MATERIALS & SHAPES

Ben Greenman
December 16, 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

MY GOALS

1. Explain the big discovery of the paper

2. Share the conclusions we drew

3. Convince you that we've acted sensibly

THE PROBLEM

- Type-safe equality in object-oriented languages
  - `Cat() == Animal()` ✓ Cast to common super
  - `42 == "forty-two"` ✗ Type error
  - `λx.42 == λx.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`

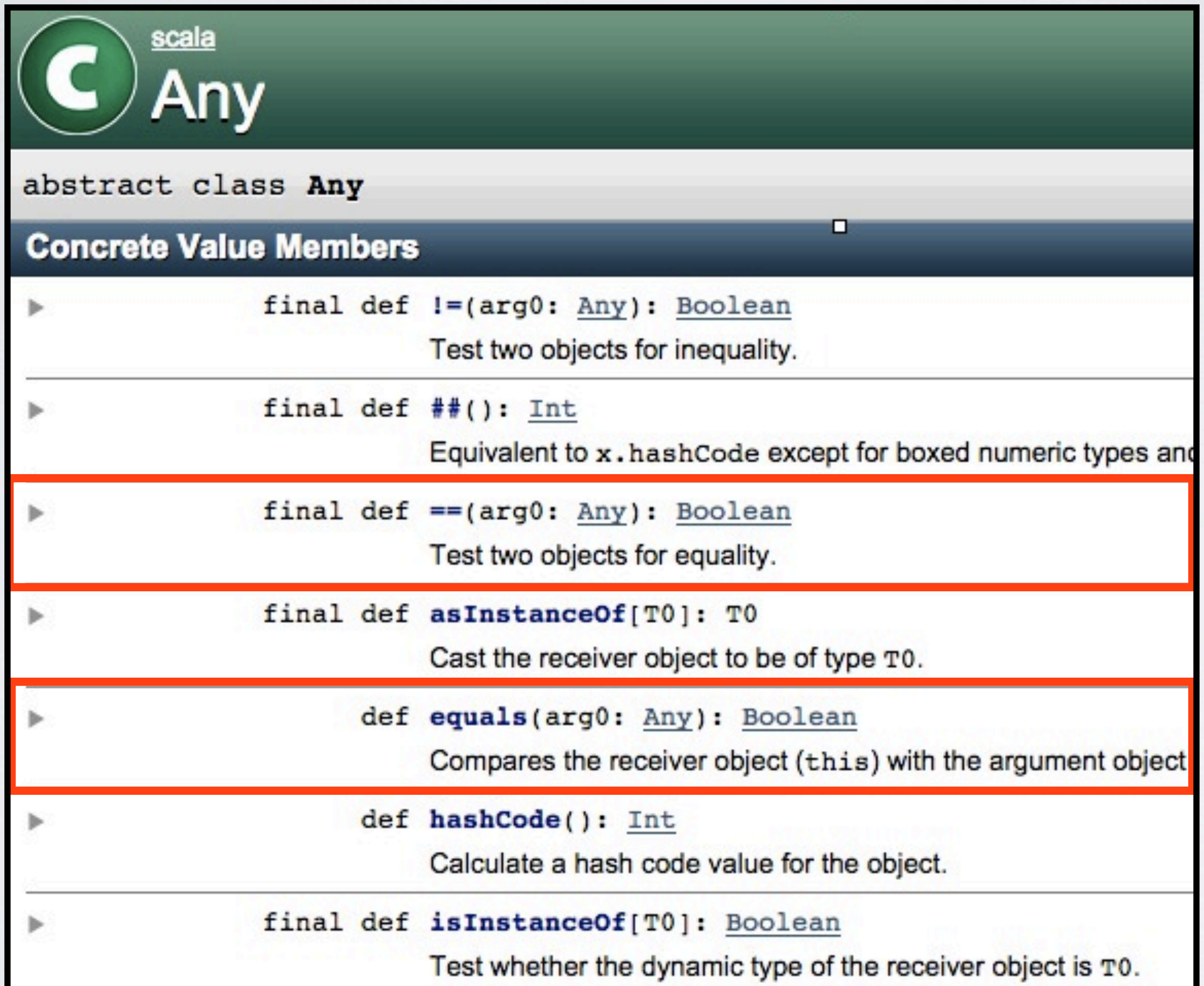
Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including array

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

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

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


scala

Any

abstract class **Any**

Concrete Value Members

final def **!=**(arg0: Any): Boolean

Test two objects for inequality.

final def **##**(): Int

Equivalent to `x.hashCode` except for boxed numeric types and

final def **==**(arg0: Any): Boolean

Test two objects for equality.

final def **asInstanceOf**[T0]: T0

Cast the receiver object to be of type `T0`.

def **equals**(arg0: Any): Boolean

Compares the receiver object (`this`) with the argument object

def **hashCode**(): Int

Calculate a hash code value for the object.

final def **isInstanceOf**[T0]: Boolean

Test whether the dynamic type of the receiver object is `T0`.

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
- ❌ Run-time cast
- ❌ Dynamic check
- ❌ Lots of boilerplate

