



Or:

Why you're going to (eventually)
miss latent typing in Java

Ask questions during the presentation



- Slides are different than handouts
- New material is in my weblogs

Loss of Perspective



- "I think one could argue that fetishizing any technology, technique, or process, from design patterns to agile programming to RUP to Java itself causes a strange inversion where the fetishized object of desire, having become an end in itself, destroys the end for which it was created."
Mike Loukides
- Critical thinking requires that you question something. Scientists do not simply accept what someone says until the experiment has been reproduced.
- Knowing the boundaries gives you power

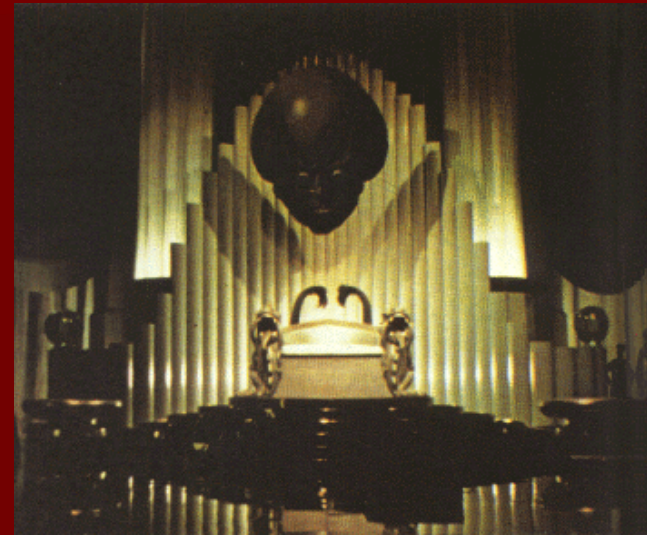
What are generics?



- Classical generics: parameterized types
- Parametric polymorphism
 - Allowing different types to be used for the same parameter in a function, or the same declaration in a class

What are Java Generics?

- Pay no attention to that man behind the curtain.
- 1) They could be just for better static type checking
- Aren't they just for type-safe collections?
- That's what I thought and taught for a long time, about C++ templates



Many writers/teachers think this

- William Grosso writes:
- “the goal of the generics specification...:
By using generics, a programmer can convert casting errors from run-time exceptions to compiler errors. This results in fewer programming errors and better code.”
- Reduce `ClassCastException`

But

- 2) If you look at generics in other languages, they are about writing more “generic” code, that works with more types.

Generic functions

- “A generic function is one which can work for arguments of many types, generally doing the same kind of work independently of the argument type.”

On Understanding Types, Data Abstraction, and Polymorphism

Luca Cardelli & Peter Wegner

Generic programming

- “A style of programming that allows algorithms to be implemented once and used over and over with arbitrary data types. Generic programming empowers algorithms, allowing them to work on [more] arbitrary data types”

The Java Generic Programming System
Mueller & Jensen (on web)

Parametric Polymorphism

- "Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure."

Christopher Strachey, who coined the term

Parametric Polymorphism

- “Using parametric polymorphism, a function or datatype can be written generically so that it can deal equally well with objects of various types. For example, a function append that joins two lists can be constructed so that it does not depend on one particular type of list: it can append lists of integers, lists of real numbers, lists of strings, and so on.”

Polymorphism definition in WordIQ.com

Loosening type constraints

- `method1a(int)`
- `method1b(String)`
- `method1c(Any type you can't inherit)`
- These methods are constrained to working on a single type

More flexible

- `method2 (ABaseClass)`
- Works with `ABaseClass` and all the classes you can inherit from that
- These methods work with more types

Even more flexible

- `method3 (AnInterface)`
- Works with any class or subclass that implements that interface
- Thus crosses class boundaries -- can be applied to even more types

Generics

- `<T> returnType method4 (T)`
- In theory: works with any type
- In practice: “holder” generics work with any type but cannot do more than allowed for **Object**
- “Manipulator” generics must be bound
- Bounds must name classes or interfaces, “any” type not allowed

What is Latent typing?

