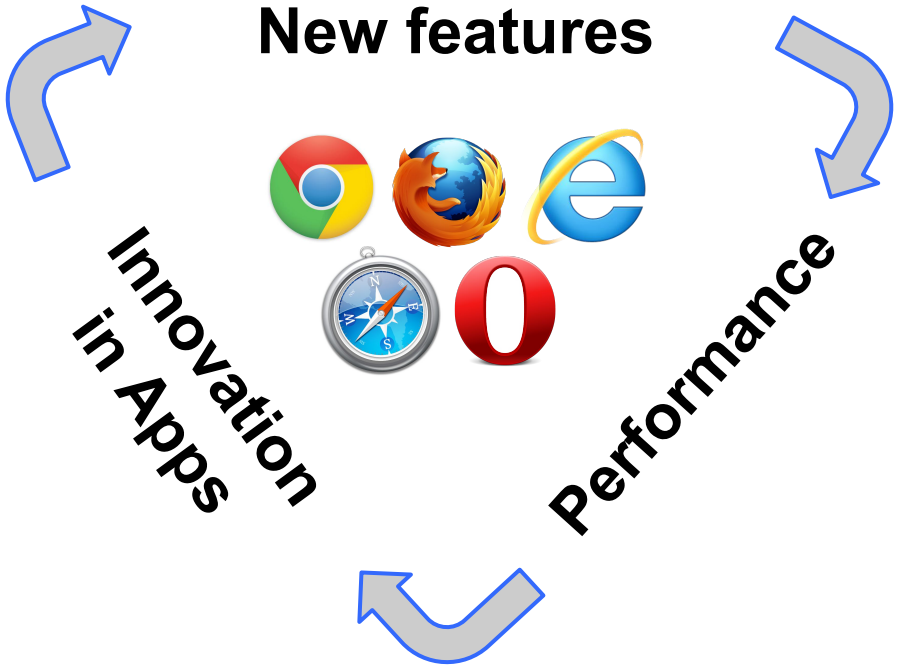
# Want To Be a Better Programmer?

Lars Bak and Kasper Lund,
Inventors of Dart,
Software engineers at Google

# Combined Experiences

# JavaScript

- We joined Google in 2006 to improve JavaScript performance
- Innovation, open source, and friendly competition made JavaScript >100x faster



# Did that make you a better programmer?

# Hmm…

```
var data = [0,1,2,3,4,5,6,7,8,9,0];
var opacity;
for(var i=0; i<data.length && i<10; i++){
  opacity = .5;
  if(i=0)
    opacity = 1;
}
```

# Okay…

```
if('Function' == Object.prototype.toString.call(this._storeUnsubscribe).slice(8, -1)) {
  // Do something
}
```

# Clearly we made you a better programmer!

.... NOT

# Wasted Effort? Nope!

- Faster JavaScript enables innovation on the web
  - Enables richer frameworks and better abstractions
  - Allows for much larger applications

- Developers still suffer from puzzling semantics and hard-to-identify errors
  - There is almost no declarative syntax and dependencies are hard to find
  - Errors are often *absorbed* and values are implicitly converted

# How Do People Get By Today?

- TypeScript
- CoffeeScript
- Ceylon
- Scala.js
- Haxe
- Elm
- ClojureScript
- GWT and Closure compiler
- or Dart

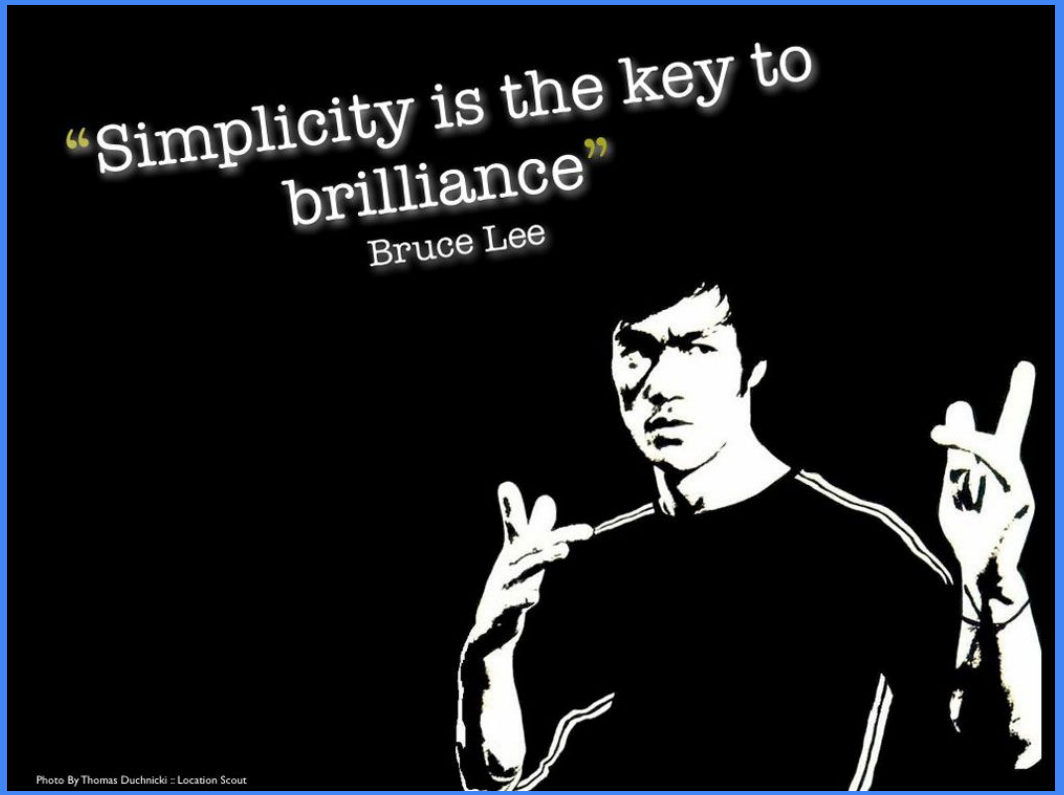
# What Makes You a Better Programmer?

