Optional Typing in Dart: Purity vs. Practice

Gilad Bracha
Google

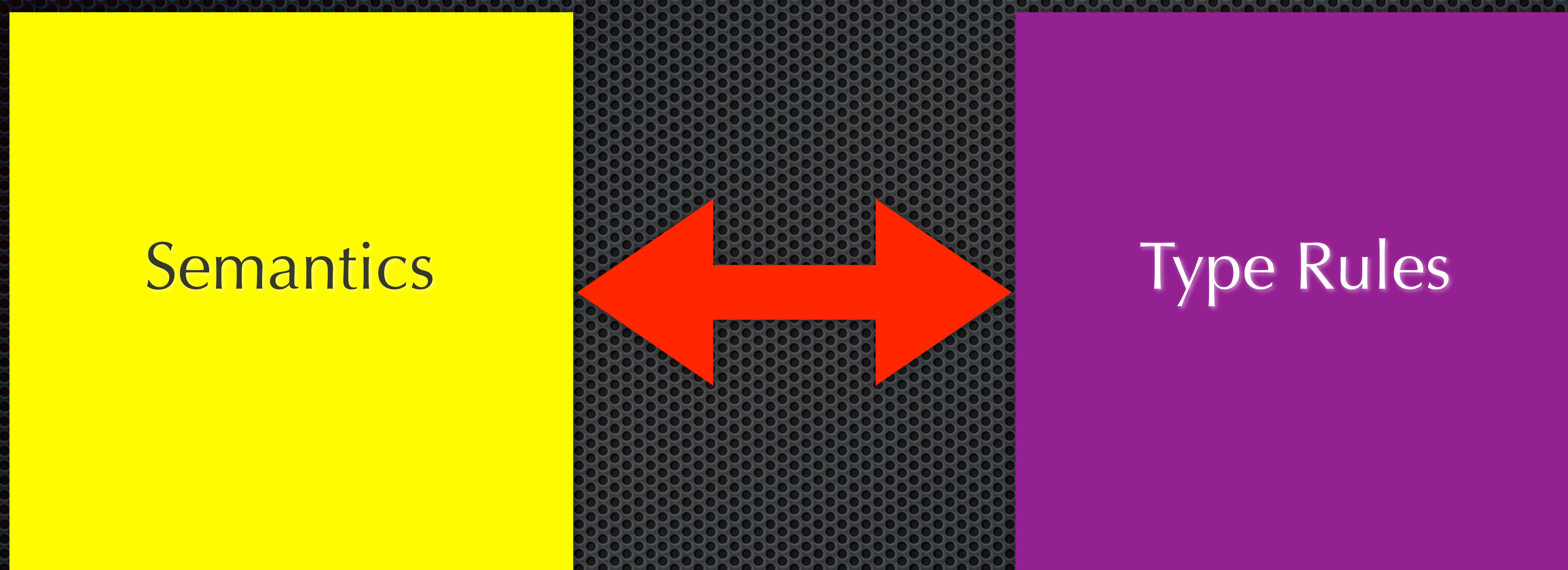
Optional Types

- A type system is optional if
  - It has no effect on the run time semantics
  - It is syntactically optional

