A Walk on the Dart Side

A Quick Tour of

DART

Gilad Bracha

Joint Work with the Dart Team

Dart at 50,000 feet

Language for Web Programming

Sophisticated Web Applications need not be a tour de force

Google

Constraints

Instantly familiar to the mainstream programmer

Efficiently compile to Javascript

Google

Dart in a Nutshell

Purely Object-Oriented, optionally typed, class-based, single inheritance with actor-based concurrency

Google™

So what's so interesting?

*Pure Object-Oriented, **optionally typed**, class-based, single inheritance with actor-based concurrency*

Google™

Some Modest Innovations

Optional types

Built-in Factory Support

ADTs without types

Google

Some Modest Innovations

Optional types

ADTs without types

Built-in Factory Support

Google

Some Modest Innovations

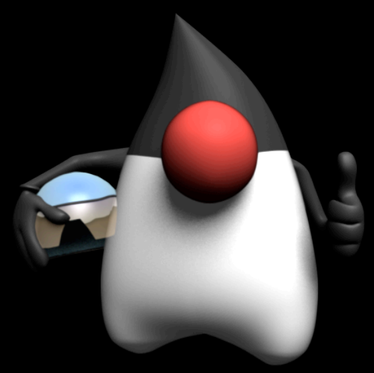
Optional types

ADTs without types

Built-in Factory Support

Google

Mandatory Types



Optional Types

DART

Google

Mandatory Types

Static type system regarded as mandatory

Maltyped programs are illegal

Google™

A Brief History of non-mandatory Types

Common Lisp

Scheme (soft typing)

Cecil

Erlang

Strongtalk

BabyJ

Gradual Typing

Google™

A Brief History of non-mandatory Types

Common Lisp

Scheme (soft typing)

Cecil

Erlang

Strongtalk

BabyJ

Gradual Typing

Google™

Optional Types

Syntactically optional

Do not affect run-time semantics

Google™

What does it look like?

Checked Mode

```
1  class Point {
2    Point(this.x, this.y);
3    var x, y;
4    operator +(other) => new Point(x + other.x, y + other.y);
5    scale(factor) => new Point(x * factor, y * factor);
6    distance() {
7      return Math.sqrt(x*x + y*y);
8    }
9  }
10
11 main() {
12   var a = new Point(10, 10);
13   var b = new Point(2, 3).scale(10);
14   print("distance=${(a+b).distance()}");
15 }
```

DART

Google

Mandatory Types: Pros

In order of importance:

Machine-checkable documentation

Types provide conceptual framework

Early error detection

Performance advantages

Google™

Mandatory Types: Cons

Expressiveness curtailed

Imposes workflow

Brittleness

Google™

Optional Types:
Can we have our Cake and Eat it Too?

Documentation (for humans and machines- but not verifiable)

Types provide conceptual framework

Early error detection

Performance advantages (much attenuated)

Google

Optional Typing Precludes ...

