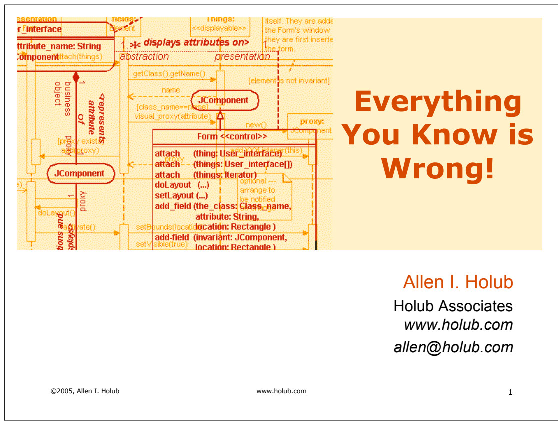
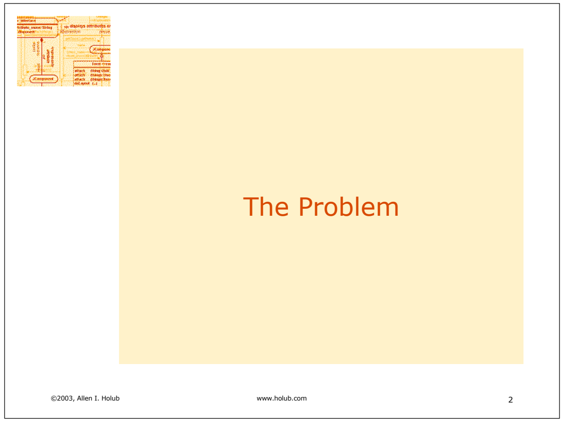
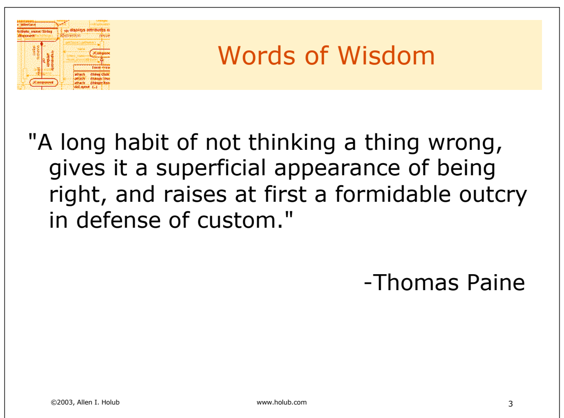






© 2005, Allen I. Holub. All Rights Reserved.

DO NOT DUPLICATE

People don’t know they don’t know

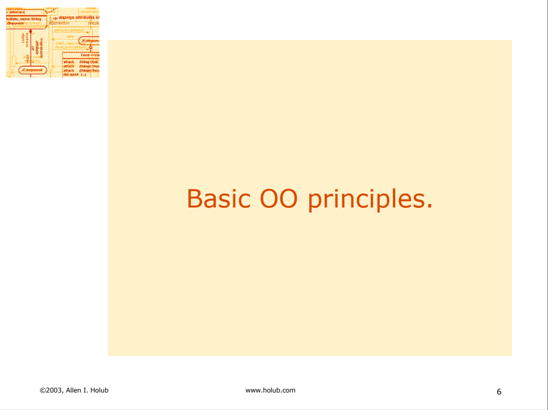
- *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments*
  - » Justin Kruger and David Dunning, Department of Psychology, Cornell University.
  - http://www.apa.org/journals/psp/psp7761121.html
- I’ve gotten death threats when I’ve written about this stuff.