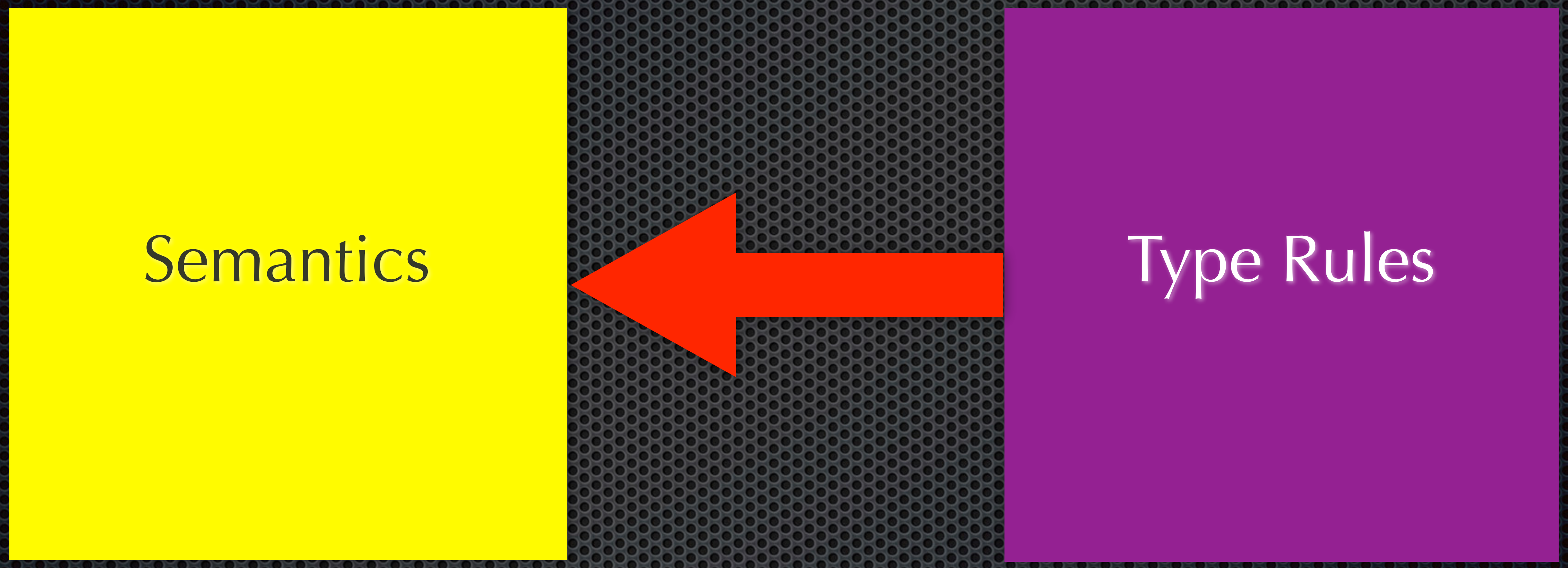
Optional Types

- A type system is optional if
  - **It has no effect on the run time semantics**
  - It is syntactically optional

Runtime dependent on Type System



Runtime not dependent on Type System



Dart Types Overview

- Nominal
- Interface-based
- Unsound for multiple reasons

Dart Types Overview

- Two Modes:
  - Checked ~ Gradual. Check runtime type against declaration at assignment, parameter passing, function return
  - Production ~ Optional. Type annotations have no effect.

Checked Mode undermines Optional Typing

- Code will be used in both checked and production modes
  - Checked mode gives annotation meaning
  - Hence annotations are not truly optional
- But checked mode is very useful

Controlling Checked Mode

- One needs finer grain control over checked mode
  - Ideally, one could choose on a library or method basis whether to do the dynamic checks
- Checked mode should be a feature of the tooling, not the language

Tangent: PX not PL

- Programming experience (PX) is what matters
- PX holistically integrates language, tools, libraries, performance etc.
- Separating PL is a very useful level of abstraction, but one needs to know when to do.

Pluggable Types

- If type systems are optional, one can treat them as plug ins
- Different type systems for different needs, e.g.:
  - Aliasing/Ownership/Capability tracking
  - Traditional types

Pluggable Types in Dart?

- No. Type rules are in the language spec.
  - Reason: worries about fragmentation, interop
- Yet pluggability arose in practice, in “strong-mode”, and its subsets, which we’ll discuss later