// It just gets worse

- **`instanceof`** checks show up everywhere
- 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! */
  }
}
```

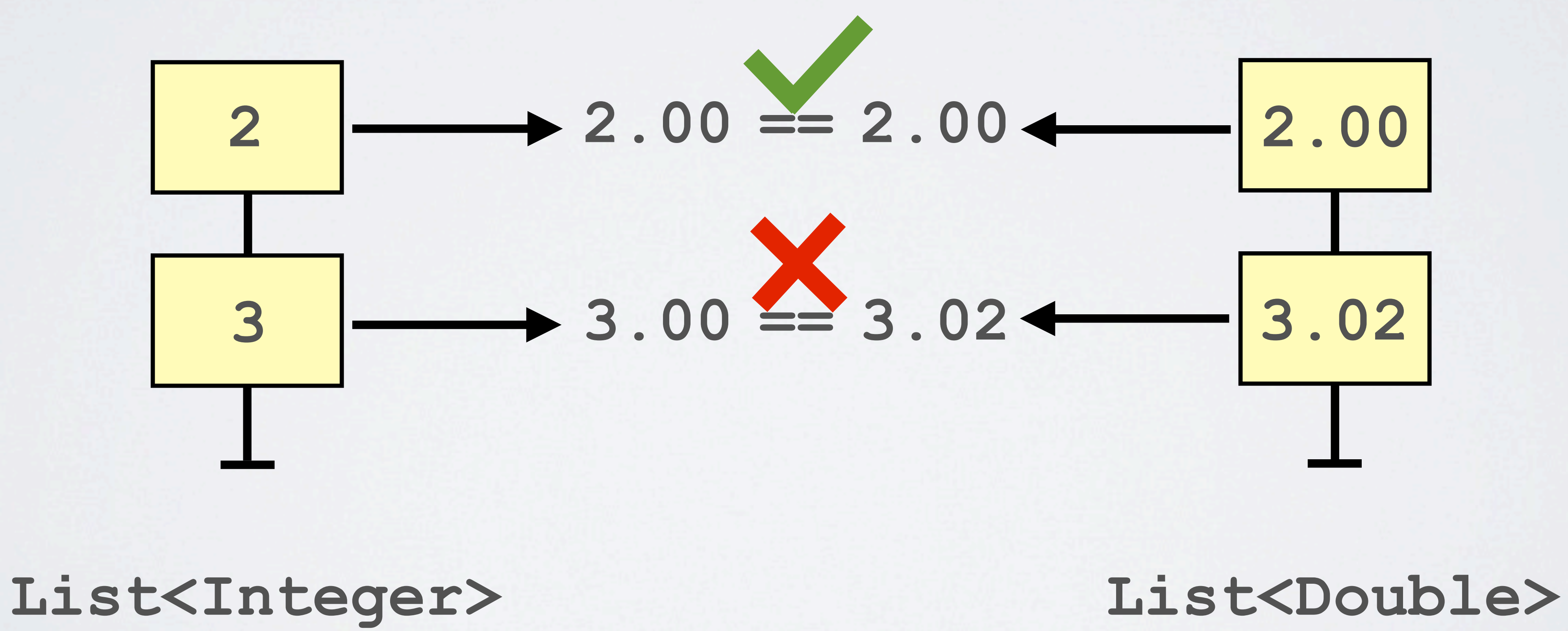
We can do better!

- 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>>>`

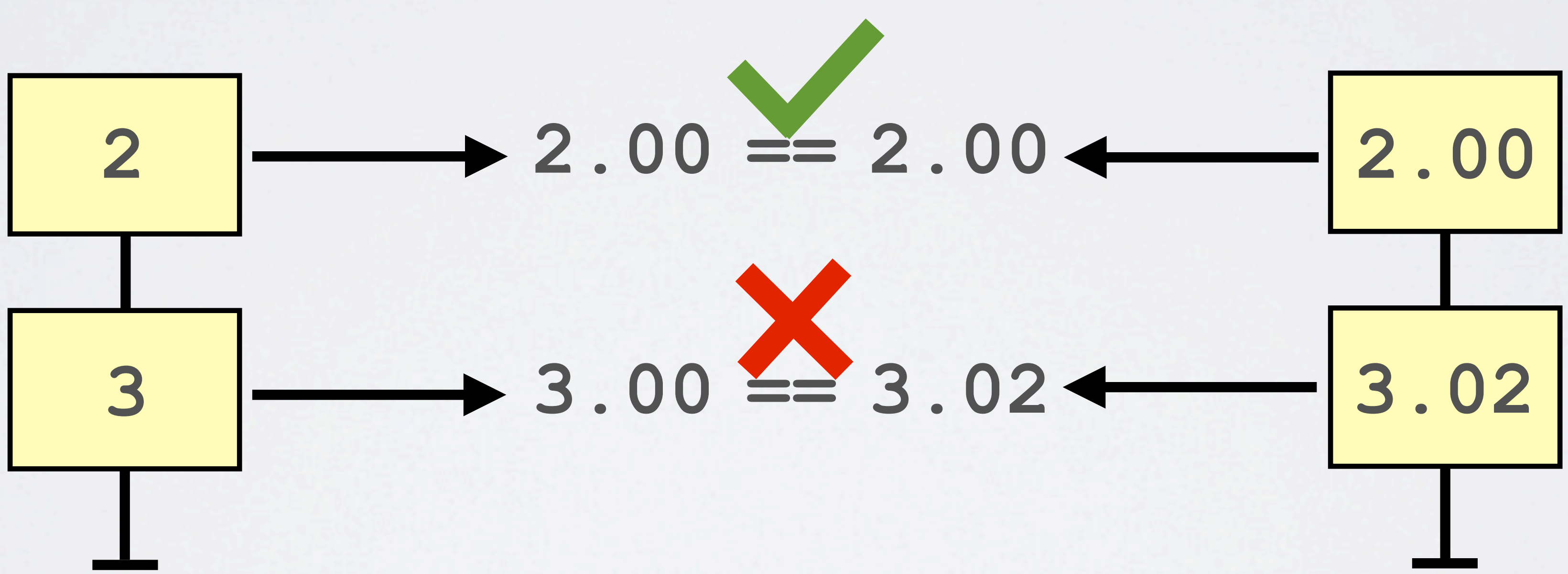


An example: List

Almost!

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

