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
Constraints

Instantly familiar to the mainstream programmer

Efficiently compile to Javascript
Dart in a Nutshell

Purely Object-Oriented, optionally typed, class-based, single inheritance with actor-based concurrency
So what’s so interesting?

Pure Object-Oriented, optionally typed, class-based, single inheritance with actor-based concurrency
Some Modest Innovations

Optional types

Built-in Factory Support

ADTs without types
Some Modest Innovations

Optional types

ADTs without types

Built-in Factory Support
Some Modest Innovations

*Optional types*

*ADTs without types*

*Built-in Factory Support*
Mandatory Types

Scala

Optional Types

DART
Mandatory Types

Static type system regarded as mandatory

Maltyped programs are illegal
A Brief History of non-mandatory Types

Common Lisp
Scheme (soft typing)
Cecil
Erlang
Strongtalk
BabyJ
Gradual Typing
A Brief History of non-mandatory Types

Common Lisp
Scheme (soft typing)
Cecil
Erlang
Strongtalk
BabyJ
Gradual Typing
Optional Types

Syntactically optional

Do not affect run-time semantics
What does it look like?

```dart
class Point {
  Point(this.x, this.y);
  var x, y;
  operator +(other) => new Point(x + other.x, y + other.y);
  scale(factor) => new Point(x * factor, y * factor);
  distance() {
    return Math.sqrt(x*x + y*y);
  }
}

main() {
  var a = new Point(10, 10);
  var b = new Point(2, 3).scale(10);
  print("distance=${(a+b).distance()}");
}
```
Mandatory Types: Pros

In order of importance:

- Machine-checkable documentation
- Types provide conceptual framework
- Early error detection
- Performance advantages
Mandatory Types: Cons

*Expressiveness curtailed*

*Imposes workflow*

*Brittleness*
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)
Optional Typing Precludes ...

*Type-based overloading*

*Type based initialization, e.g.,*

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

*Type classes, C# extension methods* ...
So what’s actually new?

Didn’t we have all this in Strongtalk in 1993?
Type Assertion Support

Dart’s optional types are best thought of as a type assertion mechanism, **not** a static type system
Dart Types at Runtime

- *During development one can choose to validate types*
  
  \[
  T \ x = o; \quad \text{assert}(o == null || o is T);
  \]

- *By default, type annotations have no effect and no cost*

  - Code runs free
Checked Mode

http://localhost:4020/s/Jw

Checked Mode
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*
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
Runtime dependent on Type System
Runtime Independent of Type System
What about type inference?

Type Inference relates to Type Checking as Type Checking to Execution

Type inference best left to tools
Type System dependent on Type Inference
Type System Independent of Type Inference
Don’t get Boxed-In
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
Generics

Reified

Covariant subtyping

```java
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
Optional Types and Reified Types

Annotations do not affect semantics

Type arguments to constructors? Interfaces?
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
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*
But is it Dynamic?

noSuchMethod

Mirrors & Debugging
Some Modest Innovations

Optional types

ADTs without types

Built-in Factory Support
Libraries and ADTs

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

Libraries may be mutually recursive

Libraries are units of encapsulation
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 :-)

Interfaces vs. ADTs

How to reconcile?

- interfaces based on externally visible behavior

- ADTs based on implementation
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());
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());
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());
Interfaces vs. ADTs

```dart
class B implements A {

  int get _foo()=> 42;

  noSuchMethod(msg){
    msg.name = '_foo' ? msg.sendTo(this): super.noSuchMethod(msg);
  }
}
```
Some Modest Innovations

Optional types

ADTs without types

Built-in Factory Support
Factories

*Constructors without tears*

*Use caches, return other types of objects*

*Instance creation expressions based on interfaces*

*Minimize need for Dependency Injection*
Factories

```dart
interface Person

class PersonFactory {
  Person(name);
  final name;
}

class PersonFactory {
  factory Person(name) {
    if (name == null) return const Ghost();
    return new RealPerson(name);
  }
}

class RealPerson implements Person {
  RealPerson(this.name);
  final name;
}

class Ghost implements Person {
  const Ghost();
  get name() => "ghost";
}

main() {
  print(new Person("gilad") is RealPerson);
  print(new Person(null) is Ghost);
}
```
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?