CZ: Multiple Inheritance Without Diamonds

Donna Malayeri

Carnegie Mellon University

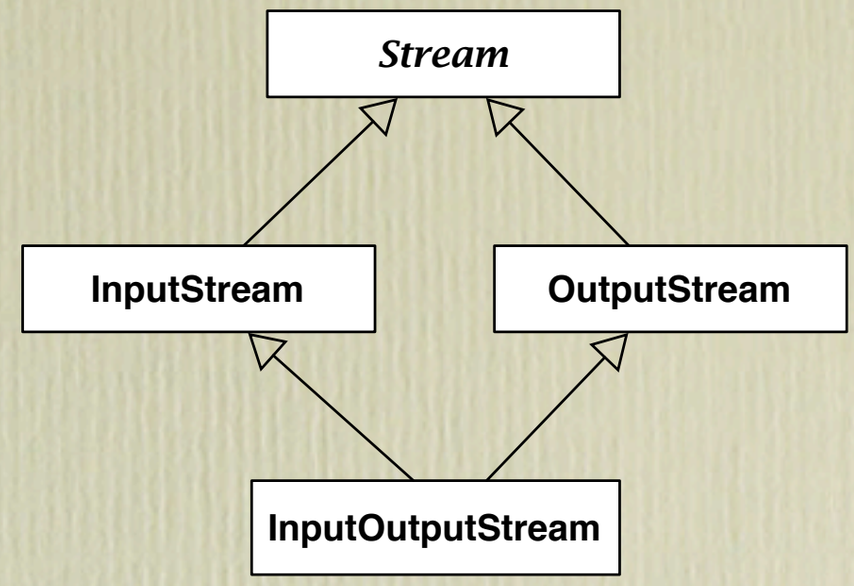
Multiple inheritance can be useful

- More flexible than single inheritance
- More opportunities for code reuse
- Can express multiple “dimensions” of behavior implemented by a class

Multiple inheritance can be problematic

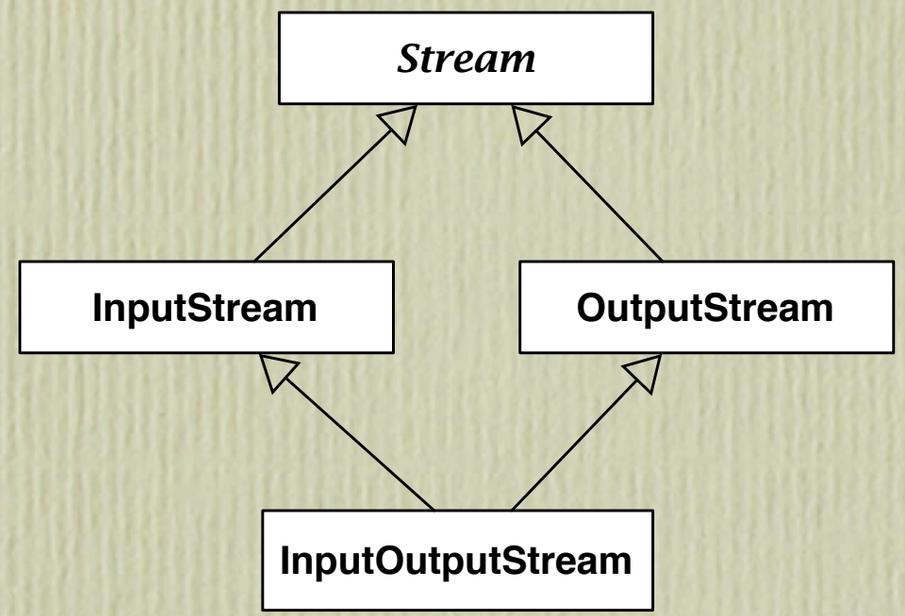
- Name clashes
  - Possible solution: allow renaming (e.g. Eiffel)
- Inheritance diamonds
  - Possible solution: restrict the language
    - Mixins: explicit linearization, no relationships between mixins
    - Traits: no state, only methods
- Our goal: can we solve the problem with fewer language restrictions?

The Diamond Problem



- Occurs when a class inherits an ancestor through more than one path
- Should `InputOutputStream` have multiple copies of `Stream`'s instance variables, or just one?
  - Problem 1: if only one copy, how do we initialize `InputOutputStream` properly?
- Problem 2: how to perform modular typechecking of external methods/multimethods?