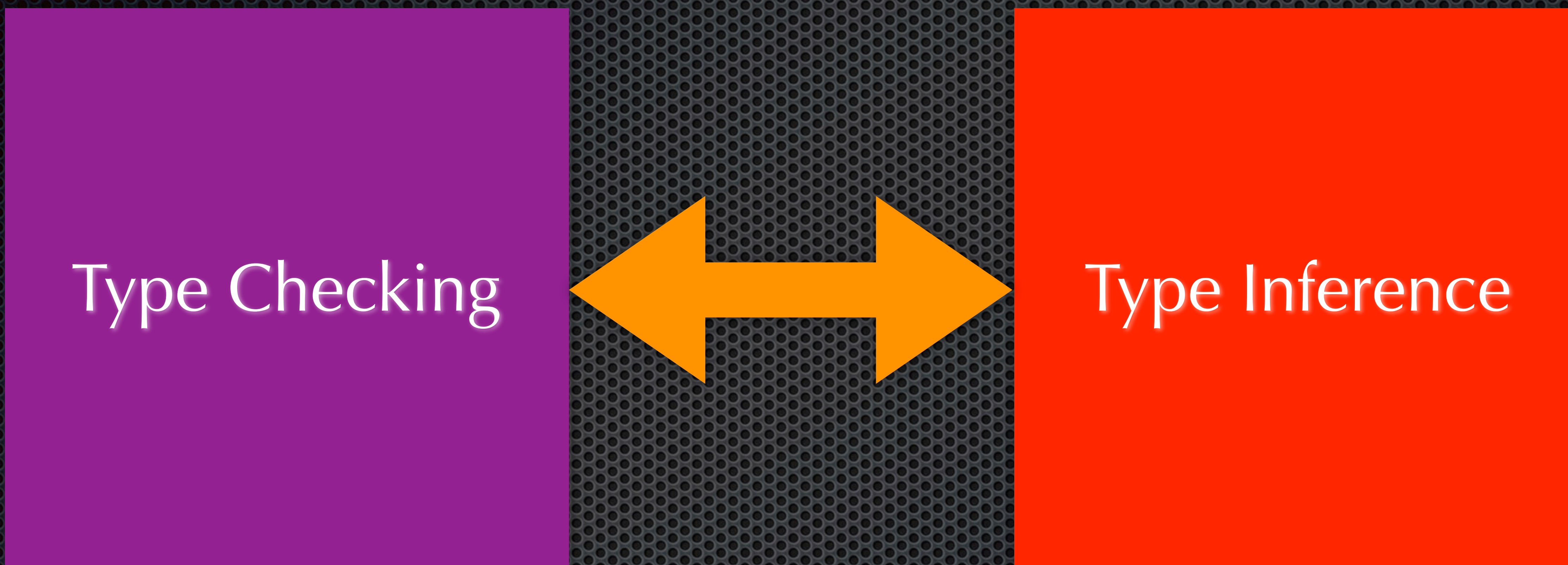
Soundishness

- Dart types are unsound in at least 3 ways:
  - Covariant generics
  - Implicit downcasting on assignment
    - The two above interact in odd function rules
  - Library privacy (ADTs) vs. interface types

