(ELS '13)

Asynchronous Programming in Dart

Streams are the Future

Florian Loitsch, Google

DART

Who am I?

Florian Loitsch, software engineer at Google

Projects

- **Hop** - multi-tier language based on Bigloo
- **Scheme2Js** - Scheme-to-JavaScript compiler
- **V8** - high-performance JavaScript virtual machine
- **Dart** - structured programming for the web
  Tech lead of the core libraries for the last 9 months.

What is Dart?

What is Dart?

- Unsurprising object-oriented programming language
- Class-based single inheritance
- Familiar syntax with proper lexical scoping
- Optional static type annotations

```
main() {
  for (int i = 99; i > 0; i--) {
    print("$i bottles of beer on the wall, ....");
    print("Take one down and pass it around ...");
  }
}
```

DART

Motivations

We want to build great web applications.

- Web apps are still rare or unconvincing.
- Big applications are a pain to write and maintain (Gmail).

We should have everything we need:

- Good VMs
- Good UI elements

Writing big applications in JavaScript is too hard.

DART

It's not Lisp or Scheme

Let's get this out of the way. Dart is not Lisp or Scheme.

Dart does not have:

- Tail call optimizations.
- Var args.
- Macros.
- Method overloading.
- Easy-to-parse syntax.
- Continuations.

Dart Features

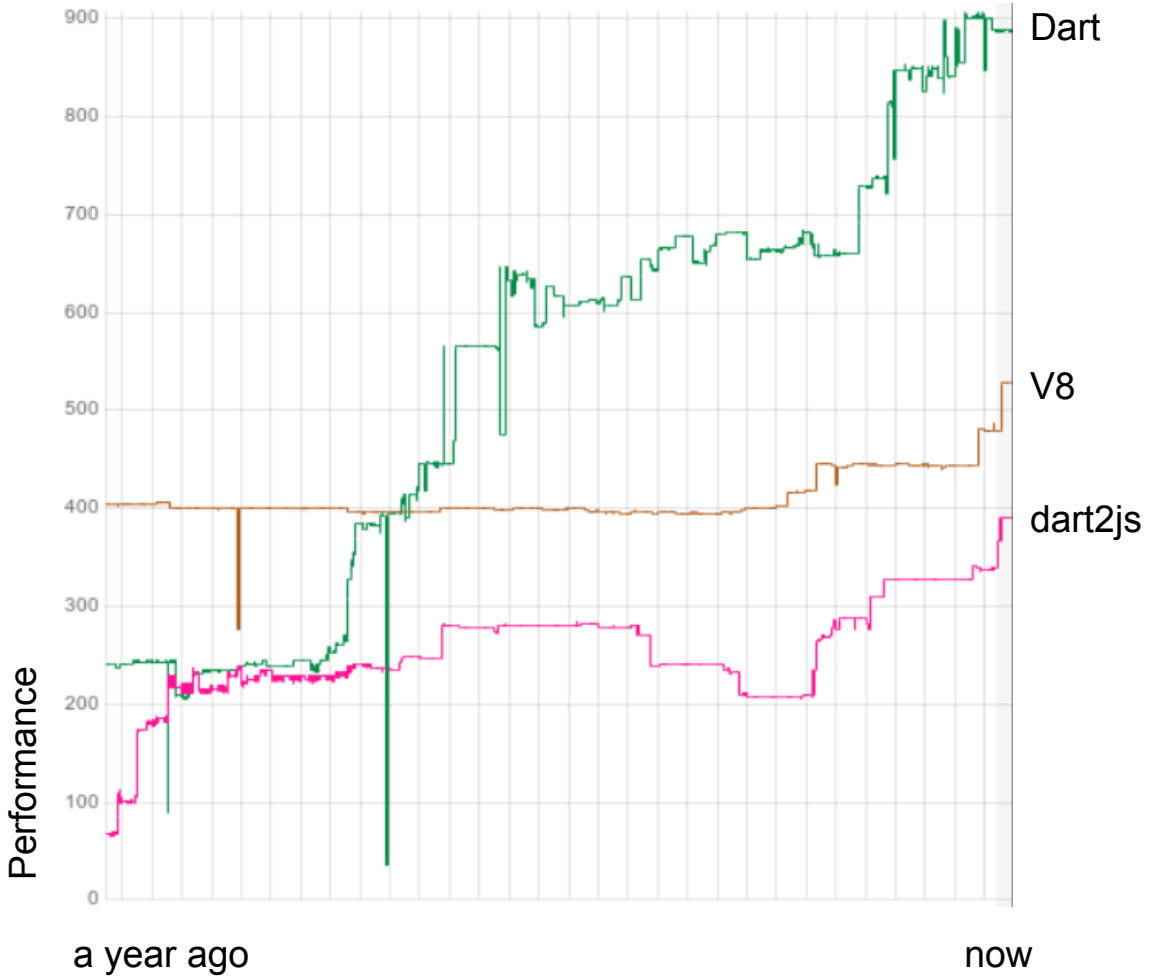
Dart features:

