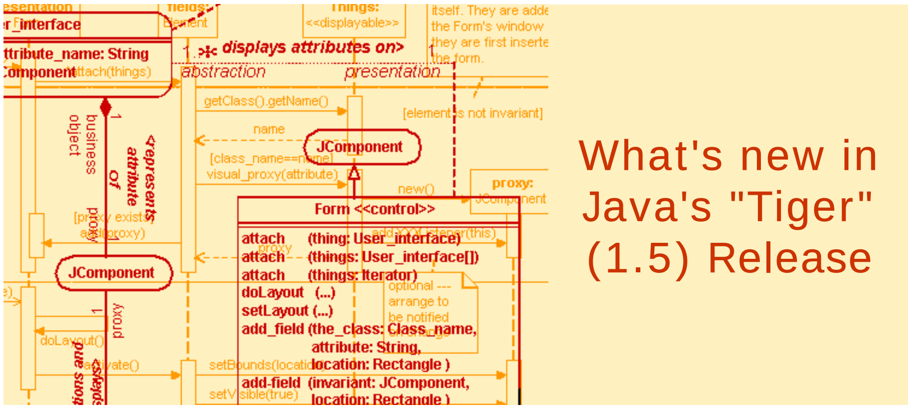
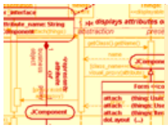


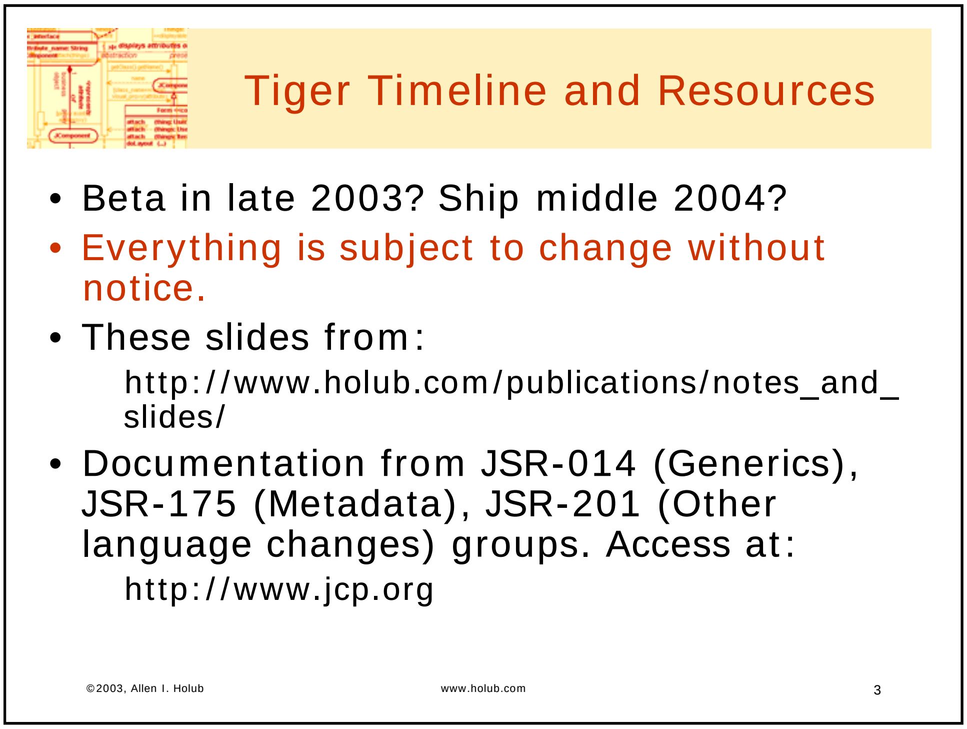
What's new in Java's "Tiger" (1.5) Release

Allen I. Holub
Holub Associates
www.holub.com
allen@holub.com



Allen Holub's Mandatory Tooting-His-Own-Horn Slide.

- Experience ranges from grunt programming to CTO.
- Been programming in Java since its inception.
  - Programmed in C++ 8 years before that.
  - Worked as a programmer since 1979.
- Author of 8 books & many articles.
  - Write the "Java Toolbox" for www.javaworld.com
- I help companies not squander money on software projects:
  - Advise Executives
  - OO Design, Design Review, Java Programming
  - Training (Java and OO) and Project Mentoring.
    - Have taught for U.C. Berkeley Extension since 1983.

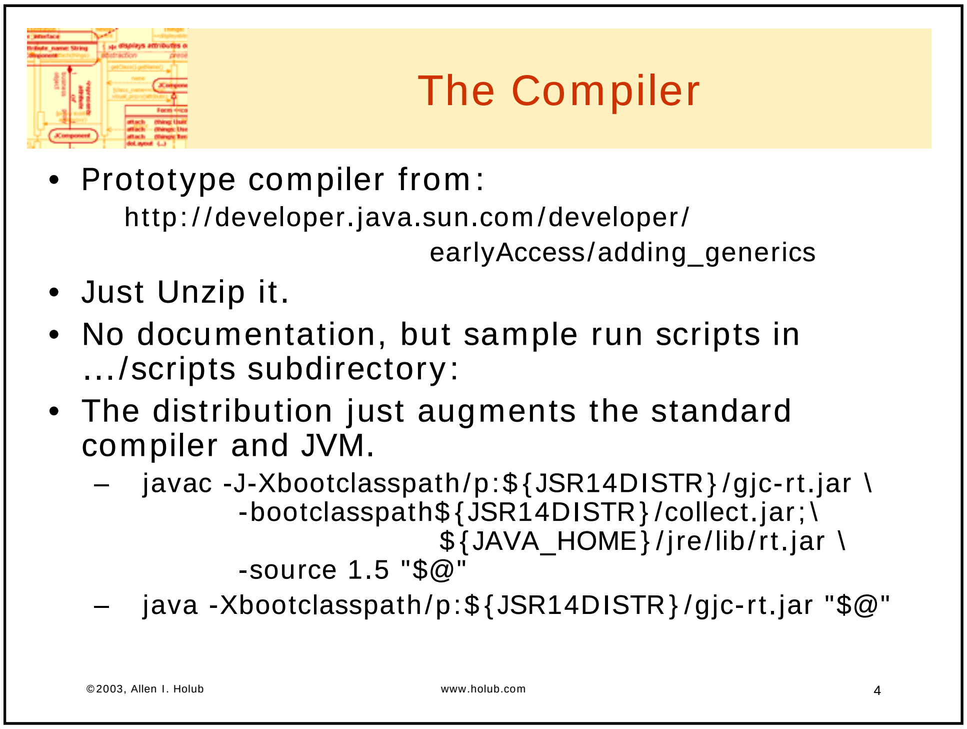


Tiger Timeline and Resources

- Beta in late 2003? Ship middle 2004?
- Everything is subject to change without notice.
- These slides from:

  http://www.holub.com/publications/notes_and_slides/
- Documentation from JSR-014 (Generics), JSR-175 (Metadata), JSR-201 (Other language changes) groups. Access at:

  http://www.jcp.org



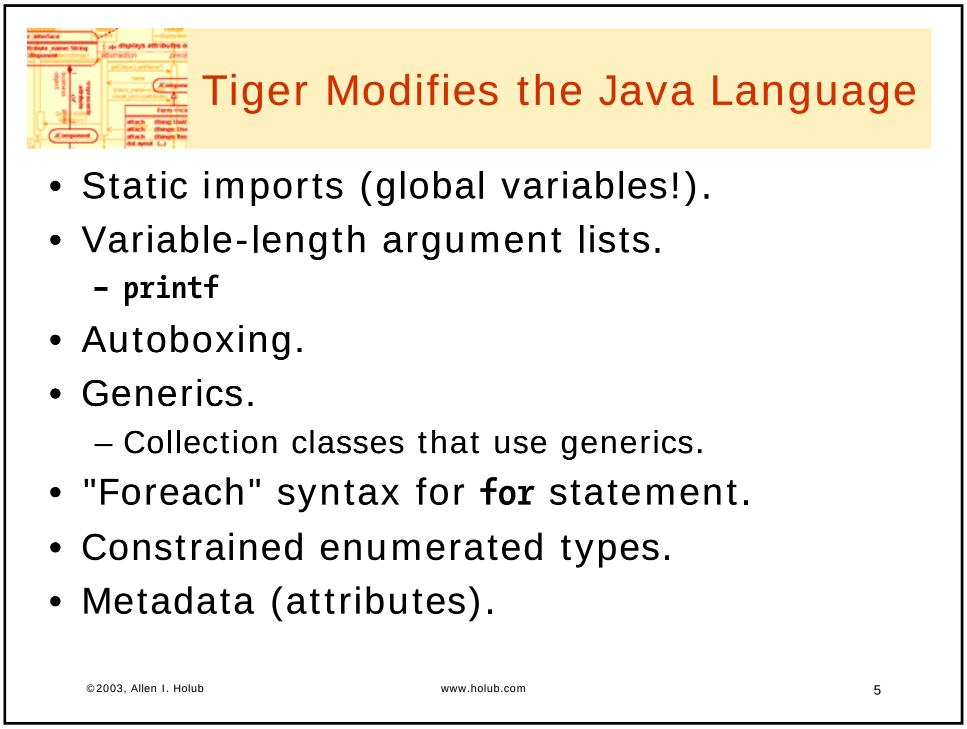
The Compiler

- Prototype compiler from:

  http://developer.java.sun.com/developer/earlyAccess/adding_generics
- Just Unzip it.
- No documentation, but sample run scripts in .../scripts subdirectory:
- The distribution just augments the standard compiler and JVM.

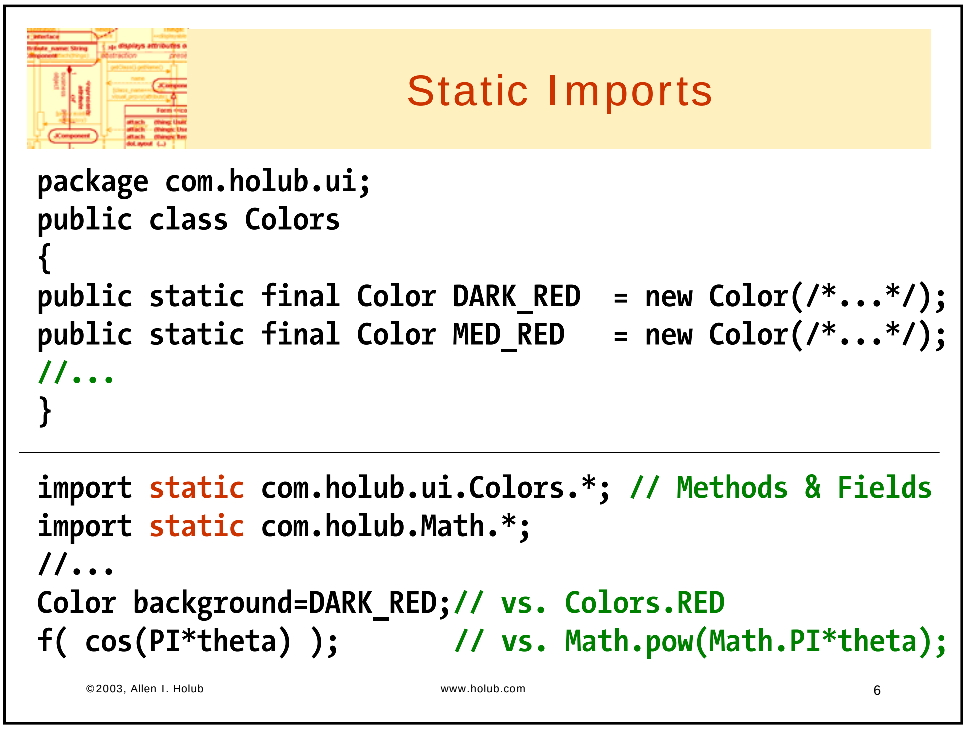
  -
    ```
    javac -J-Xbootclasspath/p:${JSR14DISTR}/gjc-rt.jar \
            -bootclasspath${JSR14DISTR}/collect.jar;\
                            ${JAVA_HOME}/jre/lib/rt.jar \
            -source 1.5 "$@"
    ```

