

MATERIALS & SHAPES

Ben Greenman
November 25, 2014

OUTLINE

- *Getting F-Bounded Polymorphism Into Shape*
 - with Fabian Muehlboeck and Ross Tate, PLDI 2014
 - and the Ceylon team
- plus some more recent developments

OUTLINE

- *Getting F-Bounded Polymorphism Into Shape* (PLDI '14)

This is an idea paper!

- Main technical result (decidable subtyping) is NOT new!
- But the ideas and insights are.

GOALS

1. Explain the big discovery
2. Convince you that we've acted sensibly

THE PROBLEM

- Type-safe equality in object-oriented languages
 - $42 == 42.00$  Cast to common super
 - $42 == "forty-two"$  Type error
 - $\lambda x. 42 == \lambda x. 42$  Type error, undecidable*

THE PROBLEM

- Type safe equality
 - `List<T>`
 - `HashMap<T>`
 - and so on ...

The state of the art? `Object.equals()`

java.lang

Class Object

java.lang.Object

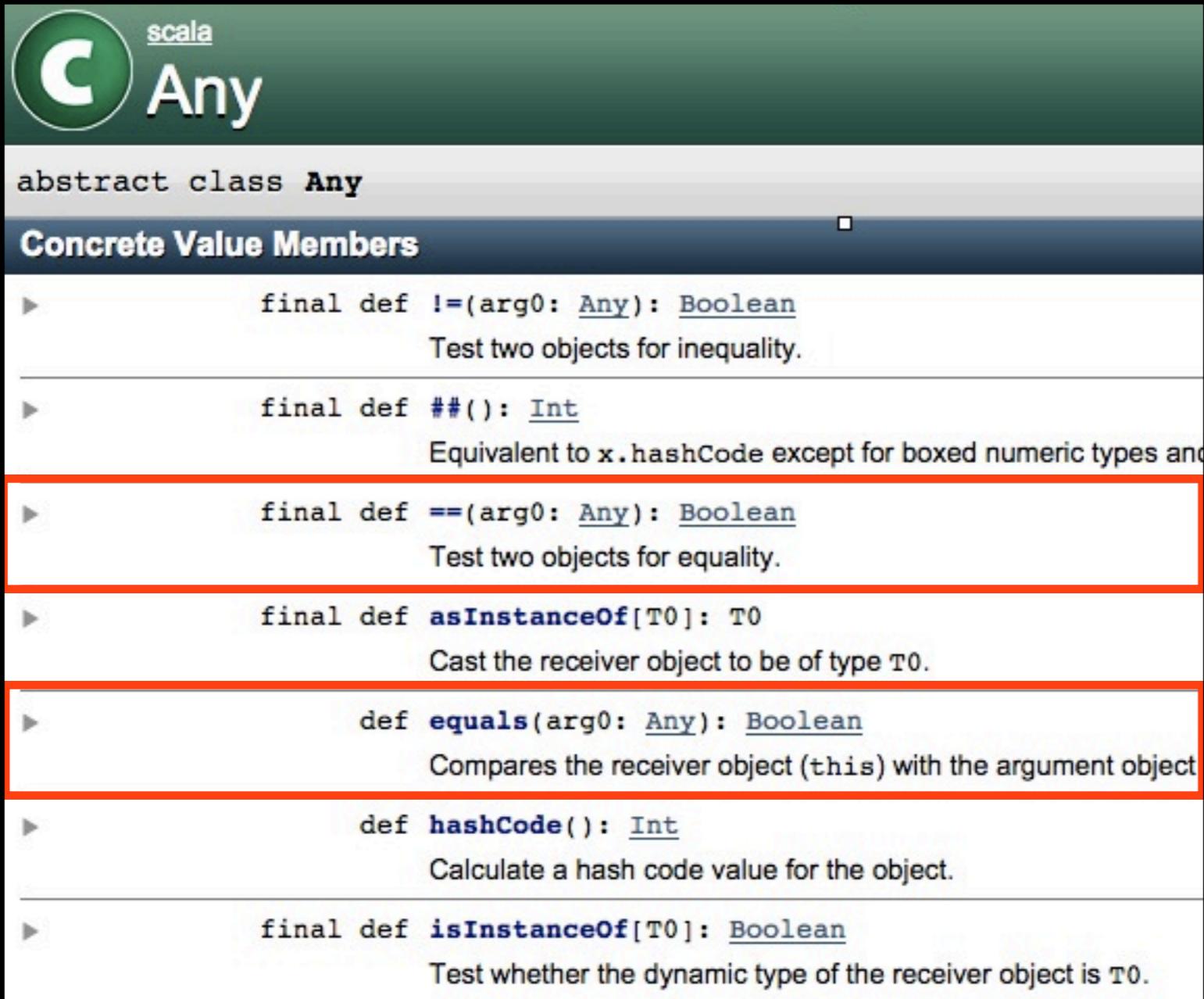
public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, and class objects are instances of class Object.

Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
protected Object	<code>clone()</code> Creates and returns a copy of this object.	
boolean	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this one.	
protected void	<code>finalize()</code> Called by the garbage collector on an object when garbage collection	
Class<?>	<code>getClass()</code> Returns the runtime class of this Object.	

The state of the art? `Object.equals()`



The screenshot shows the Scala documentation for the `Any` class. The title bar includes the Scala logo and the word `Any`. The main content area is titled `Concrete Value Members` and lists several methods:

- `!=`: A method that tests two objects for inequality.
- `##`: A method that calculates a hash code for the object, equivalent to `x.hashCode` except for boxed numeric types and strings.
- `==`: A method that tests two objects for equality.
- `asInstanceOf[T0]`: A method that casts the receiver object to be of type `T0`.
- `equals`: A method that compares the receiver object (`this`) with the argument object.
- `hashCode`: A method that calculates a hash code value for the object.
- `isInstanceOf[T0]`: A method that tests whether the dynamic type of the receiver object is `T0`.

The `equals` and `hashCode` methods are highlighted with red boxes.

WHAT'S WRONG?

- Does not scale.
 - Should there be an `Object.compareTo()` ?
- Masks errors that the static type-checker could find.
- The concept of "equality" is not defined for all objects.
- Requires dynamic dispatch

// Typical implementation

```
class Foobar extends Object {  
    boolean equals(Object obj) {  
        if (obj instanceof Foobar) {  
            Foobar that = (Foobar) obj;  
            /* Actually compare `this`  
             * and `that` */  
        }  
        return false;  
    }  
}
```

✗ Wrong arg. type

✗ Dynamic check

✗ Run-time cast

✗ Lots of boilerplate

// It just gets worse

- `instanceof` checks show up everywhere
 - +300 uses in Java Collections library
 - Repetitive, many opportunities for bugs

```
class BinaryTree<T> {  
  
    boolean contains(T elem) {  
        if (elem instanceof Comparable) {  
            /* Implement me! */  
        }  
        return false;  
    }  
  
    void remove(T elem) {  
        if (elem instanceof Comparable) {  
            /* Implement me! */  
        }  
    }  
}
```

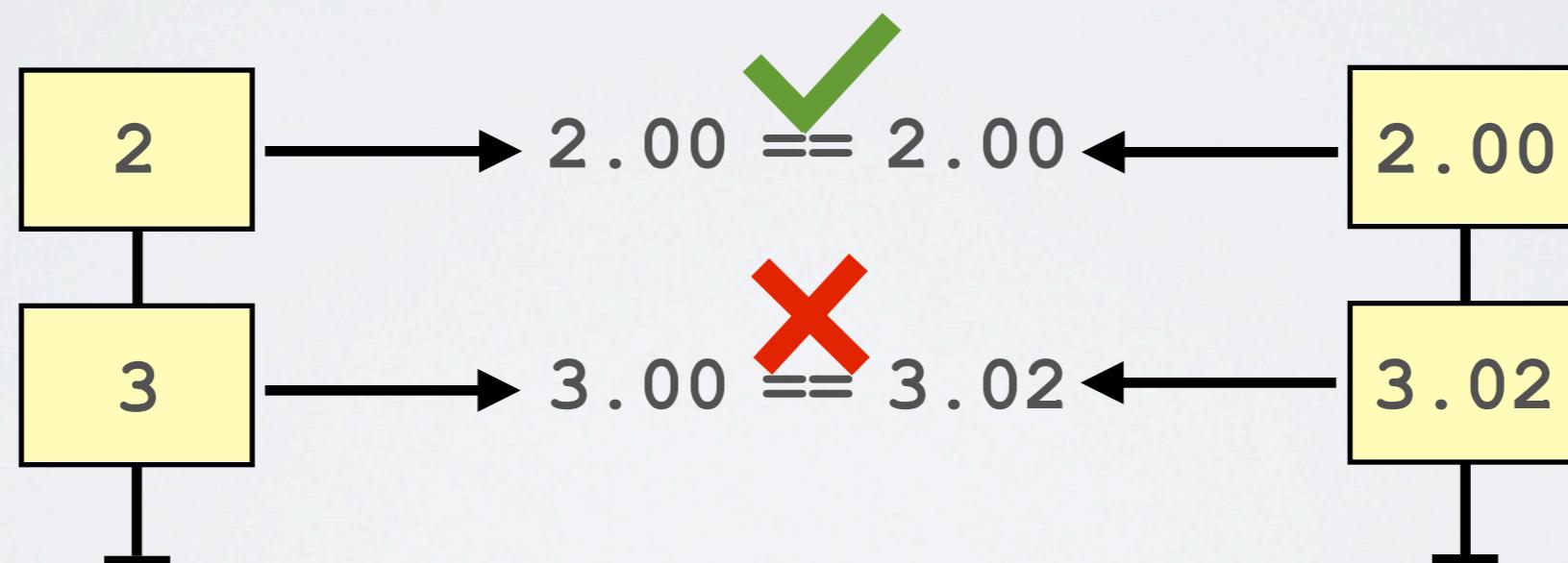
We can do better!