- Latent: “Present or potential but not evident or active”
- Sometimes called “structural typing”
- Could be called:
 - Implied type
 - Synthesized type
- Ruby calls it “Duck Typing” (if it walks like a duck, and talks like a duck, ...)
- A language allows a type to be implied based on the operations it needs, rather than requiring an explicit type definition

If Java Had Latent Typing

```
class X {  
    void foo() {}  
    void bar() {}  
}  
class Y {  
    void foo() {}  
    void baz() {}  
}
```

```
<T> void f(T t) { t.foo(); }
```

```
f(new X());  
f(new Y());
```

Why is it powerful?

- It allows more types to be used with a piece of code, without requiring explicit type definitions
- Especially important when we're dealing with generic code
- Can make the difference between whether a particular piece of generic code is useful or not

Latent typing in Python

```
def speak(anything):  
    anything.talk()
```

```
class Dog:  
    def talk(self): print "Arf!"  
    def reproduce(self): pass
```

```
class Robot:  
    def talk(self): print "Click!" # Has a talk()  
    def oilChange(self): pass # But it's a different interface
```

```
a = Dog()  
b = Robot()  
speak(a)  
speak(b)
```

Easily translates to C++

```
#include <iostream>
using namespace std;

class Dog {
public:
    void talk() { cout << "Arf!" << endl; }
    void reproduce() {}
};

class Robot {
public:
    void talk() { cout << "Click!" << endl; }
    void oilChange() {}
};
```

```
template<class T> void
    speak(T speaker) {
    speaker.talk();
}

int main() {
    Dog d;
    Robot r;
    speak(d);
    speak(r);
}
```

Checked at compile-time!

Implies a new type

- “The type acceptable to **speak()**”
- Unspecified and unwritten: *latent* type
- You can try applying the template to anything, regardless of actual type
 - Type correctness is ensured at compile time

Java/C# Generics are a different story

```
import java.util.*;

class Dog {
    public void talk() { System.out.println("Woof!"); }
    public void reproduce() { }
}

class Robot {
    public void talk() { System.out.println("Click!"); }
    public void oilChange() { }
}
```

```
public class DogAndRobotCollections {
    public static void main(String[] args) {
        List<Dog> dogList = new ArrayList<Dog>();
        List<Robot> robotList = new ArrayList<Robot>();
        for(int i = 0; i < 10; i++)
            dogList.add(new Dog());
        // dogList.add(new Robot()); // Compile-time error
        for(int i = 0; i < 10; i++)
            robotList.add(new Robot());
        // robotList.add(new Dog()); // Compile-time error
        for(Dog d : dogList) // New for-each syntax
            d.talk(); // No cast necessary
        for(Robot r: robotList)
            r.talk(); // No cast necessary
    }
}
```

But this won't work

- You cannot say “I don't care what type this is as long as it has a **talk()**” (no latent typing with Java generics)

```
class Communicate {  
    public static <T> void speak(T speaker) {  
        speaker.talk(); // Nope  
    }  
}
```

The best you can do...

- Is to only use Object methods

```
public class NothingButObject {  
    public <T> String f(T anyObject) {  
        return anyObject.toString();  
    }  
}
```

- (But this erases **T** to **Object**)
- The Dog/Robot example must use bounds

Erasure

- Java Generics erases the type
- Migration compatibility:
 - Not just compiling old code
 - Using new code with old libraries:
 - `oldLibraryMethod(List<Foo>)`
- For everything to work together, compiles to the least-common denominator: the old (non-parameterized) type.
- When you're coding, you must always remember that you don't really know the type of the parameter.



Bounds compensate for erasure

- Uses the **extends** and **super** keywords
- With either **T** or **?**



```
interface Speaks { void talk(); }
```

```
class Dog implements Speaks {  
    public void talk() { System.out.println("Woof!"); }  
    public void reproduce() { }  
}
```

```
class Robot implements Speaks {  
    public void talk() { System.out.println("Click!"); }  
    public void oilChange() { }  
}
```

```
class Communicate {  
    public static <T extends Speaks> void speak(T speaker) {  
        speaker.talk();  
    }  
}
```

```
public class DogsAndRobots {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        Robot r = new Robot();  
        Communicate.speak(d);  
        Communicate.speak(r);  
    }  
}
```

This is redundant

- Could just say:

```
class Communicate {  
    public static void speak(Speaks speaker) {  
        speaker.talk();  
    }  
}
```

- This is what erasure produces
- Redundancy like this more common in methods

Why it would be nice...