  -
    ```
    java -Xbootclasspath/p:${JSR14DISTR}/gjc-rt.jar "$@"
    ```



Tiger Modifies the Java Language

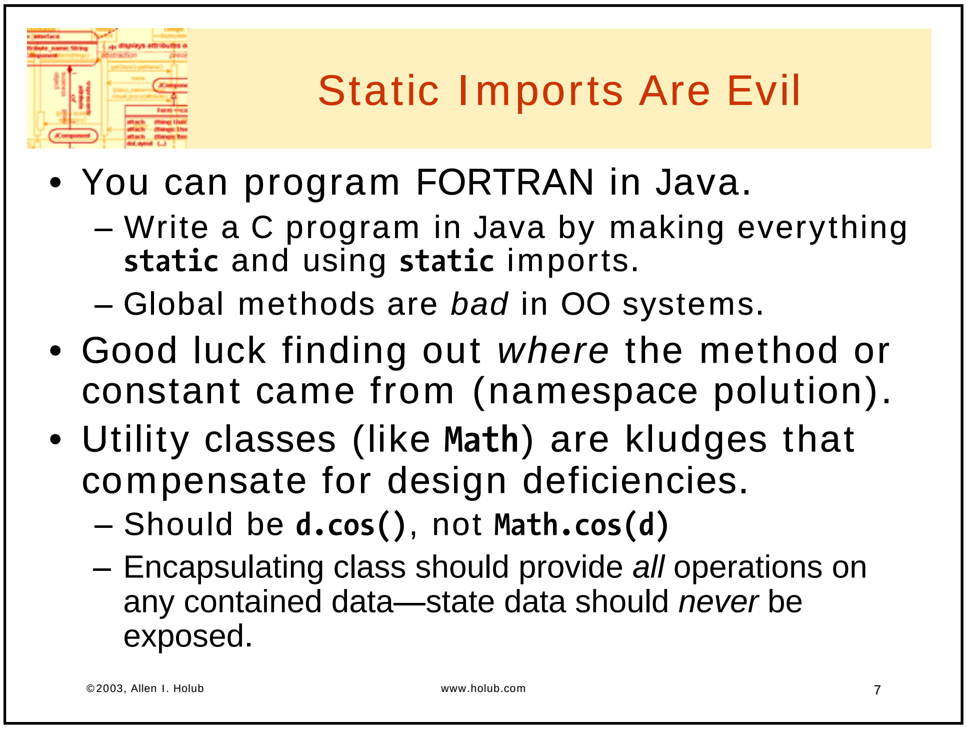
- Static imports (global variables!).
- Variable-length argument lists.
  - printf
- Autoboxing.
- Generics.
  - Collection classes that use generics.
- "Foreach" syntax for **for** statement.
- Constrained enumerated types.
- Metadata (attributes).



Static Imports

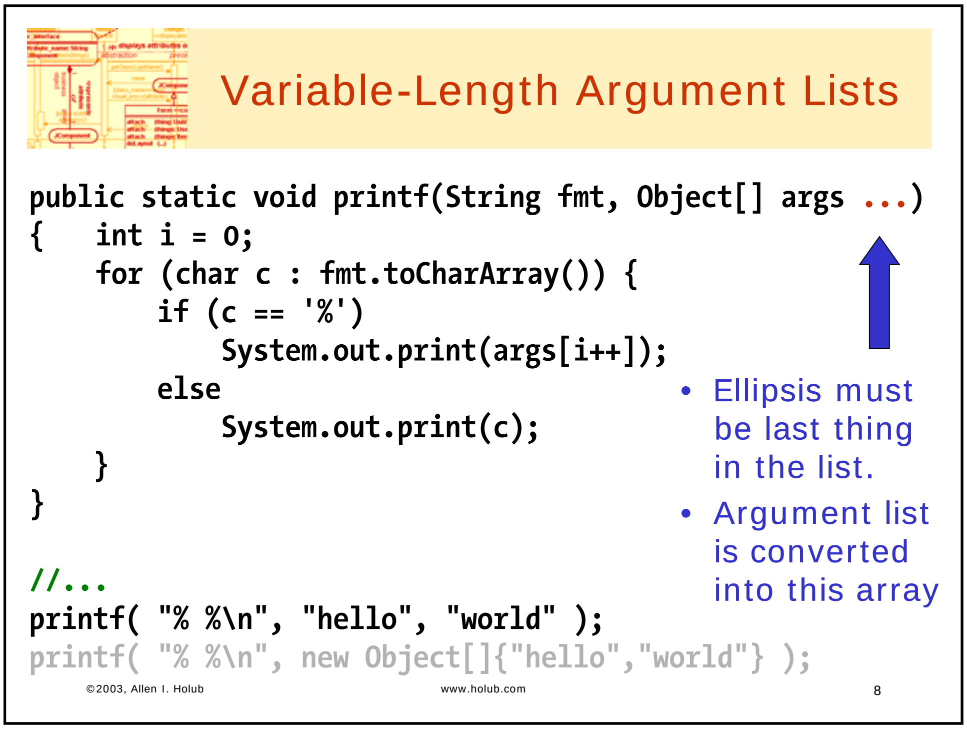
```
package com.holub.ui;
public class Colors
{
public static final Color DARK_RED  = new Color(/*...*/);
public static final Color MED_RED   = new Color(/*...*/);
//...
}
```

```
import static com.holub.ui.Colors.*; // Methods & Fields
import static com.holub.Math.*;
//...
Color background=DARK_RED;// vs. Colors.RED
f( cos(PI*theta) );       // vs. Math.pow(Math.PI*theta);
```



Static Imports Are Evil

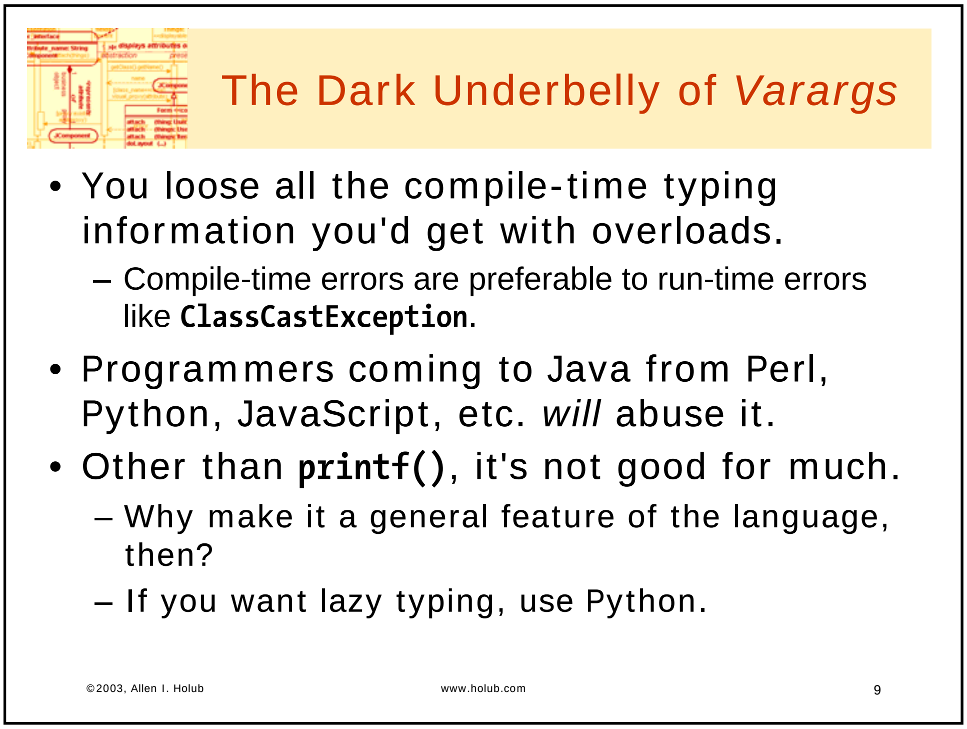
- You can program FORTRAN in Java.
  - Write a C program in Java by making everything **`static`** and using **`static`** imports.
  - Global methods are *bad* in OO systems.
- Good luck finding out *where* the method or constant came from (namespace polution).
- Utility classes (like **`Math`**) are kludges that compensate for design deficiencies.
  - Should be **`d.cos()`**, not **`Math.cos(d)`**
  - Encapsulating class should provide *all* operations on any contained data—state data should *never* be exposed.



Variable-Length Argument Lists

```
public static void printf(String fmt, Object[] args ...)
{   int i = 0;
    for (char c : fmt.toCharArray()) {
        if (c == '%')
            System.out.print(args[i++]);
        else
            System.out.print(c);
    }
}

//...
printf( "% %\n", "hello", "world" );
printf( "% %\n", new Object[]{"hello","world"} );
```

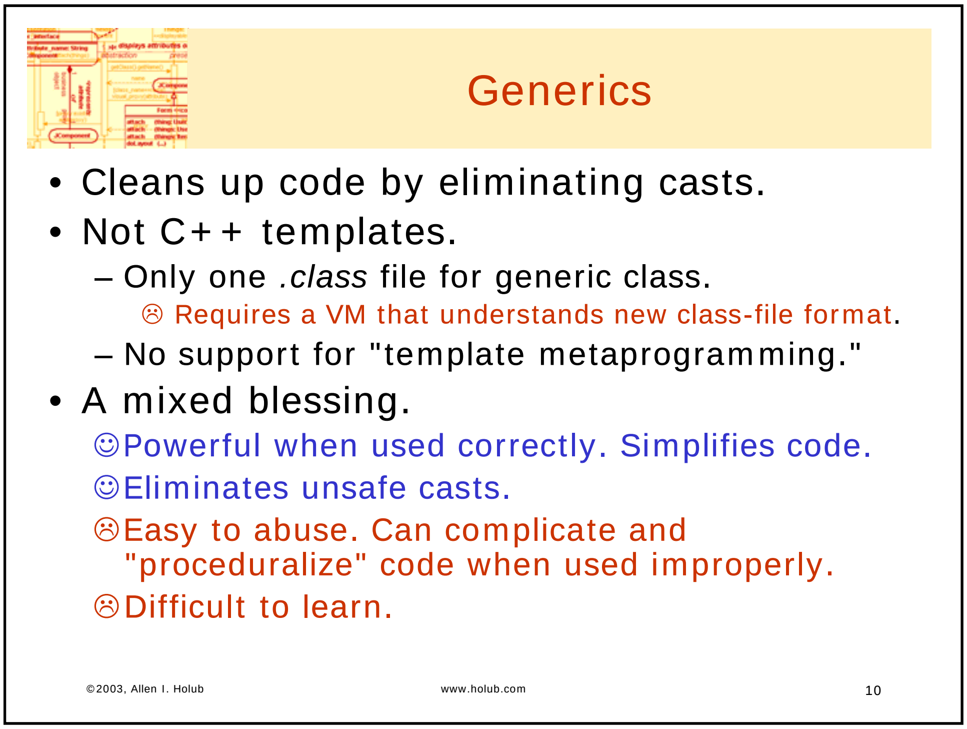
- Ellipsis must be last thing in the list.
- Argument list is converted into this array



The Dark Underbelly of *Varargs*

- You loose all the compile-time typing information you'd get with overloads.
  - Compile-time errors are preferable to run-time errors like **`ClassCastException`**.
- Programmers coming to Java from Perl, Python, JavaScript, etc. *will* abuse it.
- Other than **`printf()`**, it's not good for much.
  - Why make it a general feature of the language, then?
  - If you want lazy typing, use Python.