- fast VM and good Dart-to-JavaScript compiler.
- optional static types.
- easy to pick up.
- scales to big projects.
- fast development cycle (F5).
- closures.
- mixins.
- Erlang inspired processes ("*isolates*").
- integers and doubles.
- packages and libraries.
- good core library.

Benchmarks - Richards

OS kernel simulation benchmark, originally written in BCPL by Martin Richards.

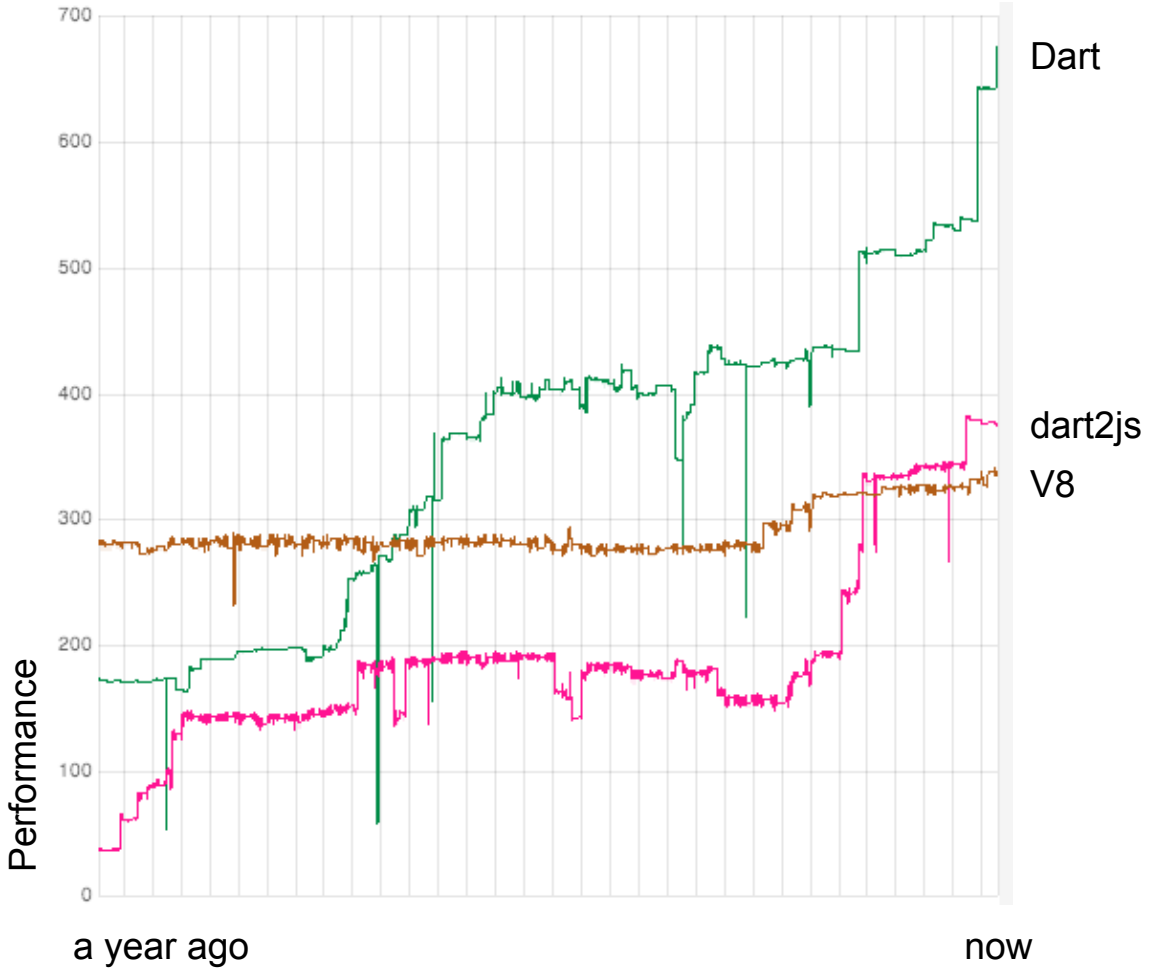
V8 has been tuned for this benchmark for the last six years.