**Simplicity and consistency!**

So we need:

- A simple and consistent programming language
- A simple and consistent application framework

# Simple language semantics



# The Dart Language Introduction in One Slide

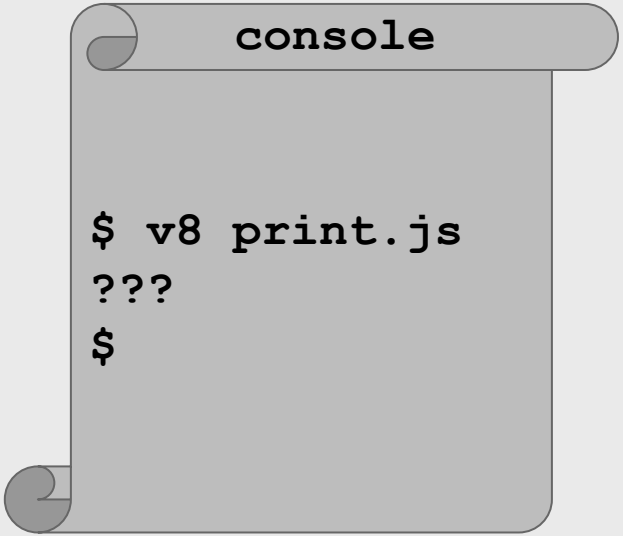
- Unsurprising object-oriented programming language
- Class-based single inheritance with interfaces and mixins
- Familiar syntax with proper lexical scoping
- Single-threaded with isolate-based concurrency
- Optional static types

Dart

# Puzzling Value Coercion in JavaScript

```
print(2.0 == '2' == new Boolean(true) == '1')
```

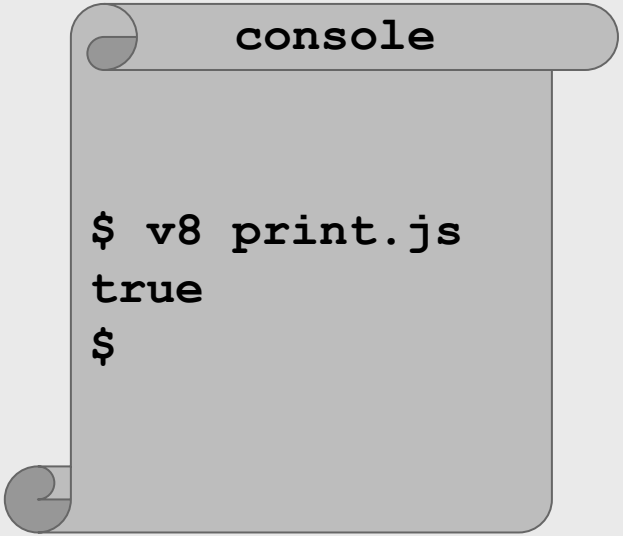
Implicit conversions will make your head explode



# Puzzling Value Coercion in JavaScript

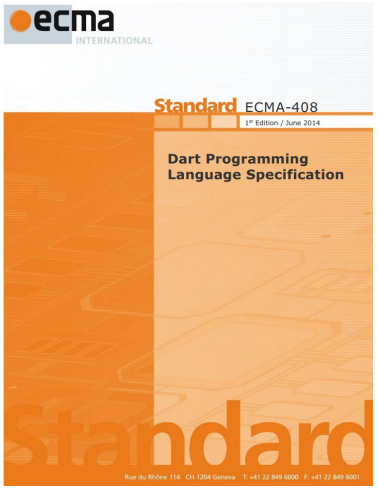
```
print(2.0 == '2' == new Boolean(true) == '1')
```

Implicit conversions will make your head explode



# Dart is an Open Standard (ECMA)

- Governed by a technical committee (TC52) since 2013
- Three editions of the specification have been published




| | |
|---|---|
| **June, 2014** | First edition |
| **December, 2014** | Enumerations, asynchrony support, and deferred loading |
| **June, 2015** | Null-aware operators and generalized tear-offs |