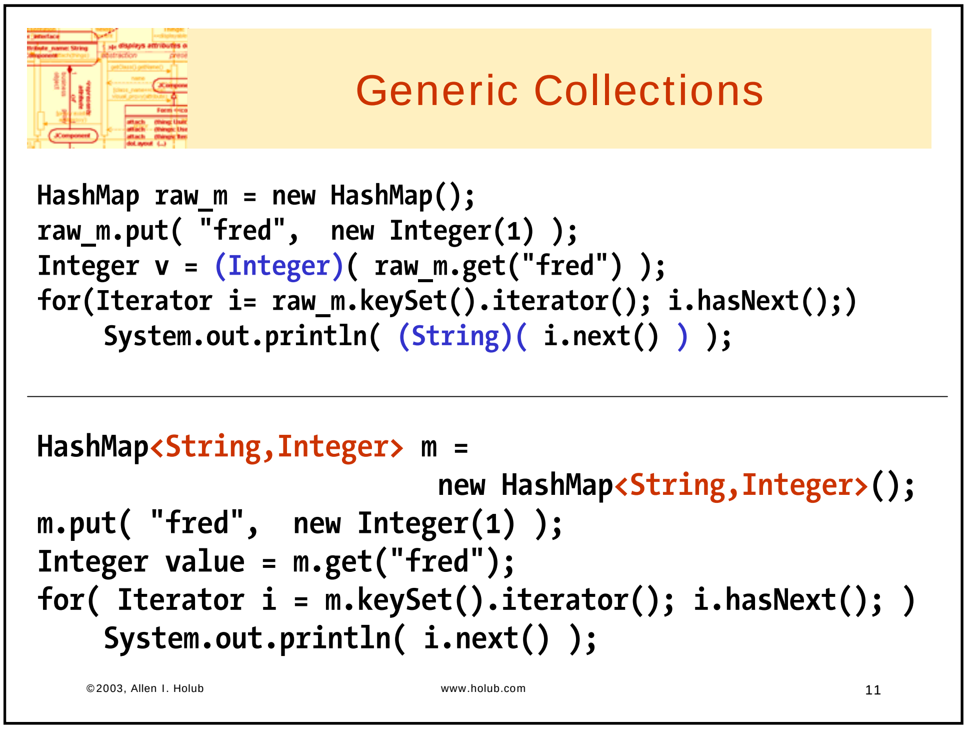
©2003, Allen I. Holub www.holub.com 9



Generics

- Cleans up code by eliminating casts.
- Not C++ templates.
  - Only one *.class* file for generic class.
    - ☹ Requires a VM that understands new class-file format.
  - No support for "template metaprogramming."
- A mixed blessing.
  - ☺Powerful when used correctly. Simplifies code.
  - ☺Eliminates unsafe casts.
  - ☹Easy to abuse. Can complicate and "proceduralize" code when used improperly.
  - ☹Difficult to learn.

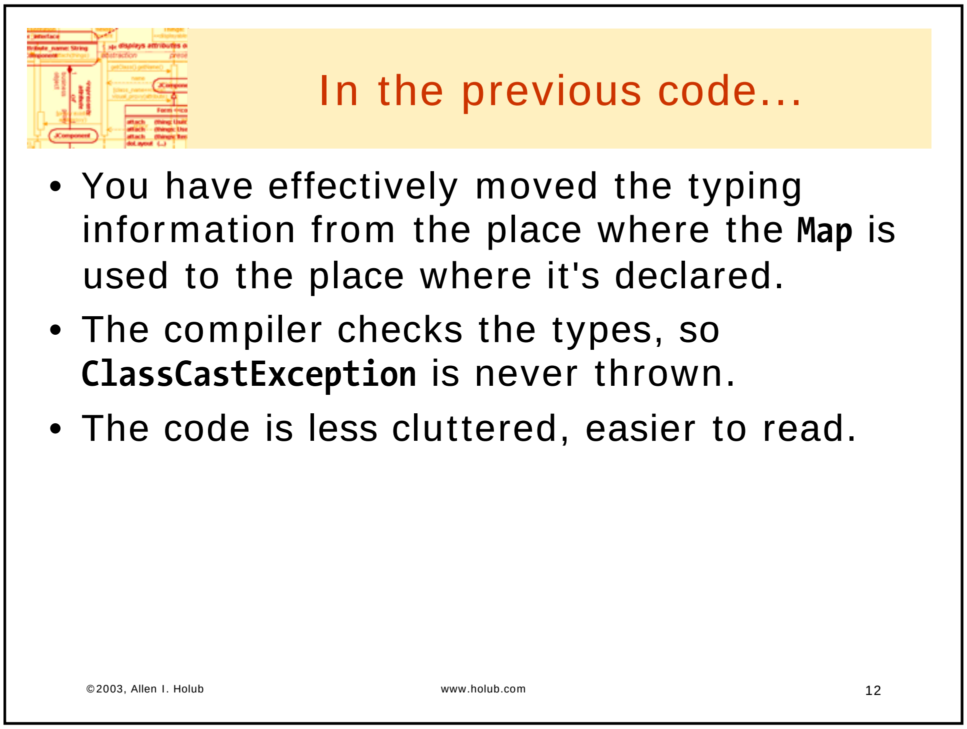
©2003, Allen I. Holub www.holub.com 10



Generic Collections

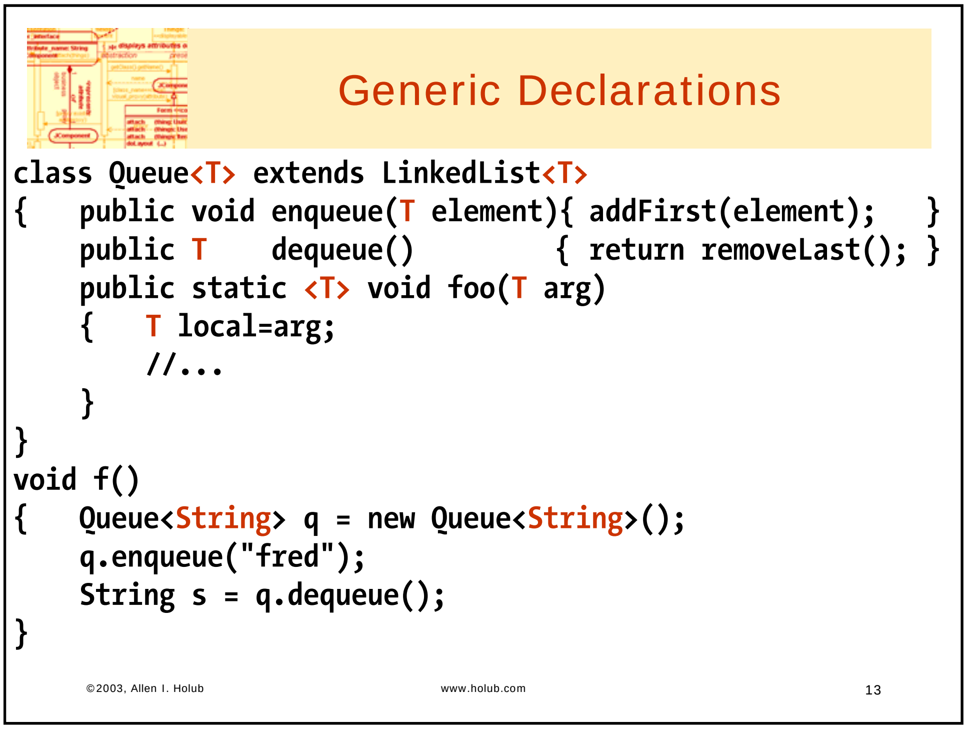
```
HashMap raw_m = new HashMap();
raw_m.put( "fred",  new Integer(1) );
Integer v = (Integer)( raw_m.get("fred") );
for(Iterator i= raw_m.keySet().iterator(); i.hasNext();)
    System.out.println( (String)( i.next() ) );
```

```
HashMap<String,Integer> m =
                         new HashMap<String,Integer>();
m.put( "fred",  new Integer(1) );
Integer value = m.get("fred");
for( Iterator i = m.keySet().iterator(); i.hasNext(); )
    System.out.println( i.next() );
```



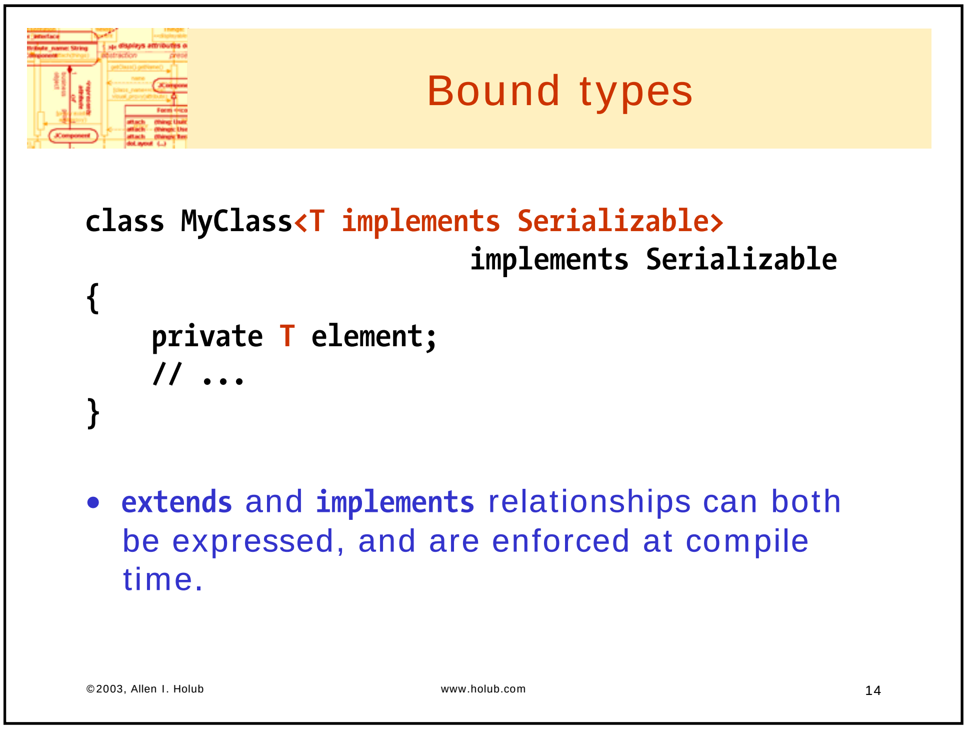
In the previous code…

- You have effectively moved the typing information from the place where the `Map` is used to the place where it's declared.
- The compiler checks the types, so **`ClassCastException`** is never thrown.
- The code is less cluttered, easier to read.