- Ideally, declare an **interface**
 - **Equatable<T>** { boolean **equalTo(T that)** ; }
- Replace **instanceof** and casts with F-Bounded polymorphism
 - **BinaryTree<T extends Equatable<T>>** { . . . }

An example: List

- Two lists are equal if their elements are pointwise equal.

`List<T> extends Equatable<List<Equatable<T>>>`



`List<Integer>`

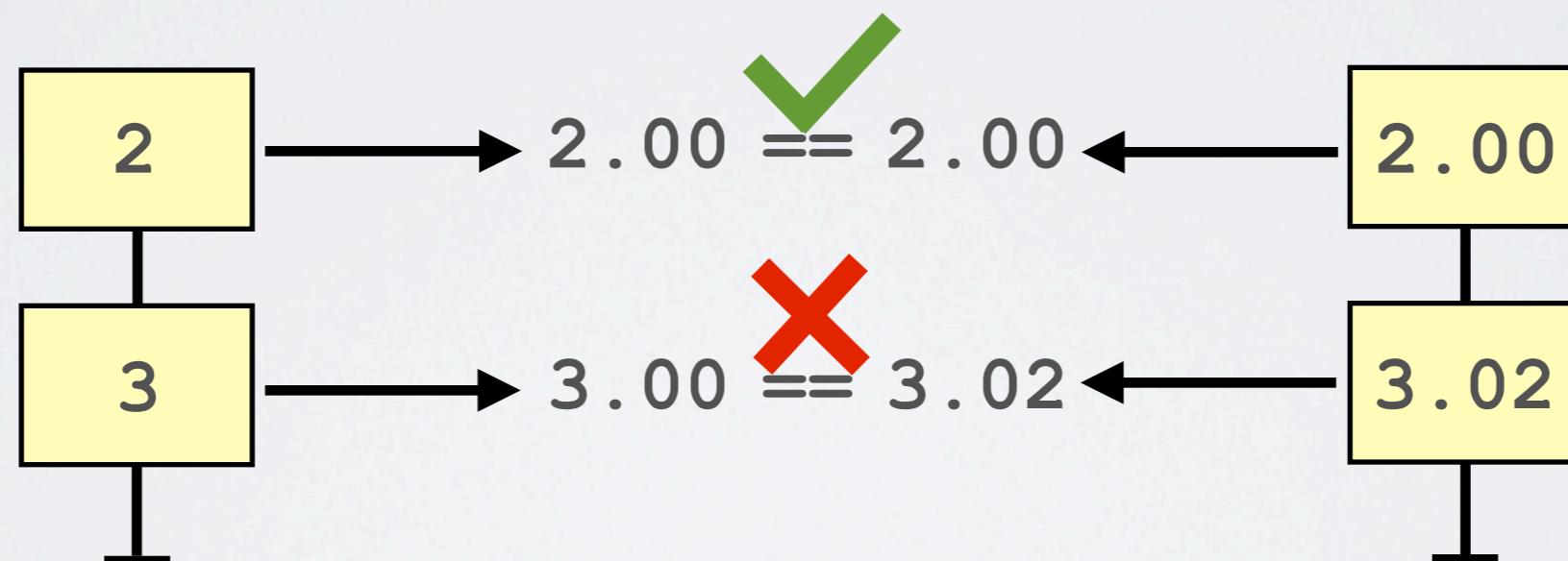
`List<Double>`

An example: List

- Two lists are equal if their elements are pointwise equal.

Almost!

`List<T> extends Equatable<List<Equatable<T>>>`



`List<Integer>`

`List<Double>`

VARIANCE

- Read-only types are covariant (**out**, **+**, **extends**, ...)
 - A **List<Integer>** can safely be treated as a **List<Double>**
- Write-only types are contravariant (**in**, **-**, **super**, ...)
 - A **Consumer<Animal>** can be treated as a **Consumer<Cat>**
- Read-Write types are invariant
 - An **Array<String>** should contain exactly **Strings**

VARIANCE

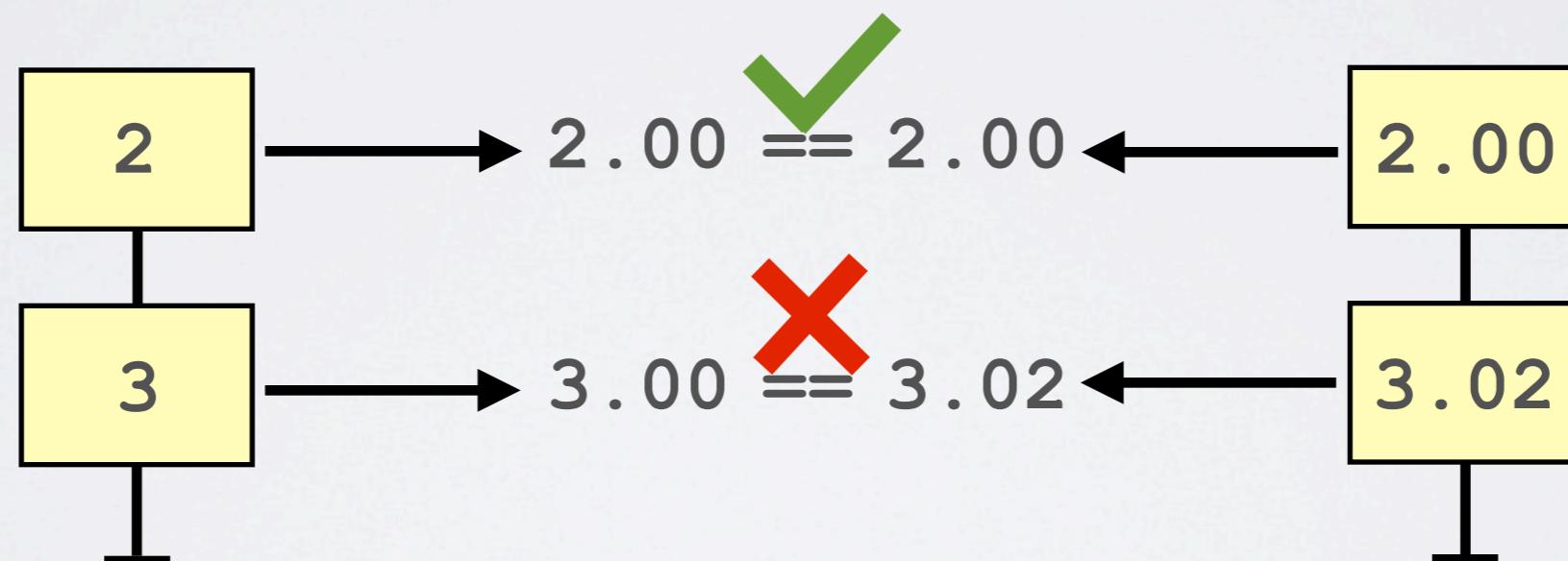
```
class Adult {}  
class Baby extends Adult {}  
  
public class ArrayHack {  
    public static void main(String[] args) {  
        Baby[] crib = new Baby[1];  
        Adult[] house = crib;  
        house[0] = new Adult();  
        System.out.printf("Success\n");  
    }  
}
```

Exception in thread "main"
java.lang.ArrayStoreException: Adult

An example: List

- Two lists are equal if their elements are pointwise equal.

`List<T> extends Equatable<List<Equatable<T>>>`

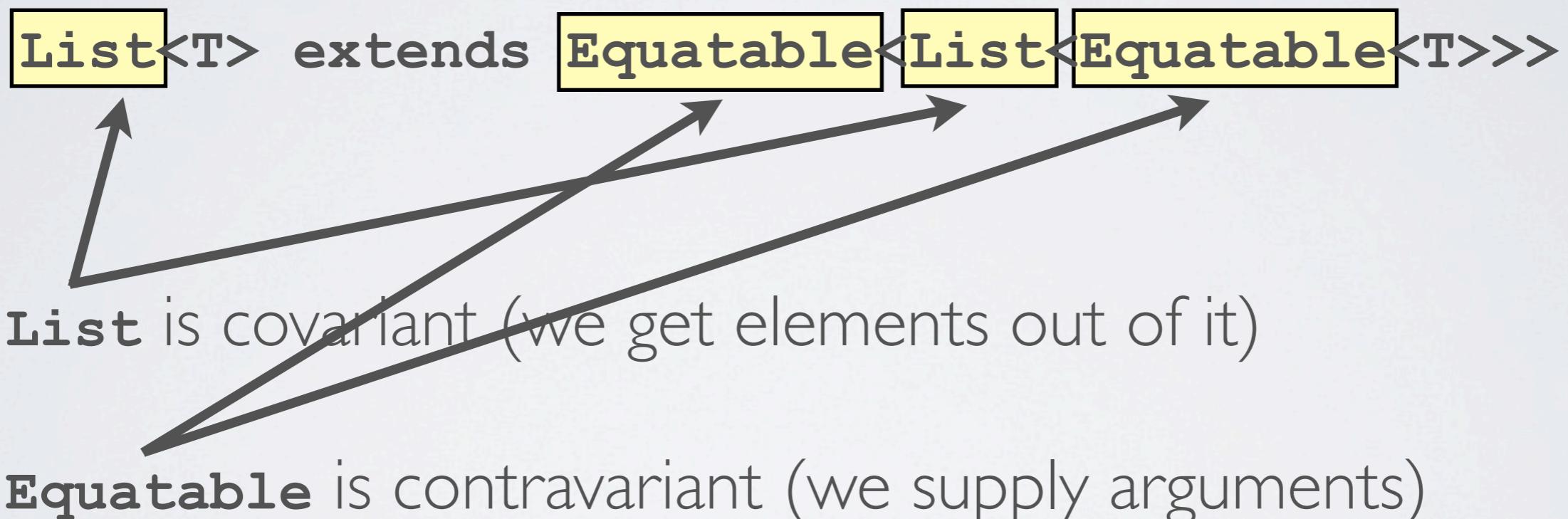


`List<Integer>`

`List<Double>`

An example: List

- Two lists are equal if their elements are pointwise equal.