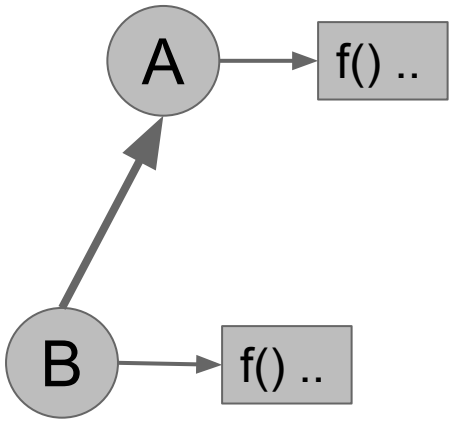
# Constructors


Google

# Constructors in Object-oriented Languages

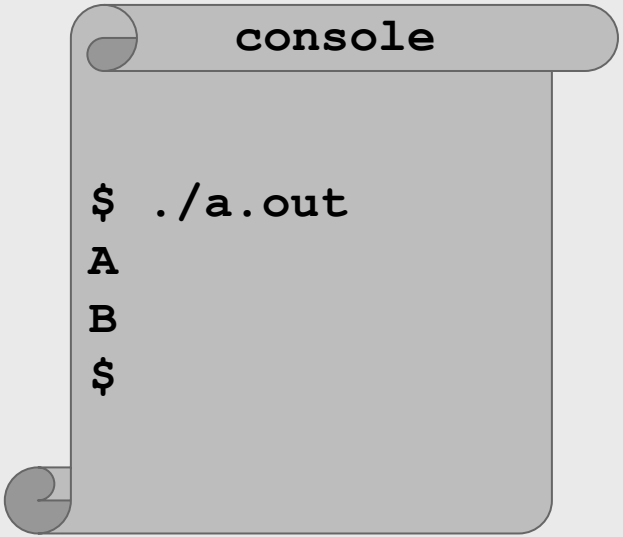
Simple example in:

- C++
- Java
- Dart



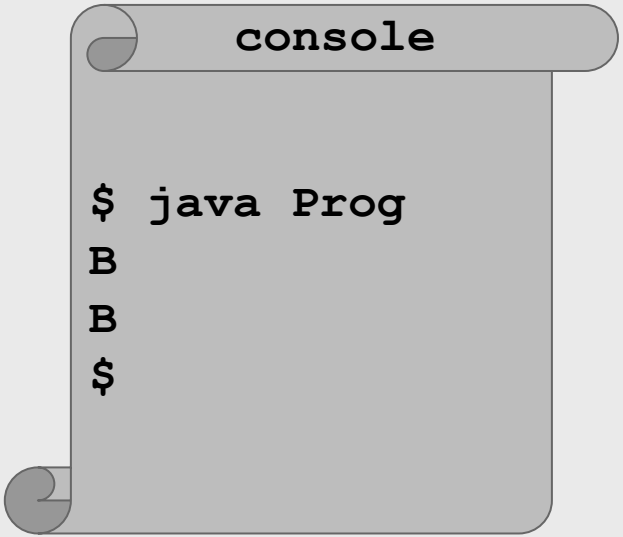
# Constructors in **C++**

```
#include <stdio.h>
class A {
 public:
  A() { f(); }
  virtual void f() { printf("A\n"); }
};
class B : public A {
 public:
  B() : A() { f(); }
  void f()  { printf("B\n"); }
};
int main() {
  B b;
}
```



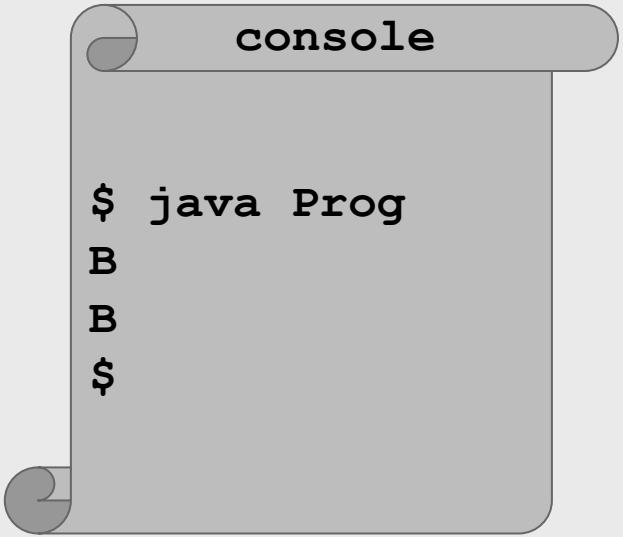
# Constructors in **Java**

```
class A {
  A() { f(); }
  void f() { System.out.println("A"); }
};
class B extends A {
  B() { f(); }
  void f() { System.out.println("B"); }
};
public class Prog {
  public static void main(String[] args) {
    new B();
  }
}
```



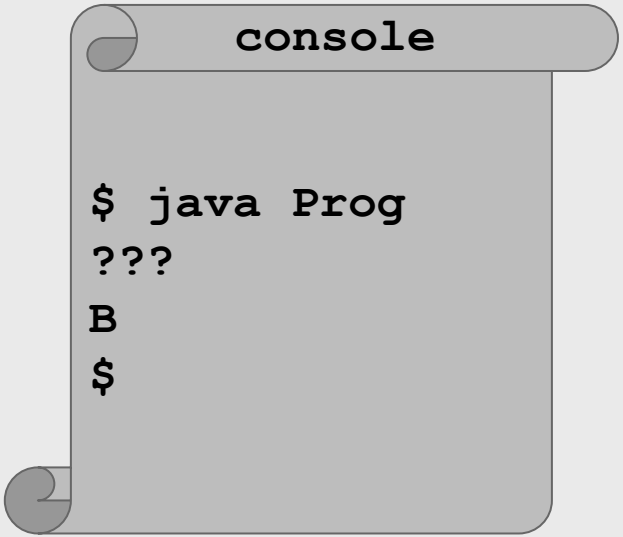
# Constructors in **Java**

```
class A {
  A() { f(); }
  void f() { System.out.println("A"); }
};
class B extends A {
  B() { f(); }
  final String x = "B";
  void f() { System.out.println(x); }
};
public class Prog {
  public static void main(String[] args) {
    new B();
  }
}
```