Generic Declarations

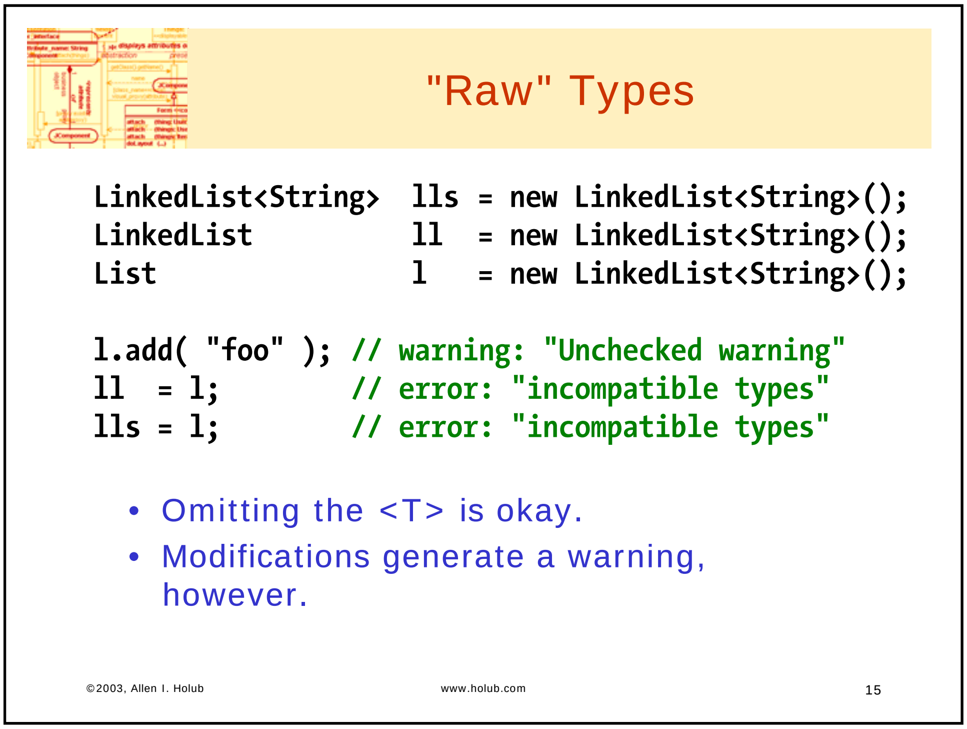
```
class Queue<T> extends LinkedList<T>
{   public void enqueue(T element){ addFirst(element);   }
    public T    dequeue()         { return removeLast(); }
    public static <T> void foo(T arg)
    {   T local=arg;
        //...
    }
}
void f()
{   Queue<String> q = new Queue<String>();
    q.enqueue("fred");
    String s = q.dequeue();
}
```



Bound types

```
class MyClass<T implements Serializable>
                        implements Serializable
{
    private T element;
    // ...
}
```

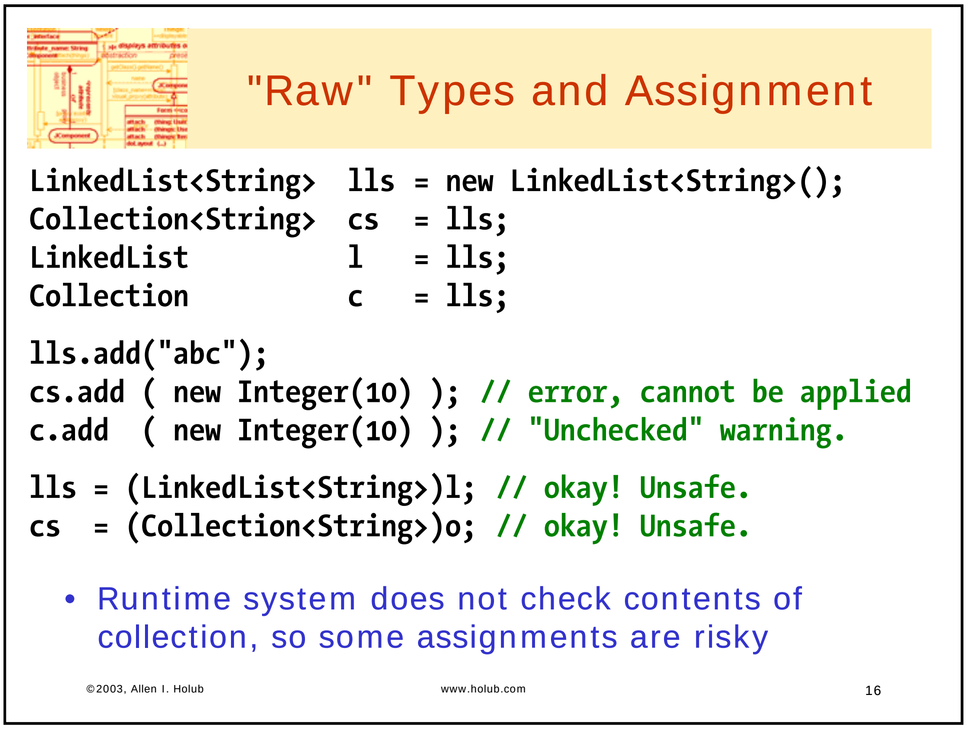
- **extends** and **implements** relationships can both be expressed, and are enforced at compile time.



"Raw" Types

```
LinkedList<String>  lls = new LinkedList<String>();
LinkedList          ll  = new LinkedList<String>();
List                l   = new LinkedList<String>();

l.add( "foo" ); // warning: "Unchecked warning"
ll  = l;        // error: "incompatible types"
lls = l;        // error: "incompatible types"
```

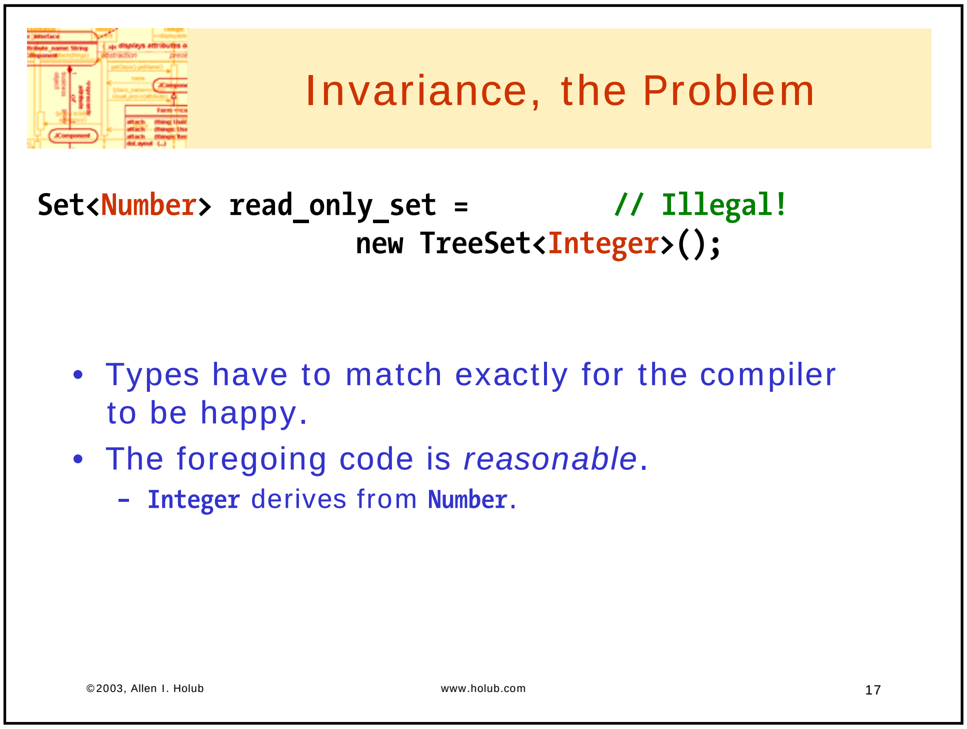
- Omitting the <T> is okay.
- Modifications generate a warning, however.



"Raw" Types and Assignment

```
LinkedList<String>  lls = new LinkedList<String>();
Collection<String>  cs  = lls;
LinkedList          l   = lls;
Collection          c   = lls;

lls.add("abc");
cs.add ( new Integer(10) ); // error, cannot be applied
c.add  ( new Integer(10) ); // "Unchecked" warning.

lls = (LinkedList<String>)l; // okay! Unsafe.
cs  = (Collection<String>)o; // okay! Unsafe.
```

- Runtime system does not check contents of collection, so some assignments are risky

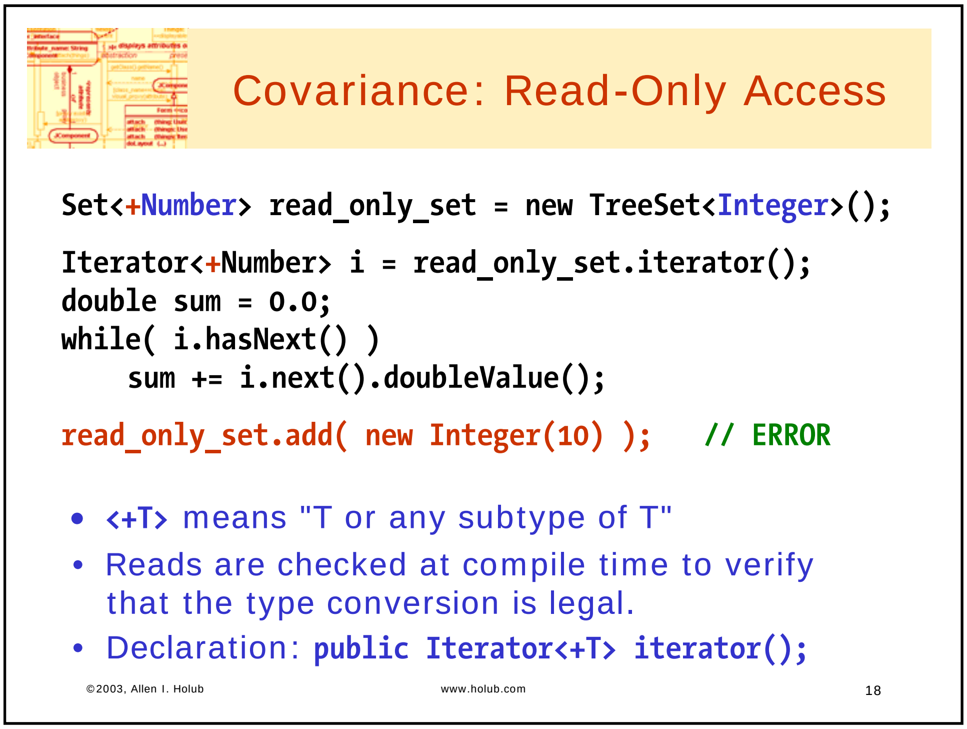


Invariance, the Problem

```
Set<Number> read_only_set =         // Illegal!
                    new TreeSet<Integer>();
```

- Types have to match exactly for the compiler to be happy.
- The foregoing code is *reasonable*.
  - `Integer` derives from `Number`.