Benchmarks - Delta Blue

One-way constraint solver, originally written in Smalltalk by John Maloney and Mario Wolczko.

V8 has been tuned for this benchmark for the last six years.



Closures

Two ways to write closures:

```
(x, y) {
  foo(x);
  bar(y);
  return 42;
}

(x) => x + 1;
(x) { return x + 1; }
```

Core Libraries

DART

Core Libraries

- `dart:core`
- `dart:collection`
- `dart:async`
- `dart:isolate`
- `dart:mirror`
- `dart:math`
- `dart:io`
- `dart:html`
- …

dart:core

Core is always automatically imported.

Contains important classes like:

- `Iterable`.
- `List` (Arrays).
- `Set`.
- `Map`.
- `RegExp`.

Iterables

Iterables are the foundation of the core library: a sequence of elements.

```
abstract class Iterable<E> {
  Iterator<E> get iterator;

  Iterable map(Function f);
  Iterable<E> where(Function f);
  E reduce(combine(E value, E element));
  E get first;
  void forEach(void f(E element));
  ...
};
```

the argument `combine` is typed as a function that takes two `E`s.

DART

Iterables cont.

Iterables are lazy.

```
[1, 2].map((x) => x + 1).forEach(print); // 2 3
[1, 2].map((x) => print(x + 1));  // Nothing.

filter-map:
  iterable.map(...)
          .where((x) => x != false)
          .toList();
```

Iterables methods

```
iterator,
map(), where(), expand(), contains(),
forEach(), reduce(), fold(), every(),
any(), join(), length, isEmpty, take(),
skip(), takeWhile(), skipWhile(), first,
last, single, firstWhere(), lastWhere(),
singleWhere(), elementAt(), toList(),
toSet()
```

dart:async

Asynchronous programming

Isolates can run in parallel but their heaps are independent. Communication between two isolates must go through ports.

Dart is single-threaded within one isolate.

Asynchronous programming with callbacks.

Callback Hell

It is easy to produce unreadable code with callbacks (and I'm not even adding the second error-handling argument).

```
asyncFind(needle, (x) {
  asyncProcess(x, (y) {
    asyncPolish(y, (z) {
      asyncSet(z, () {
        print("done");
      });
    });
  });
});
```

Callback Hell cont.

Familiar problem for Schemers and Lispers.

```
(asyncFind
    needle
    (lambda (x)
      (asyncProcess
          x
          (lambda (y)
            (asyncPolish
                y
                (lambda (z)
                  (asyncSet
                      z
                      (lambda () (print "done")))))))))
```

DART

Callback Hell - Naming methods

Avoid nesting by naming functions.

```
handleSet() => print("done");
handlePolished(x) => asyncSet(x, handleSet);
handleProcessed(x) => asyncPolish(x, handlePolished);
handleFound(x) => asyncProcess(x, handleProcessed);
asyncFind(needle, handleFound);
```

DART

Futures and Streams

dart:async makes asynchronous programming easier:

- Future (aka Promise): an asynchronous (one-shot) result.
- Stream: an asynchronous sequence of elements.

Future

Future

A future (aka "Promise") represents a future value (or error).

Allows to write previous example in concise compact way.

```
asyncFind(needle)  // returns a Future
   .then(asyncProcess)  // some magic...
   .then(asyncPolish)
   .then(asyncSet)
   .then((_) => print("done");
```

DART

Futurs - then, catchError

Users register callbacks which are executed when the value/error is available.