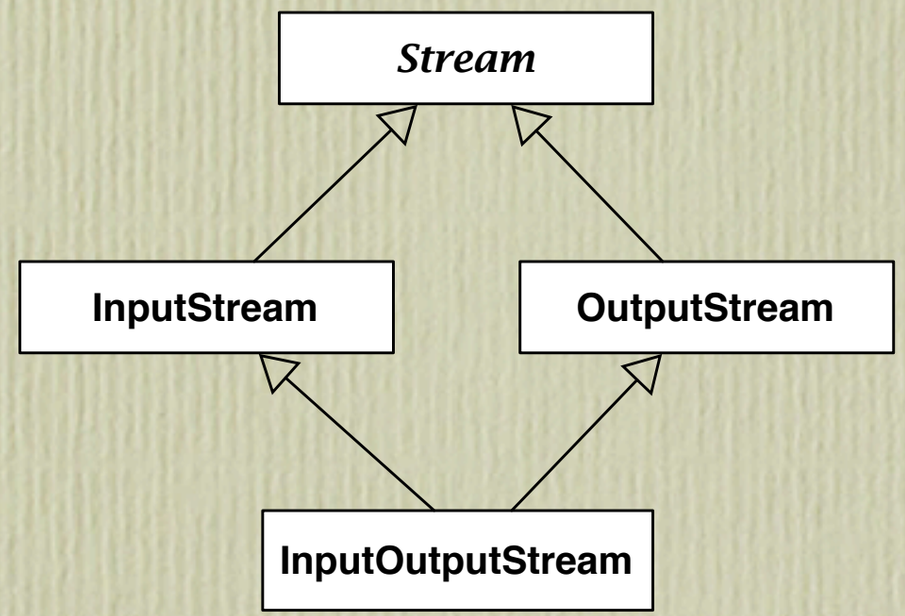
Problem 2: Typechecking External Methods




```
void Stream.seek(long pos) { }

void InputStream.seek(long pos) { … }

void OutputStream.seek(long pos) { … }
```

- External method:
  - like ordinary method, but can be defined outside class definition
  - performs *dispatch* on objects of that class's type
    - i.e., correct method selected based on run-time type
- Eliminates need for Visitor pattern

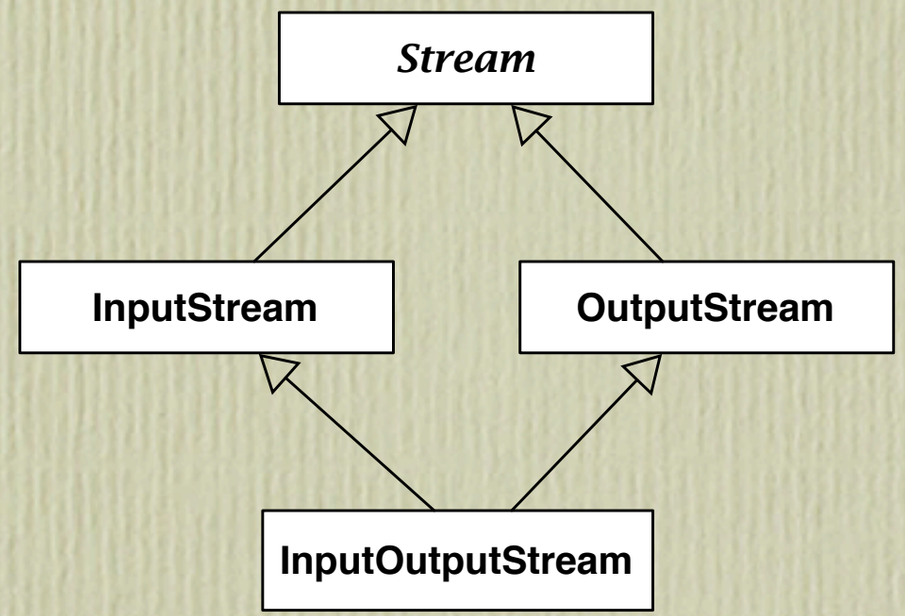
Problem 2: Typechecking External Methods




```
void Stream.seek(long pos) { }

void InputStream.seek(long pos) { … }

void OutputStream.seek(long pos) { … }
```

- What if we call `new InputOutputStream().seek()`?
- How to modularly determine that this is ambiguous?
  - When checking `InputOutputStream`, can't look for potentially ambiguous external method definitions
  - When checking `seek()`, can't look for diamonds in the class hierarchy

Problem 2: Typechecking External Methods




```
void Stream.seek(long pos) { }

void InputStream.seek(long pos) { … }

void OutputStream.seek(long pos) { … }
```

- Problem arises with any form of multiple inheritance:
  - Java-style
    - Dispatch on interfaces (JPred)
  - Traits
    - Dispatch on traits (Fortress)

Solution

- Disallow inheritance diamonds
- But, what if we need to define them?