©2003, Allen I. Holub www.holub.com 17

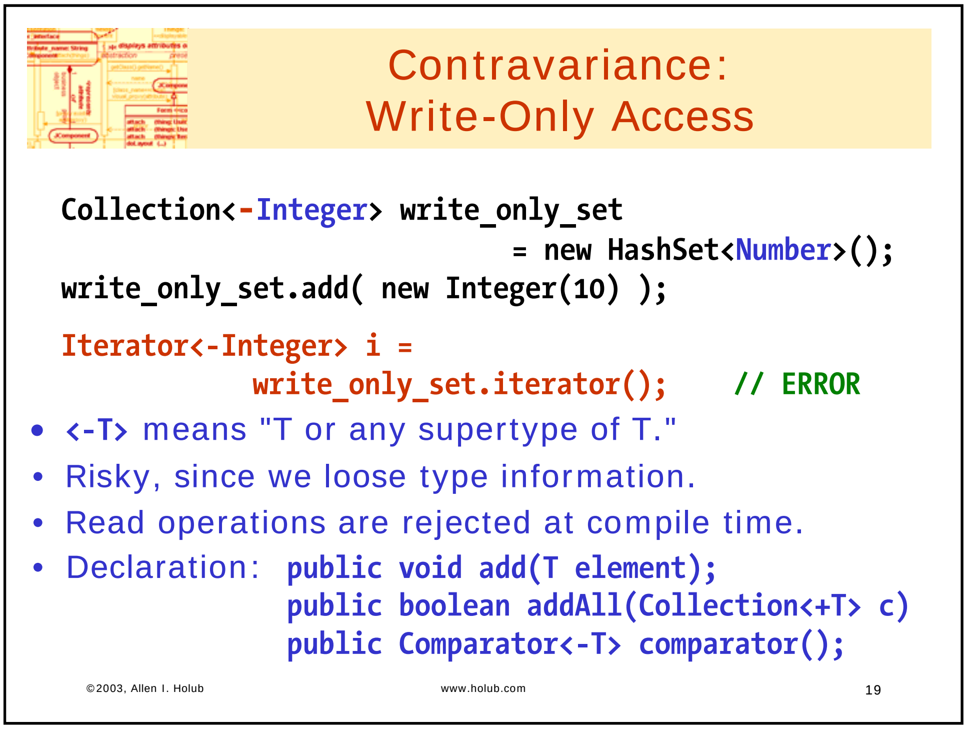


Covariance: Read-Only Access

```
Set<+Number> read_only_set = new TreeSet<Integer>();

Iterator<+Number> i = read_only_set.iterator();
double sum = 0.0;
while( i.hasNext() )
    sum += i.next().doubleValue();

read_only_set.add( new Integer(10) );   // ERROR
```

- `<+T>` means "T or any subtype of T"
- Reads are checked at compile time to verify that the type conversion is legal.
- Declaration: `public Iterator<+T> iterator();`

©2003, Allen I. Holub www.holub.com 18

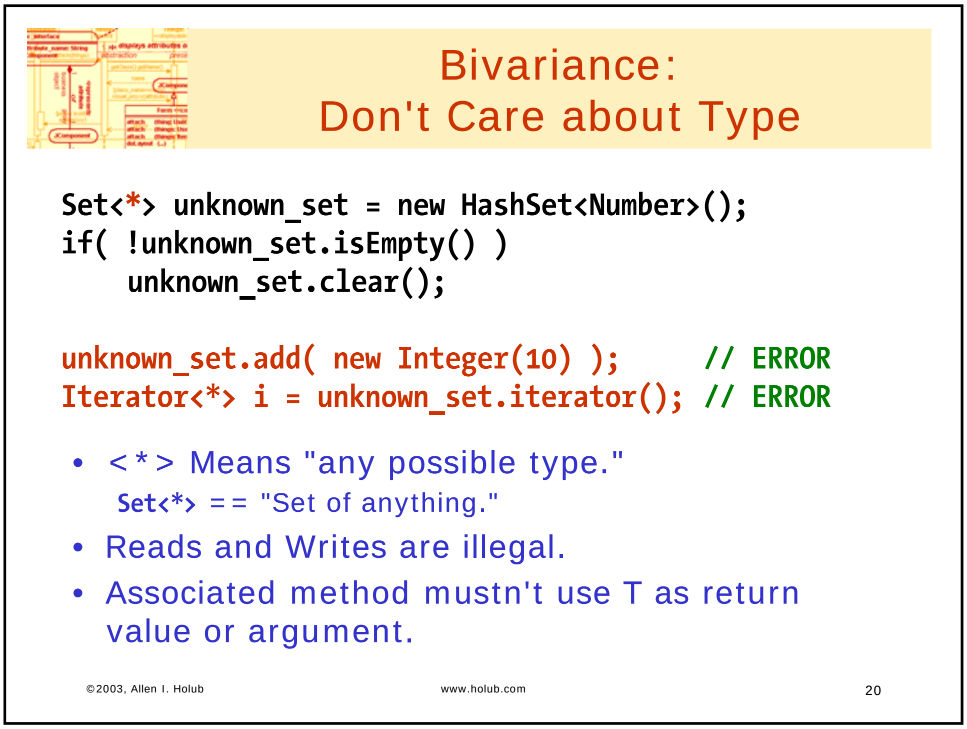


Contravariance: Write-Only Access

```
Collection<-Integer> write_only_set
                            = new HashSet<Number>();
write_only_set.add( new Integer(10) );

Iterator<-Integer> i =
            write_only_set.iterator();    // ERROR
```

- `<-T>` means "T or any supertype of T."
- Risky, since we loose type information.
- Read operations are rejected at compile time.
- Declaration:

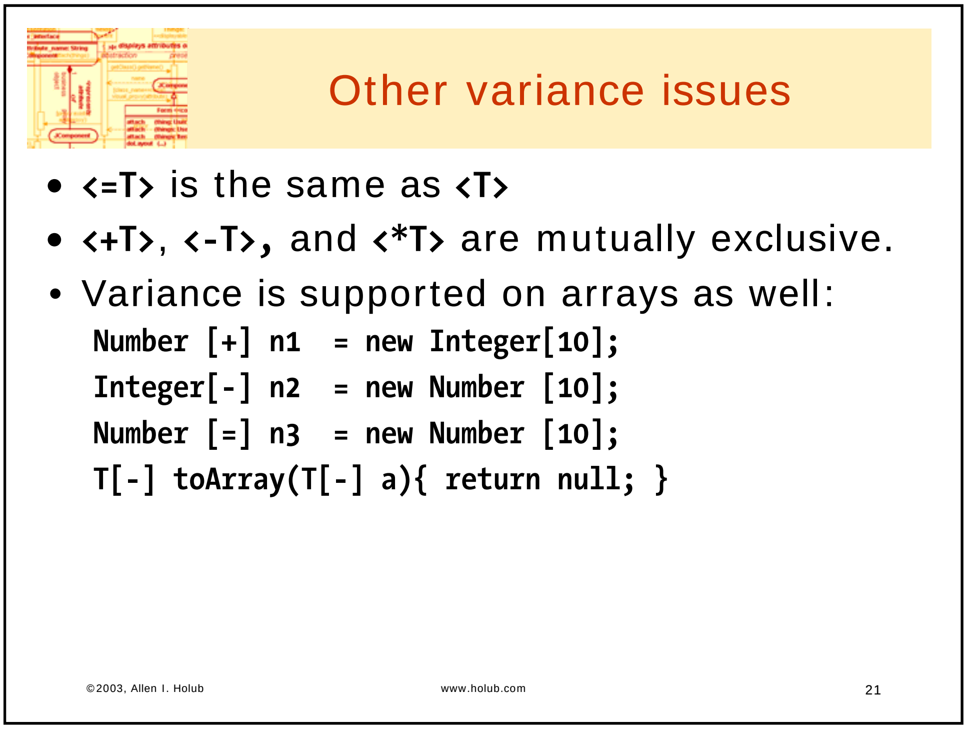
```
public void add(T element);
public boolean addAll(Collection<+T> c)
public Comparator<-T> comparator();
```



Bivariance: Don't Care about Type

```
Set<*> unknown_set = new HashSet<Number>();
if( !unknown_set.isEmpty() )
    unknown_set.clear();

unknown_set.add( new Integer(10) );     // ERROR
Iterator<*> i = unknown_set.iterator(); // ERROR
```

- <*> Means "any possible type."
  `Set<*>` == "Set of anything."
- Reads and Writes are illegal.
- Associated method mustn't use T as return value or argument.

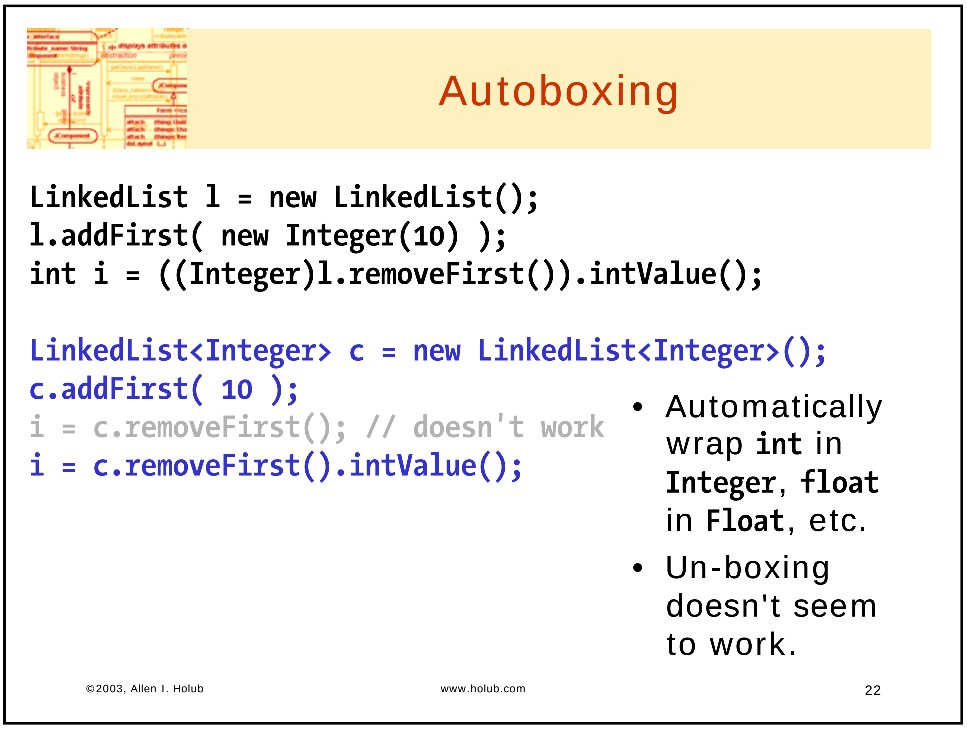


Other variance issues

- **<=T>** is the same as **<T>**
- **<+T>**, **<-T>, ** and **<*T>** are mutually exclusive.
- Variance is supported on arrays as well:

```
Number [+] n1  = new Integer[10];
Integer[-] n2  = new Number [10];
Number [=] n3  = new Number [10];
T[-] toArray(T[-] a){ return null; }
```