- ...To have latent typing in Java
- Note: it's not going to happen. I can't see how it could ever happen
- But understanding the issue – and the solution – will make you a better generic programmer

```
import java.util.*;
import java.lang.reflect.*;

// Doesn't work with "anything that has an add()."
// There is no "Addable" interface so we are narrowed
// to using a Collection.
// We cannot generalize using generics in this case.

public class Fill {
    public static <T> void
    fill(Collection<T> collection, Class<? extends T>
    classToken, int size) {
        for(int i = 0; i < size; i++)
            // Assumes default constructor:
            try {
                collection.add(classToken.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
}
```

```
// A different kind of container that is Iterable
class SimpleQueue<T> implements Iterable<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void add(T t) { storage.offer(t); }
    public T get() { return storage.poll(); }
    public Iterator<T> iterator() { return
        storage.iterator(); }
}
```

```
class Contract {}
class TitleTransfer extends Contract {}

class FillTest {
    public static void main(String[] args) {
        List<Contract> contracts = new ArrayList<Contract>();
        Fill.fill(contracts, Contract.class, 3);
        Fill.fill(contracts, TitleTransfer.class, 2);
        for(Contract c: contracts)
            System.out.println(c);
        SimpleQueue<Contract> contractQueue =
            new SimpleQueue<Contract>();
        // Won't work. fill() is not generic enough:
        // Fill.fill(contractQueue, Contract.class, 3);
    }
}
```

Working around the lack of latent typing

- What is latent typing accomplishing here?
- It means that you are able to write code saying "I don't care what type I'm using here as long as it has these methods."
- In effect, latent typing creates an implicit interface containing the desired methods.
- So it follows that if we write the necessary interface by hand (since Java doesn't do it for us), that should solve the problem.

Adapter Pattern

- Writing code to produce an interface that we want from an interface that we have is an example of the *Adapter* design pattern.
- We can use adapters to adapt existing classes to produce the desired interface, with a relatively small amount of code.
- At least one interesting/surprising result

```
interface Generator<T> { T next(); }
interface Addable<T> { void add(T t); }

public class Fill2 {
    // Classtoken version:
    public static <T> void fill(Addable<T> addable,
        Class<? extends T> classToken, int size) {
        for(int i = 0; i < size; i++)
            try {
                addable.add(classToken.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }

    public static <T> void // Generator version
    fill(Addable<T> addable, Generator<T> generator, int
    size) {
        for(int i = 0; i < size; i++)
            addable.add(generator.next());
    }
}
```

```
// If you're adapting a base type, use composition.
// Make any Collection Addable using composition:
class AddableCollectionAdapter<T> implements Addable<T> {
    private Collection<T> c;
    public AddableCollectionAdapter(Collection<T> c) {
        this.c = c;
    }
    public void add(T item) { c.add(item); }
}

// Helper method for above, captures the type:
class Adapter {
    public static <T>
    Addable<T> collectionAdapter(Collection<T> c) {
        return new AddableCollectionAdapter<T>(c);
    }
}
```

```
// If you're adapting a specific type, use inheritance:
// Make a SimpleQueue Addable using inheritance:
class AddableSimpleQueue<T>
extends SimpleQueue<T> implements Addable<T> {
    public void add(T item) { super.add(item); }
}

// Another type of container created by someone else:
class RandomList<T> {
    private ArrayList<T> storage = new ArrayList<T>();
    private Random rand = new Random();
    public void insert(T item) { storage.add(item); }
    public T select() {
        return storage.get(rand.nextInt(storage.size()));
    }
}
```

```
// Abstract composition adapter:  
abstract class AddableAdapter<S, T>  
implements Addable<T> {  
    protected S seq;  
    public AddableAdapter(S sequence) { seq = sequence; }  
    public abstract void add(T item);  
}
```

```
class Coffee {}  
class Latte extends Coffee {}  
class Mocha extends Coffee {}  
class Cappuccino extends Coffee {}  
class Americano extends Coffee {}  
class Breve extends Coffee {}
```

```
// Generate different types of Coffee:
class CoffeeGenerator implements Generator<Coffee> {
    private Class[] types = { Latte.class, Mocha.class,
        Cappuccino.class, Americano.class, Breve.class, };
    private Random rand = new Random();
    public Coffee next() {
        try {
            return (Coffee)types[
                rand.nextInt(types.length)].newInstance();
            // Report programmer errors at runtime:
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
class Fill2Test {
    public static void main(String[] args) {
        // Adapt a Collection:
        List<Coffee> coffeeCarrier = new ArrayList<Coffee>();
        Fill2.fill(new
            AddableCollectionAdapter<Coffee>(coffeeCarrier),
            Coffee.class, 3);
        // Helper method captures the type:
        Fill2.fill(Adapter.collectionAdapter(coffeeCarrier),
            Latte.class, 2);
        for(Coffee c: coffeeCarrier)
            System.out.println(c);

        System.out.println("-----");
    }
}
```

```
// Use an adapted class:
AddableSimpleQueue<Coffee> coffeeQueue =
    new AddableSimpleQueue<Coffee>();
Fill2.fill(coffeeQueue, Mocha.class, 4);
Fill2.fill(coffeeQueue, Latte.class, 1);
for(Coffee c: coffeeQueue)
    System.out.println(c);

System.out.println("++++++++++++++++++++");
```

```
RandomList<Coffee> rl = new RandomList<Coffee>();  
// An on-the-fly adapter via an anonymous class:  
Addable<Coffee> ac =  
    new AddableAdapter<RandomList<Coffee>, Coffee>(rl) {  
        public void add(Coffee item) { seq.insert(item);  
        }  
    };  
Fill2.fill(ac, new CoffeeGenerator(), 7);  
for(int i = 0; i < 10; i++)  
    System.out.println(rl.select());  
}
```