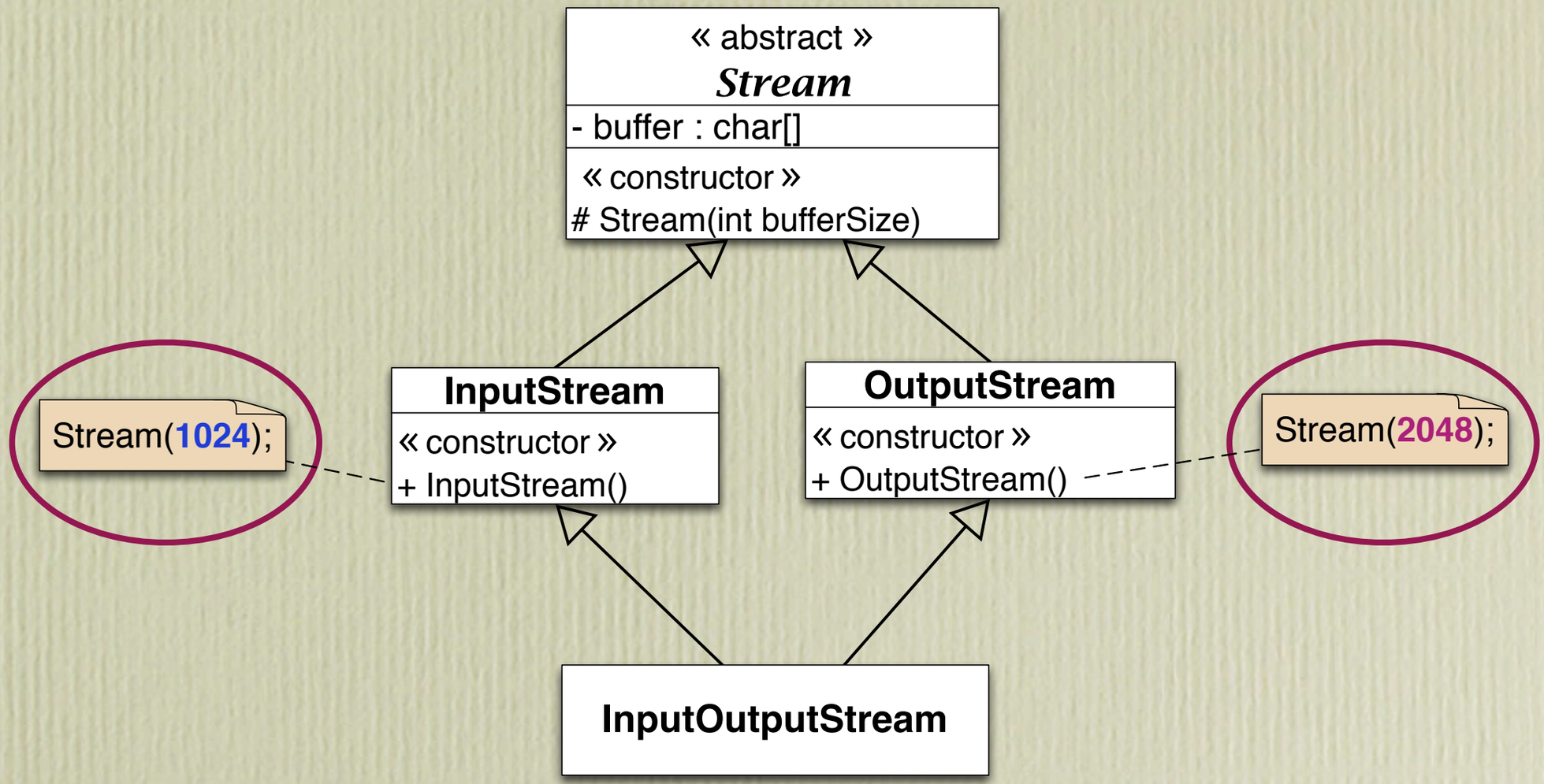
When an inheritance (subclass) diamond is *needed*:

- Use `requires` instead (generalization of a Scala construct)
- Captures notion of inheritance dependency
- If *A* `requires` *B*
  - *A* is a sub*type* of *B* (but not a sub*class*)
  - *A* is abstract
  - Concrete subclasses of *A* must `extend` *B*