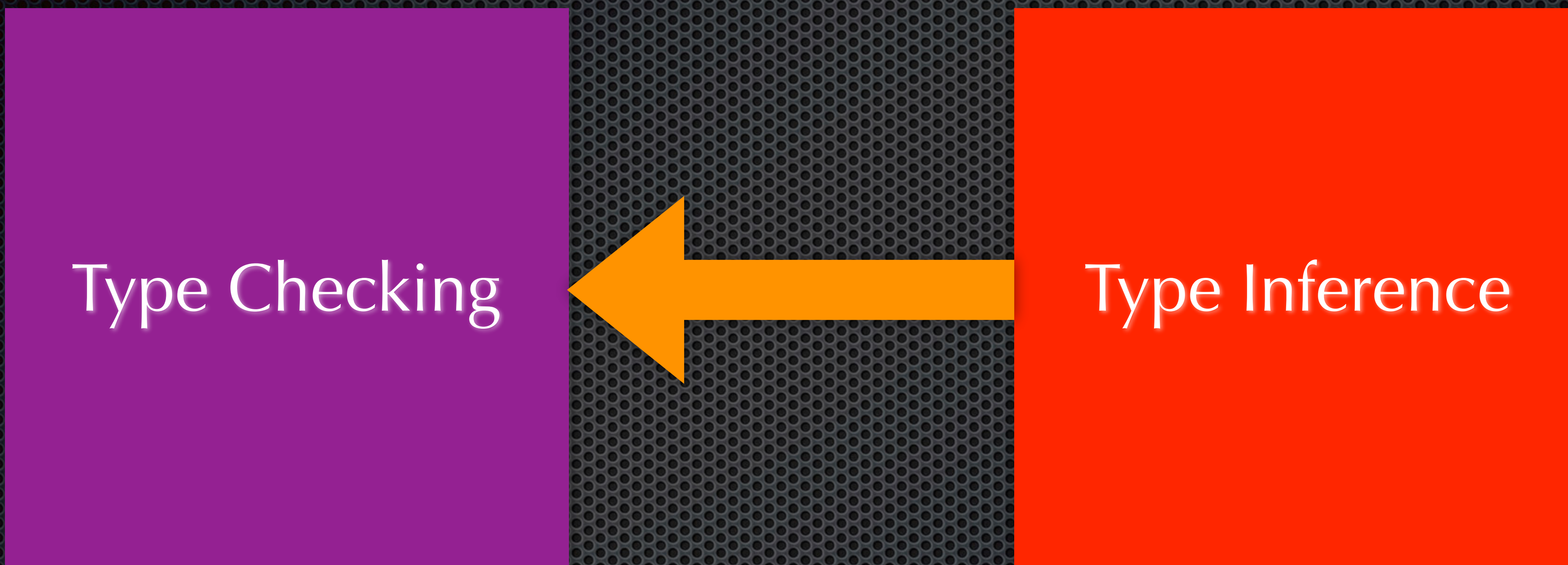
Type Inference

- Programmers want type inference
  - They don’t want to have to write types because they hate typing (with their fingers)
  - They don’t even want to read types when the types are obvious
    - ***var*** *i = 0;* // expect *i* to be inferred as *int*