©2003, Allen I. Holub www.holub.com 4

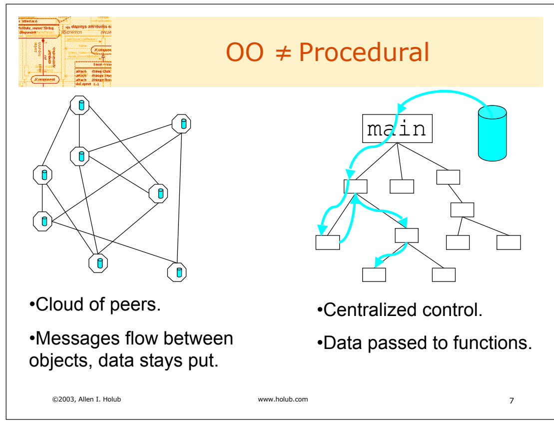
Procedural thinking is everywhere

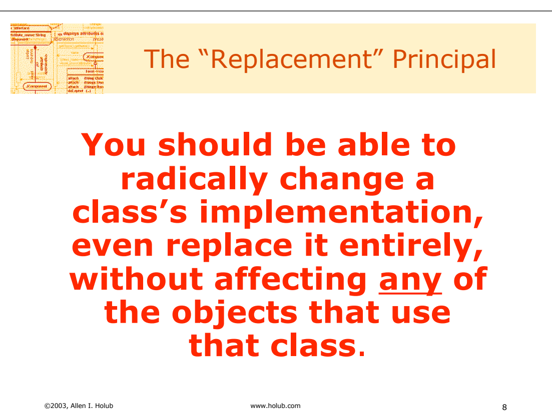
- OO really is a different way of thinking about programming.
- Don’t confuse familiar with “right.”
  - Procedural methods are familiar.
  - Many commonly used libraries (particularly open source [e.g. Struts, JavaBeans]) are fundamentally procedural.
- It takes as long (or longer) to learn design as it does to learn how to program.
  - Programming and design are different disciplines.

©2003, Allen I. Holub www.holub.com 5



© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE





Data abstraction

- The less you know about how objects work, the more maintainable your code.
- The less you know about the actual classes you’re using, the more maintainable your code. (*Abstraction*)
  - Program in terms of an abstraction layer (interfaces), not concrete classes.

© 2005, Allen I. Holub. All Rights Reserved.

DO NOT DUPLICATE

What is an object?

- **Objects are defined by what they *do***, not what they contain.
  - objects ≠ data + functions.
  - objects have responsibilities, not data.
- The way in which the object does the work should be completely hidden (*Encapsulation*).

Ask for help, not for information

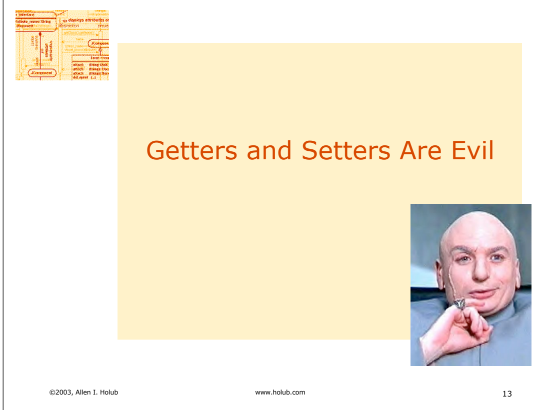
Don't ask an object to give you the data you need to do something — ask the object that has the information to do the work for you. (*Delegation*)

Some Examples

- An Employee doesn't need a `getName()`
  - `exportAsXML("name", Writer out);`
- A String doesn't need a `getBytes()`
  - The String class should support all necessary string operations.
  - `String.print(Writer)`
- An EJB running on a bank's server does not need a `getBalance()`.

```
Boolean IsBalanceGreaterThan(
                    Money requestedFunds);
```


© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE



More on Data Abstraction

- The goal is to be able to change a class's implementation without impacting the code that uses that class.
- Implications of implementation hiding:
  - public fields are bad
    - They expose implementation.
  - Getter/setter (accessor/mutator) methods are bad
    - They are just complicated ways of making a field public.

©2003, Allen I. Holub www.holub.com 14

An example of why accessors are bad

- Consider this class:

```
class Money
{       double value;
        double getValue();
}
```

- What if you need to support multiple currencies?
  - getValue() fails: we don't know the currency.
  - Adding getCurrency() doesn't help
    - All the code that uses getValue() must be modified to use getCurrency().
  - Operations like comparing values represented in different currencies are now complicated, and must be performed all over your code.
  - **Can't be fixed with automated refactoring.**

©2003, Allen I. Holub www.holub.com 15

© 2005, Allen I. Holub. All Rights Reserved. **DO NOT DUPLICATE**

The problem is endemic

- A procedural programmer sees nothing wrong.
  - Many books recommend putting mutators and accessors on all fields!
- JavaBeans introduced the getter/setter "design pattern" because it was "easy."
  - There's a better alternative, called a BeanCustomizer, but nobody uses it.
  - "Metadata" (Java 1.5) is vastly better:

```
@property private int someProperty;
```

- People have blindly copied the idiom without considering the consequences.