An example: List

- Two lists are equal if their elements are pointwise equal.

List<T> extends Equatable<List<Equatable<T>>>

This actually works!

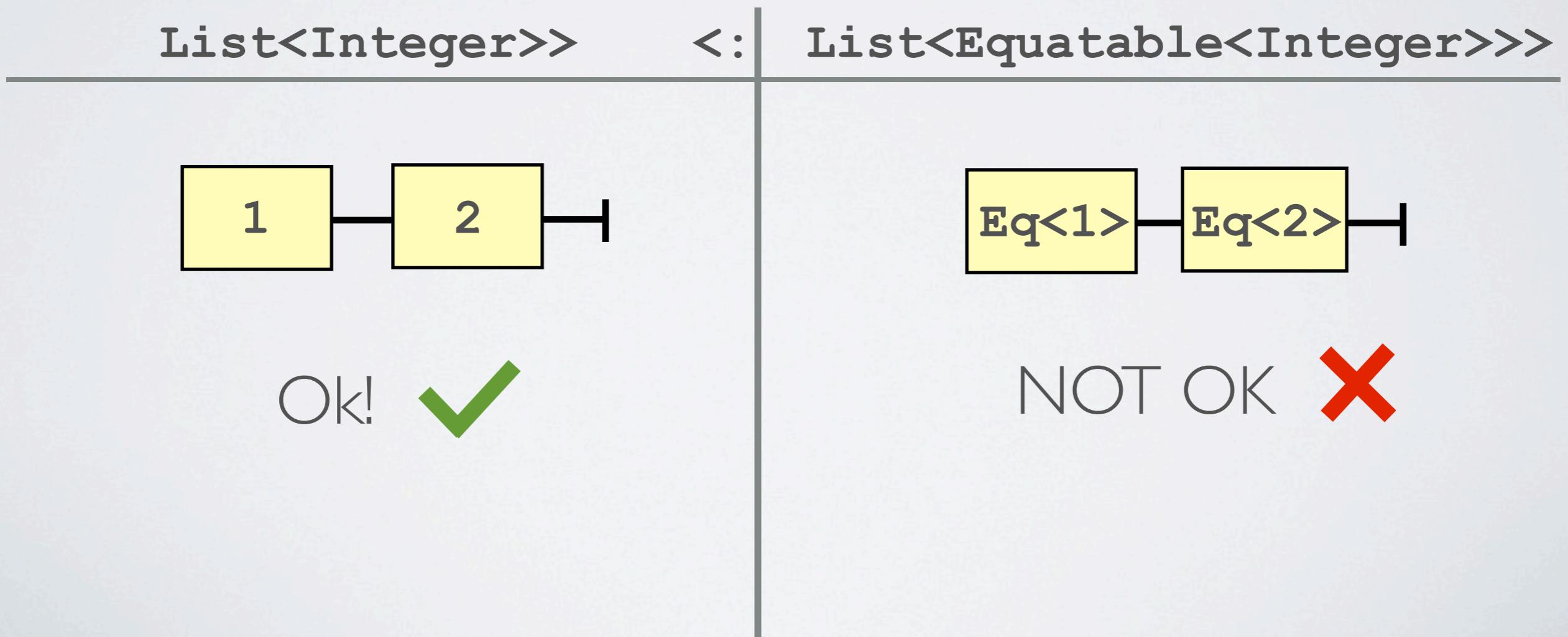
THE BIG DISCOVERY

- Backstory: the Ceylon team refused to `Object.equals()`
- An "anonymous researcher" suggested the above solution
- Ceylon's response:

NO.

THE BIG DISCOVERY

- "A `List<Equatable<T>>` is nonsense!"
 - Lists contain data, but **Equatable** is an abstract concept.



THE BIG DISCOVERY

- "A `List<Equatable<T>>` is nonsense!"
- Lists contain data, but **Equatable** is an abstract concept.

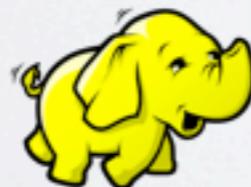
Equatable is a constraint on **Integers**

Integers are a valid instantiation for **List<T>**

You never want a "list of constraints"

EXPERIMENT

- Ceylon is only one project. We weren't convinced.
- Surveyed 60 Open-Source Java projects
 - ~13.5 million lines of code (avg. 242,113 med. 60,062)
 - ~100,000 classes (avg. 1,962 med. 487)
 - ~10,000 interfaces (avg. 202 med. 41)



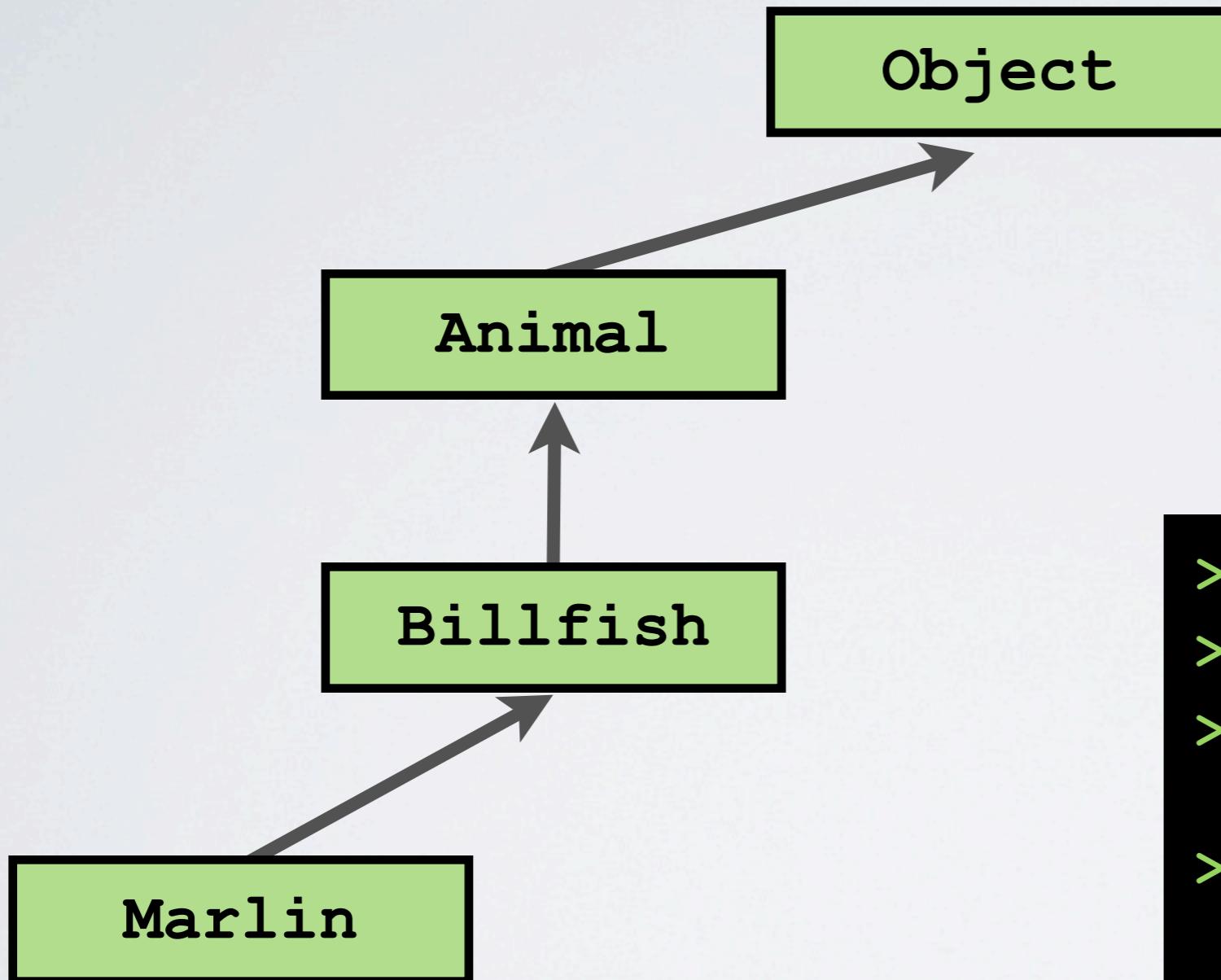
EXPERIMENT

You never want a "list of constraints"