Type-based overloading

Type based initialization, e.g.,

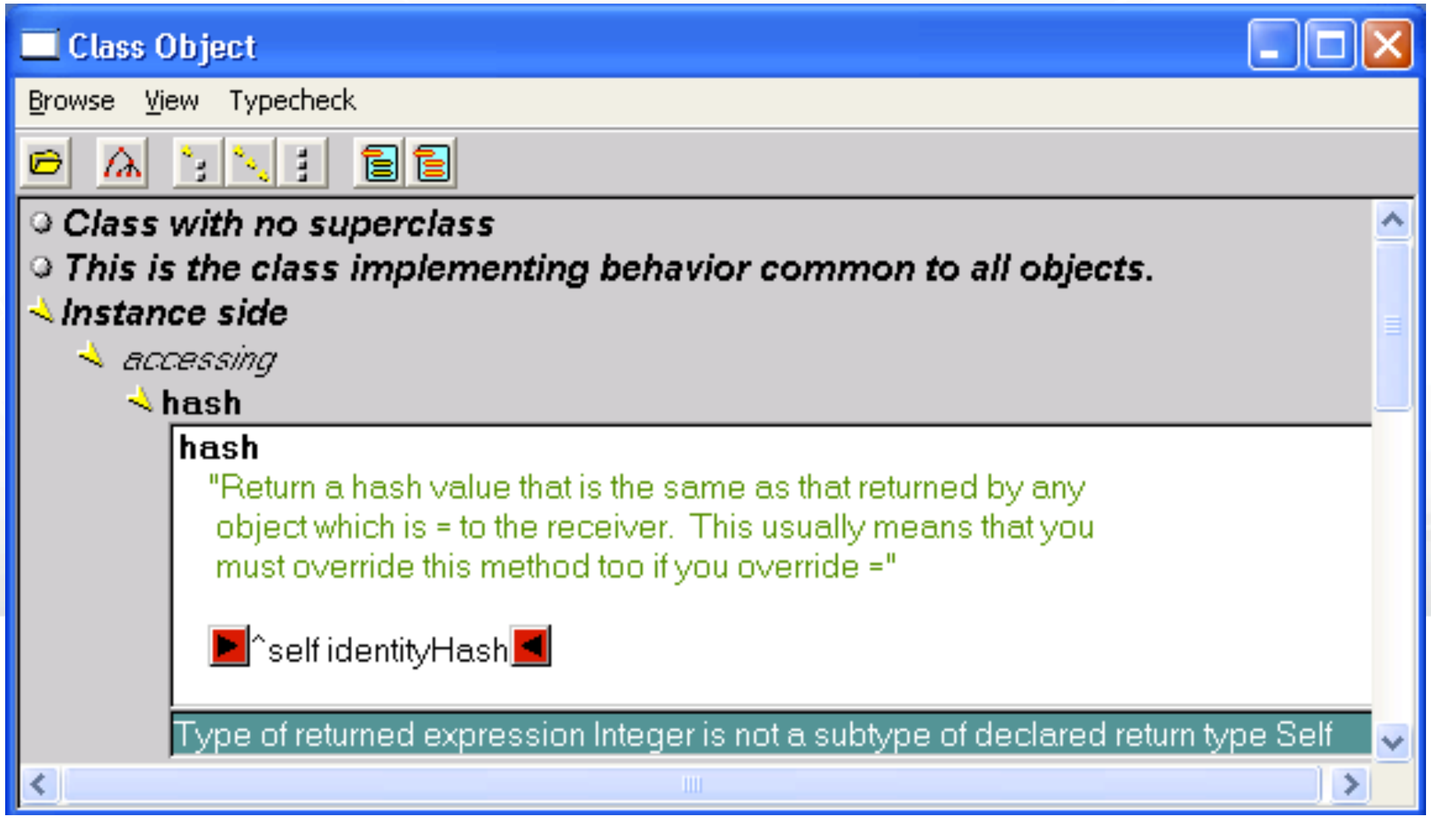
int i; *cannot mean* **var i: int = 0;**

Type classes, C# extension methods ...

Google

So what's actually new?

Didn't we have all this in Strongtalk in 1993?



DART

Google

Type Assertion Support

*Dart's optional types are best thought of as a type assertion mechanism, **not** a static type system*

Google™

Dart Types at Runtime

- *During development one can choose to validate types*
  - `T x = o;` ⟶ `assert(o === null || o is T);`
- *By default, type annotations have no effect and no cost*
- *Code runs free*

Google™

Checked Mode

DART

Google™

Not your Grandfather's Type System

Not a type system at all -

rather a static analysis tool based on heuristics, coupled to a type assertion mechanism