©2003, Allen I. Holub

How should it work?

- Don't ask for the information that you need to do the work; ask the object that has the information to do the work for you.

```
class Money
{ private double value;
  public  Money  addTo    ( Money x )   {/*...*/}
  public  int    compare  ( Money x )   {/*...*/}
  public  Money  printTo  ( Writer w )  {/*...*/}
  public  String asXML    ()            {/*...*/}
  public  String toString ()            {/*...*/}
}
```

©2003, Allen I. Holub

Ramifications in the UI

- An object must be responsible for building its own UI.
  - or at least providing generic representations of its attributes.
- (Simplistically) not

```
System.out.println( obj.getAttribute() );
```

but

```
obj.printTo( Writer w );
```

or

```
JComponent c =
    object.getRepresenationOf("attribute");
```

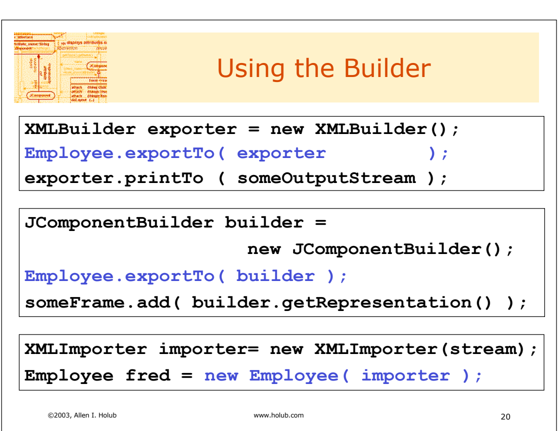
©2003, Allen I. Holub

© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE

A realistic solution

- You can't add a billion `printYourselfAsXXX()` methods for different representations.
- exportAsXML() can work, but is awkward.
- Solve the problem generically with the GoF *Builder* design pattern.
  - An object (the "director") builds a representation to itself by passing information to a builder object, which is passed into the director.
  - Different builders build different products.
  - The director doesn't know what is build.
  - Accessors, if reqired, are part of the builder, not the director, so changes in the director ripple only to the builders.

©2003, Allen I. Holub www.holub.com 19



Builder (1)

```
public class Employee
{   private Name        name;
    private EmployeeId  id;
    private Money       salary;

    public interface Exporter
    {   void addName    ( String name   );
        void addID      ( String id     );
        void addSalary  ( String salary );
    }

    public interface Importer
    {   String provideName();
        String provideID();
        String provideSalary();
        void   open();
        void   close();
    }
```

→

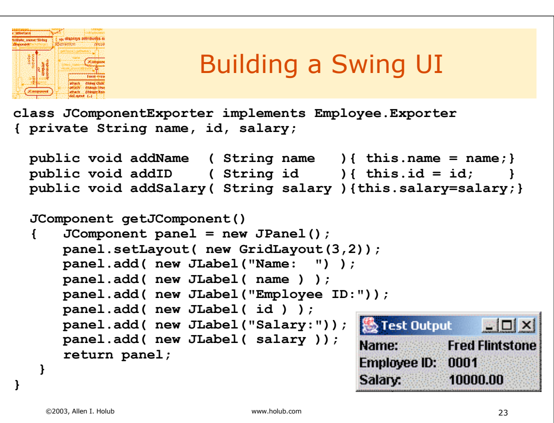
©2003, Allen I. Holub www.holub.com 21

© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE

Builder (2)

```
  public Employee( Importer builder )
  {   builder.open();
      this.name = new Name      (builder.provideName() );
      this.id   = new EmployeeId(builder.provideID()   );
      this.salary = new Money   (builder.provideSalary(),
                                  new Locale("en","US"));
      builder.close();
  }

  public void export( Exporter builder )
  {   builder.addName   ( name.toString()   );
      builder.addID     ( id.toString()     );
      builder.addSalary( salary.toString() );
  }
  //...
}
```