```
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

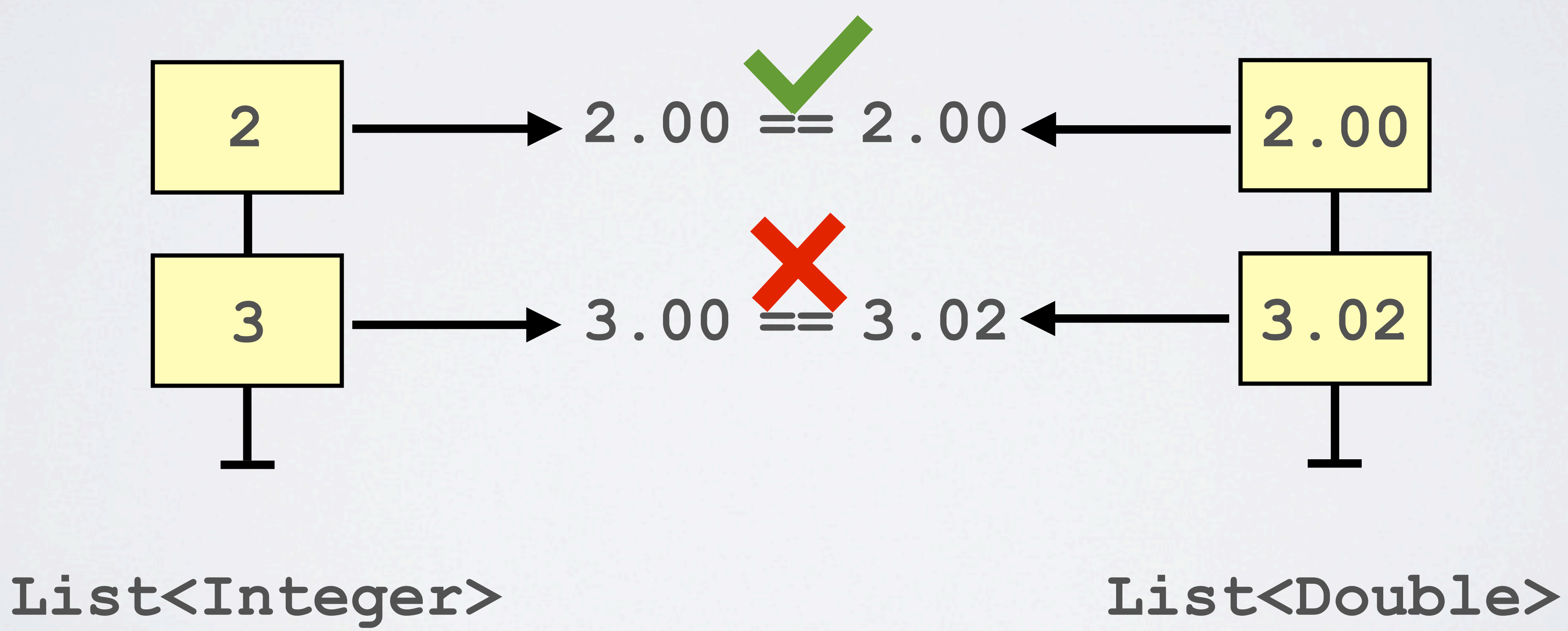
```
class Adult {}
class Baby extends Adult {}

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>>>



An example: List

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

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

  - **`List`** is covariant (we get elements out of it)
  - **`Equatable`** is contravariant (we supply arguments)

An example: List

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

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

This actually works!

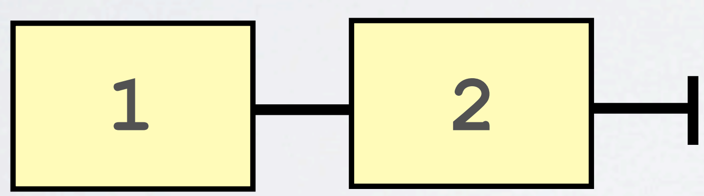
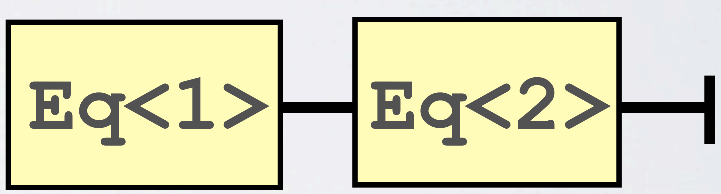
THE BIG DISCOVERY

- The Ceylon team wanted to avoid `Object.equals()`