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

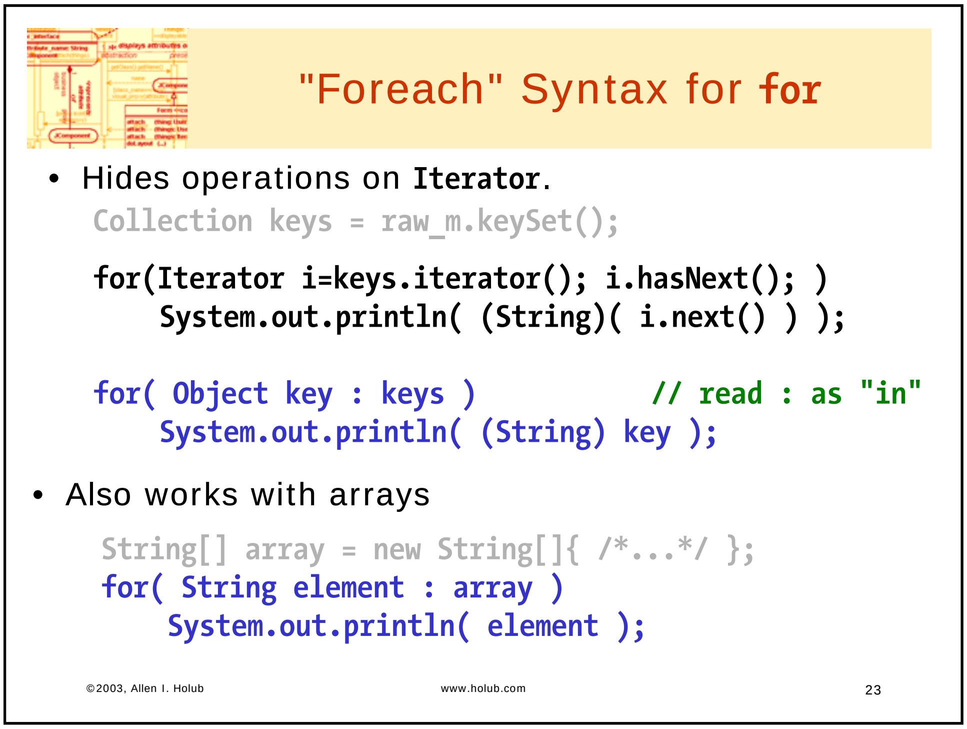


Autoboxing

```
LinkedList l = new LinkedList();
l.addFirst( new Integer(10) );
int i = ((Integer)l.removeFirst()).intValue();

LinkedList<Integer> c = new LinkedList<Integer>();
c.addFirst( 10 );
i = c.removeFirst(); // doesn't work
i = c.removeFirst().intValue();
```

- Automatically wrap **int** in **Integer**, **float** in **Float**, etc.
- Un-boxing doesn't seem to work.

©2003, Allen I. Holub www.holub.com 22



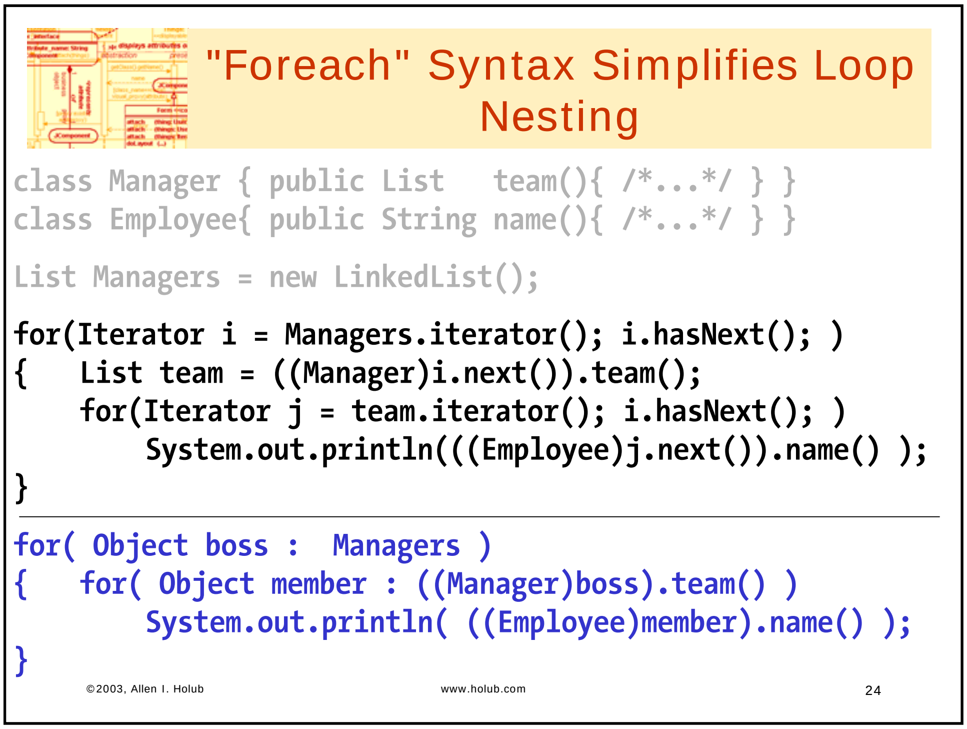
"Foreach" Syntax for for

- Hides operations on Iterator.

```
Collection keys = raw_m.keySet();

for(Iterator i=keys.iterator(); i.hasNext(); )
    System.out.println( (String)( i.next() ) );

for( Object key : keys )          // read : as "in"
    System.out.println( (String) key );
```

- Also works with arrays

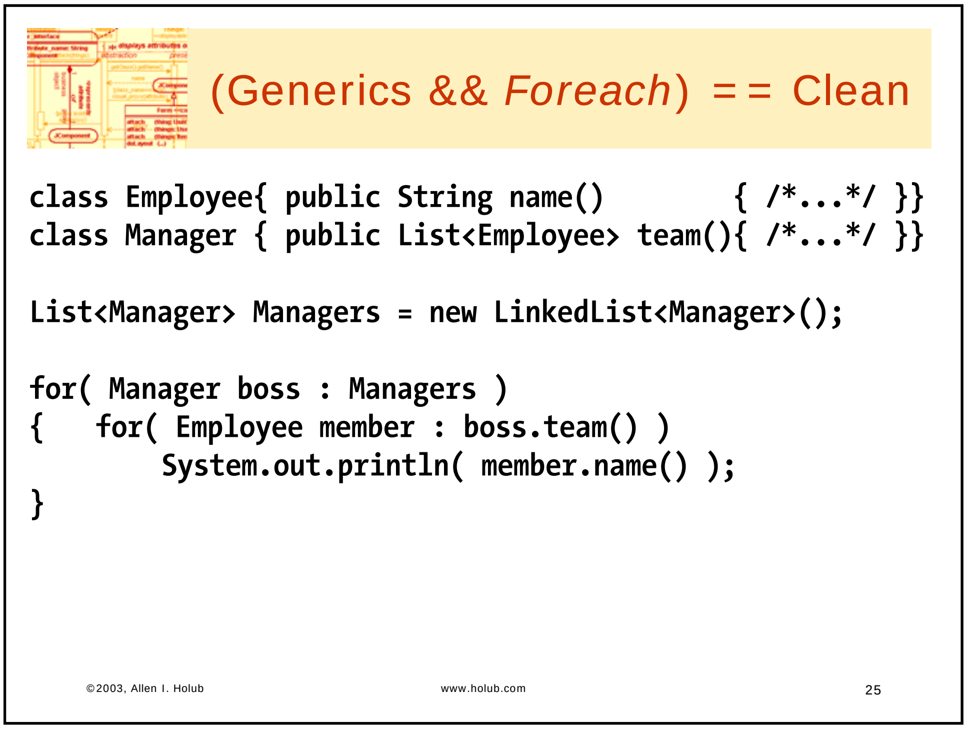
```
String[] array = new String[]{ /*...*/ };
for( String element : array )
    System.out.println( element );
```



"Foreach" Syntax Simplifies Loop Nesting

```
class Manager { public List   team(){ /*...*/ } }
class Employee{ public String name(){ /*...*/ } }

List Managers = new LinkedList();

for(Iterator i = Managers.iterator(); i.hasNext(); )
{   List team = ((Manager)i.next()).team();
    for(Iterator j = team.iterator(); i.hasNext(); )
        System.out.println(((Employee)j.next()).name() );
}
```

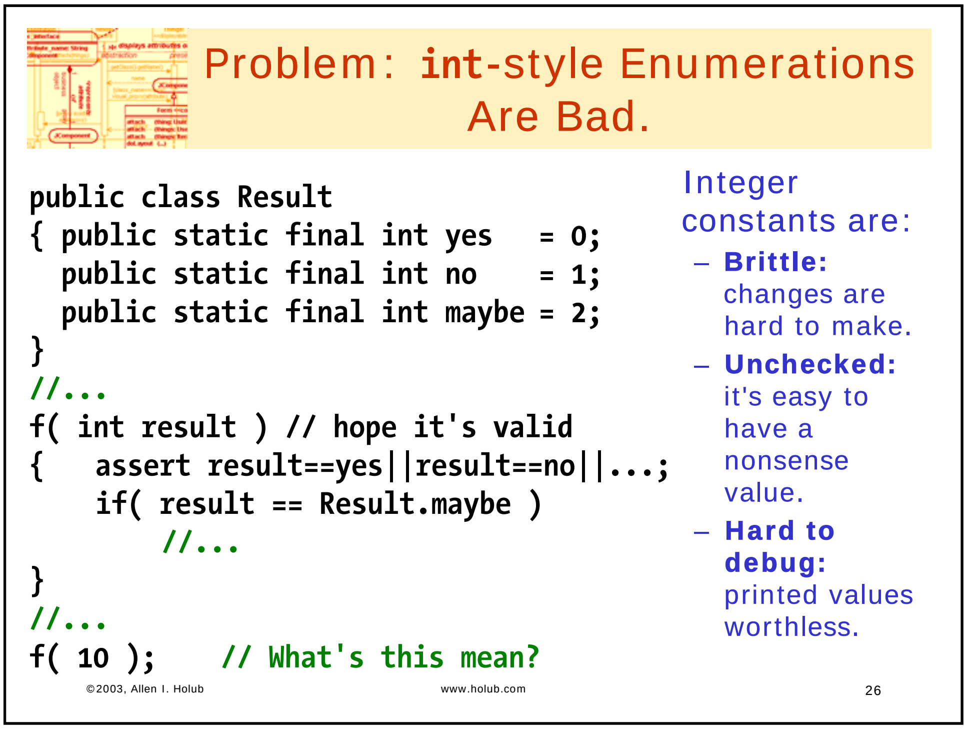
```
for( Object boss :  Managers )
{   for( Object member : ((Manager)boss).team() )
        System.out.println( ((Employee)member).name() );
}
```