Type System dependent on Inference



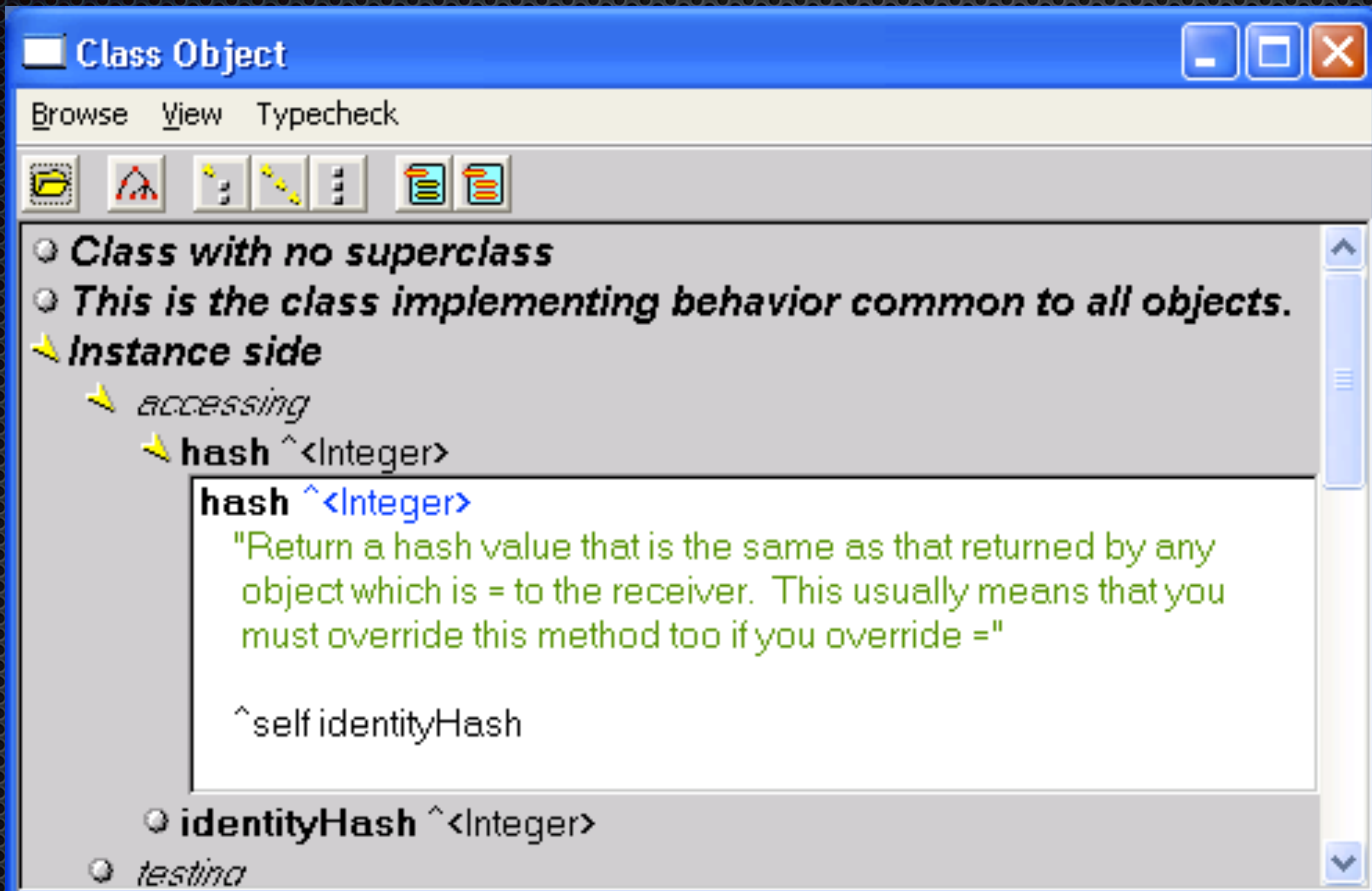
Type System not dependent on Inference