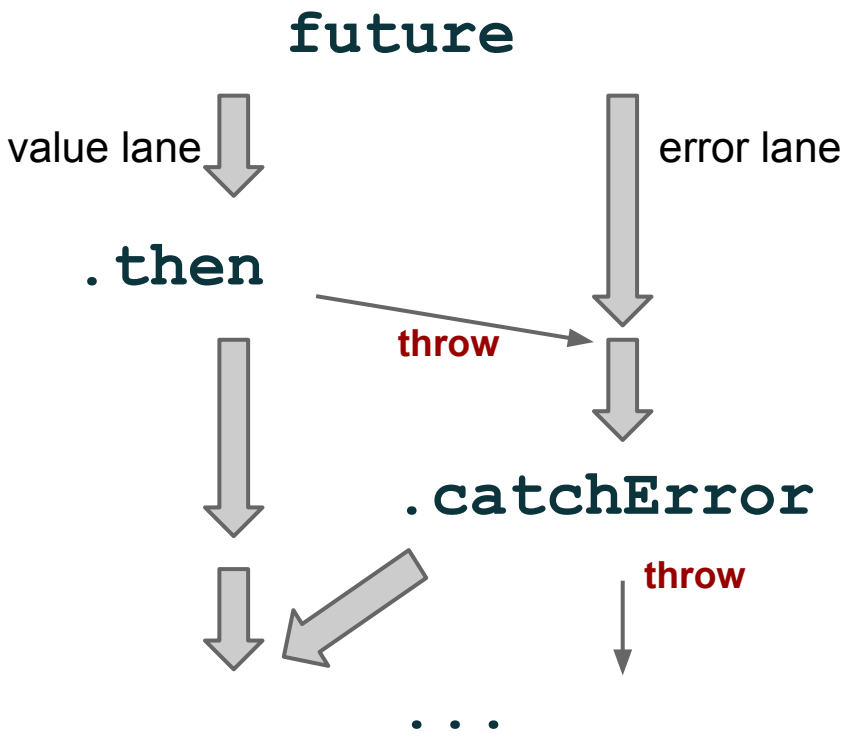
Futures can be completed in two ways:

- with a value. Callback registered with `then` is executed.
- with an error. Callback of `catchError` is executed.

Registering a callback returns a new Future.

Two Lanes

Two lanes for values and errors:



DART

Future - running example

```
asyncFind(needle)
   .then((x) {  // Longer version.
     var future = asyncProcess(x);
     return future;
   })
   .then(asyncPolish)  // Chains the result.
   .then(asyncSet)
   .then((_) => print("done");
```

Futures - Some Magic

- Futures chain automatically

```
future.then((x) {
  return anotherFuture;
}).then((y) {  // Waits for anotherFuture.
  print(y);  // Prints result of anotherFuture.
});
```

One of the very few cases where the core library looks at the type.

Streams

Streams

Heavily inspired by Rx (Reactive Extensions).
"Your Mouse is a Database" by Erik Meijer

Idea: a **`Stream`** is the asynchronous version of an **`Iterable`**.

- Iterables are pulled.
- Streams push.

Stream - Push

```
// Iterables are pulled.
var iterator = [1, 2, 3].iterator;
while (iterator.moveNext()) {
  print(iterator.current);
}

// Streams push:
var stream = element.onClick;
stream.listen(print);
```

Stream - Subscription

`listen` returns a **`StreamSubscription`**.

`StreamSubscription` has `cancel`, `pause` and `resume` methods.

```
abstract class StreamSubscription<T> {
  void cancel();
  void pause([Future resumeSignal]);
  void resume();
  ...
}
```

the argument **`resumeSignal`** is optional.

A listener gets all elements unless it cancels its subscription.

DART

Stream - Subscription cont.

Different from DOM.

In JavaScript:

```
var div = document.getElementById('div');
var listener = function (event) { /* do something here */ };
div.addEventListener('click', listener, false);
div.removeEventListener('click', listener, false);
```

Streams Everywhere

Streams are used extensively in Dart.