Google™

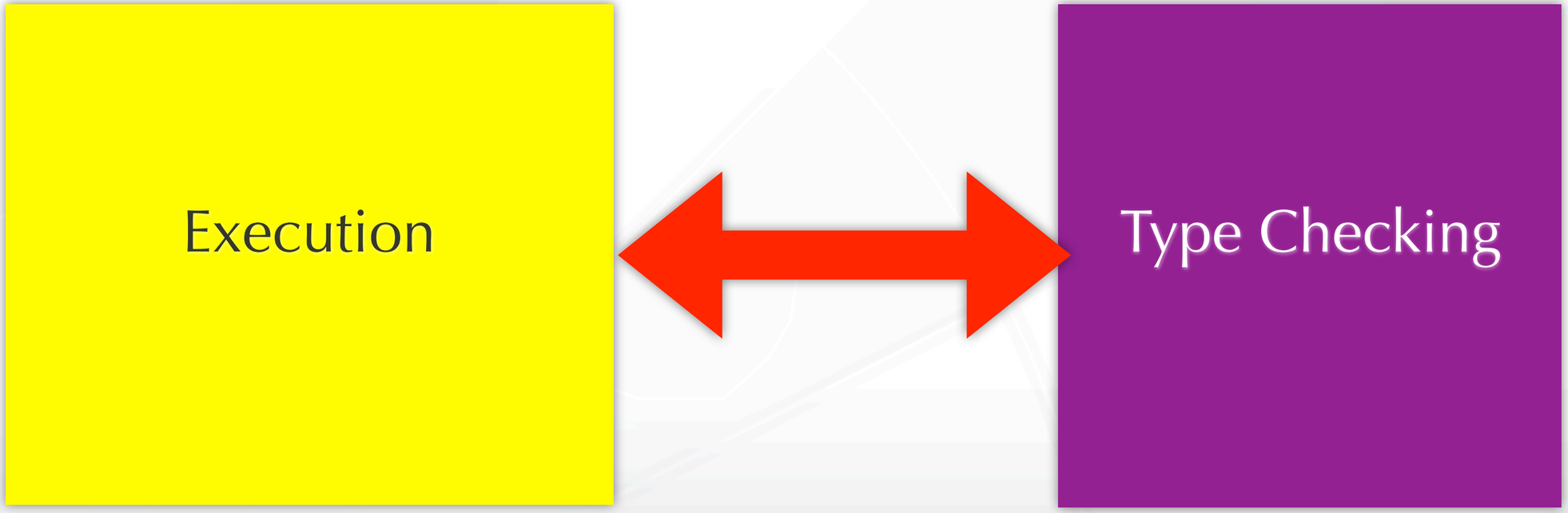
What about a real, sound, type system?

There is no privileged type system, but pluggable types are possible

For example, one can write a tool that interprets existing type annotations strictly