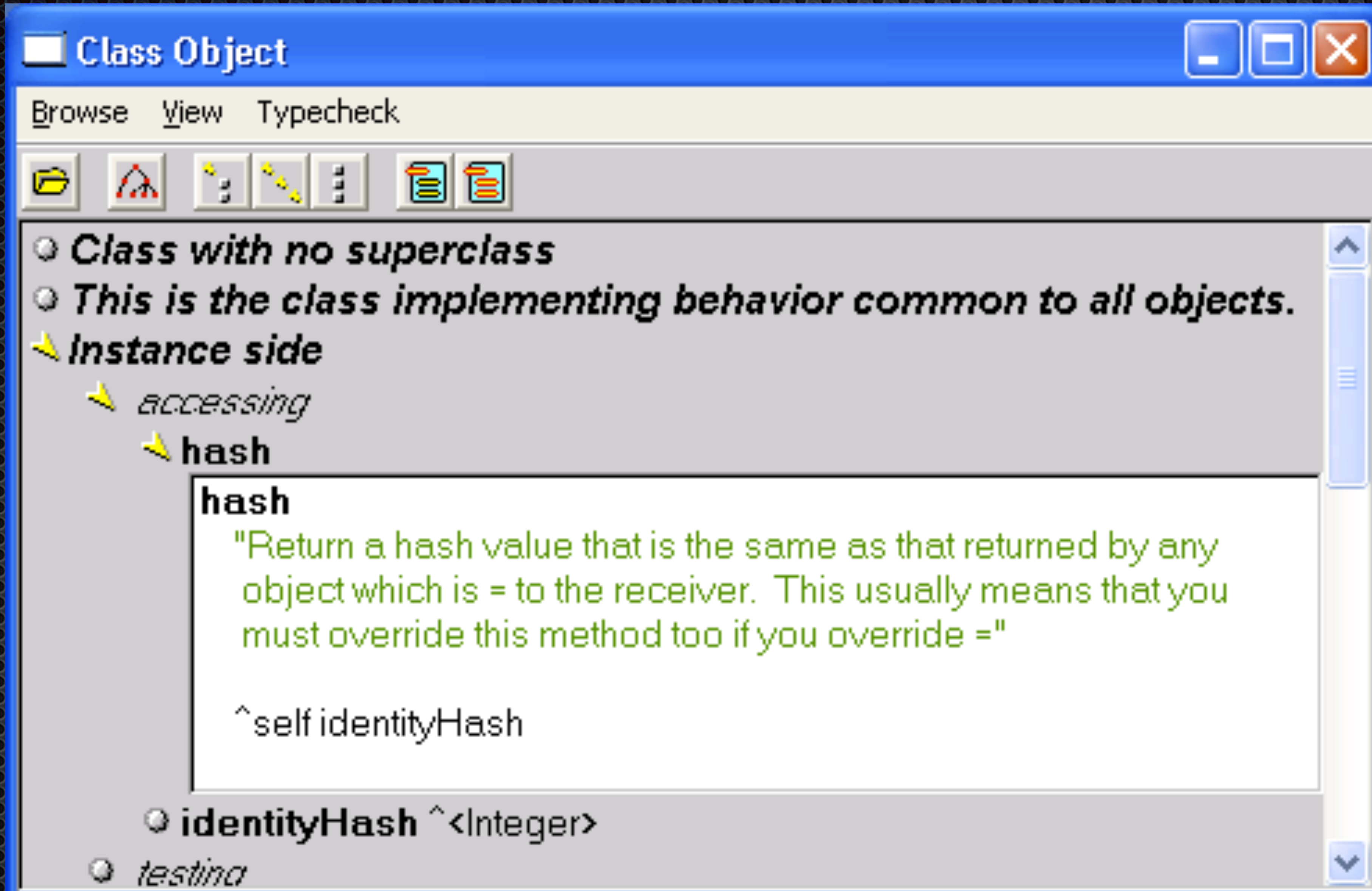
Optional Typing Requires Smart, Integrated Tools