- Ross 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.

List<Integer>> <:	**List<Equatable<Integer>>>**
	
Easy to imagine	Not so easy to understand

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

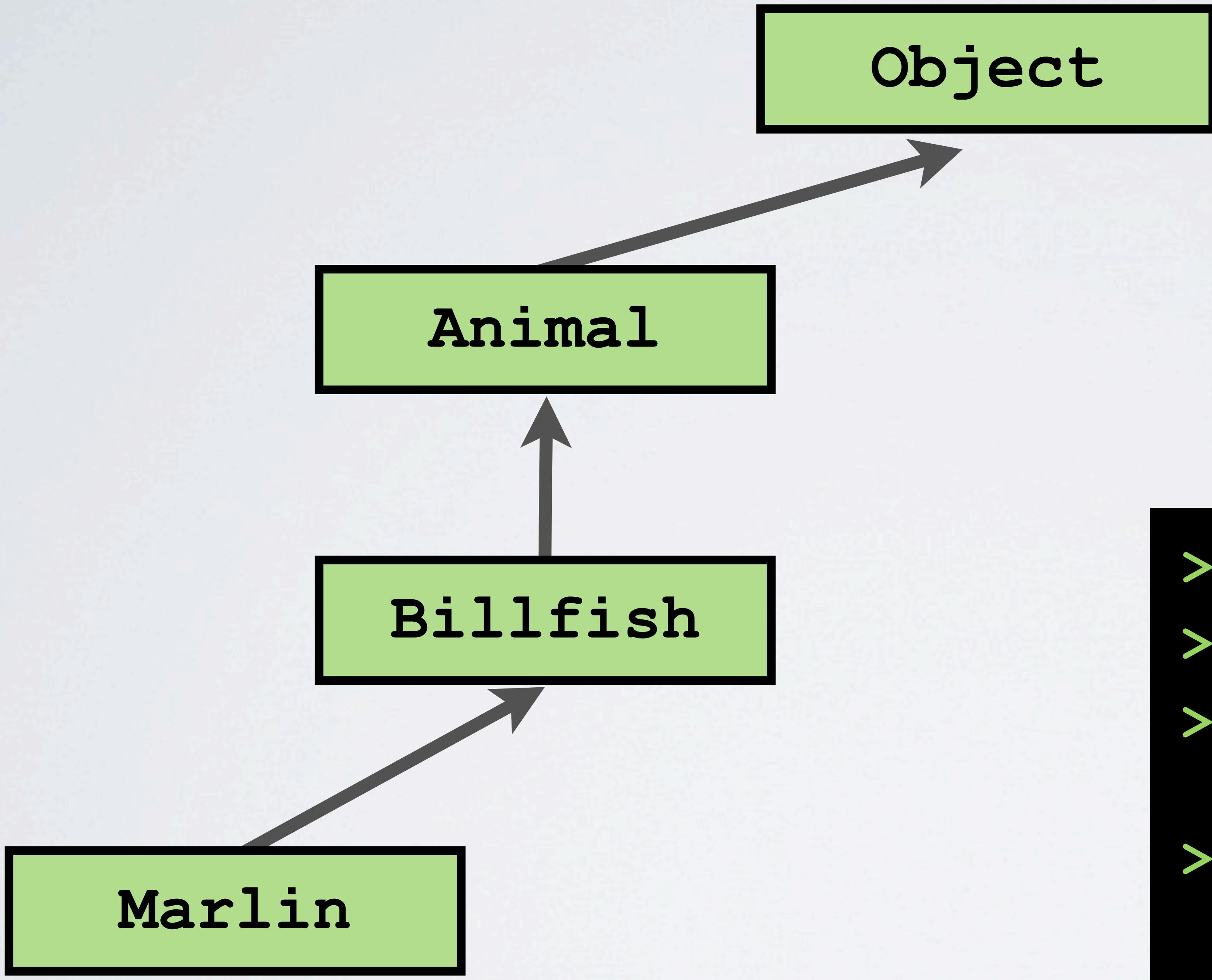
You never want a "list of constraints" **?**

- We can't tell what programmers were thinking
- Or they challenges they faced in development
- But, we can formalize Ceylon's opinion in the Java compiler without breaking backwards-compatibility

EXPERIMENT

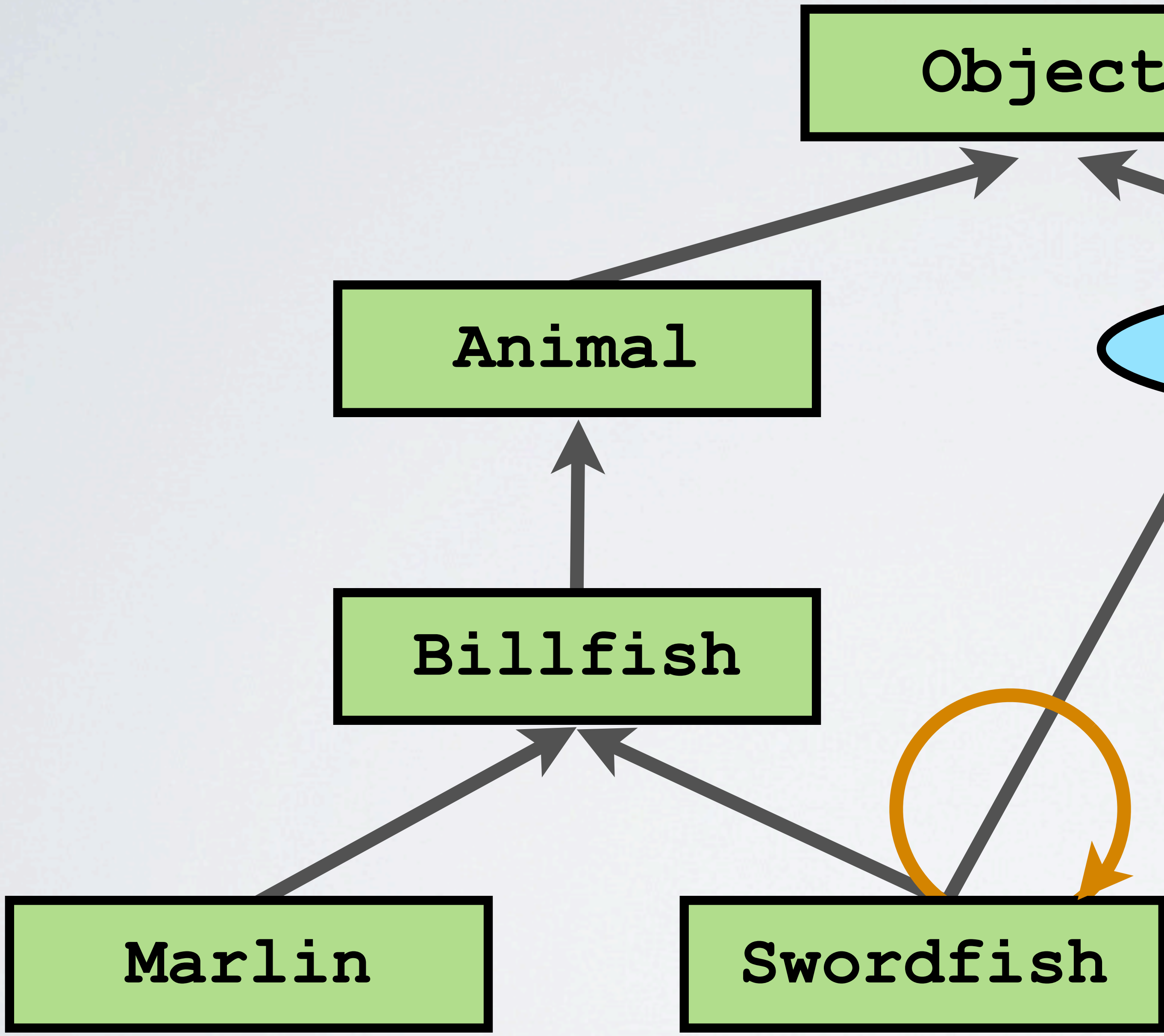
- Types like `Equatable<Integer>` were never used as:
  - Type Parameters
  - Function arguments or return types
  - Local variables or fields

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



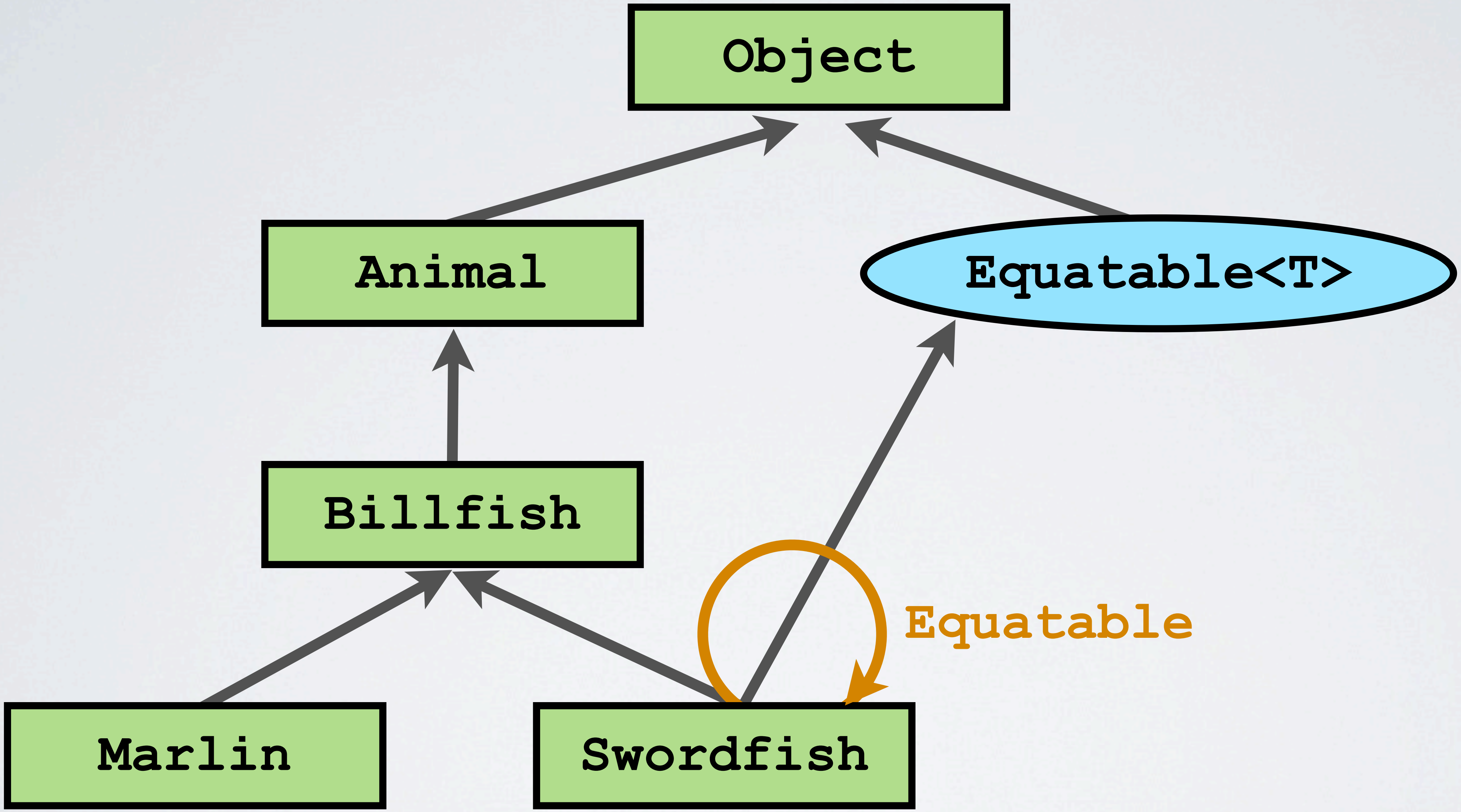