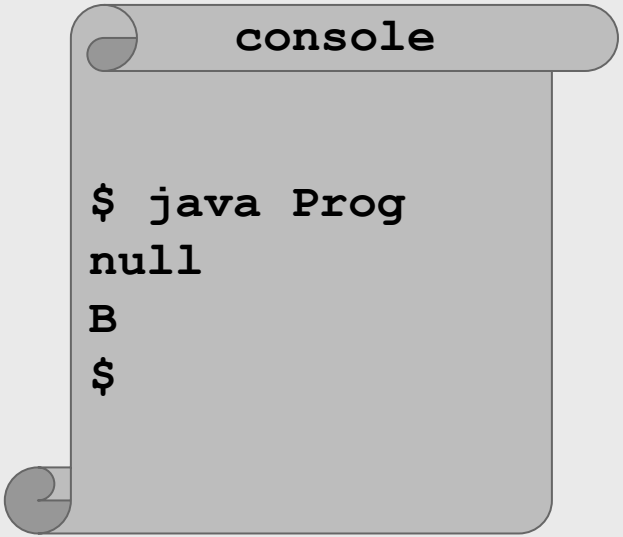
# Constructors in **Java**

```
class A {
  A() { f(); }
  void f() { System.out.println("A"); }
};
class B extends A {
  B() { f(); }
  final String x = "B".trim();
  void f() { System.out.println(x); }
};
public class Prog {
  public static void main(String[] args) {
    new B();
  }
}
```



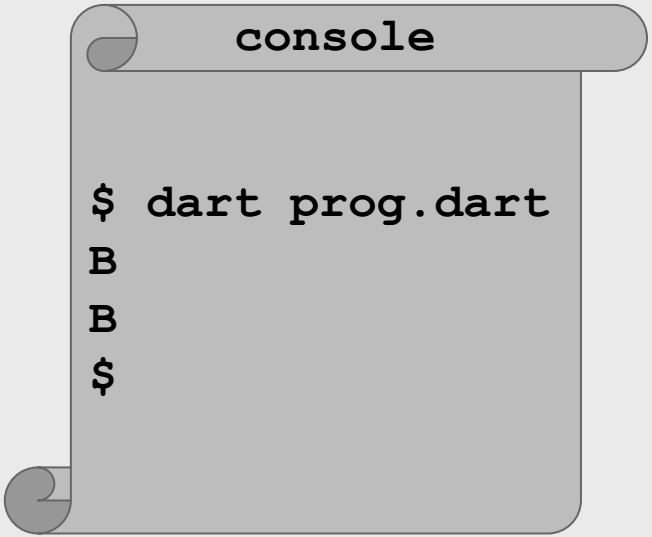
# Constructors in **Java**

```
class A {
  A() { f(); }
  void f() { System.out.println("A"); }
};
class B extends A {
  B() { f(); }
  final String x = "B".trim();
  void f() { System.out.println(x); }
};
public class Prog {
  public static void main(String[] args) {
    new B();
  }
}
```



# Constructors in **Dart**

```
class A {
  A() { f(); }
  f() => print("A");
}
class B extends A {
  B() { f(); }
  final x = "B".trim();
  f() => print(x);
}
main() => new B();
```