CZ Design

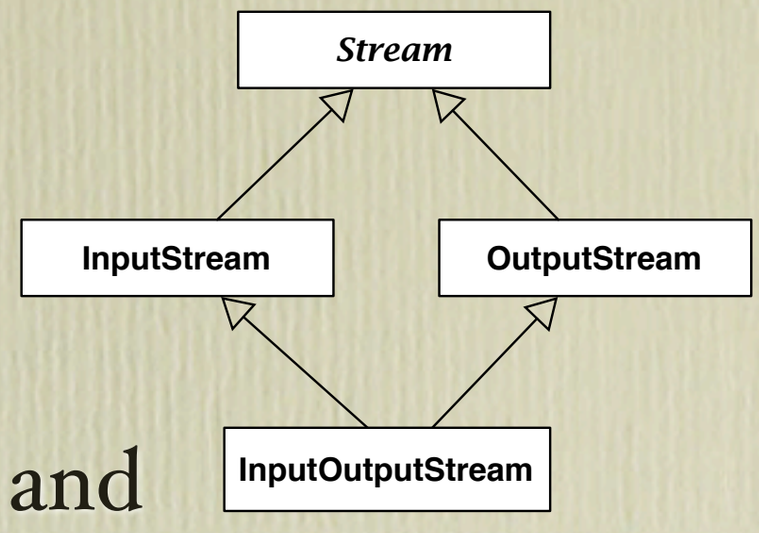
- Disallow inheritance diamonds
- When needed, use `requires` instead
  - Expresses inheritance dependency and provides subtyping

Problem 1: Object Initialization



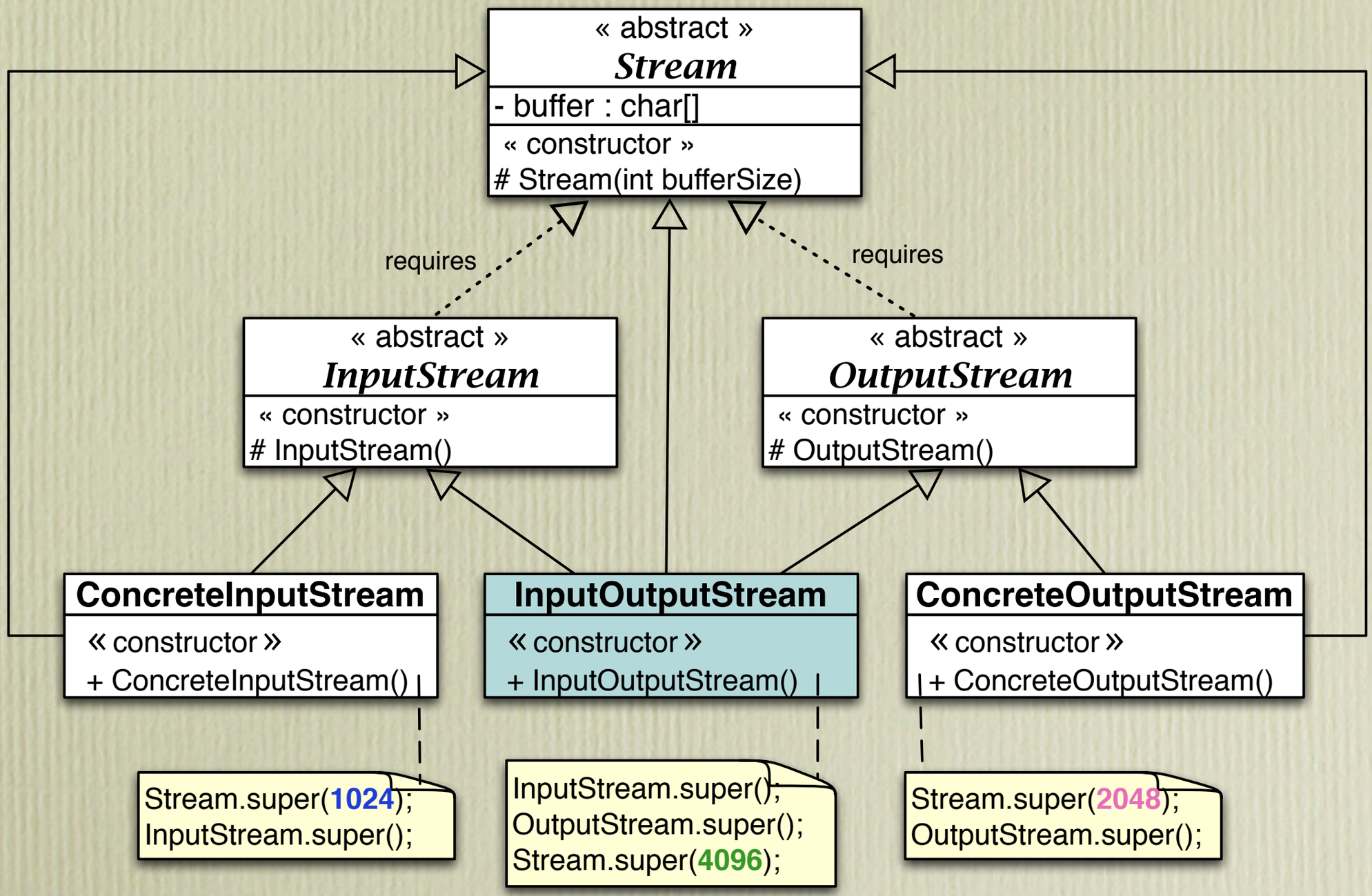
When initializing `InputOutputStream`, How many times and with which argument is `Stream` constructor called?