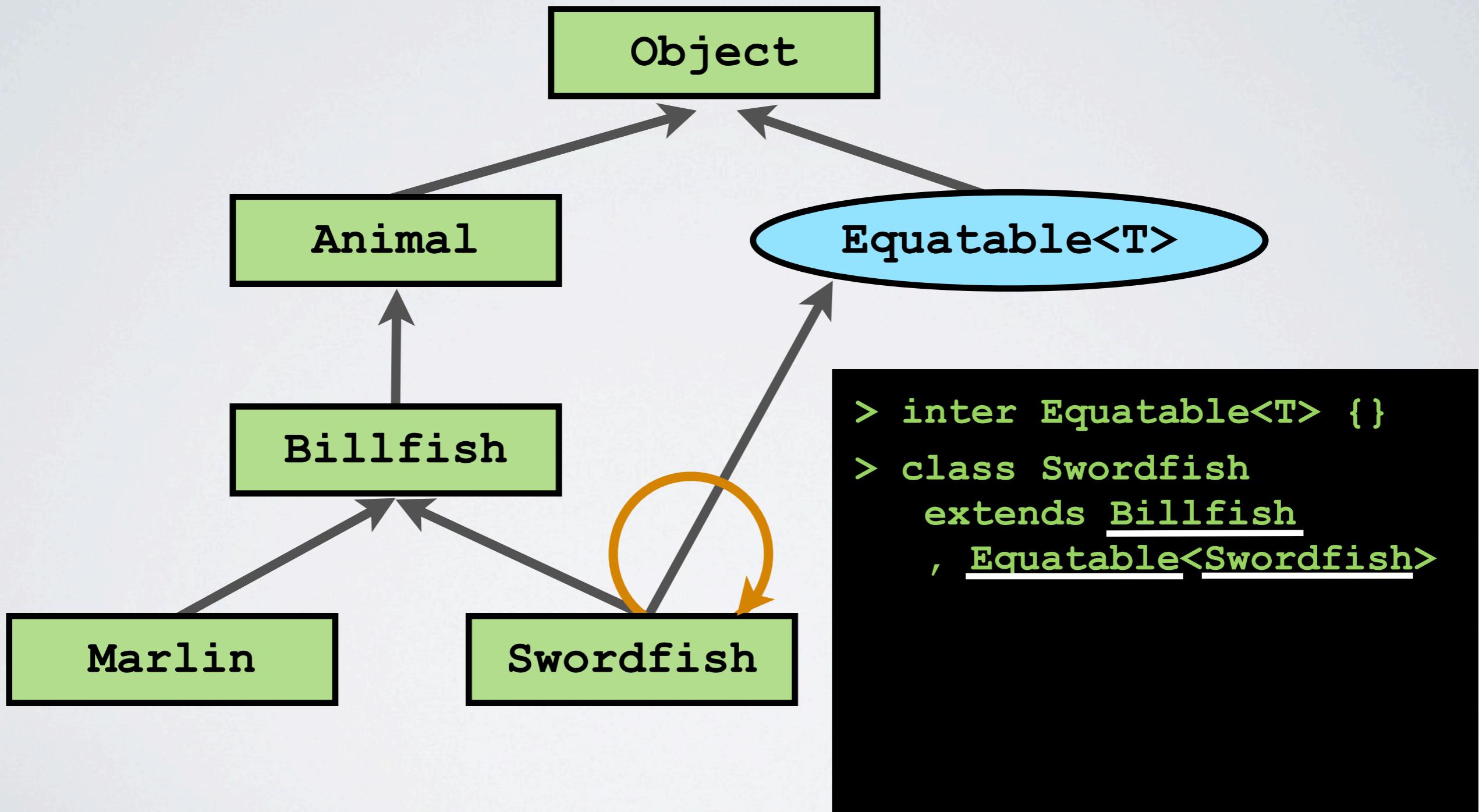
- Types like **Equatable<Integer>** were never used as:
 - Type Parameters
 - Function arguments or return types
 - Local variables or fields

What is a "type like" `Equatable<Integer>` ?

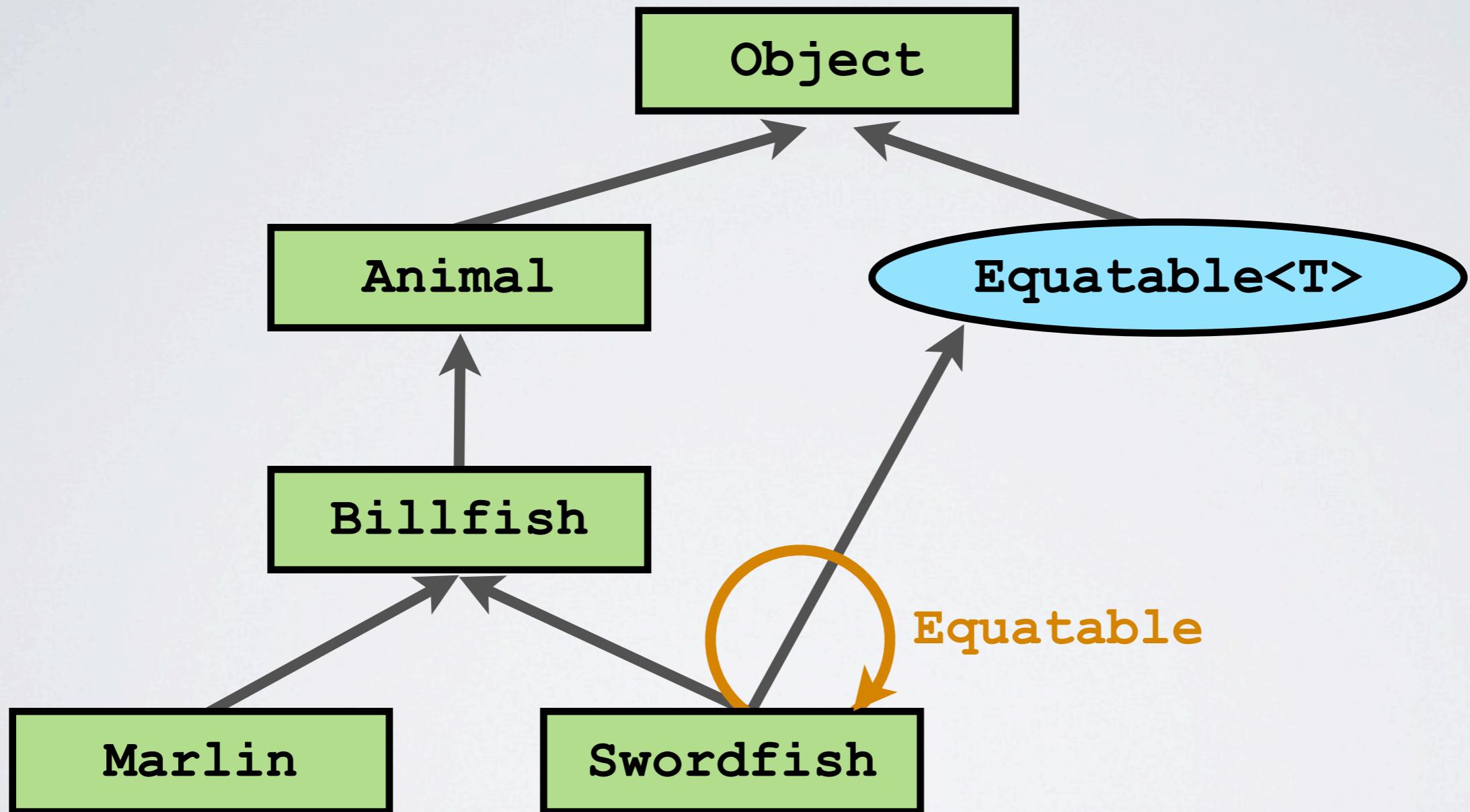


```
> class Object {}  
> class Animal {}  
> class Billfish  
    extends Animal {}  
> class Marlin  
    extends Billfish {}
```

What is a "type like" `Equatable<Integer>` ?



What is a "type like" `Equatable<Integer>` ?