# Clean Semantics Make You Better

In Dart, constructors enforce two pass initialization

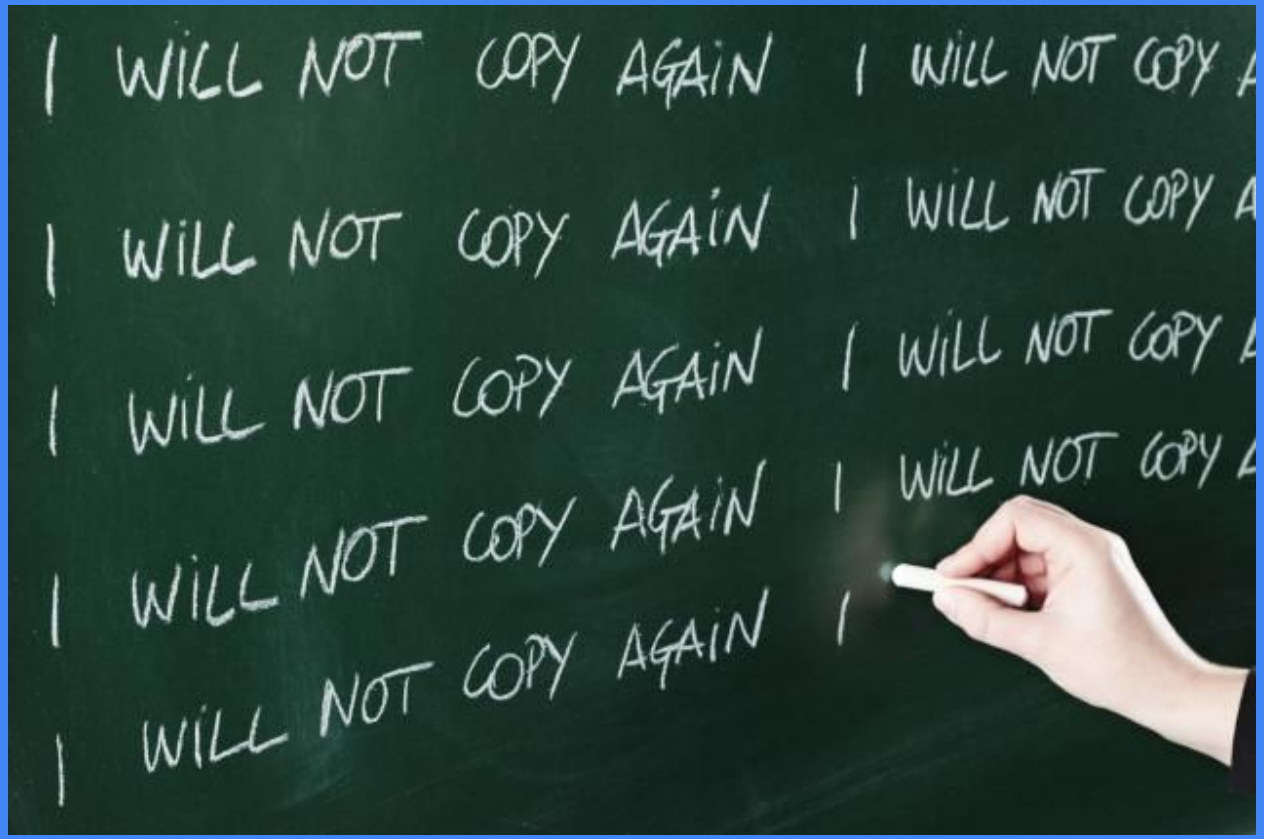
- All fields are initialized before ...
- … constructor bodies are executed

# Constructors in Dart

```
class Symbol {
  final String name;
  static Map<String, Symbol> _cache = <String, Symbol>{};
  factory Symbol(String name) {
    if (_cache.containsKey(name)) return _cache[name];
    return _cache[name] = new Symbol._internal(name);
  }
  Symbol._internal(this.name);
}

main() {
  print(new Symbol("X") == new Symbol("X"));
}
```

# Boilerplate



# Simple Constructors

Dart

```
class Point {
  final num x, y;
  Point(this.x, this.y);
}
```

Java

```
public class Point {
  public final double x, y;
  public Point(double x,
               double y) {
    this.x = x;
    this.y = y;
  }
}
```

# No Need for Explicit Accessors

Dart

```
class Point {
  final num x, y;
  Point(this.x, this.y);
}
```

Java

```
public class Point {
  private final double x, y;
  public Point(double x,
               double y) {
    this.x = x;
    this.y = y;
  }
  public double getX() {
    return x;
  }
  public double getY() {
    return y;
  }
```



intentional overflow

# Classes Can Act as Interfaces

Dart

```
class PolarPoint implements Point {
  num get x => ...;
  num get y => ...;
}
```

Java

```
public interface Point {
  ...
}

public class CartesianPoint
    implements Point {
  ...
}

public class PolarPoint
    implements Point {
  ...
}
```



intentional overflow

# Generic Types Are Reified

Dart

```
List<String> listify(Set<String> s)
    => s.toList();
```

Java

```
String[] listify(Set<String> s) {
  return s.toArray(
    new String[s.size()]);
}
```

# Cascaded Calls

```
void drawCircle(CanvasElement canvas, int x, int y, int size) {
  canvas.context..beginPath()
                ..arc(x, y, size, 0, PI * 2, false)
                ..fill()
                ..closePath()
                ..stroke();
}
```

# Cascaded Calls for Initialization

```
Set initialSet() => new Set()..add(42);
```

# Cascaded Method Invocations

- Enables the programmer to apply **method chaining** to any object
- Expression always returns cascaded receiver object
- Inspiration from Smalltalk

# Asynchrony FTW

# What About IO?

- Browser enforces single threaded execution
- Blocking IO would allow one operation at a time
  - … and kill responsiveness
- Why not solve it with multi-threading?