```
> class Object {}
> class Animal {}
> class Billfish
    extends Animal {}
> class Marlin
    extends Billfish {}
```

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




```
> inter Equatable<T> {}
> class Swordfish
    extends Billfish
     , Equatable<Swordfish>
```

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



EXPERIMENT
(more precisely)

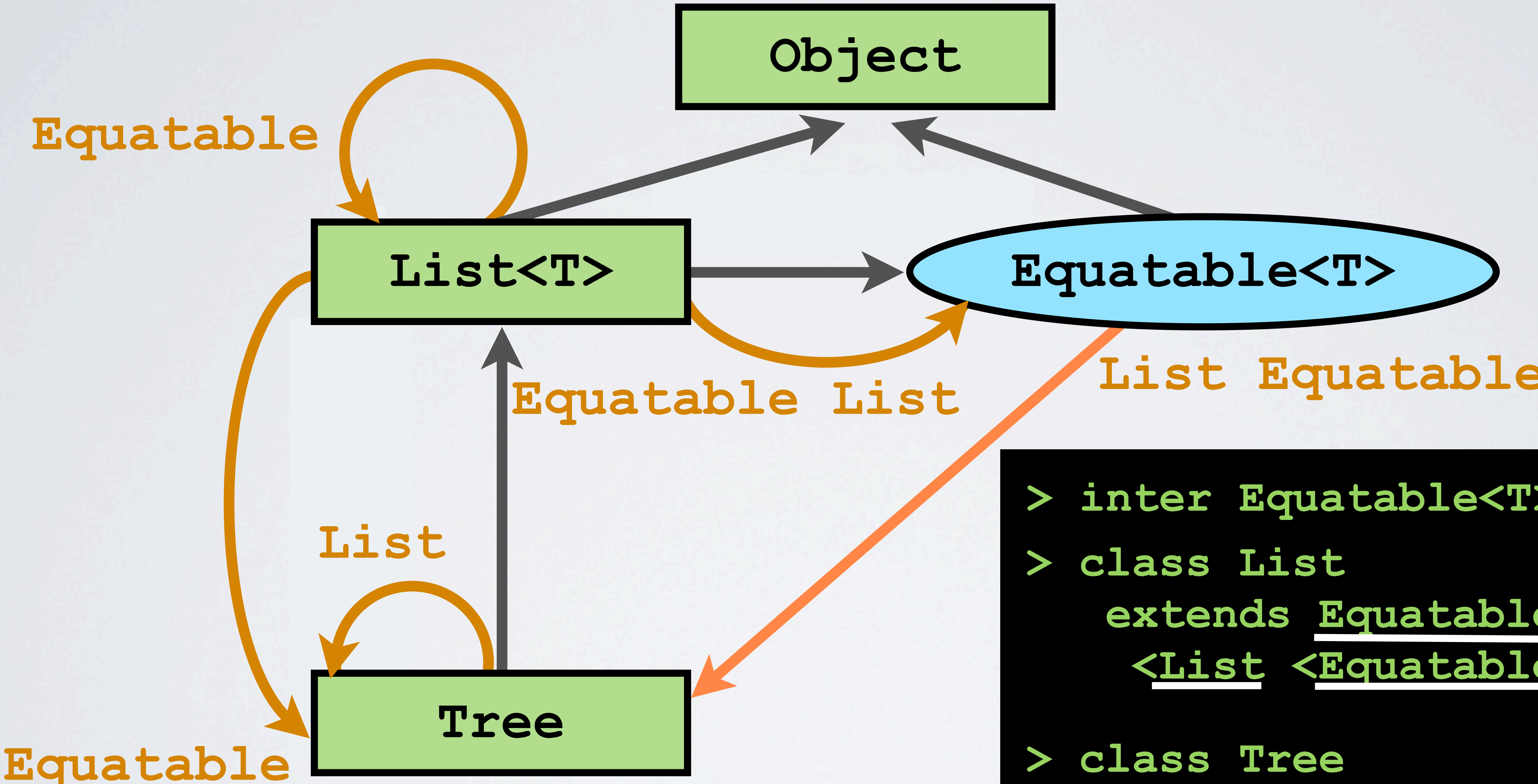
- Parameterized types used to complete cycles in the inheritance hierarchy were never used as:
  - Type Parameters
  - Function arguments or return types
  - Local variables or fields

RECAP

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

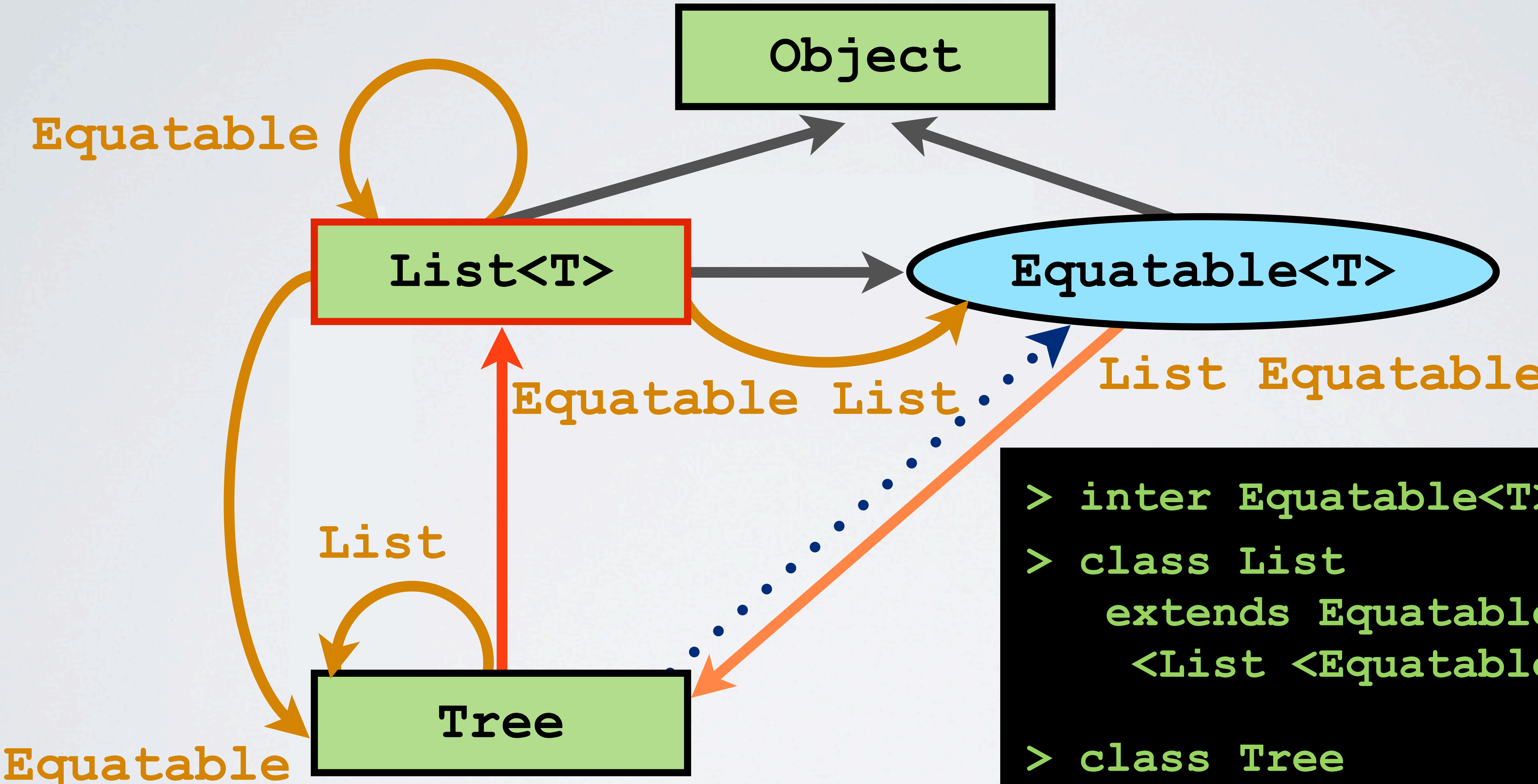
Next Up: the research perspective

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



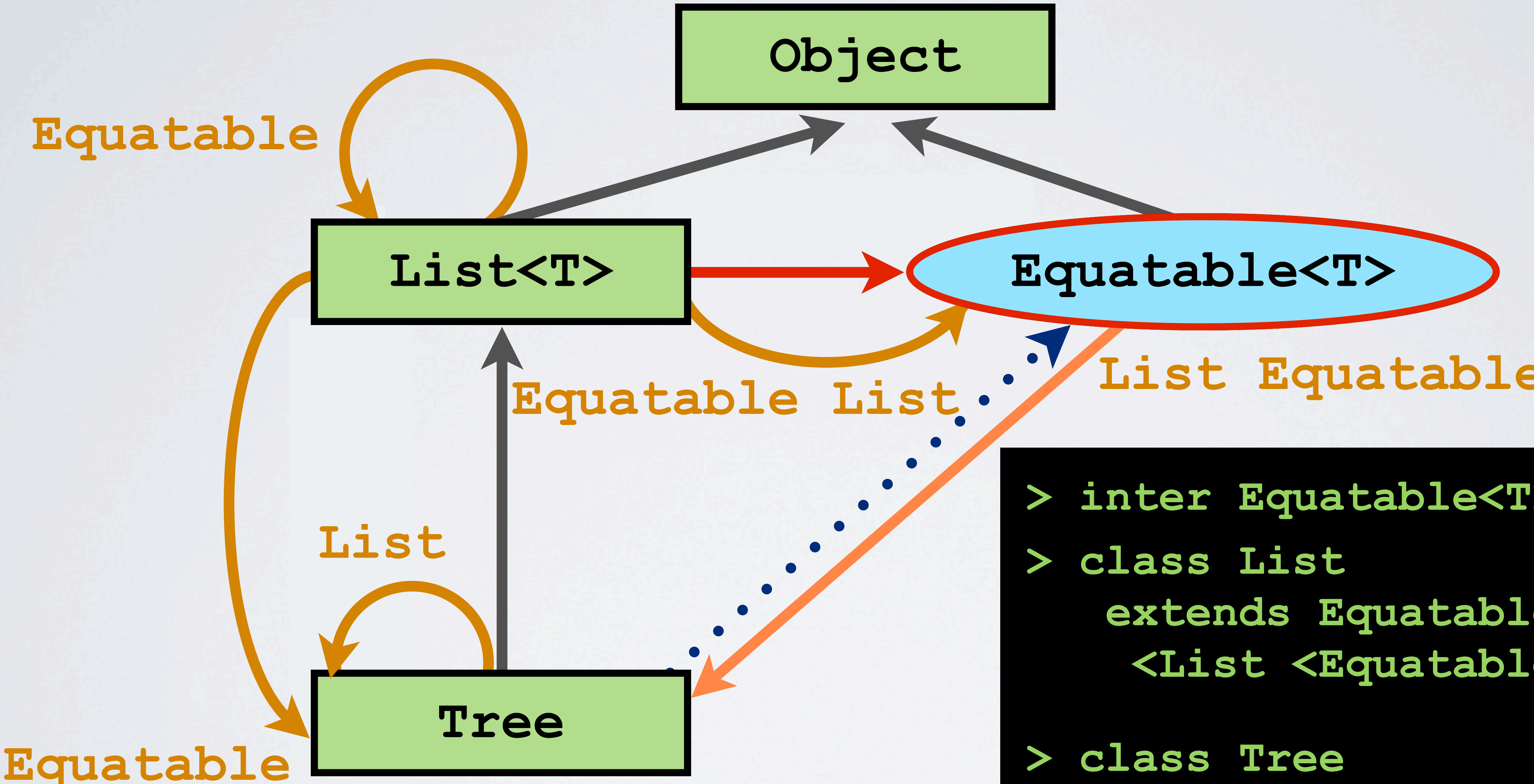

