ArrayList and consequently it gains a performance boost…

Revised Stack using Arrays

```
class Stack
{
  private int stack_pointer = -1;
  private Object[] stack = new Object[1000];

  public void push( Object article )
  {
    assert stack_pointer < stack.length;
    stack[ ++stack_pointer ] = article;
  }
  public Object pop()
  {
    assert stack_pointer >= 0;
    return stack[ stack_pointer-- ];
  }
  public void push_many( Object[] articles )
  {
    assert (stack_pointer + articles.length) < stack.length;
    System.arraycopy(articles, 0, stack, stack_pointer+1,
                                          articles.length);
    stack_pointer += articles.length;
  }
}
```

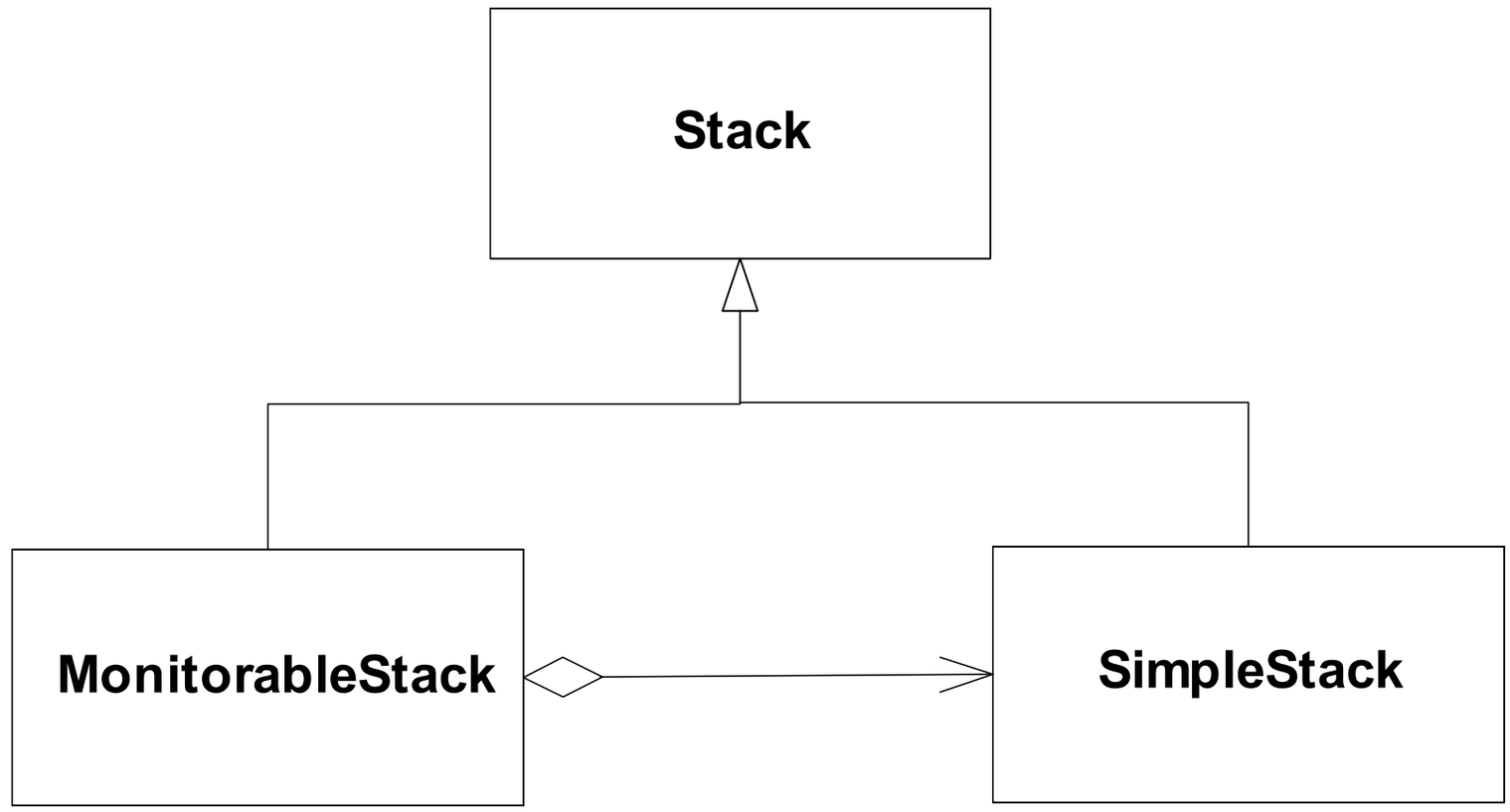
Problems?

```
void f(Stack s)
{
  //...
  s.push_many (someObjectArray);
  //...
}
```

- If **s** is a MonitorableStack, is high_water_mark updated?
- No – because the new Stack base class push_many() implementation does not call push() at all
- LSP Violation: i.e. function f() will not appropriately operate a Stack derived object

Solution

```
interface Stack
{
  void push( Object o );
  Object pop();
  void push_many( Object[] source );
}
```



Simple_Stack

```
class Simple_Stack implements Stack
{
  private int stack_pointer = -1;
  private Object[] stack = new Object[1000];

  public void push( Object article )
  {
    assert stack_pointer < stack.length;
    stack[ ++stack_pointer ] = article;
  }
  public Object pop()
  {
    assert stack_pointer >= 0;
    return stack[ stack_pointer-- ];
  }
  public void push_many( Object[] articles )
  {
    assert (stack_pointer + articles.length) < stack.length;
    System.arraycopy(articles, 0, stack, stack_pointer+1,
                                          articles.length);
    stack_pointer += articles.length;
  }
}
```

```
class Monitorable_Stack implements Stack
{
  private int high_water_mark = 0;
  private int current_size;
  Simple_stack stack = new Simple_stack();
  public void push( Object o )
  {
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    stack.push(o);
  }
  public Object pop()
  {
    --current_size;
    return stack.pop();
  }
  public void push_many( Object[] source )
  {
    if( current_size + source.length > high_water_mark )
      high_water_mark = current_size + source.length;
    stack.push_many( source );
  }
  public int maximum_size()
  {
    return high_water_mark;
  }
}
```

Consult Stack API

Constructor Summary

Stack()
Creates an empty Stack.

Method Summary

boolean	**empty**() Tests if this stack is empty.
E	**peek**() Looks at the object at the top of this stack without removing it from the stack.
E	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
E	**push**(E item) Pushes an item onto the top of this stack.
int	**search**(Object o) Returns the 1-based position where an object is on this stack.

Methods inherited from class java.util.Vector

add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, lastIndexOf, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toArray, toString, trimToSize

Stack is Derived from Vector

Overview Package **Class** Use Tree Deprecated Index Help

Java™ 2 Platform Standard Ed. 5.0

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.util

Class Stack<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractList<E>
      └ java.util.Vector<E>
        └ **java.util.Stack<E>**

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Holub's Advice

- In general, it's best to avoid concrete base classes and extends relationships in favour of interfaces and implements relationships.
- Rule of thumb : 80 percent of code at minimum should be written entirely in terms of interfaces.
  - E.g. never use references to a HashMap, use references to the Map
- The more abstraction you add, the greater the flexibility.
- In today's business environment, where requirements regularly change as the program develops, this flexibility is essential.