- dart:html:

```
element.onMouseDown.listen(...);
element.onScroll.listen(...);
```

- dart:io:

```
file.openRead.listen(...);
new Directory("/tmp")
   .list(recursive: true)
   .listen(...);
```

DART

Stream - Iterable

Streams are extremely similar to Iterables

```
abstract class Stream<E> {
  listen(...);

  Stream map(Function f);
  Stream<E> where(Function f);
  Future<E> reduce(combine(E value, E element));
  Future<E> get first;
  void forEach(void f(E element));
  ...
};
```

Stream - Iterables 2

For comparison. Here is the Iterables class.

```
abstract class Iterable<E> {
  Iterator<E> get iterator;

  Iterable map(Function f);
  Iterable<E> where(Function f);
  E reduce(combine(E value, E element));
  E get first;
  void forEach(void f(E element));
  ...
};
```

DART

Streams methods

```
listen,
map(), where(), expand(), contains(),
forEach(), reduce(), fold(), every(),
any(), join(), length, isEmpty, take(),
skip(), takeWhile(), skipWhile(), first,
last, single, firstWhere(), lastWhere(),
singleWhere(), elementAt(), toList(),
toSet(),

transform, pipe
```

Streams Example

```
Socket.connect("localhost", 8888)
  .then((socket) {
    new File("/etc/passwd")
      .openRead()
      .transform(new StringDecoder())
      .transform(new LineTransformer())
      .map((line) => line.split(":")[4] + "\n")
      .take(3)
      .transform(new StringEncoder())
      .pipe(socket);
  });
```

```
$ nc -l 8888
root
daemon
bin
```

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
...
```

Stream - Lifetime

A stream is a description. The subscription is the active part. Same as for Iterables:

```
[1, 2].map((x) => print(x + 1));  // Nothing.
new Stream.fromIterable([1, 2]).map(print);
```

- `listen` starts sequence.
- `cancel` stops sequence.

Example:

```
new File(...).openRead();
```

Stream - Values, Errors, Done

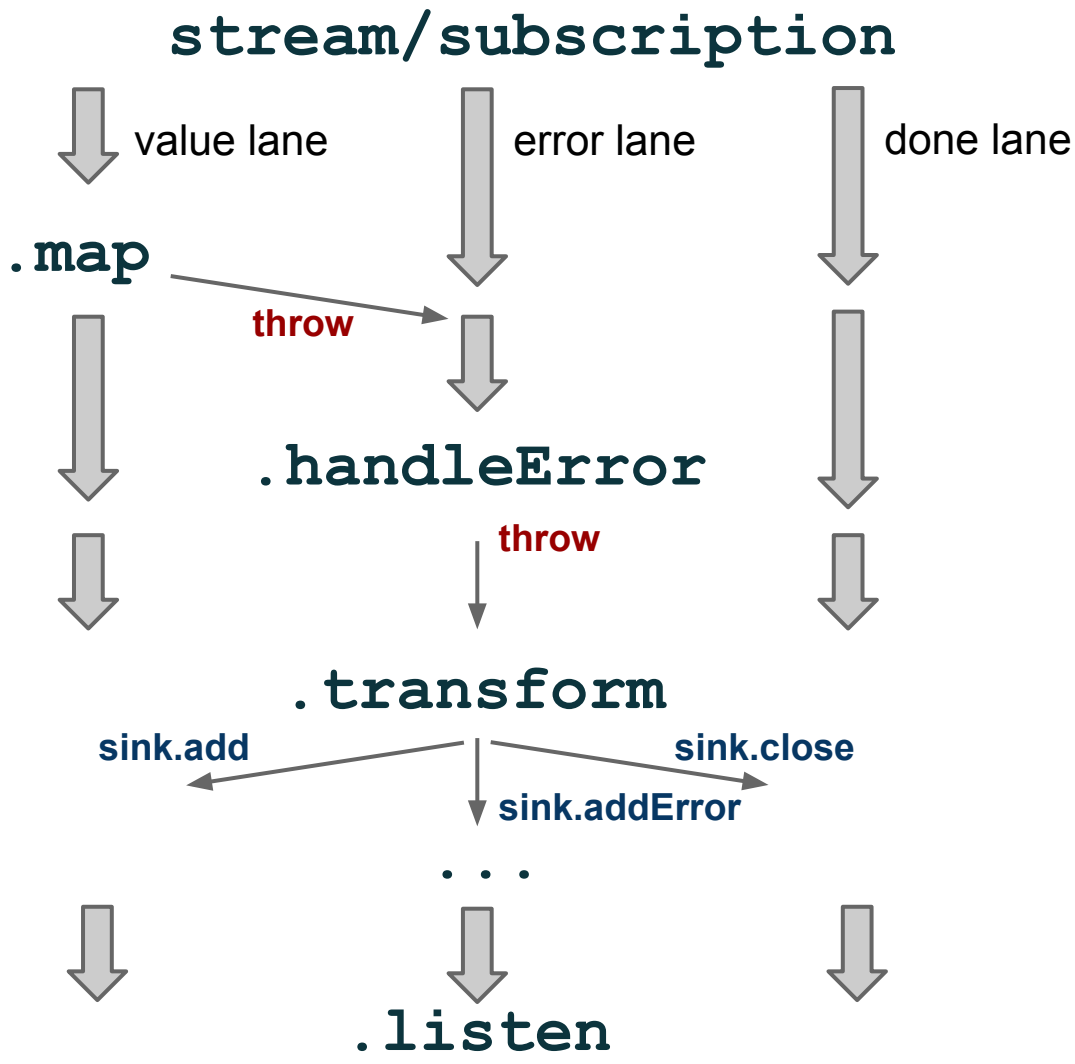
Streams have three lanes.

- values: **`map`**, **`listen`**, ...
- errors: **`handleError`**, ...
- done-event: **`listen(..., onDone: doneHandler)`**