EXPERIMENT

(more precisely)

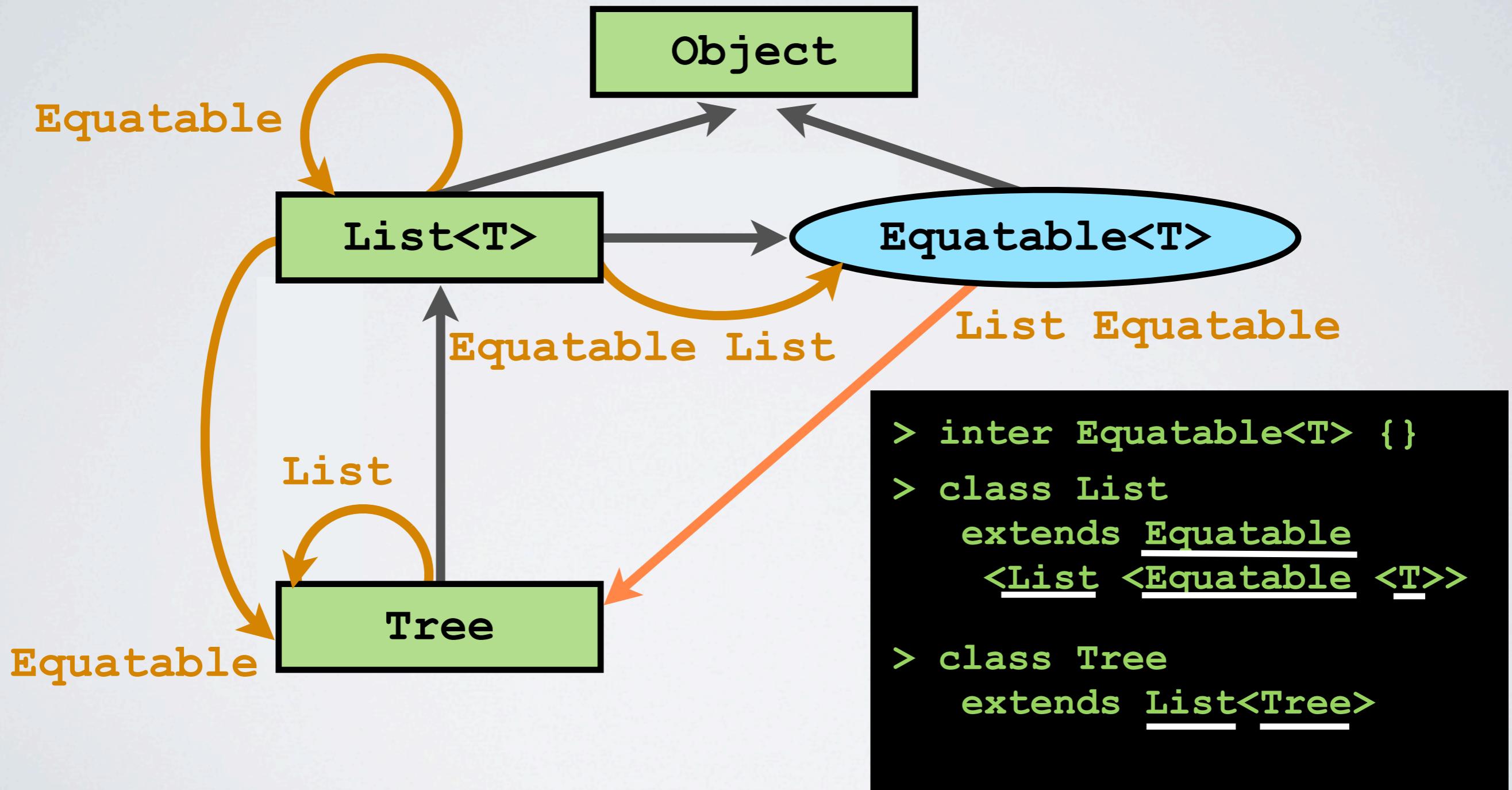
- Parameterized types used to complete cycles in the inheritance hierarchy were never used as:
 - Type Parameters
 - Function arguments or return types
 - Local variables or fields

RECAP

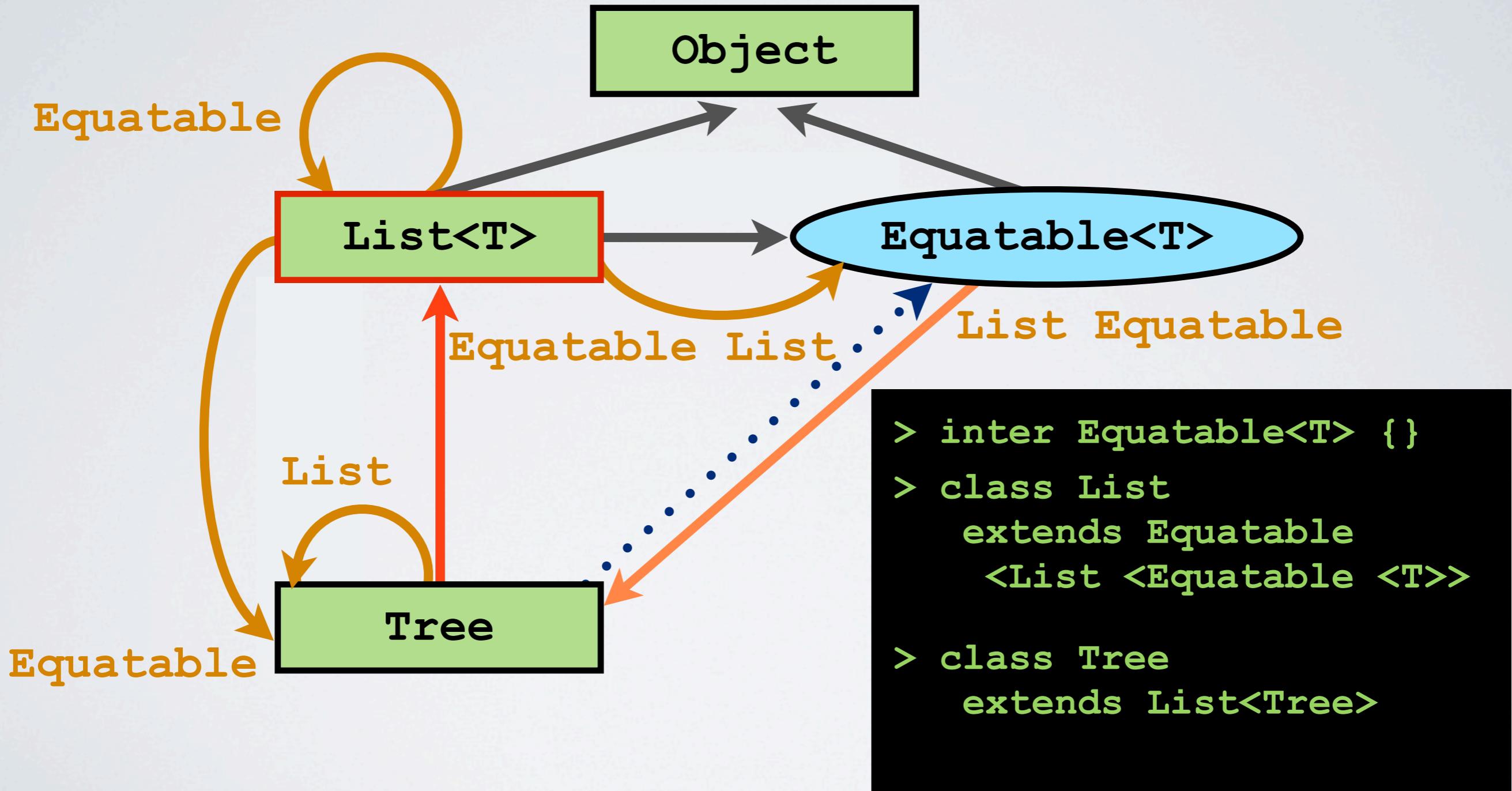
1. The problem: type-safe equality
2. Proposed solution: **Equatable** and F-Bounded Polymorphism
3. Strong Reject from industry
4. **Equatable** is a constraint, and causes cyclic inheritance