Google™

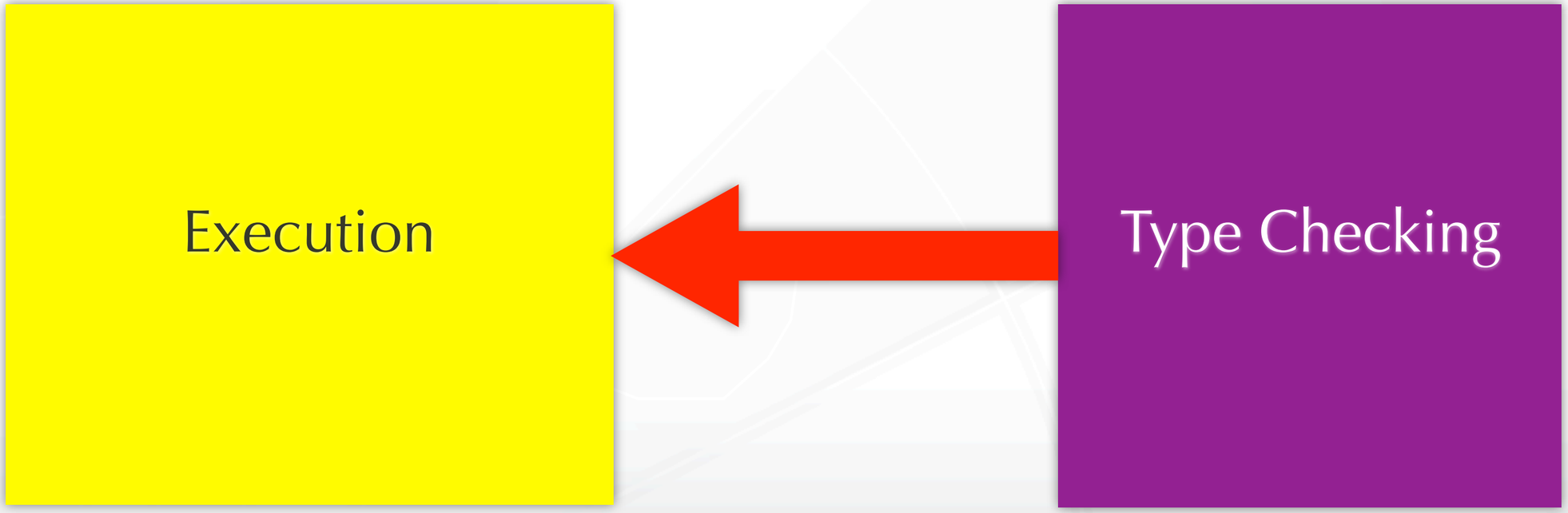
Runtime dependent on Type System



DART

Google

Runtime Independent of Type System



DART

Google™

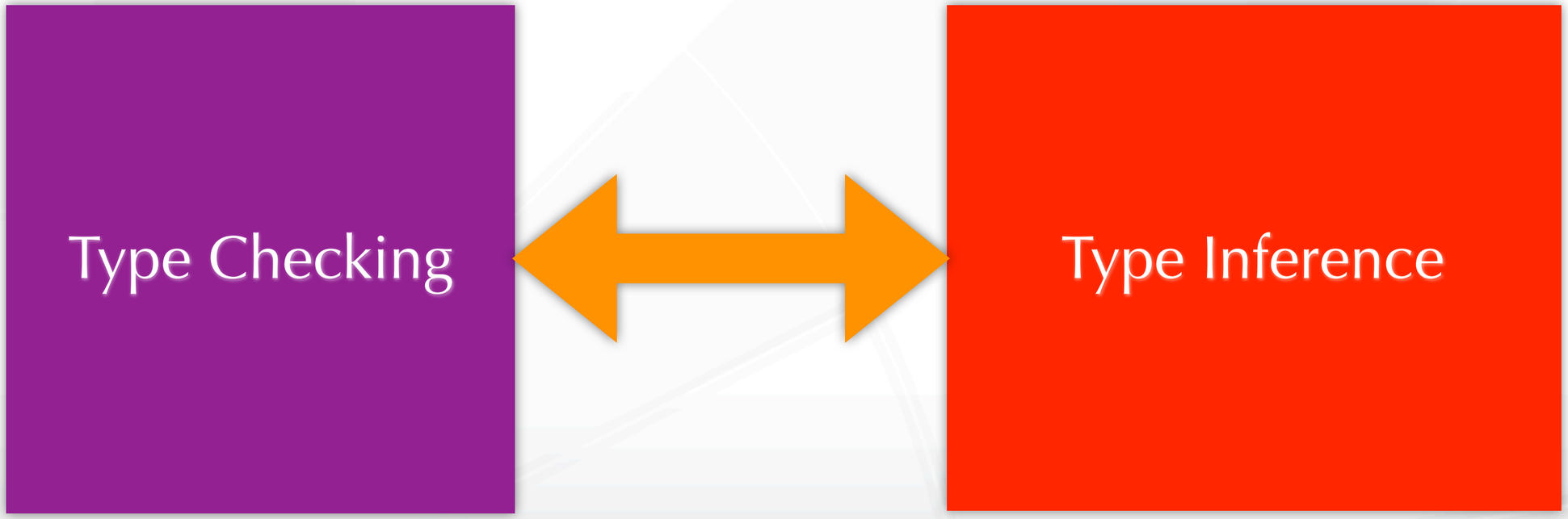
What about type inference?