Exporting to HTML

```
HTMLExporter implements Employee.Exporter
{   private final String       HEADER = "<table border=\"0\">\n";
    private final StringBuffer out    = new StringBuffer(HEADER);

    public void addName( String name )
    {   out.append("\t<tr><td>Name:</td><td>" );
        out.append("<input type=\"text\" name=\"name\"value=\"");
        out.append( name );
        out.append( "\"></td></tr>\n" );
    }
    public void addID( String id     ) { /*.. .*/ }
    public void addSalary( String salary ) { /*.. .*/ }
    String getHTML()
    {   out.append("</table>");
        String html = out.toString();
        out.setLength(0);                  // erase the buffer
        out.append(HEADER);
        return html;
    }
}
```

```
HTML Exporter e = new HtmlExporter;
someEmployee.export( e );
someStream.print( e.getHTML() );
```


© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE

Initializing from an HTML form

```
class HTMLImporter implements Employee.Importer
{   ServletRequest request;
    public void open() { /*nothing to do*/ }
    public void close(){ /*nothing to do*/ }
    public HTMLImporter( ServletRequest request )
    {   this.request = request;
    }
    public String provideName()
    {   return request.getParameter("name");
    }
    public String provideID()
    {   return request.getParameter("id");
    }
    public String provideSalary()
    {   return request.getParameter("salary");
    }
}

      Employee e =
            new Employee( new HTMLImporter(request) );
```

©2003, Allen I. Holub www.holub.com 25

Ask for help, not for information

- Eliminate getters/setters by rethinking how the object works.
- **Ask the object that has the information to do the work for you**.
  - then you don't need to "get" anything.
- Develop code using accepted OO-Design processes.
  - Code that develops from use-case analysis and dynamic modeling doesn't have getters & setters because they simply aren't necessary.

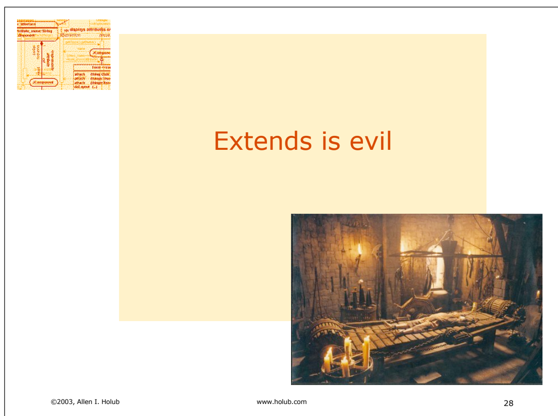
©2003, Allen I. Holub www.holub.com 26

When are getters/setters okay?

- Returning an object in terms of an interface that it implements can reduce the consequences, but should be avoided if possible.
  - Eg. Collection.iterator().
  - The returned object must hid its own implementation.
- Accessors/mutators are mandatory at the "procedural boundary layer."
  - The database.
  - The Operating System.
  - The UI Toolkit.
- Designers of generic toolkits must accessors and mutators because they don't know how the objects will be used, so can't define the operations.

©2003, Allen I. Holub www.holub.com 27

© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE



Inheritance

- There are two types of inheritance:
  - Implementation inheritance (**`extends`**)
    - The base class has methods and fields which are effectively part of (are inherited by) the derived class.
  - Interface inheritance (**`implements`**)
    - The base class is nothing but prototypes of methods that are implemented by the derived class.
- Implementation inheritance is risky, and can almost always be replaced by interface inheritance and delegation.

©2003, Allen I. Holub www.holub.com 29

Interfaces make your code more flexible.

```
LinkedList list = new LinkedList();
g( list );

g( LinkedList list )
{    list.add( ... );
    g2( list )
}
```

Changing the list type impacts g().

– vs. –

```
Collection list = new LinkedList();
g( list );

g( Collection list )
{  list.add( ... );
    g2( list )
}
```

Changing the collection type doesn't impact g().

©2003, Allen I. Holub www.holub.com 30

© 2005, Allen I. Holub. All Rights Reserved.

DO NOT DUPLICATE

Design patterns can add even more flexibilty

```
void f2()
{   Collection c = new HashSet();
    //...
    g2( c.iterator() );
}

void g2( Iterator i )
{   while( i.hasNext() ;)
        do_something_with( i.next() );
}
```


When is implementation inheritance appropriate?

- Implementation normalization.
  - Encapsulate into a base class operations that would otherwise be implemented identically in several derived classes.
- Compile-time restriction of activities.