Synchronous code:

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

# Asynchronous Callback

```
readWrite() {
  read((c) { write(c, handleError); },
       handleError);
}

// Finally block cannot be handled.
// Easy to make mistakes in error handling.
// … and fairly unreadable.
```

Synchronous code:

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

# Futures Makes It Better

```
readWrite() {
  Future f = read();
  return f.then((c) => write(c))
          .catchError(handleError)
          .whenComplete(close);
}

// Control flow must be dealt with in library.
// Chaining of futures is tedious.
```

Synchronous code:

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

# Solving the Callback Morass

- Hired Erik Meijer to help with asynchronous design
- Introduced special `async` methods
- Libraries were already based in futures and streams
- Inspired by C#

# Async/await Feature

```
readWrite() async {
  try {
    var c = await read();
    await write(c);
  } catch (e) {
    handleError(e);
  } finally {
    await close();
  }
}

// await suspends the activation in a non-blocking way!
```

Synchronous code:

```
readWrite() {
  try {
    var c = read();
    write(c);
  } catch (e) {
    handleError(e);
  } finally {
    close();
  }
}
```

# Reflections on async/await

Pros

- Restores “normal” control flow
  - Including exception and finally code
- Incremental migration of code

Cons

- Dual mode makes reasoning complicated
- Stack traces disappear
- Debugging is not intuitive

# Simple and Consistent Language ✓

- Dart is a simple and consistent programming language
  - Unsurprising and familiar
  - Concise and useful
- Now we just have to fix the framework...

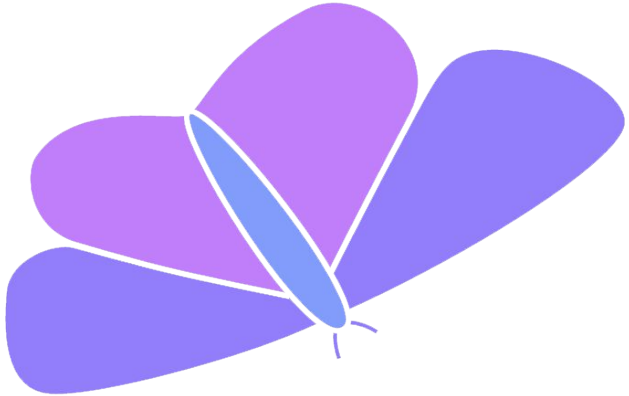
# The Butterfly Effect with Flutter

# Making Mobile Development Easier and Cheaper

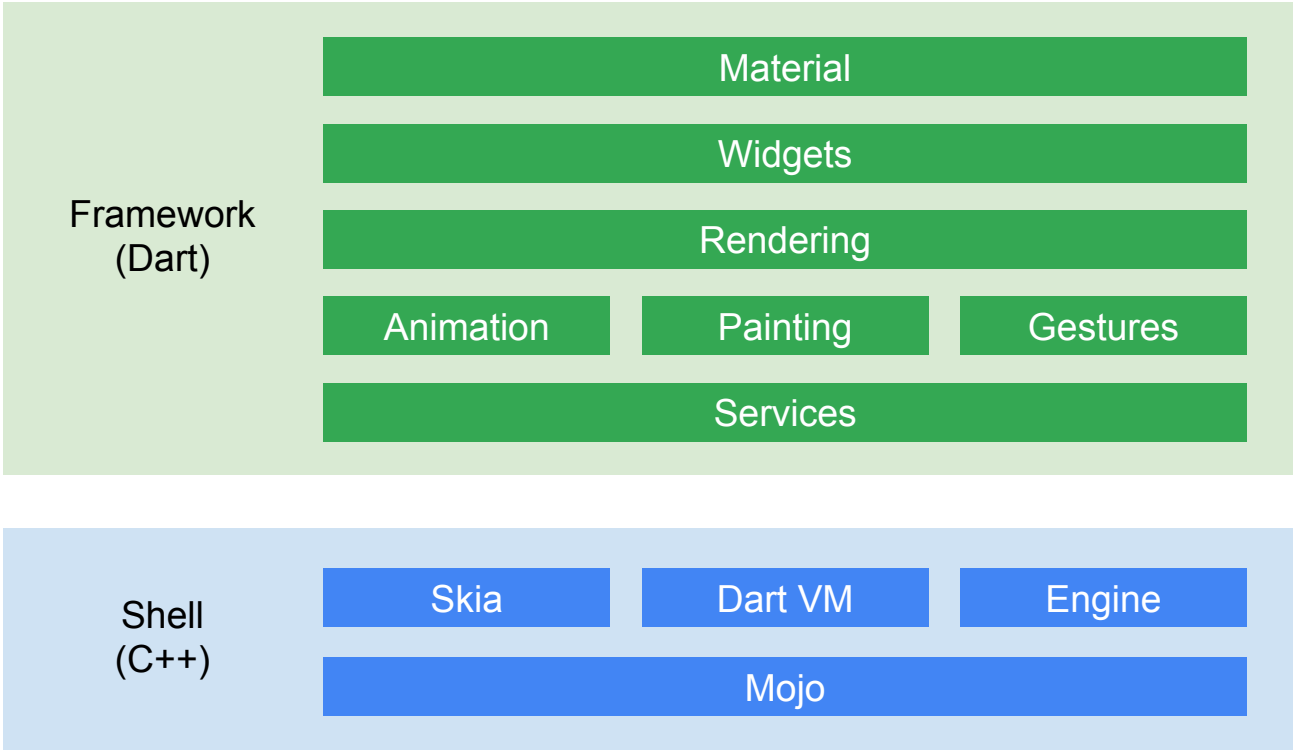
Flutter is a new project to help developers build high-performance, high-fidelity, mobile apps for iOS and Android from a single codebase **in Dart**