A second example

- Try to create generic methods that operate on numbers
- Stymied because **Number** base class is so limited

```
import java.util.*;

public class
SequenceCalc<T extends Number, SEQ extends Iterable<T>> {
    private SEQ sequence;
    public SequenceCalc(SEQ seq) { sequence = seq; }
    public T sum() {
        Iterator<T> it = sequence.iterator();
        if(it.hasNext()) {
            // Numbers are read-only; don't need to clone
            T n = it.next();
            while(it.hasNext())
                n += it.next();
            return n;
        } else
            return null; // Or throw exception
    }
}
```

**Nope. Number interface doesn't
provide what we need.**

Solution

- Adaptation is provided by function objects
- Function objects are passed into generic methods as strategies (*Strategy* design pattern)

```
import java.math.*;
import java.util.*;
import java.util.concurrent.atomic.*;

// Different types of function objects:
interface Combiner<T> { T combine(T x, T y); }
interface UnaryFunction<R,T> { R function(T x); }
interface Collector<T> extends UnaryFunction<T,T> {
    T result(); // Extract result of collecting parameter
}
interface UnaryPredicate<T> { boolean test(T x); }
```

```
public class Functional {
    // Calls the Combiner object on each element to combine
    // it with a running result, which is finally returned:
    public static <T> T
    reduce(Iterable<T> seq, Combiner<T> combiner) {
        Iterator<T> it = seq.iterator();
        if(it.hasNext()) {
            T result = it.next();
            while(it.hasNext())
                result = combiner.combine(result, it.next());
            return result;
        }
        // If seq is the empty list:
        return null; // Or throw exception
    }
}
```

```
// Take a function object and call it on each object in
// the list, ignoring the return value. The function
// object may act as a collecting parameter, so it is
// returned at the end.
public static <T> Collector<T>
forEach(Iterable<T> seq, Collector<T> func) {
    for(T t : seq)
        func.function(t);
    return func;
}
```

```
// Creates a list of results by calling a function
// object for each object in the list:
public static <R,T> List<R>
transform(Iterable<T> seq, UnaryFunction<R,T> func) {
    List<R> result = new ArrayList<R>();
    for(T t : seq)
        result.add(func.function(t));
    return result;
}
```

```
// Applies a unary predicate to each item in a sequence,  
// and returns a list of items that produced "true":  
public static <T> List<T>  
filter(Iterable<T> seq, UnaryPredicate<T> pred) {  
    List<T> result = new ArrayList<T>();  
    for(T t : seq)  
        if(pred.test(t))  
            result.add(t);  
    return result;  
}
```

```
// To use the above generic methods, we need to create
// function objects to adapt to our particular needs:
static class IntegerAdder implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
        return x + y;
    }
}
static class
IntegerSubtractor implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
        return x - y;
    }
}
```

```
static class
BigDecimalAdder implements Combiner<BigDecimal> {
    public BigDecimal combine(BigDecimal x, BigDecimal y) {