```
stream
   .map(...)  // value lane (unless throw).
   .handleError(...)  // error lane (consumes).
   .transform(...)  // can switch lanes.
   .listen(valueHandler,
           onError: errorHandler,
           onDone: doneHandler);
```

DART

Three Lanes



DART

Stream - Line splitter

```
class LineSplitter extends StreamEventTransformer {
  String _carry = "";

  void handleData(String data, EventSink<String> sink) {
    data = _carry + data;
    int pos;
    while ((pos = data.indexOf("\n")) >= 0) {
      sink.add(data.substring(0, pos));
      data = data.substring(pos + 1);
    }
    _carry = data;
  }

  void handleDone(EventSink<String> sink) {
    if (_carry != "") sink.add(_carry);
    sink.close();  // Send done-event.
  }
}
```

DART

Uncaught Errors

Error Handling

Error handling is **hard**.

Design principle:

- Never miss uncaught errors. Runtime must not swallow errors.

Difficulties:

- When is an error uncaught?
- Where are errors reported?

Uncaught Errors

An error is uncaught if there is no error handler for it.

What if the error handler is just late?

```
var errorFuture = future.then((x) {
  throw "uncaught?";
});
new Timer(new Duration(seconds: 2), () {
  errorFuture.catchError(print);
});
```

DART

Uncaught Errors cont.

Other similar future/promise systems require a termination call:

```
var errorFuture = future.then((x) {
  throw "uncaught?";
});
new Timer(new Duration(seconds: 2), () {
  errorFuture.terminate();
});
```

Error-prone if optional.

Lazy (stream-like) semantics too weird.

DART

Uncaught Errors in Dart

Wait for one cycle before declaring an error as uncaught:

```
var errorFuture = future.then((x) {
  throw "uncaught?";
});
new Timer(new Duration(seconds: 2), () {
  // Too late.
  errorFuture.catchError(print);
});
```

Errors - Where?

Difficult to know where errors are reported.

Socket has synchronous interface (same interface as StringBuffer):

```
void socket.write("some header");
```

Error is reported on the next asynchronous operation or on the close future:

```
socket.addStream(file.openRead())
      .catchError(...)
// or
socket.close().catchError(...);
```

DART

Improvements - WIP

Error handling still too hard.

Possible improvements:

- better debugging support (more logging).
- zone-based try/catch.

Zone: dynamic extent including asynchronous callbacks declared in the zone.

```
tryAsync(() {
    future.then((x) { throw "uncaught?"; });
  },
  handleErrors: print);
```

DART

Conclusion

Conclusion

- Dart is single-threaded inside an isolate.
- Heavy use of futures and streams.

- Futures and Streams make code much more manageable.

- Stream is the push version of Iterables.

- Error handling is hard.

Questions?