Type Inference relates to Type Checking as Type Checking to Execution

Type inference best left to tools

Google™

Type System dependent on Type Inference



DART

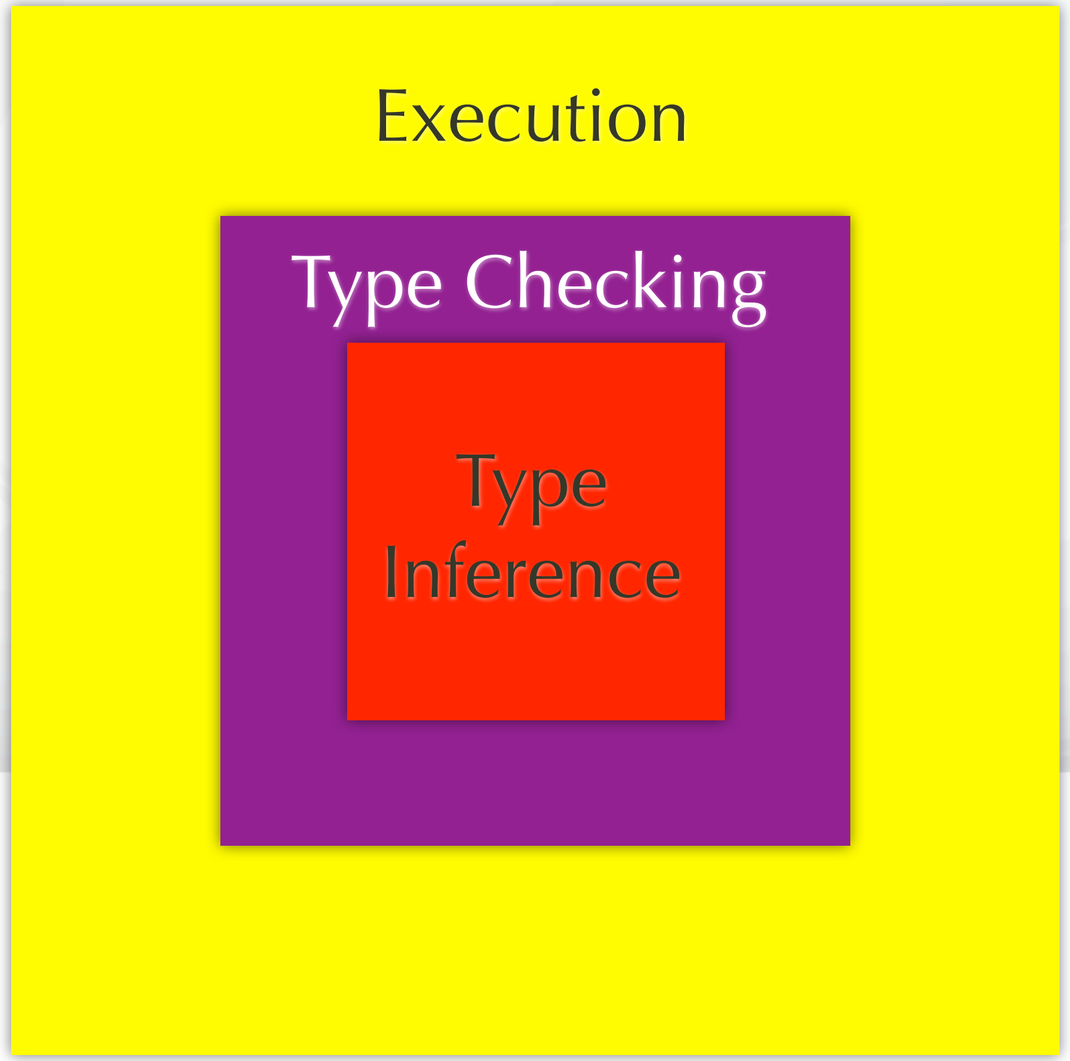
Google

Type System Independent of Type Inference

DART

Google

Don’t get Boxed-In



DART

Google™

Interfaces

Every class induces an implicit interface

Interfaces are reified at runtime

Type tests are interface based

You can implement the interface of another class without subclassing it

Google

Generics

Reified

Covariant subtyping

```
print(new List<String>() is List<Object>);
print(new List<Object>() is List<String>);
print(new List<String>() is List<int>);
print(new List<String>() is List);
print(new List() is List<String>);
```

Yes, Virginia, it isn't sound

DART

Google

Optional Types and Reified Types

Annotations do not affect semantics

Type arguments to constructors? Interfaces?

Google™

Optional Types and Reified Types

Annotations do not affect semantics

Type arguments to constructors? Interfaces?

Type Arguments to constructors are optional, but are reified

Type tests are a dynamic construct that relies on reified interfaces

Google

Summary: Optional Types

- *Static checker provides warnings; tuned to be unobtrusive*
- *Type annotations have no effect except ...*
- *During development, you can check dynamic types against declarations*

Google

But is it Dynamic?

noSuchMethod

Mirrors & Debugging

Google™

Some Modest Innovations

Optional types

ADTs without types

Built-in Factory Support

Google

Libraries and ADTs

A Library is a set of top-level classes, interfaces and functions

Libraries may be be mutually recursive

Libraries are units of encapsulation

Google

Libraries and ADTs

Library based privacy

- based on names

- _foo is private to the library

- naming and privacy are not orthogonal :-(

- privacy can be recognized context-free :-)

Google

Interfaces vs. ADTs

How to reconcile?

- interfaces based on externally visible behavior

- ADTs based on implementation