        return x.add(y);
    }
}

static class
BigIntegerAdder implements Combiner<BigInteger> {
    public BigInteger combine(BigInteger x, BigInteger y) {
        return x.add(y);
    }
}

static class
AtomicLongAdder implements Combiner<AtomicLong> {
    public AtomicLong combine(AtomicLong x, AtomicLong y) {
        // Not clear whether this is meaningful:
        return new AtomicLong(x.addAndGet(y.get()));
    }
}
```

```
// Whatever an ulp is, make a UnaryFunction with it.
static class BigDecimalUlp
implements UnaryFunction<BigDecimal, BigDecimal> {
    public BigDecimal function(BigDecimal x) {
        return x.ulp();
    }
}

static class GreaterThan<T extends Comparable<T>>
implements UnaryPredicate<T> {
    private T bound;
    public GreaterThan(T bound) { this.bound = bound; }
    public boolean test(T x) {
        return x.compareTo(bound) > 0;
    }
}
```

```
static class MultiplyingIntegerCollector
implements Collector<Integer> {
    private Integer val = 1;
    public Integer function(Integer x) {
        val *= x;
        return val;
    }
    public Integer result() { return val; }
}
```

```
public static void main(String[] args) {
    // A very cool effect of generics, varargs & boxing:
    List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
    Integer result = reduce(li, new IntegerAdder());
    System.out.println(result);

    result = reduce(li, new IntegerSubtractor());
    System.out.println(result);

    System.out.println(
        filter(li, new GreaterThan<Integer>(4)));

    System.out.println(forEach(li,
        new MultiplyingIntegerCollector()).result());

    System.out.println(
        forEach(filter(li, new GreaterThan<Integer>(4)),
            new MultiplyingIntegerCollector()).result());
}
```

```
MathContext mc = new MathContext(7);
List<BigDecimal> lbd = Arrays.asList(
    new BigDecimal(1.1, mc), new BigDecimal(2.2, mc),
    new BigDecimal(3.3, mc), new BigDecimal(4.4, mc));
BigDecimal rbd = reduce(lbd, new BigDecimalAdder());
System.out.println(rbd);

System.out.println(filter(lbd,
    new GreaterThan<BigDecimal>(new BigDecimal(3))));

// Use the prime-generation facility of BigInteger:
List<BigInteger> lbi = new ArrayList<BigInteger>();
BigInteger bi = BigInteger.valueOf(11);
for(int i = 0; i < 11; i++) {
    lbi.add(bi);
    bi = bi.nextProbablePrime();
}
System.out.println(lbi);
```

```
BigInteger rbi = reduce(lbi, new BigIntegerAdder());
System.out.println(rbi);
// The sum of this list of primes is also prime:
System.out.println(rbi.isProbablePrime(5));

List<AtomicLong> lal = Arrays.asList(
    new AtomicLong(11), new AtomicLong(47),
    new AtomicLong(74), new AtomicLong(133));
AtomicLong ral = reduce(lal, new AtomicLongAdder());
System.out.println(ral);

System.out.println(
    transform(lbd, new BigDecimalUlp()));
}
}
```

Epilog

- I've demonstrated the problem of trying to apply Java Generics across types, because of the lack of latent typing
- And shown the solution:
 - Write the interface
 - Create an adapter for each new class that you want to apply the generic to
- But...

What is generic programming?

■ This?

1) Define generic f()

2) Use it:

```
f(aType1);
```

```
f(aType2);
```

```
f(aType3);
```

...