Problem 1: Previous Solutions



- Eiffel, Stateful traits: call `InputStream` and `OutputStream` constructors explicitly; `Stream` constructor is called twice
  - What if `Stream()` performs a stateful operation?
- C++: call `Stream()` directly from `InputOutputStream` and ignore other calls to `Stream()`
  - What if the ignored calls were important?
- Scala: trait constructors may not take parameters
  - Elegant, but restrictive

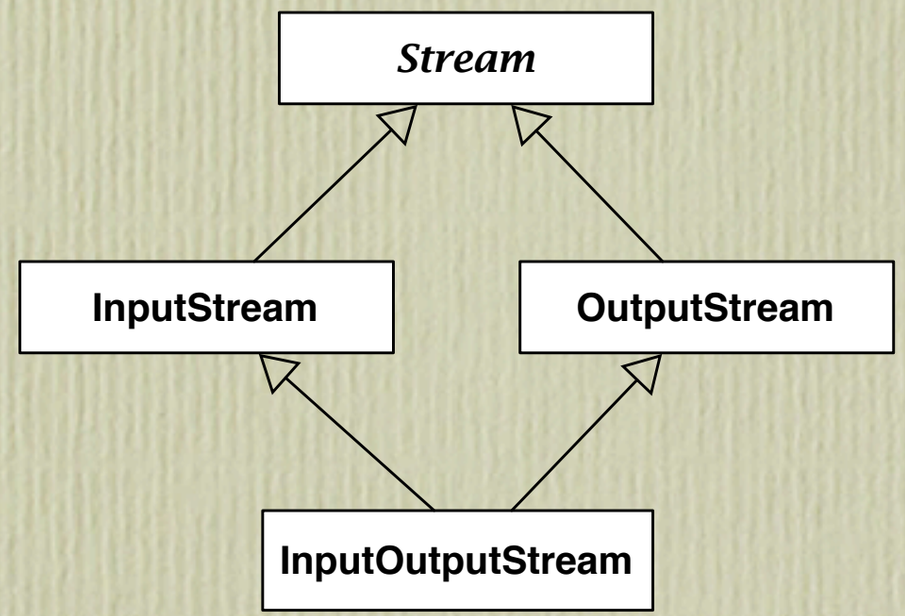
Problem 1: Streams in CZ



CZ Solves Problem 1

- No diamonds means:
  - graceful co-existence of state and multiple inheritance
  - No ignored constructor calls
  - No restrictions on constructors
  - Correct initialization semantics

Recall Problem 2: Typechecking External Methods



void Stream.seek(**long** pos) { }

void InputStream.seek(**long** pos) { … }

void OutputStream.seek(**long** pos) { … }

- What if we call `new InputOutputStream().seek()`?
- How to modularly determine that this is ambiguous?