- Checked mode control
- Type checking selectively
- Using metadata to disable undesired warnings

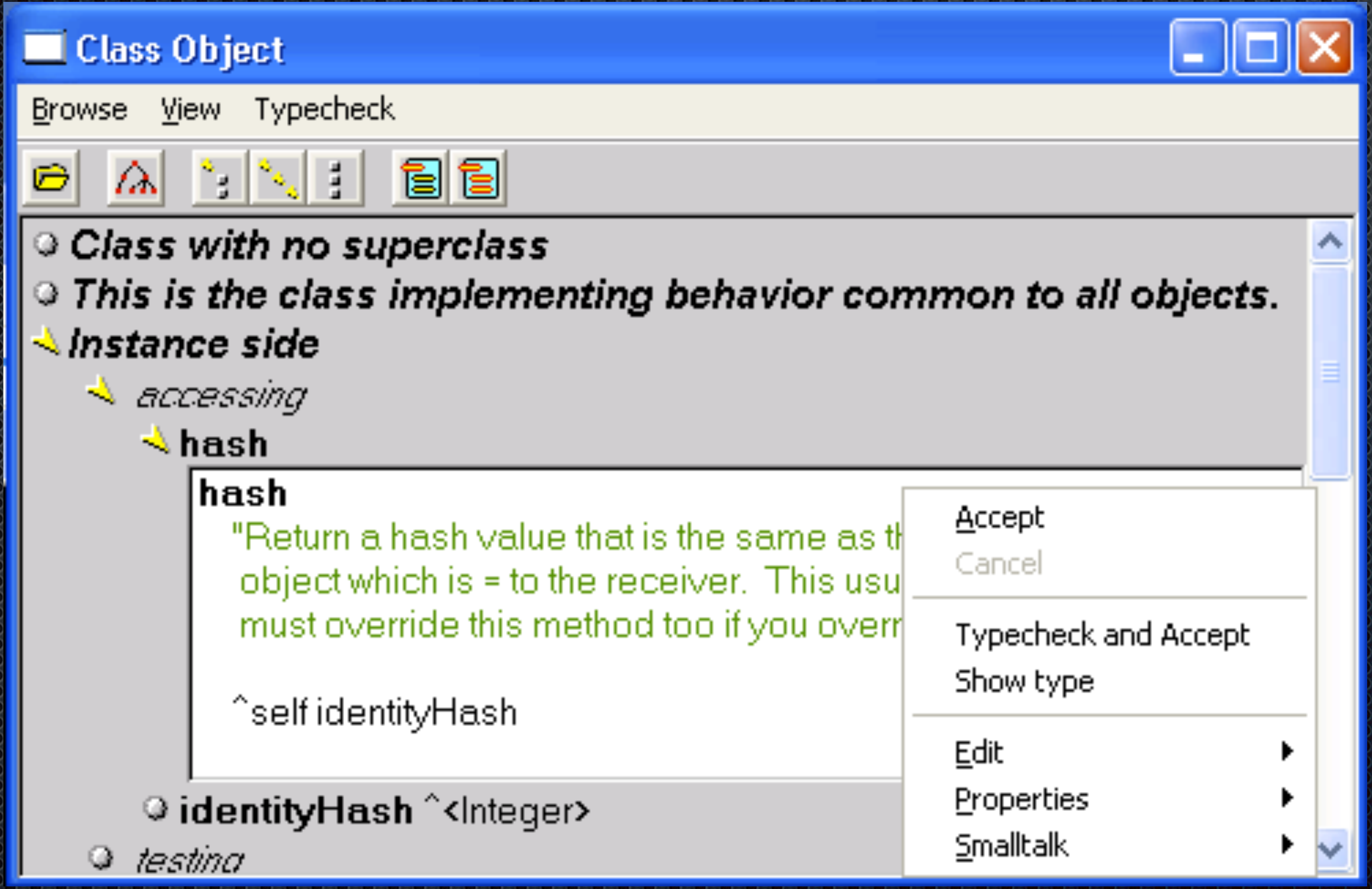
Object>>hash


Class Object
Browse View Typecheck
Class with no superclass
This is the class implementing behavior common to all objects.
Instance side
accessing
hash ^<Integer>
hash ^<Integer>
"Return a hash value that is the same as that returned by any object which is = to the receiver. This usually means that you must override this method too if you override ="
^self identityHash
identityHash ^<Integer>
testing