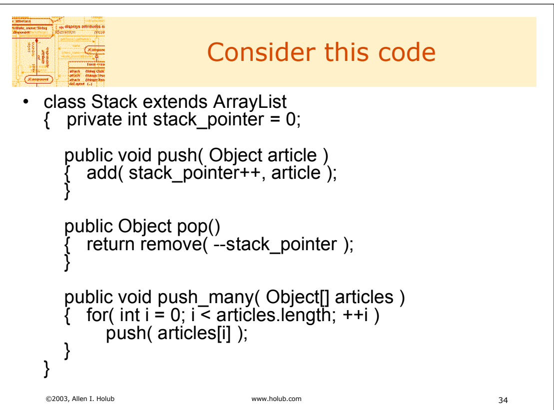
```
Class Employee { /*…*/ }
Class Manager extends Employee
{       do_manager_stuff(){/*…*/}
}
```

- Other reasonable uses of extends all involve design trade offs.

Fragile base classes

- The main problem with implementation inheritance is "fragility."
  - Derived classes often depend on base class behaving in a certain way.
  - If you change the behavior of a base-class method, you can break the derived class.
  - This base-class change is often an IMPROVEMENT.

© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE



Consider this code

```
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] );
    }
}
```

©2003, Allen I. Holub www.holub.com 35

So what's wrong?

- What if a user leverages inheritance and uses the ArrayList's clear() method to pop everything off the stack:

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

- The stack pointer is not modified, so the stack now holds garbage.

©2003, Allen I. Holub www.holub.com 35

How about using encapsulation?

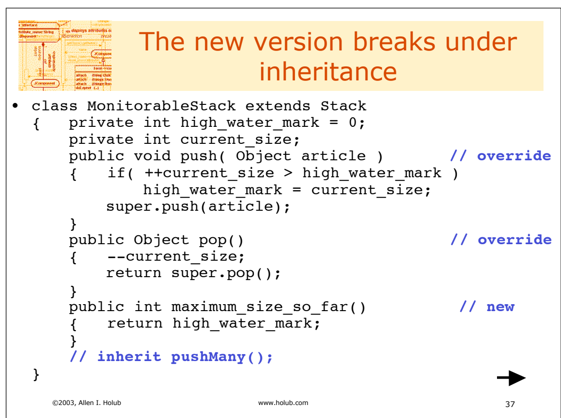
```
class Stack
{    private int stack_pointer  = 0;
     private ArrayList the_data = new ArrayList();
     public void push( Object article )
     {    the_data.add( stack_pointer++, article );
     }
     public Object pop()
     {    return the_data.remove( --stack_pointer );
     }
     public void push_many( Object[] articles )
     {    for( int i = 0; i < o.length; ++i )
               push( articles[i] );
     }
}
```

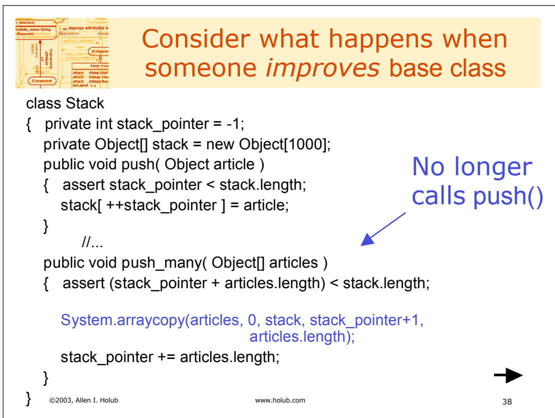
- There's no **clear()** [that's good].   But ...

©2003, Allen I. Holub www.holub.com 36

© 2005, Allen I. Holub. All Rights Reserved.

DO NOT DUPLICATE





The improvement broke the derived class.