```
> inter Equatable<T> {}
> class List
   extends Equatable
    <List <Equatable <T>>
> class Tree
   extends List<Tree>
```

```
-? Tree <: Equatable<Tree>
```

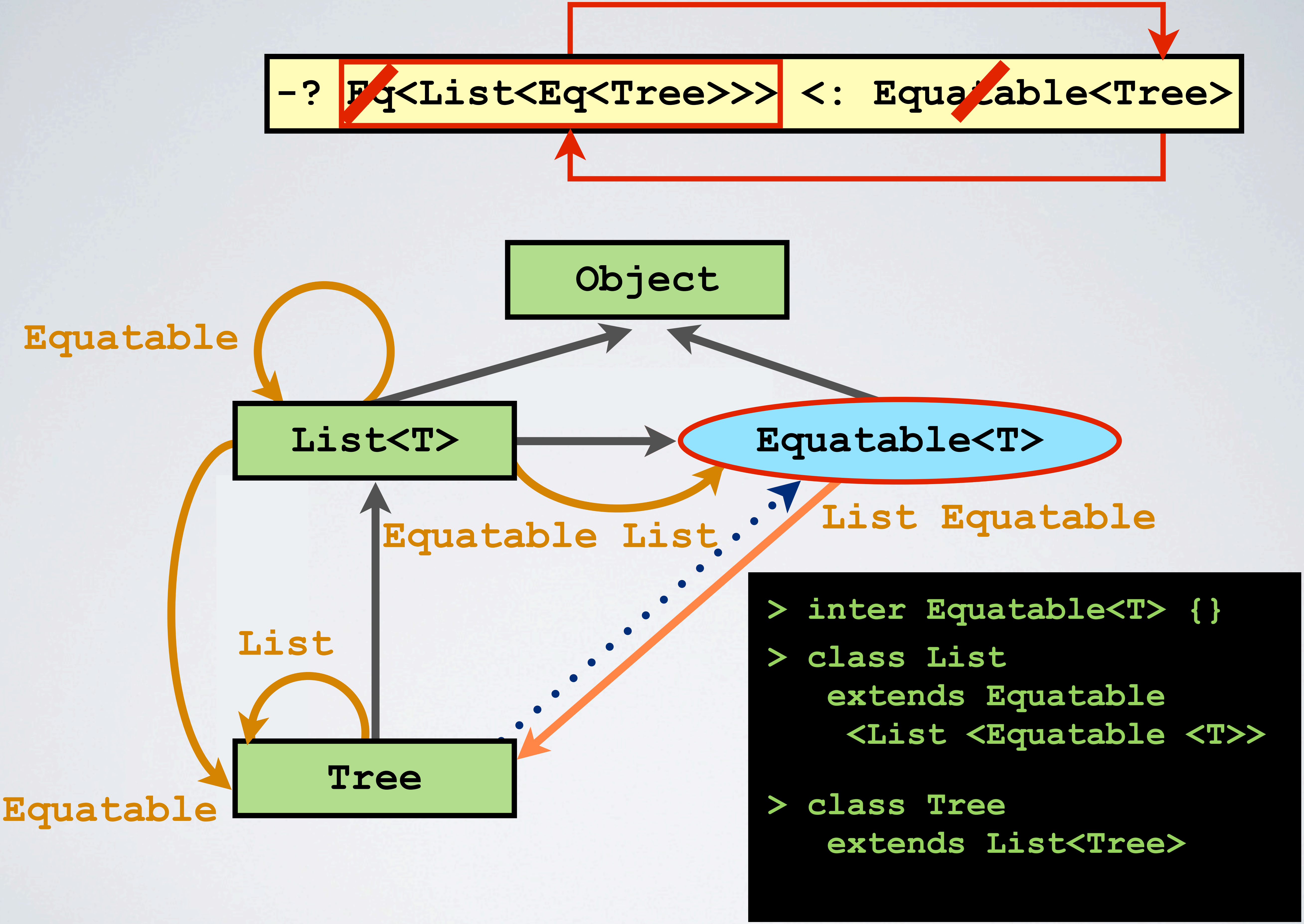


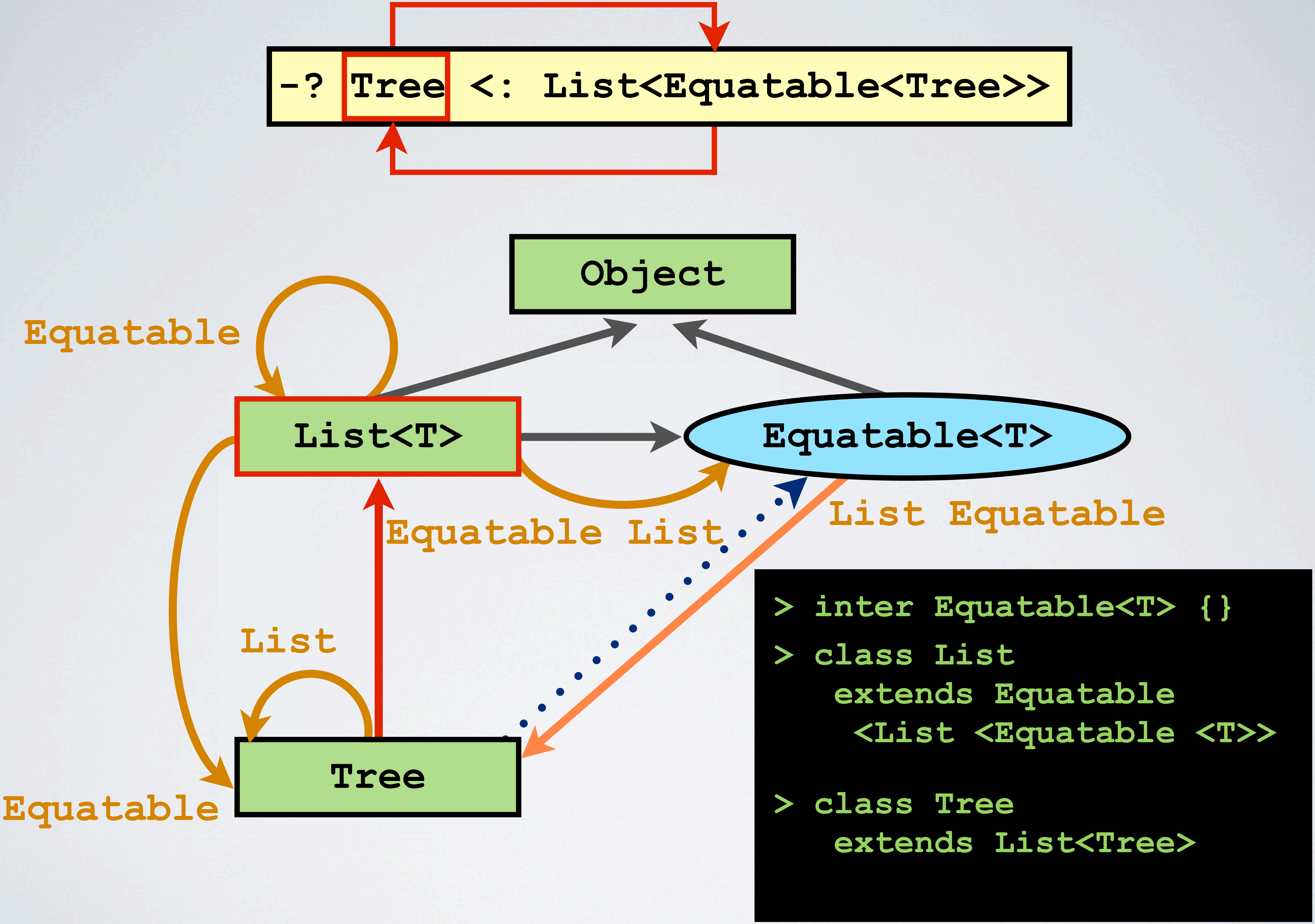