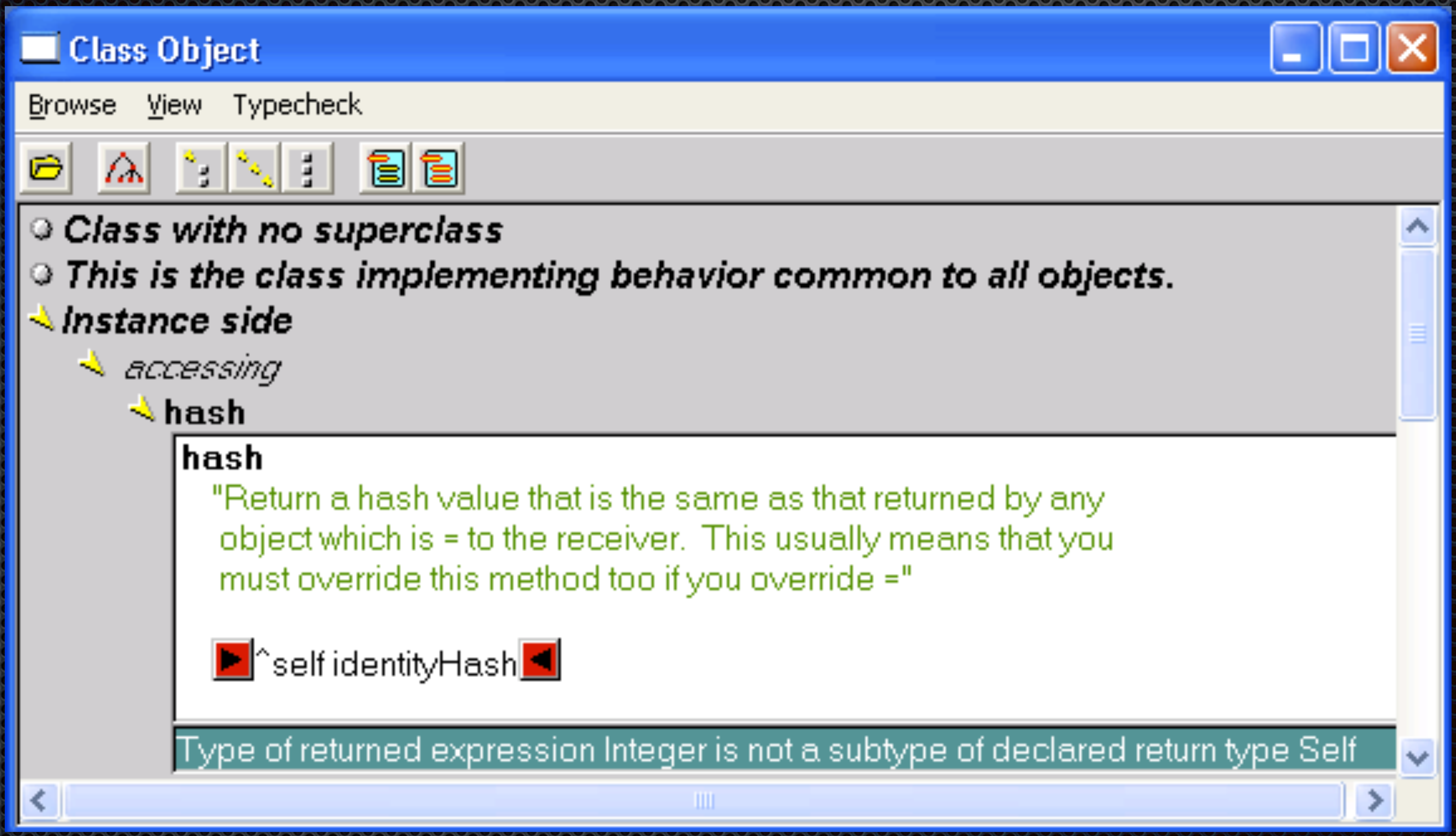
Object>>hash maltyped



Invoking the Typechecker



Type Errors


Class Object
Browse
View
Typecheck
Class with no superclass
This is the class implementing behavior common to all objects.
Instance side
accessing
hash
hash
"Return a hash value that is the same as that returned by any object which is = to the receiver. This usually means that you must override this method too if you override ="
^self identityHash
Type of returned expression Integer is not a subtype of declared return type Self

Type Annotations Create Expectations of Behavior

int i; // people expect *i* to be initialized to 0

C Syntax Aggravates

Given

var *i;*

engineers think ***var*** is a type meaning *dynamic*.

Rational Syntax is Resisted

***var** i: int := 0;*

Complaint is that this is too verbose, too unfamiliar

Types are Knowledge

Knowledge is Power

Implementors Lust for Power

- Especially true when classic VM technology is restricted, as when targeting the web or iOS

Size is the Big Problem

- Size of download on the web (more due to JS parse time than actual download)
- On iOS, no JIT, so we use AOT compilation to machine code, which gets big
- IOT - devices are super small

Size is the Big Problem

- In both web and mobile (even Android) non-native platform is at huge disadvantage; always a second-class citizen

The Return of Pluggable Types?

- Fully type programs prior to deployment
- Check programs under sound rules
- Capitalize on types in implementation

The Return of Pluggable Types?

- Dart's *strong mode* is somewhat similar
  - Check programs under sound-ish rules
  - Some teams define their own subsets
- One has to implement both behaviors :-). But really just like -Oxxx

Liveness

- Dart now allows code to be changed and reloaded without restarting
- Even if your code is full type safe, the pre-existing heap and stack may not conform
- If you rely on the types … Boom!
- So you need a mode that does not rely on types anyway

Conclusion

- Easier for pre-existing language; core language rules fixed, will keep you honest
- Hard to retrofit into conventional design
- Requires tight control over entire programming experience; not just language, but tools