Google™

Interfaces vs. ADTs

What happens when we implement an interface with private members?

```
// in library 1

class A { var _foo = 0;}

foo(A a) => a._foo;

// in library 2

class B implements A {int get _foo()=> 42;}

foo(new B());
```

Google

Interfaces vs. ADTs

What happens when we implement an interface with private members?

```
// in library 1

class A { var _foo = 0;}

foo(A a) => a._foo

// in library 2

class B implements A {int get _foo()=> 42;} // Warning?

foo(new B());
```

Google

Interfaces vs. ADTs

What happens when we implement an interface with private members?

```
// in library 1

class A { var _foo = 0;}

foo(A a) => a._foo; // Warning?

// in library 2

class B implements A {int get _foo()=> 42;}

foo(new B());
```

Google

Interfaces vs. ADTs

```
class B implements A {

  int get _foo()=> 42;

  noSuchMethod(msg){

    msg.name = '_foo' ?msg.sendTo(this): super.noSuchMethod(msg);

  }

}
```

Google

Some Modest Innovations

Optional types

ADTs without types

Built-in Factory Support

Google™

Factories

Constructors without tears

Use caches, return other types of objects

Instance creation expressions based on interfaces

Minimize need for Dependency Injection

Google

Factories

Checked Mode

```
1  interface Person factory PersonFactory {
2    Person(name);
3    final name;
4  }
5
6  class PersonFactory {
7    factory Person(name) {
8      if (name == null) return const Ghost();
9      return new RealPerson(name);
10   }
11 }
12
13 class RealPerson implements Person {
14   RealPerson(this.name);
15   final name;
16 }
17
18 class Ghost implements Person {
19   const Ghost();
20   get name() => "ghost";
21 }
22
23 main() {
24   print(new Person("gilad") is RealPerson);
25   print(new Person(null) is Ghost);
26 }
```

DART

Google

Dart is not Done

- *Mixins?*
- *Reflection*
- *High level actor semantics: await? Erlang-style pattern matching? Promise-pipelining?*
- *Class nesting? First class libraries? Non-nullable types?*
- *Metadata? Pluggable types?*

Google

Q & A

Google