■ Or this:

1) Define interface

2) Define generic f()
that uses interface

3) For each new type
you want to apply f()
to, write adapter

4) Call f() with adapter

Other Generic Limitations

- You can't make an instance of a Generic param:

```
class Foo<T> {  
    T x = new T(); // Oops! Illegal.  
}
```

- In C++ (& C#) this is natural and straightforward:

```
template<class T> Foo {  
    T x;  
    T* y;  
public:  
    Foo() { y = new T(); }  
};
```

Factories

- The solution in Java is to pass in a factory object, and use that to make the object
- A convenient factory object is just the class object:

```
class Foo<T> {
    T x;
    public Foo(Class<T> kind) {
        try {
            x = kind.newInstance();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class Employee {}

public class InstantiateGenericType {
    public static void main(String[] args) {
        Foo<Employee> fe =
            new Foo<Employee>(Employee.class);
        System.out.println("Foo<Employee> succeeded");
        try {
            Foo<Integer> fi =
                new Foo<Integer>(Integer.class);
        } catch(Exception e) {
            System.out.println("Foo<Integer> failed");
        }
    }
}
```

Some folks say “bad”

- Loss of compile-time checking, can fail at runtime
- Instead, use an explicit factory and constrain the type so that it only takes a class that implements this factory:

```
interface FactoryI<T> {
    T create();
}

class Foo2<T> { // Generic class
    private T x;
    // Generic method:
    public <F extends FactoryI<T>> Foo2(F factory) {
        x = factory.create();
    }
    // ...
}
```

```
class IntegerFactory implements FactoryI<Integer> {
    public Integer create() {
        return new Integer(0);
    }
}

class Widget {
    public Widget() {}
    public static class Factory implements FactoryI<Widget> {
        public Widget create() {
            return new Widget();
        }
    }
}

public class FactoryConstraint {
    public static void main(String[] args) {
        new Foo2<Integer>(new IntegerFactory());
        new Foo2<Widget>(new Widget.Factory());
    }
}
```

Can't create arrays of generics

- Won't work. You have to use containers (OK much of the time, but not always)

```
Future<String>[] results =  
    new Future<String>[SZ]; // Nope!
```

- Trivial in C++

Can't use primitive types in generics

■ Trivial in C++/C#:

```
template<class T> class AllTypes {};  
  
class ClassType {};  
  
int main() {  
    AllTypes<ClassType> a;  
    AllTypes<int> b;  
};
```

```
import java.util.*;

interface Generator<T> { T next(); }

/* Can't do this at all:
class Test implements Generator<int> {
    private int i = 0;
    public int next() { return i++; }
} */

class RandIntGenerator implements Generator<Integer> {
    private int mod = 10000;
    private static Random r = new Random();
    public RandIntGenerator() {}
    public RandIntGenerator(int modulo) { mod = modulo; }
    public Integer next() { return r.nextInt(mod); }
}
```

```
class ArrayUtilities {
    // Fill an array using a generator:
    public static <T> void fill(T[] a, Generator<T> gen) {
        fill(a, 0, a.length, gen);
    }
    public static <T> void
    fill(T[] a, int from, int to, Generator<T> gen) {
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }
}

public class PrimitiveGenericTest {
    public static void main(String[] args) {
        RandIntGenerator gen = new RandIntGenerator(1000);
        Integer[] a = new Integer[100];
        ArrayUtilities.fill(a, gen);
        int[] b = new int[100];
        // Autoboxing won't save you here:
        // ArrayUtilities.fill(b, gen); // Won't compile
    }
}
```

Can't overload on generics

- Won't work:

```
public class ManipulateLists<W,T> {  
    void manipulate(List<T> v) { }  
    void manipulate(List<W> v) { }  
}
```

Can't implement parameterized interfaces differently

```
class A implements Comparable<A> {  
    public int compareTo(A other) {...}  
}  
class B extends A implements Comparable<B> {  
    public int compareTo(B other) {...}  
}
```