```
> inter Equatable<T> {}
> class List
   extends Equatable
    <List <Equatable <T>>

> class Tree
   extends List<Tree>
```

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



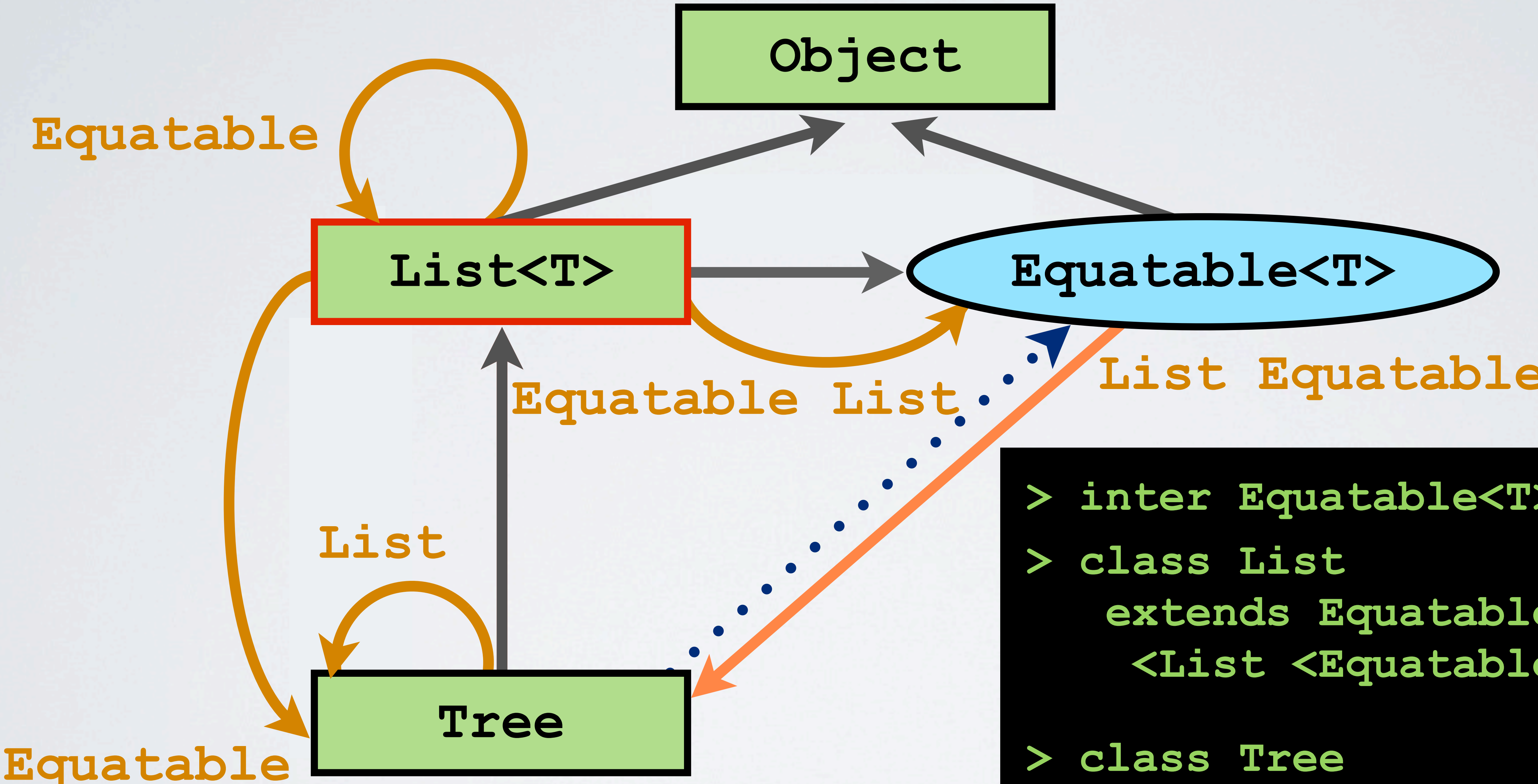