`http://flutter.io`

Flutter is open source



# Flutter Architectural Overview



# Flutter is a Functional-Reactive Framework

```
class CounterState extends State {
  int counter = 0;

  void increment() {
    setState(() { counter++; });
  }

  Widget build(BuildContext context) =>
    new Scaffold(
      toolBar: new ToolBar(
        center: new Text("Flutter Demo")),
      body: new Material(
        child: new Center(
          child: new Text("Button tapped $counter times."))));
}
```

*"**Dart** has been a great fit for mobile.*
*It's familiar, flexible, and fast. We're building our entire framework, widgets, and developer tools in **Dart**."*

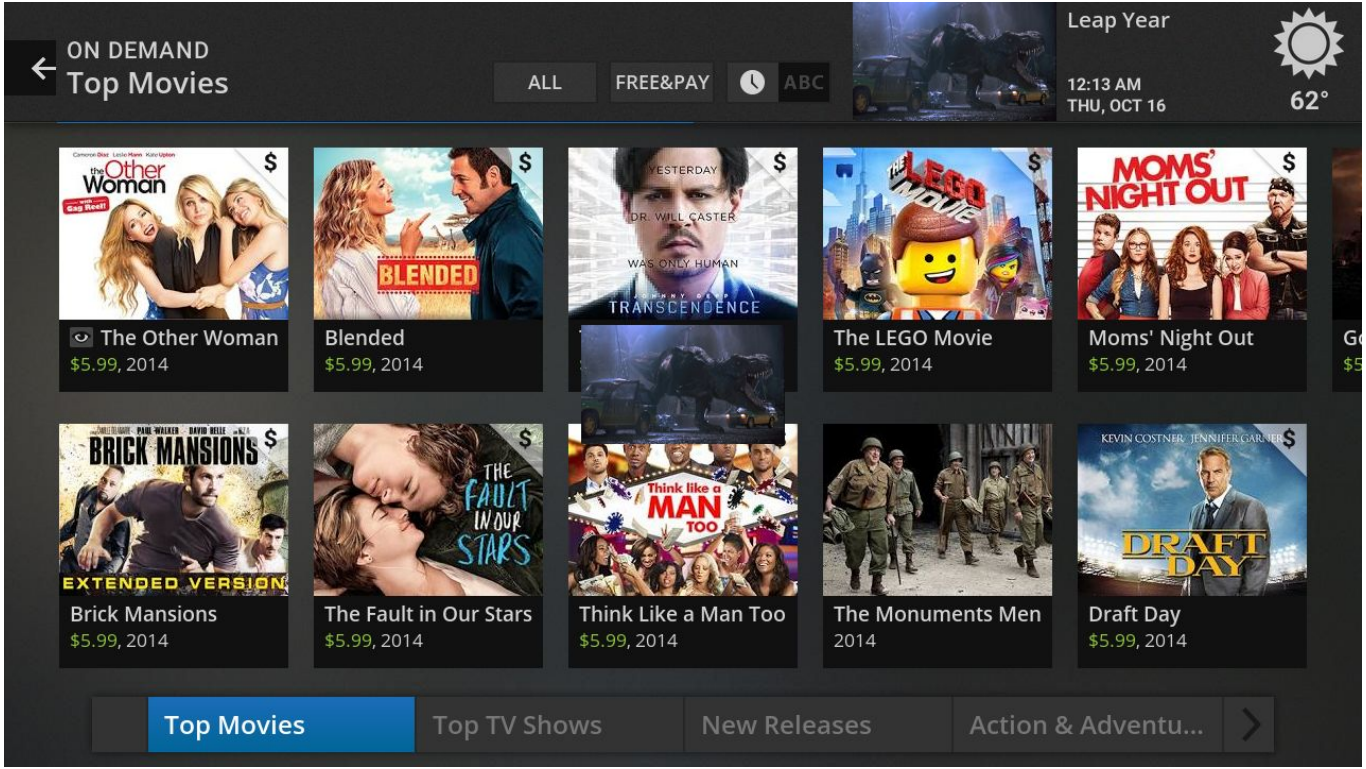
Seth Ladd, Product Manager for Flutter at Google