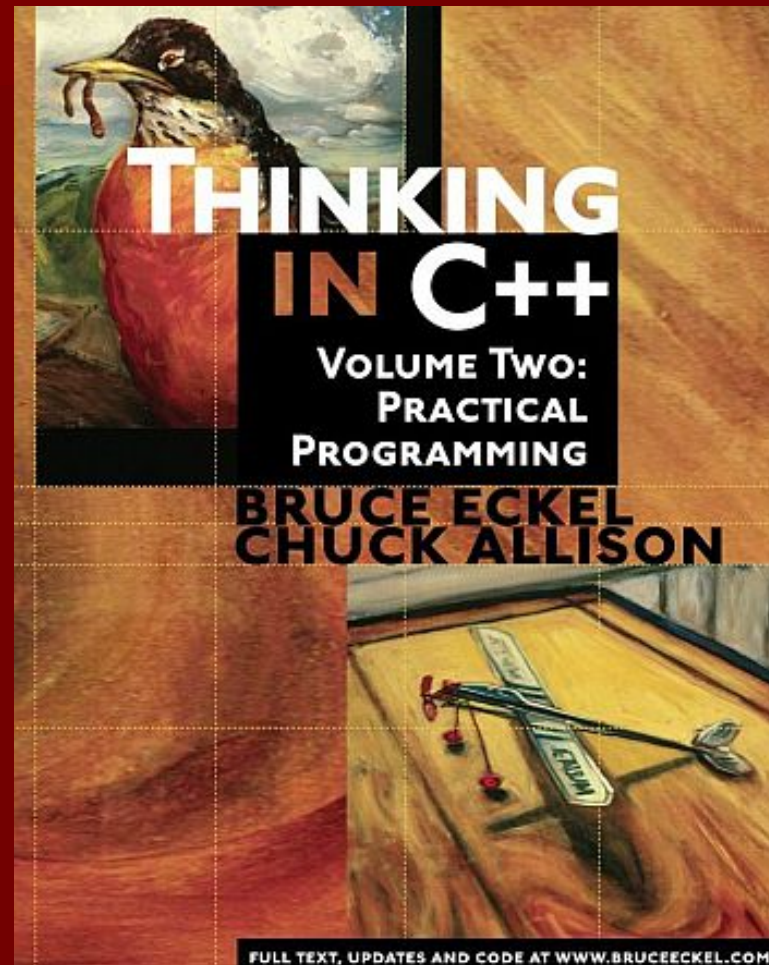
- Erasure reduces this to “implements Comparable” in both cases

Varargs

- Can't use varargs with a parameterized type

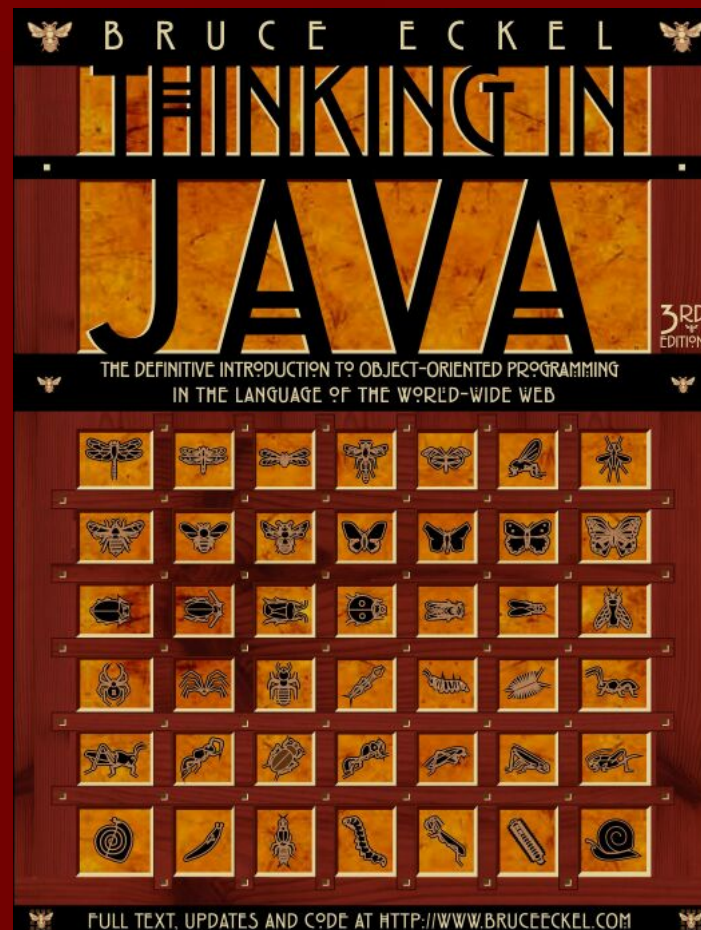
Latest book

- Intermediate & Advanced topics
 - Templates
 - STL
 - Design Patterns
 - Multithreading
- Downloadable from web site



Working On

- Thinking in Java, 4th Edition
- Rewritten for JDK 5
- To be in print February



Will someday finish...

- Thinking in Patterns: Problem-Solving Techniques using Java
- Trying to make it a fresh and insightful look at Design Patterns, taking Java into account



Questions