(Generics && *Foreach*) == Clean

```
class Employee{ public String name()        { /*...*/ }}
class Manager { public List<Employee> team(){ /*...*/ }}

List<Manager> Managers = new LinkedList<Manager>();

for( Manager boss : Managers )
{   for( Employee member : boss.team() )
        System.out.println( member.name() );
}
```

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



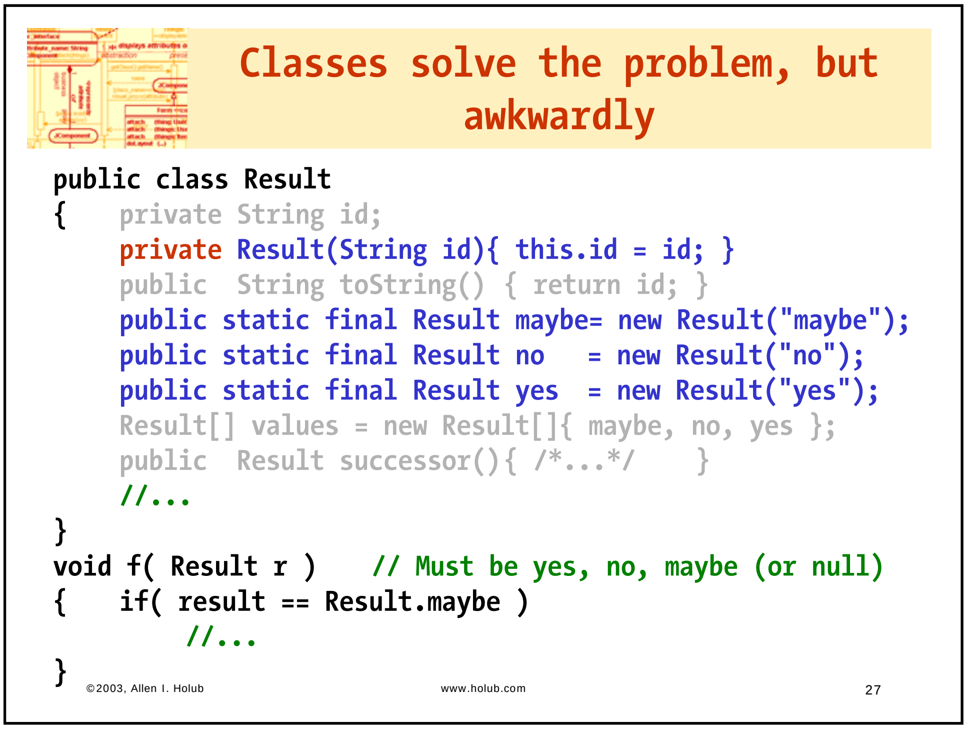
Problem: **int**-style Enumerations Are Bad.

```
public class Result
{ public static final int yes   = 0;
  public static final int no    = 1;
  public static final int maybe = 2;
}
//...
f( int result ) // hope it's valid
{   assert result==yes||result==no||...;
    if( result == Result.maybe )
        //...
}
//...
f( 10 );    // What's this mean?
```

Integer constants are:

- **Brittle:** changes are hard to make.
- **Unchecked:** it's easy to have a nonsense value.
- **Hard to debug:** printed values worthless.

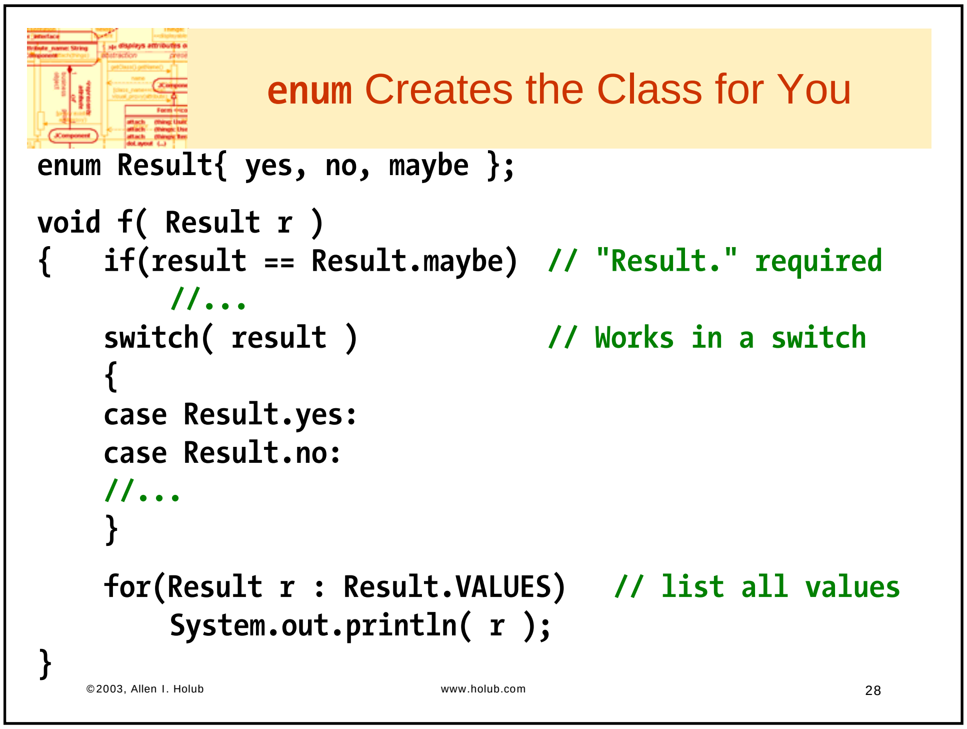
©2003, Allen I. Holub www.holub.com 26



Classes solve the problem, but awkwardly

```
public class Result
{   private String id;
    private Result(String id){ this.id = id; }
    public  String toString() { return id; }
    public static final Result maybe= new Result("maybe");
    public static final Result no   = new Result("no");
    public static final Result yes  = new Result("yes");
    Result[] values = new Result[]{ maybe, no, yes };
    public  Result successor(){ /*...*/    }
    //...
}
void f( Result r )    // Must be yes, no, maybe (or null)
{   if( result == Result.maybe )
        //...
}
```

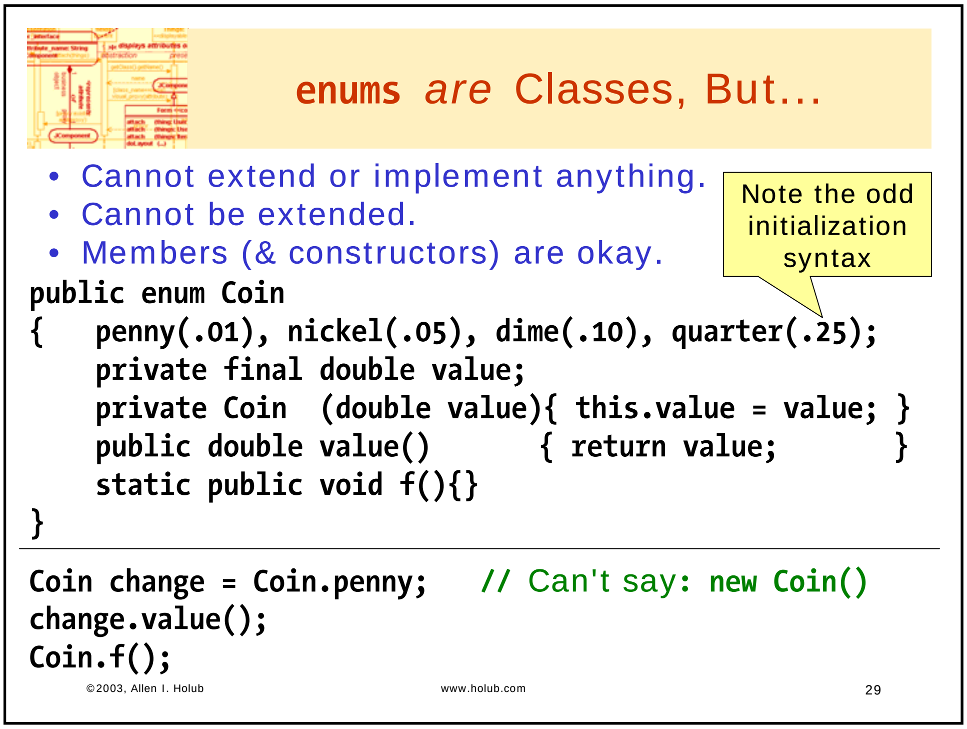
©2003, Allen I. Holub www.holub.com 27



enum Creates the Class for You

```
enum Result{ yes, no, maybe };

void f( Result r )
{   if(result == Result.maybe)  // "Result." required
        //...
    switch( result )            // Works in a switch
    {
    case Result.yes:
    case Result.no:
    //...
    }

    for(Result r : Result.VALUES)   // list all values
        System.out.println( r );
}
```

©2003, Allen I. Holub www.holub.com 28



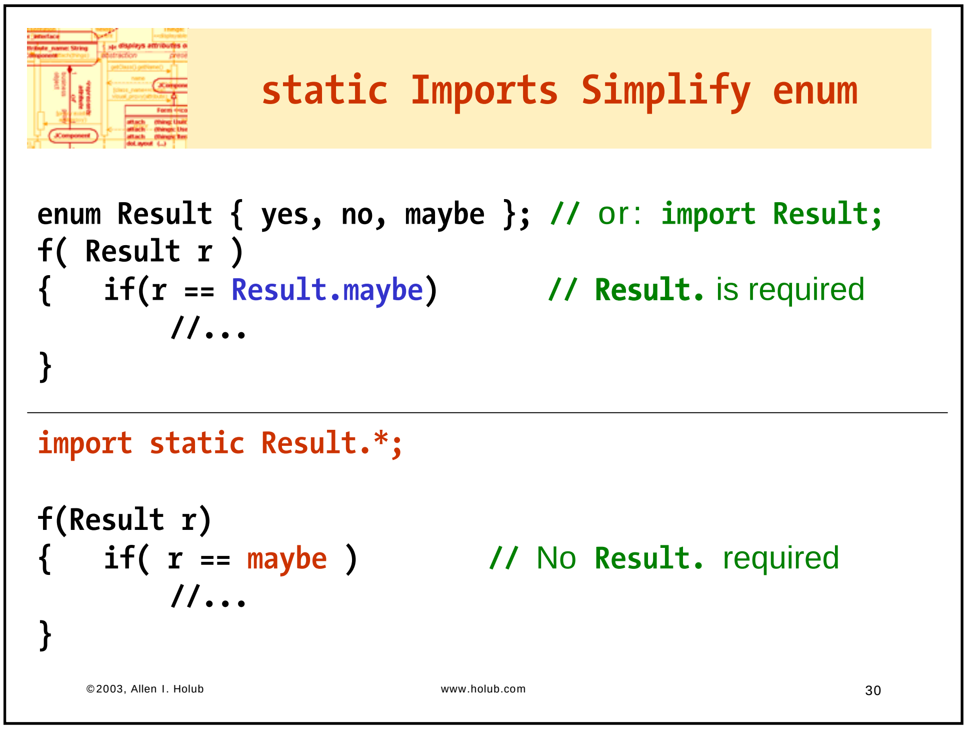
enums *are* Classes, But...

- Cannot extend or implement anything.
- Cannot be extended.
- Members (& constructors) are okay.