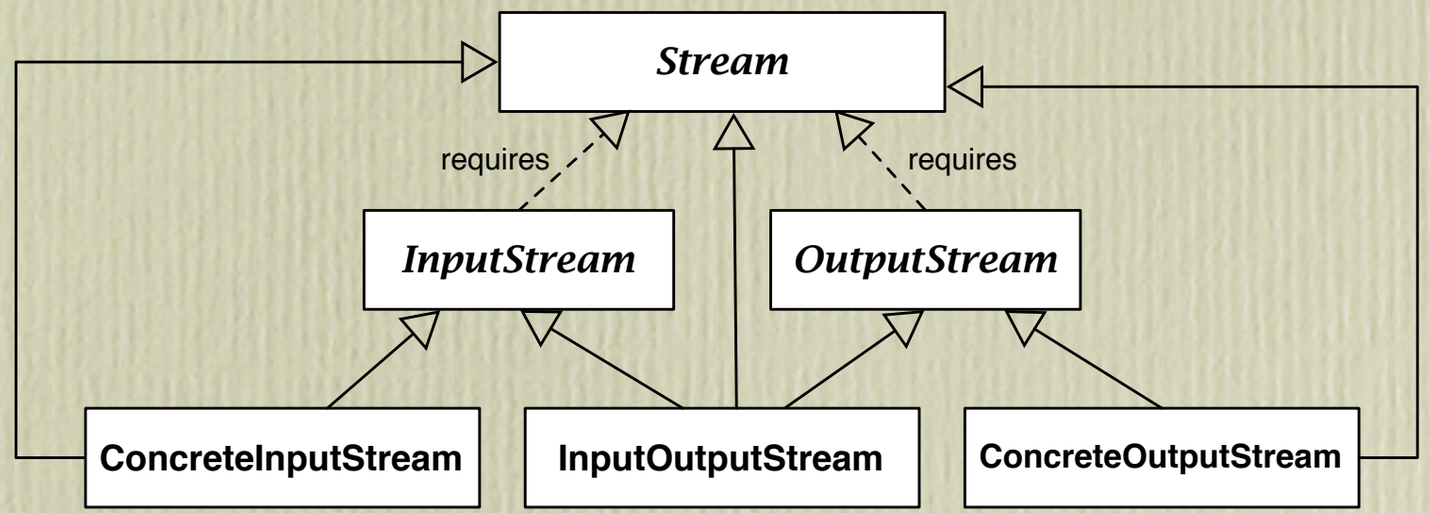
Problem 2: Previous Solutions

- Disallow multiple inheritance across module boundaries (Dubious language)
- Require disambiguating methods for *every subset of incomparable types* (JPred, Fortress)

```
void (InputStream && OutputStream).seek() { … }
```

- Even if type like `InputOutputStream` is never created
- Exponential in the number of incomparable types
  - **3** incomparable types => **4** additional methods
  - **4** incomparable types => **11** additional methods
  - **5** incomparable types => **26** additional methods

Problem 2: Typechecking external methods



- External override based on inheritance (subclassing)

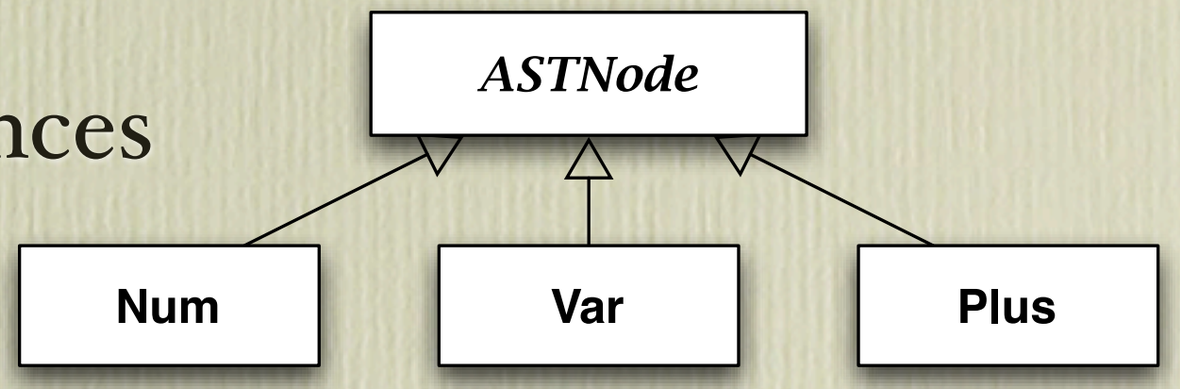
```
Stream.seek(int pos) { … }
ConcreteInputStream.seek(int pos) { … }
ConcreteOutputStream.seek(int pos) { … }

InputOutputStream.seek(int pos) { … }
```

CZ Solves Problem 2

- No diamonds means
  - Ambiguity checking becomes easy
  - No manual disambiguation required

When to use `requires` rather than `extends`

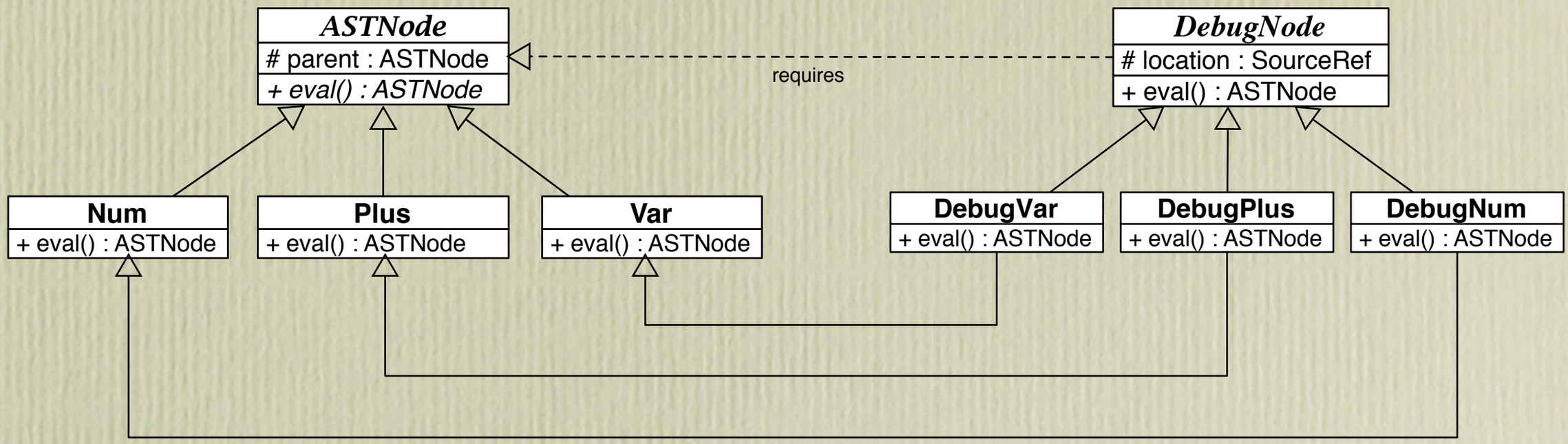
- Observe that a class serves two roles:
  - Generator of instances
  - Unit of reuse