Next Up: the research perspective

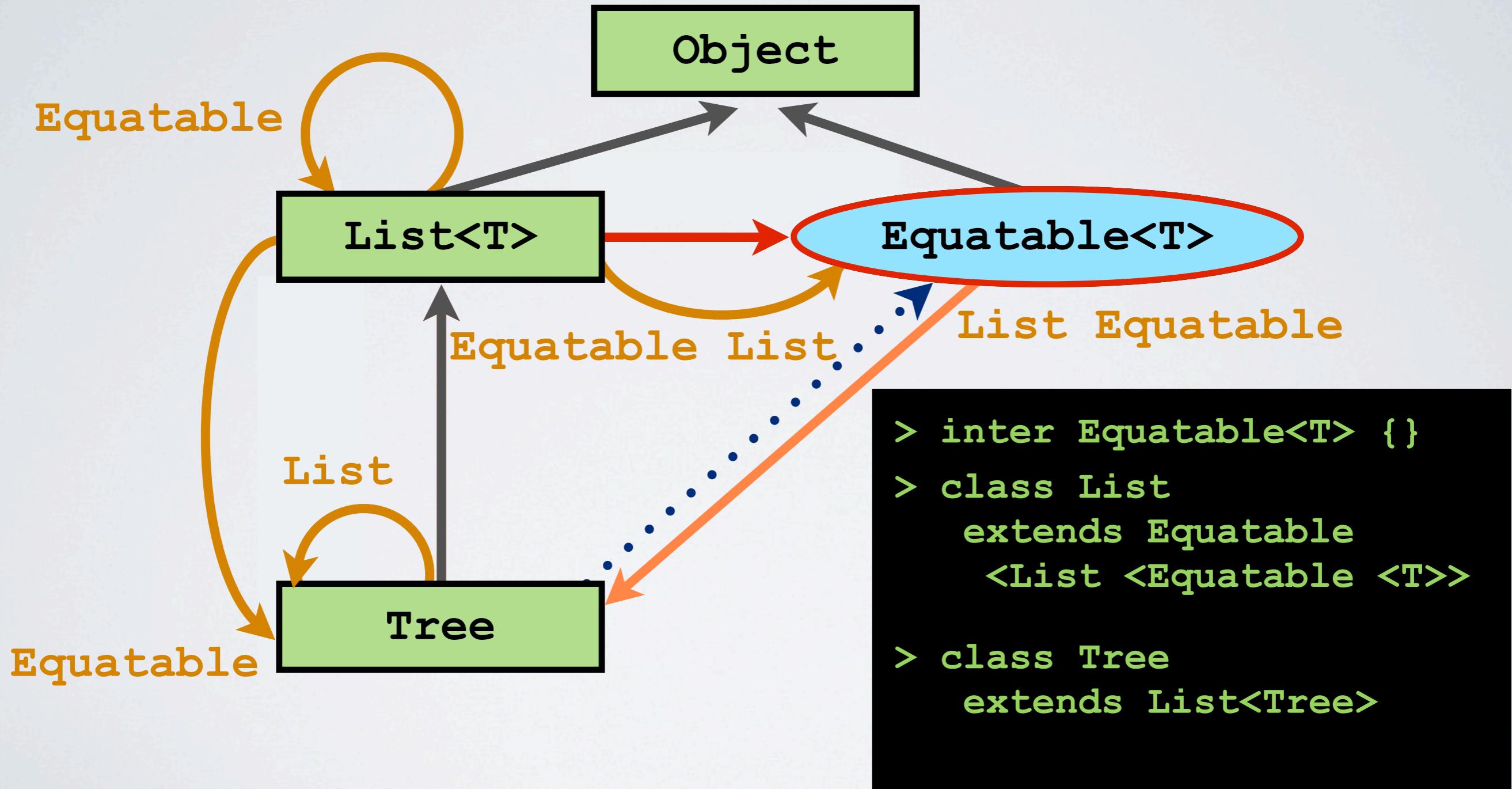
The problem with `Equatable<List<...>>`

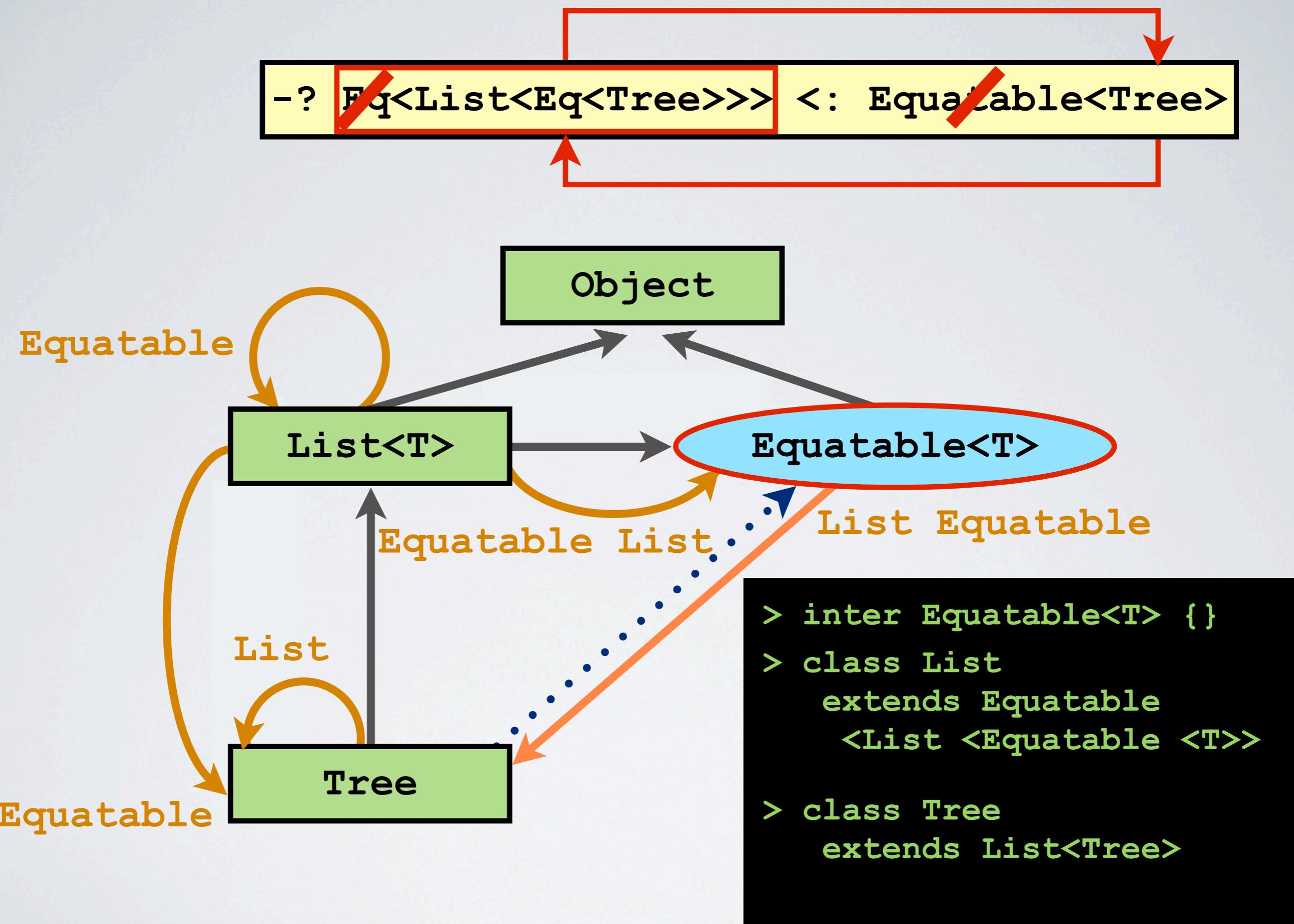


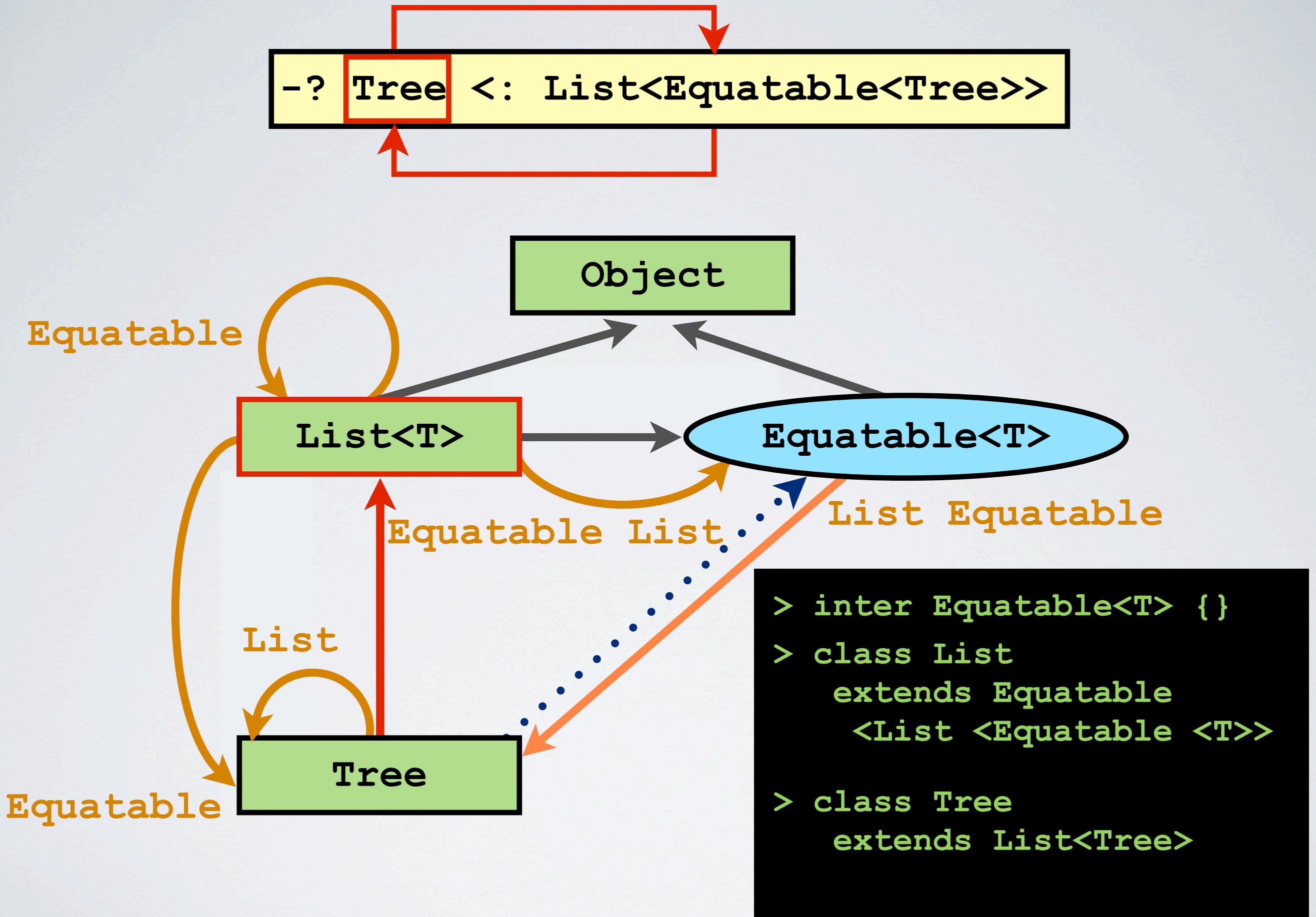
-? Tree <: Equatable<Tree>



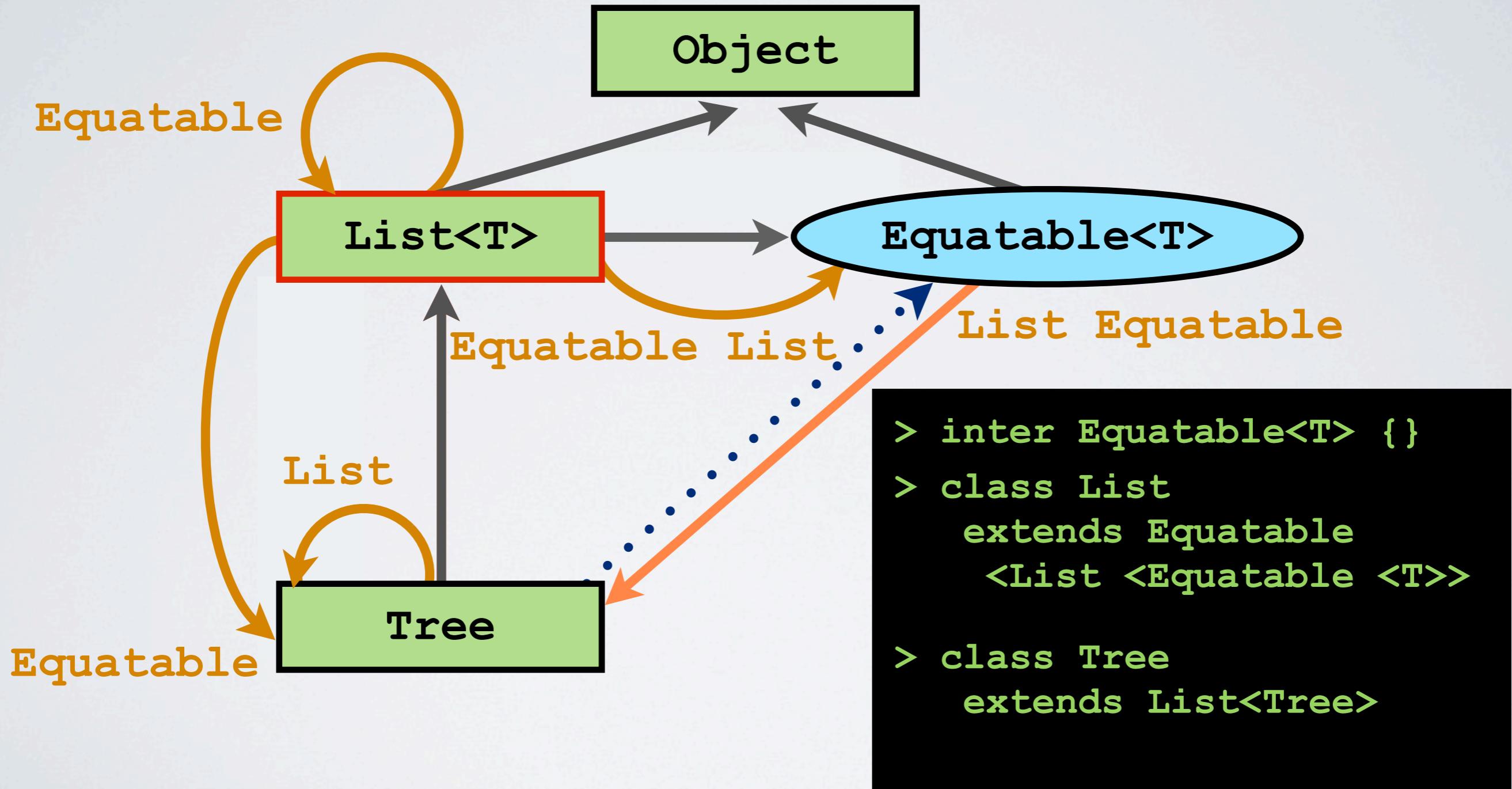
-? List<Tree> <: Equatable<Tree>





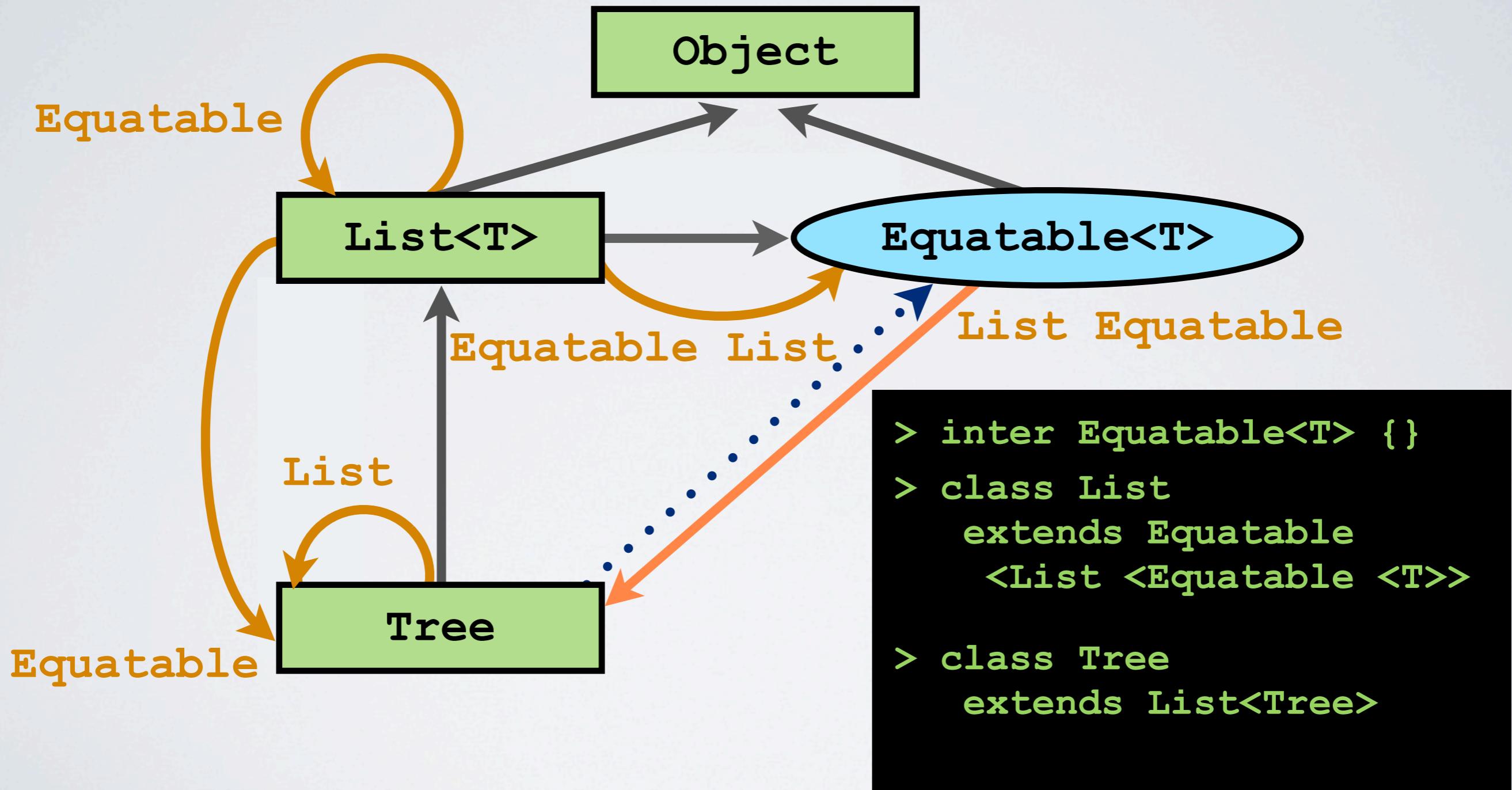


-? ~~List<Tree> <: List<Equatable<Tree>>~~



-? Tree <: Equatable<Tree>

Cycle!



PRIOR WORK

- *On the Decidability of Nominal Subtyping with Inheritance*
 - Andrew Kennedy & Benjamin Pierce, FOOL 2007
- The general problem is undecidable
- Can recover decidability by removing either:
 - I. Contravariance
 2. Expansive Inheritance
 3. Multiple Instantiation Inheritance*

PRIOR WORK

I. Remove Contravariance

For all types $C<*>$, $D<*>$, and all values x , y :

$C<x>$ is a subtype of $D<y>$

if

x is a subtype of y

PRIOR WORK

2. Remove Expansive Inheritance

Suppose $C<X>$ inherits $D<Y>$,

Either $X=Y$

or

X does not appear in Y

(Y is no "larger" than X)

PRIOR WORK

3. Remove Multiple Instantiation Inheritance*

For all types c , $D<*>$, and all values x, y :

c cannot inherit
both
 $D<x>$ and $D<y>$

* All expansive-recursive type parameters must be invariant and linear

PRIOR WORK

- *Taming Wildcards in Java's Type System*
 - Ross Tate, Alan Leung, Sorin Lerner, PLDI 2011