Note the odd initialization syntax

```
public enum Coin
{   penny(.01), nickel(.05), dime(.10), quarter(.25);
    private final double value;
    private Coin  (double value){ this.value = value; }
    public double value()      { return value;       }
    static public void f(){}
}
```

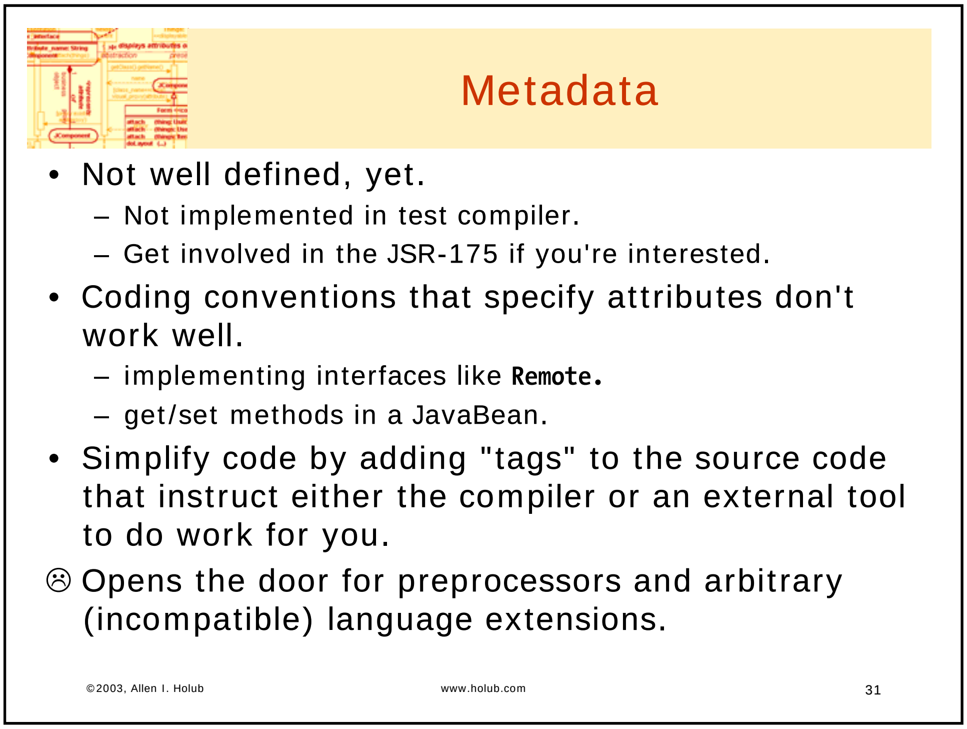
```
Coin change = Coin.penny;   // Can't say: new Coin()
change.value();
Coin.f();
```



static Imports Simplify enum

```
enum Result { yes, no, maybe }; // or: import Result;
f( Result r )
{   if(r == Result.maybe)       // Result. is required
        //...
}
```

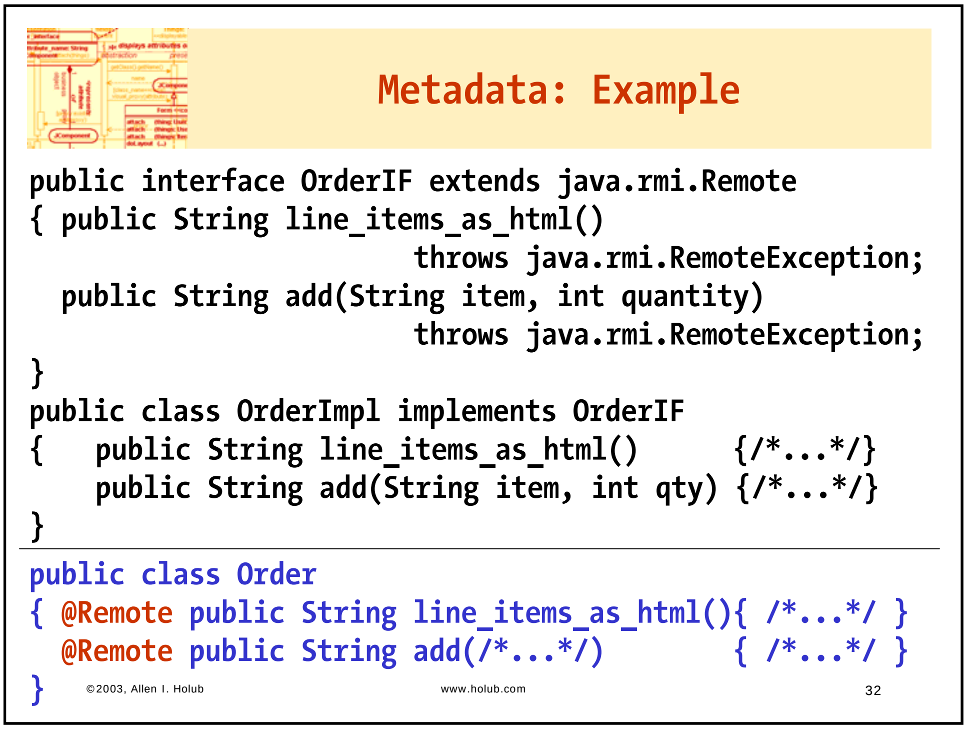
```
import static Result.*;

f(Result r)
{   if( r == maybe )         // No Result. required
        //...
}
```



Metadata

- Not well defined, yet.
  - Not implemented in test compiler.
  - Get involved in the JSR-175 if you're interested.
- Coding conventions that specify attributes don't work well.
  - implementing interfaces like **Remote.**
  - get/set methods in a JavaBean.
- Simplify code by adding "tags" to the source code that instruct either the compiler or an external tool to do work for you.

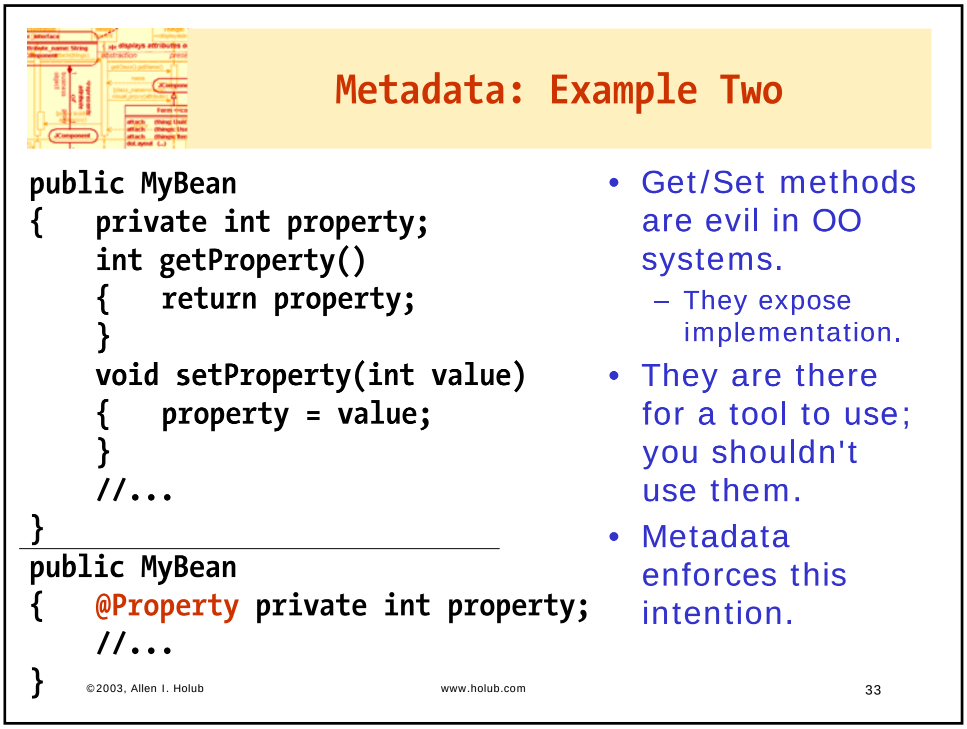
☹ Opens the door for preprocessors and arbitrary (incompatible) language extensions.



Metadata: Example

```
public interface OrderIF extends java.rmi.Remote
{ public String line_items_as_html()
                        throws java.rmi.RemoteException;
  public String add(String item, int quantity)
                        throws java.rmi.RemoteException;
}
public class OrderImpl implements OrderIF
{   public String line_items_as_html()      {/*...*/}
    public String add(String item, int qty) {/*...*/}
}
```

```
public class Order
{ @Remote public String line_items_as_html(){ /*...*/ }
  @Remote public String add(/*...*/)        { /*...*/ }
}
```

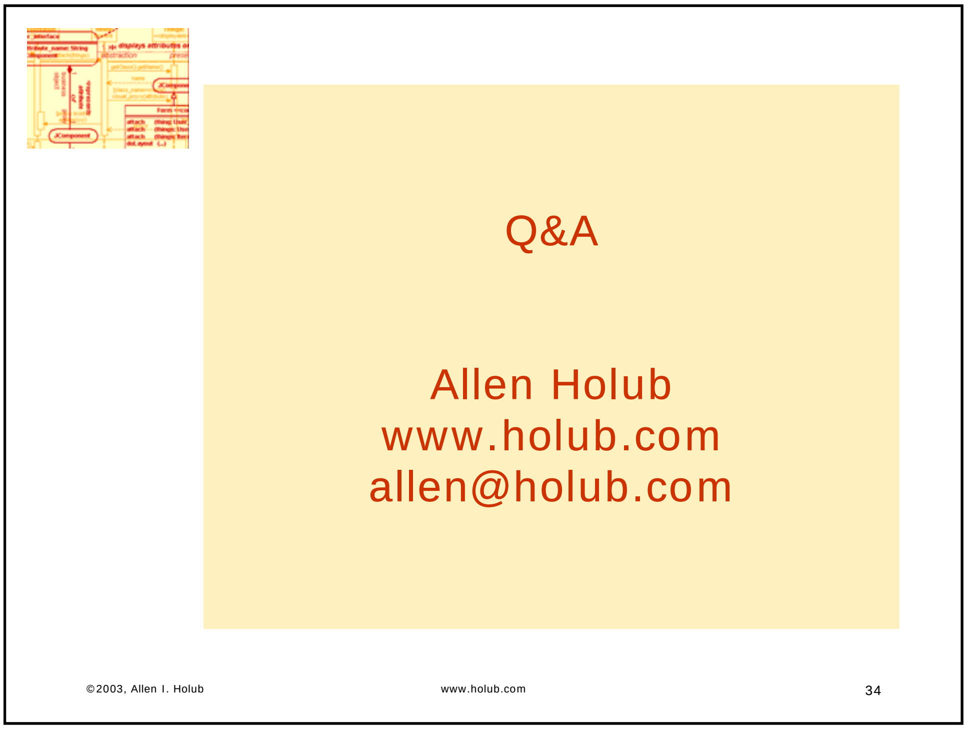


Metadata: Example Two

```
public MyBean
{   private int property;
    int getProperty()
    {   return property;
    }
    void setProperty(int value)
    {   property = value;
    }
    //...
}
```

```
public MyBean
{   @Property private int property;
    //...
}
```

- Get/Set methods are evil in OO systems.
  - They expose implementation.
- They are there for a tool to use; you shouldn't use them.
- Metadata enforces this intention.



Q&A

Allen Holub
www.holub.com
allen@holub.com