# Simple Unified Application Framework

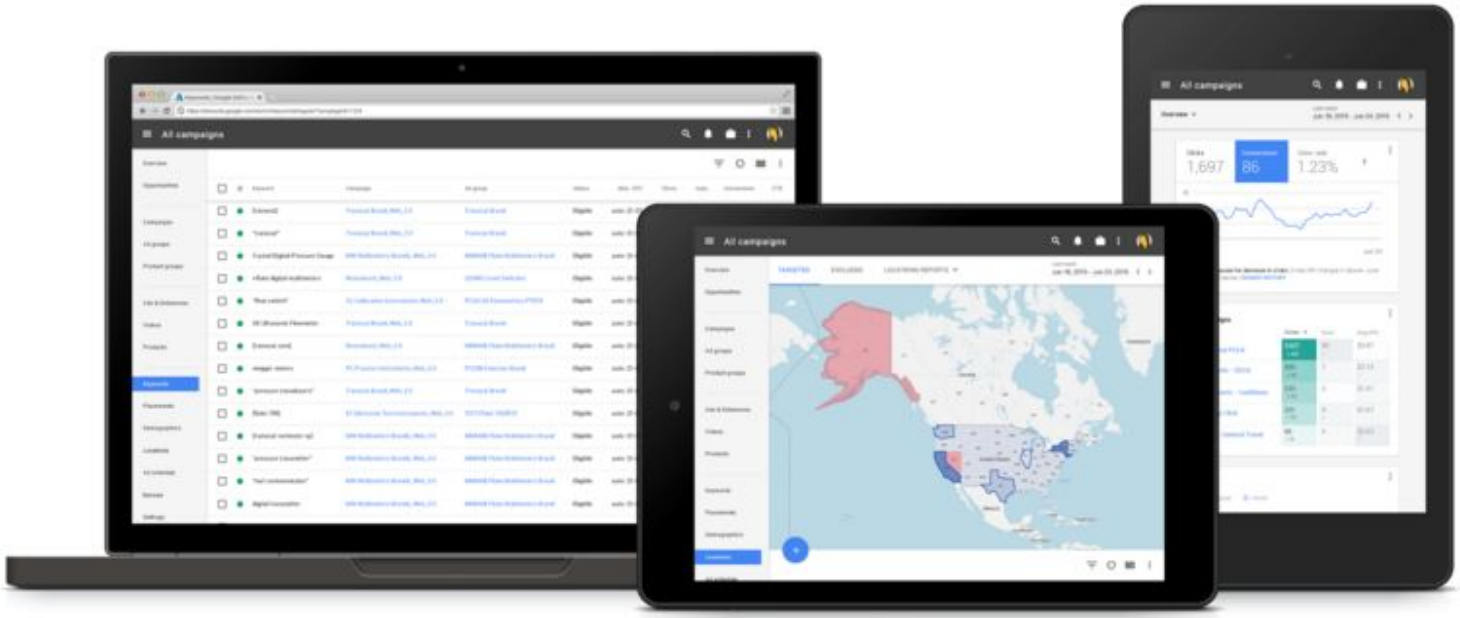
- One language to rule the entire stack
  - Same semantics
  - Same great tooling experience (debugging, etc.)
- Contrast to Dart + Angular + HTML

# Dart in 2016

# Google Fiber



# Google AdWords



# Dart for Web Apps

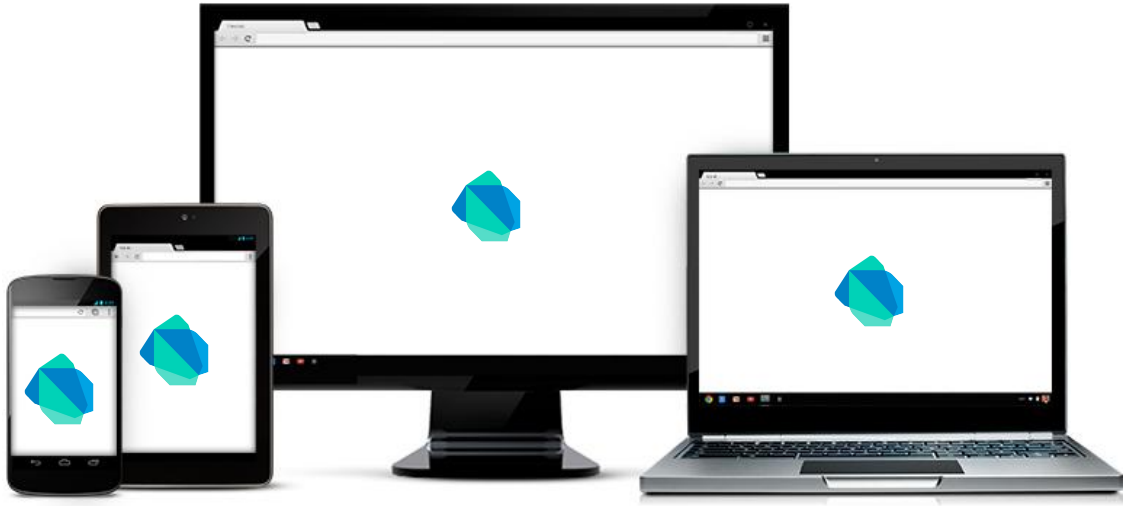
- Google is shipping large, mission critical web apps built in Dart
- Every day lots of developers are spending all their time writing in Dart

## Why?

*“Our engineers were **highly productive** as their projects scaled to many engineers. Dart’s support for strong typing, its object-oriented nature, and its excellent IDE integration really worked for us.”*

Joshy Joseph, Distinguished Engineer, Google AdWords

# Dart Runs Everywhere...



Browsers: Runs translated to JavaScript
Mobile: Runs on optimized Dart VM (Flutter)

Servers: Runs on optimized Dart VM


IoT: Runs on embedded MCUs

# Dartino: Dart for Embedded Devices

**Work in progress...**

Early preview SDK supporting Coretex M4/M7 microcontrollers is available.

```
http://dartino.org/
```

Small, efficient runtime for Dart:

- 32-bit microcontrollers (ARM, MIPS)
- 128 KB of RAM
- 512 KB of Flash

# Conclusions

# Summary

- We designed a pragmatic OO programming language
  - It is readable and well-defined
  - It increases programmer productivity
  - It has nice systems properties
- Projects u
- Several m

Both creators of Dart hereby claim that Dart makes you a better programmer


EPIC WIN
This is the face of an epic win
CanItBeSaturdayNow.com

# Questions