No nested contravariance in:
inheritance clauses
or
type parameters

`List<T> extends Equatable`
`<List <Equatable <T>>`

- ✗ Contravariance
- ✗ Nested Contravariance
- ✗ Expansive Inheritance



- ✗ Bad design

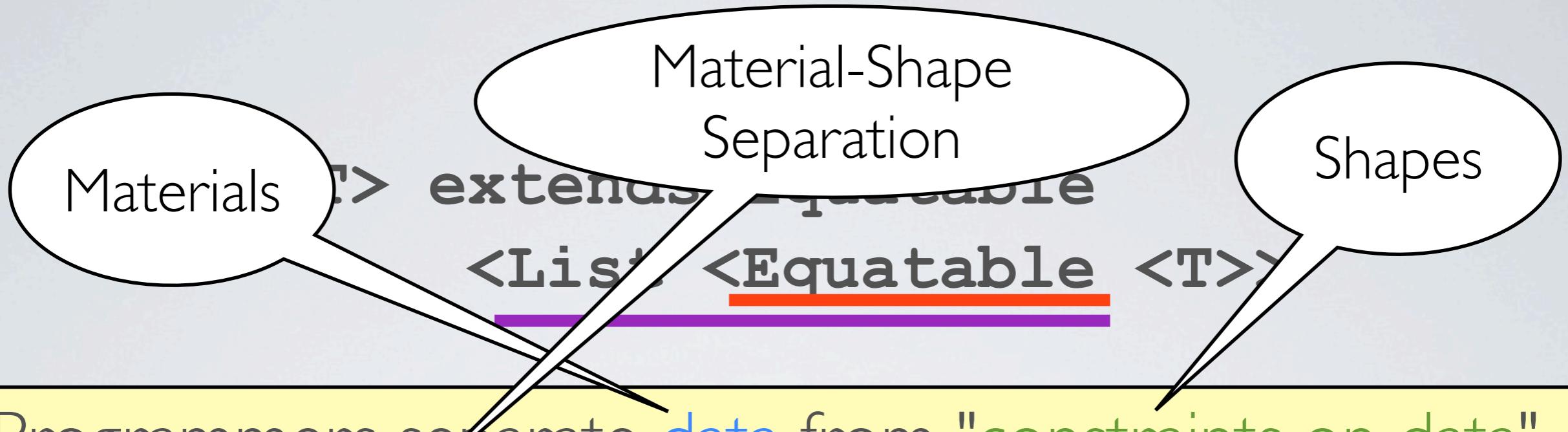


`List<T> extends Equatable`
`<List <Equatable <T>>`

Programmers separate `data` from "constraints on data".
This separation leads to decidable subtyping.

✗ Nested Contravariance ⊂ ✗ Bad design





Programmers separate **data** from "constraints on data".
This separation leads to decidable subtyping.

✗ Nested Contravariance ⊂ ✗ Bad design



Programmers separate **data** from "constraints on data".
This separation leads to decidable subtyping.

Materials

- **Object**
- **List<T>**
- **Swordfish**

Cycle-free inheritance

Shapes

- **Equatable<T>**
- **Cloneable<T>**
- **Addable<T>**

Never used as type parameters

SUMMARY

- While studying type-safe equality, we found a strange pattern
 - **Equatable**, **Comparable**, **Hashable** are different!
- Following this pattern intuitively gives decidable subtyping
- These **Shapes** describe the structure and constraints of data
- In contrast, **Materials** are the data used and exchanged

MATERIALS & SHAPES

SUB-GOALS

1. Outline proof of decidable subtyping
2. Type equality, decidable joins
3. Conditional inheritance
4. Shape shifters

DECIDABLE SUBTYPING

- Strategy: define a well-founded measure on judgments $x <: y$
- Key idea: inheritance never introduces new shapes
- We use a two-part measure:
 - The number of shapes appearing in each type
 - The maximum number of proof steps until the next shape

TYPE EQUALITY

- Suppose the type system has intersection types, **X&Y**
- Is **List<X&Y>** equal to **List<Y&X>** ? (It should be!)
 - Not true in Java
 - Not true using Kennedy & Pierce's technique
 - Not true using Tate et al.'s technique

TYPE EQUALITY

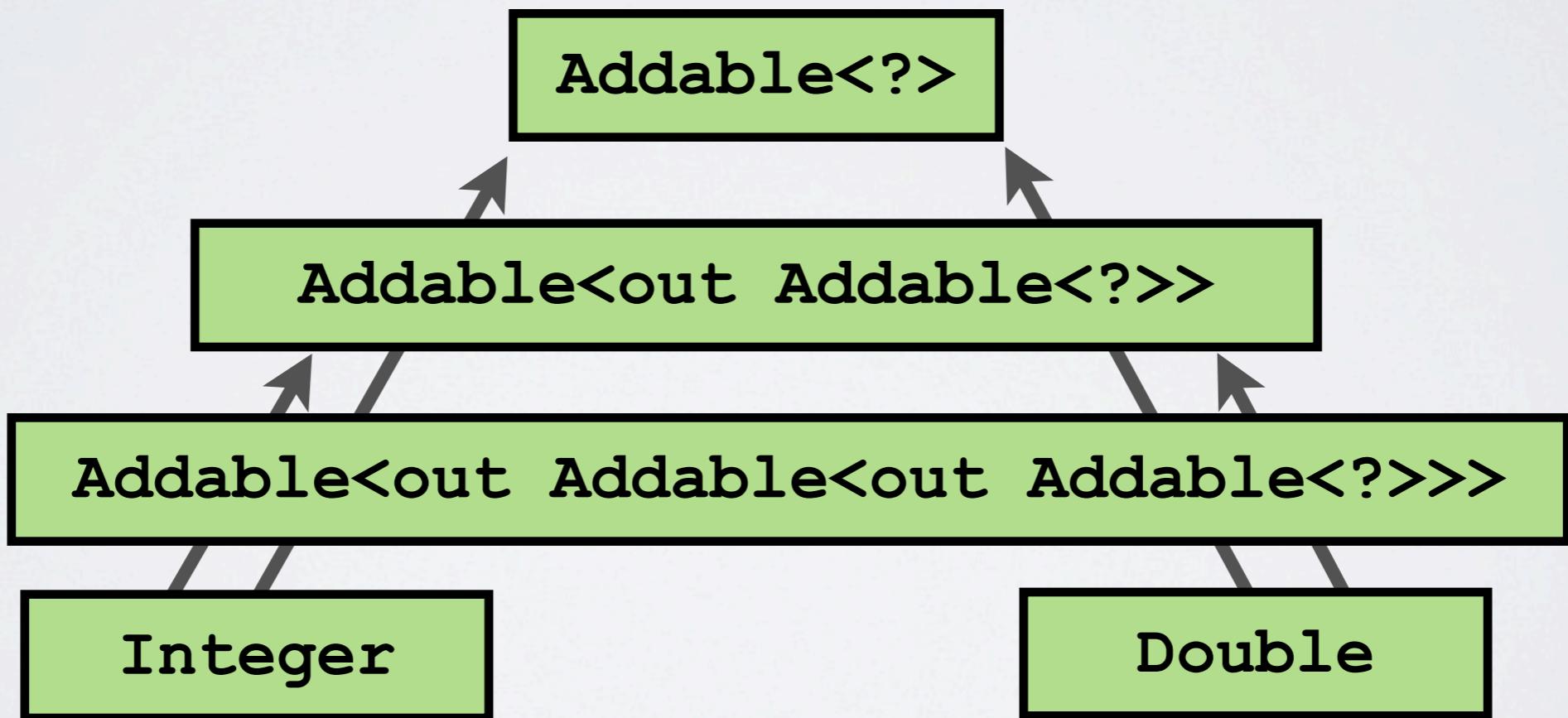
- Our subtyping algorithm only depends on recursion
 - Never uses syntactic equivalences
- We get equality for free: $(A = B)$ iff $(A <: B \text{ and } B <: A)$

JOINS

- $\mathbf{A} \sqcup \mathbf{B}$ is the least common supertype of \mathbf{A} and \mathbf{B}
- Useful for type-checking conditional statements.
 - **if** (C) **then** \mathbf{A} **else** \mathbf{B} has type $\mathbf{A} \sqcup \mathbf{B}$
- In many languages, arbitrary joins do not exist

JOINS

```
interface Addable<out T> {}  
class Double implements Addable<Double> {}  
class Integer implements Addable<Integer> {}
```



JOINS

- Our system: the join of two materials always exists
 - Because material inheritance is decidable
- Note: **Addable<*>** was never the desired result
 - The result of any computation must be a material

CONDITIONAL INHERITANCE

- Unanswered question: type-safe equality for `List<T>`
- First solution, again: `List<T> extends Eq<List<Eq<T>>>`
 - Bad style
 - Nested contravariance & expansive inheritance
 - List elements forced to extend `Eq` -- cannot make a `List<Object>`

CONDITIONAL INHERITANCE

- Ideally, `List<T>` is **Equatable** if and only if its elements are
 - `List<out T> satisfies Equatable`
`given T satisfies Equatable`
- "**satisfies**" indicates that shapes are constraints, orthogonal to material classes and interfaces
- "**given**" denotes a condition that holds for certain instances

CONDITIONAL INHERITANCE

- Surprisingly challenging! Consider:

```
> interface List<out T> satisfies Cloneable
    given T satisfies Cloneable

> class Array<inv T> extends List<T>
    satisfies Cloneable
    given T satisfies Cloneable

> class B satisfies Cloneable

> class A extends B
```

- What is the result of invoking `Array<A>.clone()` ?

SHAPE SHIFTERS

- Code reuse is fundamental to object-oriented programming
- Shapes express constraints at the class / interface level
- **Shape Shifters** are a proposal for type variable-level reasoning

`Set<String with CaseInsensitive>`

`Set<Function<Int, Int> with RefEqual>`

The End