- If *A* is an extension of *B*, and is intended to be reused => use `requires`
  (e.g. `InputStream requires Stream`)

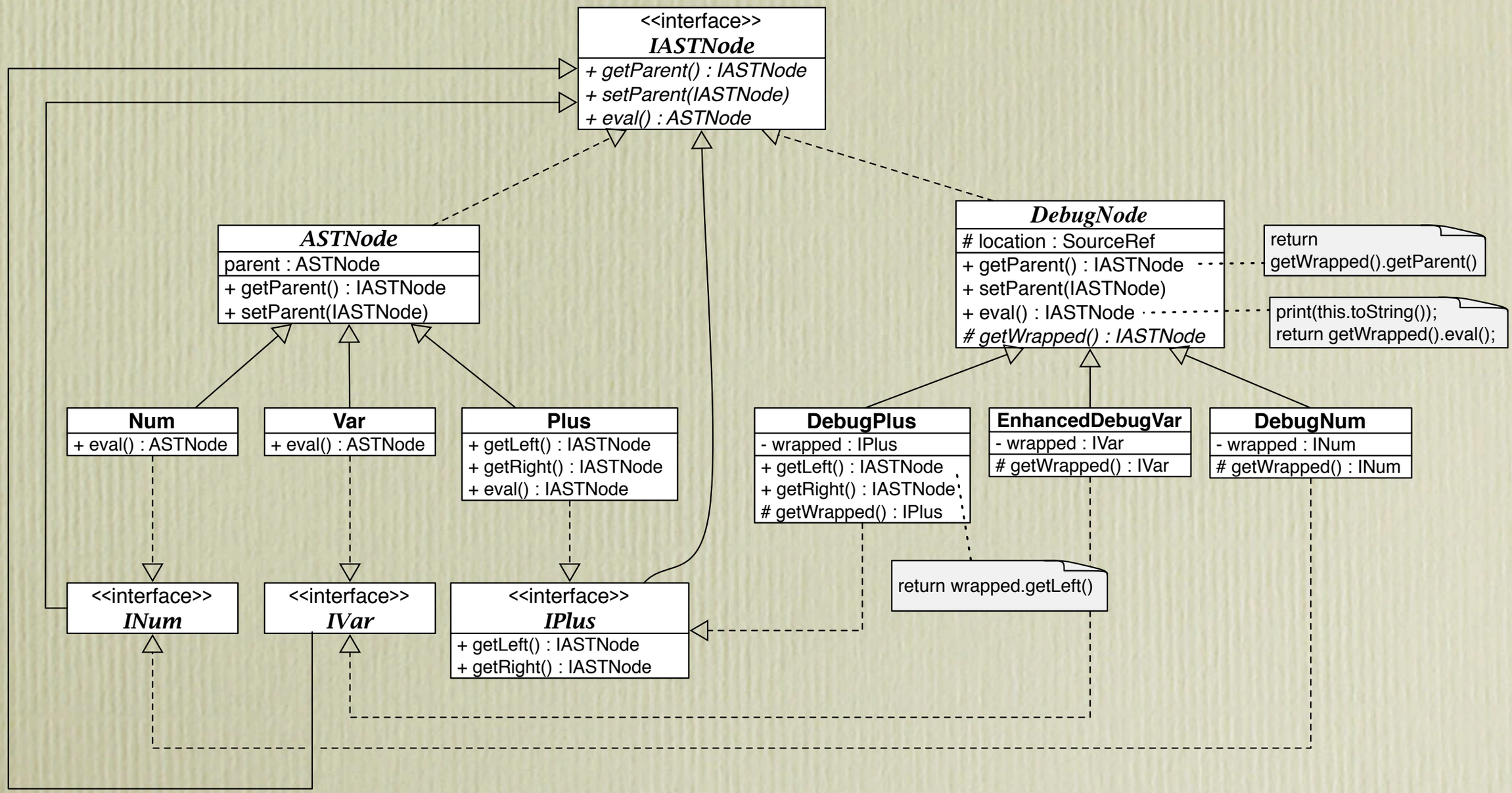
- When subclasses of *A* are disjoint, use `extends`
  (e.g. can have no subclass of `Num` and `Var`)