- The MonitorableStack did not override pushMany() because it expected pushMany() to call push() (which it did override).
- If someone calls pushMany(), then the high-water-mark will not be adjusted.
- A solution using interfaces and encapsulation fixes the problem permanently. (next slide)

©2003, Allen I. Holub www.holub.com

© 2005, Allen I. Holub. All Rights Reserved. **DO NOT DUPLICATE**

An improved version (1)

- First, introduce an interface:

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

- Implement SimpleStack just like before, but implement the interface:

```
class SimpleStack implements Stack
{ //... as in eariler slide.
}
```


An improved version (2)

```
class MonitorableStack implements Stack
{   private SimpleStack stack = new SimpleStack();
    private int high_water_mark = 0, current_size;
    public void push( Object o )
    {   if( ++current_size > high_water_mark )
            high_water_mark = current_size;
        stack.push(o);
    }
    //...
    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 );
    }
    //...
}
```


The "inheritance" pattern

- Rather than:

```
class Simple{ void f(){ /*...*/ }   }
class Specialization extends Simple{ /*...*/ }
```

Use:

```
interface Simple
{      void f();
       static class Implementation implements Simple
       {      void f(){ /* does some work */ }
       }
}
class Specialization implements Simple
{      Simple delegate = new Simple.Implementation();
       void f(){ delegate.f(); }
}
```


© 2005, Allen I. Holub. All Rights Reserved.

DO NOT DUPLICATE

A few observations

- At any time in the future, anyone can add a method to a base class (e.g. `clear()`) that might break the derived class.
- Avoid "Framework" architectures. (in which you must use implementation inheritance to customize base-class behavior)
- Since you can implement as many interfaces as you like, you can use the "inheritance" pattern to implement multiple inheritance in Java.

What this all means

There is no such thing as perfect

- Design is a series of trade-offs.
- Assess risk, then make reasonable decisions.
  - If you use implementation inheritance, then you run the risk of a fragile-base-class related bug.
  - If you expose implementation (with getters and setters) then you run the risk of a change to the exposing class rippling out to the entire program, with concomitant maintenance headaches.
  - *That might be okay. Use your brain!*

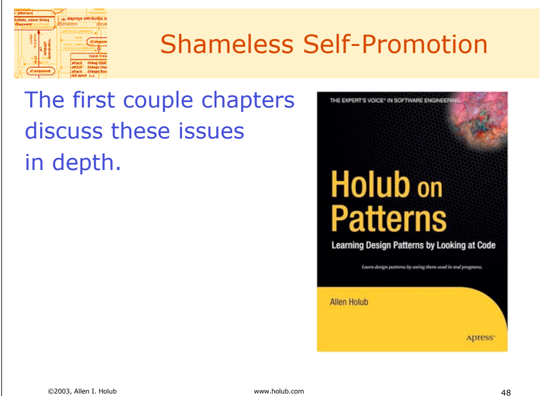
© 2005, Allen I. Holub. All Rights Reserved. **DO NOT DUPLICATE**

There’s often a better solution

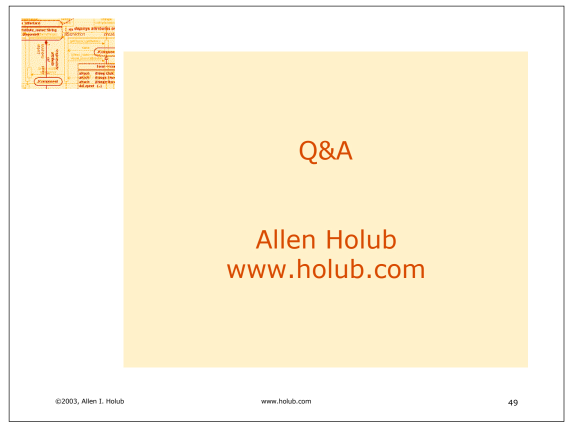
- Approach popular libraries with skepticism
  - Use them if they help, but don’t hold them out as a model of good design.
- There’s almost always a way to do it “right.”
  - Move the work into the class that has the information needed to do the work.
  - Replace implementation inheritance with interface inheritance.
- You *will* learn to think in an OO way with enough practice.
- Study design.
  - Know the Gang-of-Four design patterns cold.
  - Read code.
  - Learn at least two OO languages.

References

- *These slides*
  - *http://www.holub.com/publications/notes_and_slides*
- *Why extends is evil: Improve your code by replacing concrete base classes with interfaces*
  - http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html
- *Why getter and setter methods are evil: Make your code more maintainable by avoiding accessors*
  - http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html
- *More on getters and setters: Build user interfaces without getters and setters*
  - http://www.javaworld.com/javaworld/jw-01-2004/jw-0102-toolbox.html



© 2005, Allen I. Holub. All Rights Reserved. DO NOT DUPLICATE


Q&A
Allen Holub
www.holub.com
©2003, Allen I. Holub
www.holub.com
49