```
> inter Equatable<T> {}
> class List
   extends Equatable
    <List <Equatable <T>>
> class Tree
   extends List<Tree>
```


-? Eq<List<Eq<Tree>>> <: Equatable<Tree>
Object
Equatable
List<T>
Equatable<T>
Equatable List
List Equatable
List
Tree
Equatable
> inter Equatable<T> {}
> class List
extends Equatable
<List <Equatable <T>>
> class Tree
extends List<Tree>



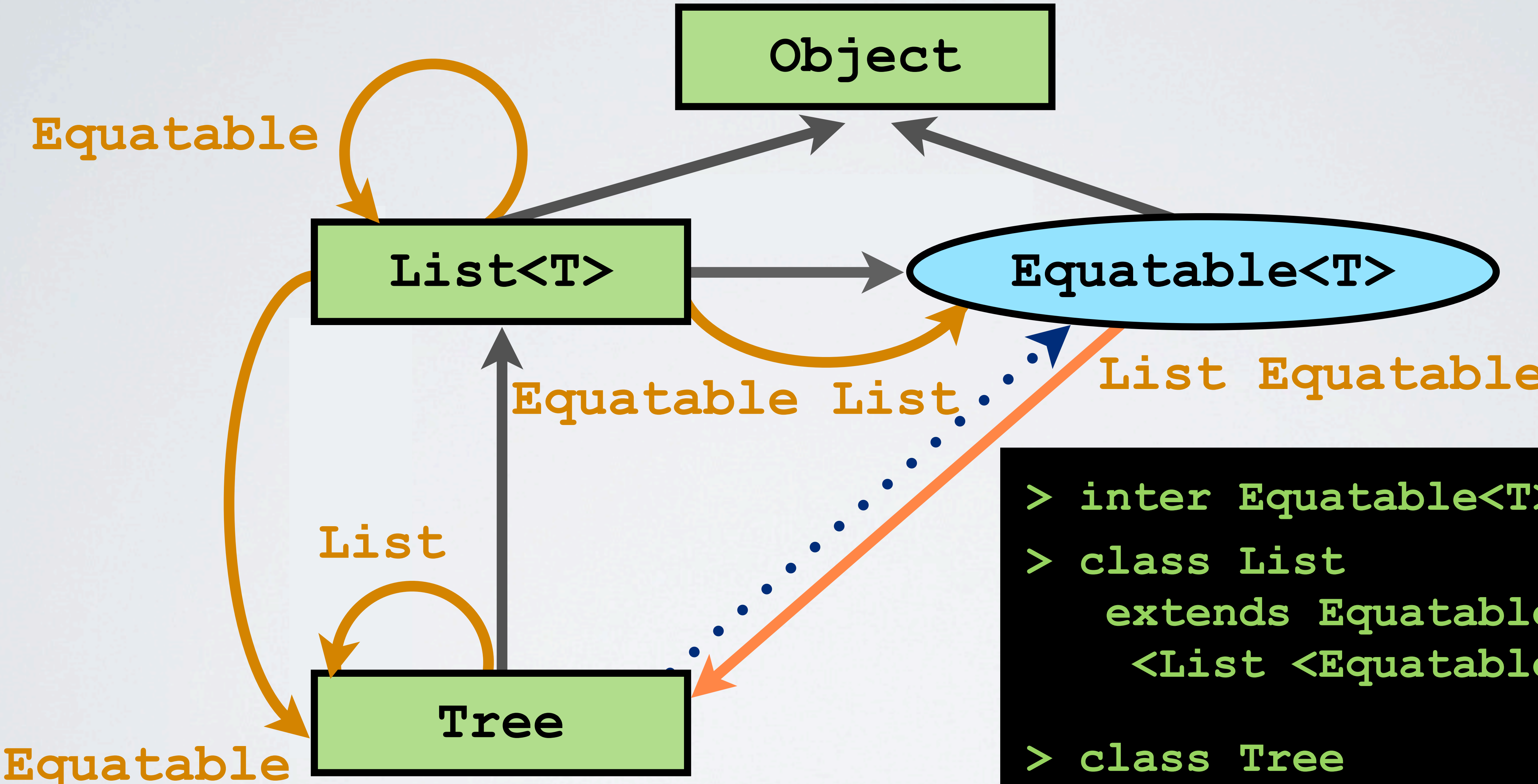
-? Tree <: List<Equatable<Tree>>
Object
Equatable
List<T>
Equatable<T>
Equatable List
List Equatable
List
Tree
Equatable
> inter Equatable<T> {}
> class List
extends Equatable
<List <Equatable <T>>
> class Tree
extends List<Tree>

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




```
> inter Equatable<T> {}
> class List
   extends Equatable
    <List <Equatable <T>>

> class Tree
   extends List<Tree>
```

`-? Tree <: Equatable<Tree>` ✗ Cycle!




```
> inter Equatable<T> {}
> class List
   extends Equatable
    <List <Equatable <T>>

> class Tree
   extends List<Tree>
```

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:

1. Contravariance
2. Expansive Inheritance
3. Multiple Instantiation Inheritance*

PRIOR WORK

1. 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

Materials

Material-Shape Separation

Shapes

```
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.

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

i.e. "where can we go from here?"

1. Decidable subtyping
2. Type equality, decidable joins
3. Conditional inheritance
4. Shape shifters

WELL-FOUNDED INHERITANCE

- Undecidability results were caused by cyclic inheritance
    - Impossible to predict how type parameters would expand
- Without shapes, inheritance is well-founded
    - No more cycles!
    - An object's inheritance graph is known at compile-time
- Many applications

DECIDABLE SUBTYPING

- Strategy: define a measure on judgments `X <: Y`
- Key idea: inheritance never introduces new shapes
- Two components:
  - 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

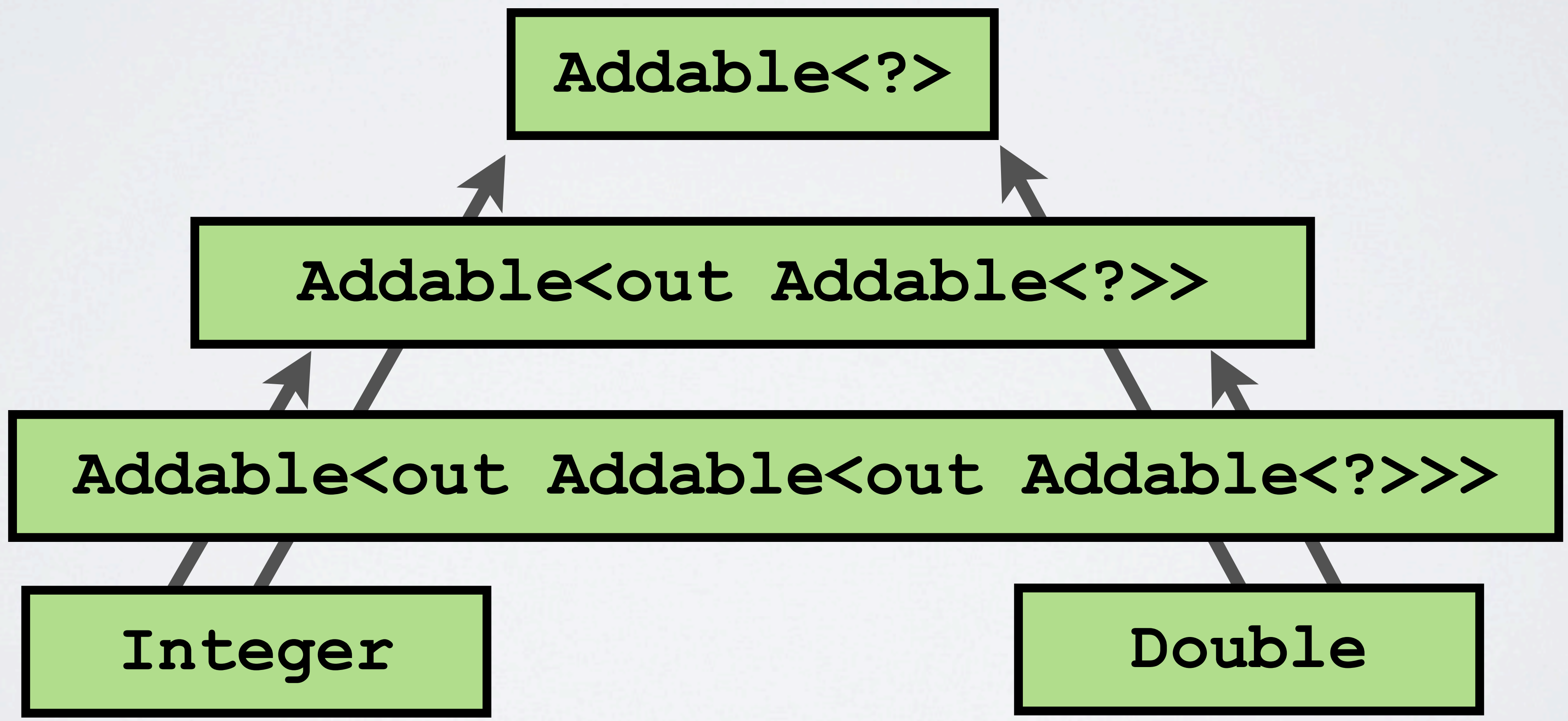
- Our subtyping algorithm only depends on recursion
  - Never uses syntactic equivalences
- We get equality for free: **`(A = B)`** iff **`(A <: B`** and **`B <: A)`**

JOINS

- **`A ⊔ B`** is the least common supertype of **`A`** and **`B`**
- Useful for type-checking conditional statements.
  - **`if (C) then A else B`** has type **`A ⊔ 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