Example: Abstract Syntax Trees

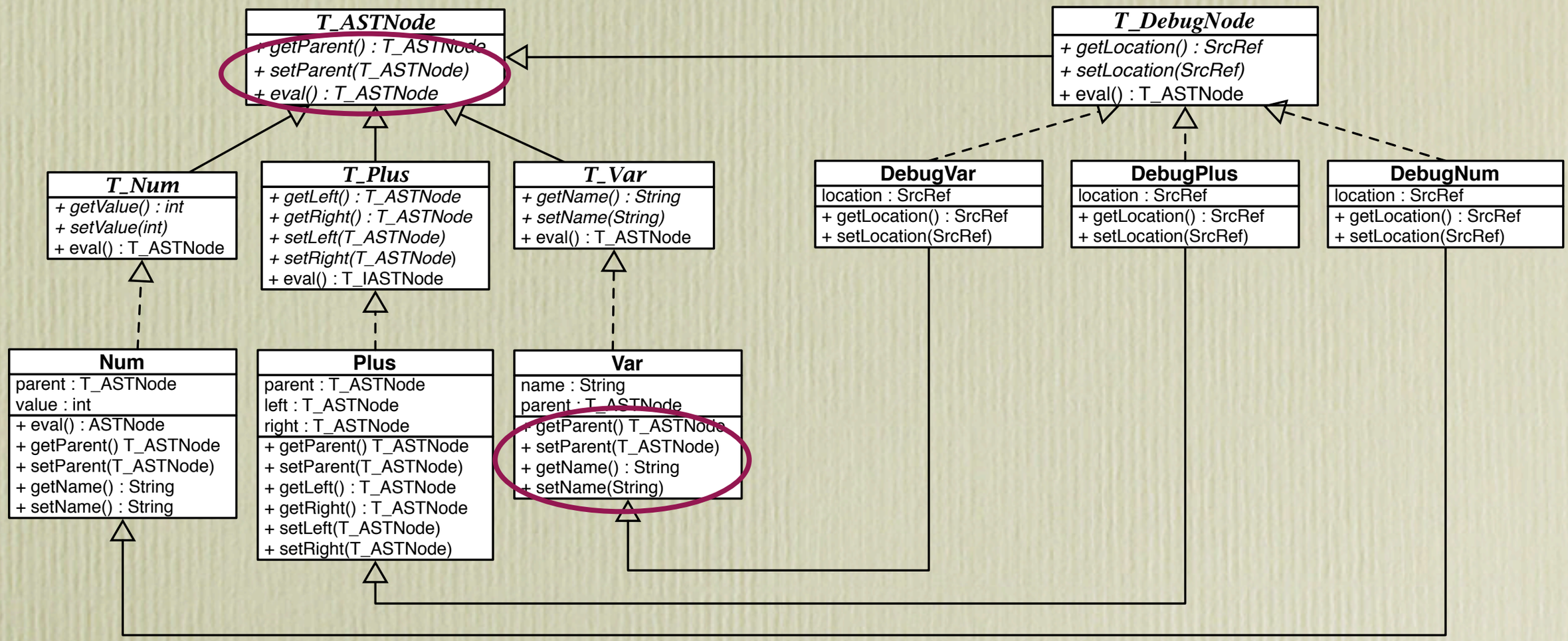



```
void ASTNode.defCheck() { ... }
void Var.defCheck()     { ... }
void Plus.defCheck()    { ... }
void Num.defCheck()     { ... }
```

Alternative Design: Single Inheritance



Alternative Design: Traits



- How to define “state” that is private to a trait?
- Can only define public and protected accessors

Advantages of the design

- Allow reuse of both methods *and* state
- Capture design intent through `requires`
- Allows subtyping without inheritance
- Can typecheck external methods in a straightforward, modular way

CZ Status

- Language formalized, extension of Featherweight Java
  - Proved sound, shown to be modular
- Designed an extension of Java that provides (no-diamond) multiple inheritance and `requires`
  - Implemented typechecker (using JastAdd)

Related Work

- Stateful traits: Bergel et al. [CLSS '05]
- Scala: Odersky and Zenger [OOPSLA '05]
  Odersky [tech report]
- Dubious: Millstein and Chambers, [I&C '02]
- JPred: Millstein et al [FOOL '06, TOPLAS '09]
- Fortress: Allen et al. [SAC '07]

Summary

- CZ takes a novel approach to the diamond problem
- Less restrictive than other solutions
- Allows multiple inheritance to co-exist with
  - State and initialization
  - Modular typechecking of multiple dispatch