



DART talk

An introduction to Dart
for Smalltalkers

March 19, 2012

Eric Clayberg
Engineering Manager
Dart Editor
Google, Inc.

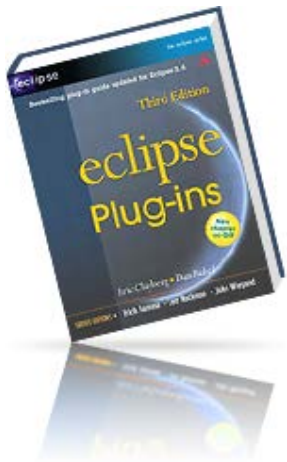


Who Am I



Eric Clayberg

- Engineering Manager for Dart Editor at Google
- Former V.P. of Product Development for Instantiations
- Started with Smalltalk in 1991, Java in 1996, and Dart in 2011
- Co-author of Eclipse Plug-ins and Eclipse Graphical Editing Framework (GEF)
- Project manager & architect of VA Assist, CodePro, WindowBuilder and over a dozen other commercial software products
- Project Lead for Eclipse.org WindowBuilder project





Agenda

- Motivation
- Language
- Isolates
- Code samples
- Smalltalk vs. Dart
- Dart Editor

Special thanks to Seth Ladd, Florian Loitsch, Gilad Bracha, Dan Rubel, Steve Messick, Brian Wilkerson, and Alan Knight for slides and ideas.



Web Programming

- Small-to-medium apps are easy
- Platform independent
- No installation
- Platform improving fast
- Everywhere
 - And getting more modern:
 - ~50% users on IE9/FF7/Chrome/Safari



Why create Dart?

- Developing large applications is hard
 - Hard to find program structure
 - No static types
 - No support for libraries
 - Weak tool support
 - Slow startup
- Lots of cruft after 15 years



**Help app developers
write complex, high
fidelity client apps
for the modern web.**

Our goal



Dart is...

- Structured Web Programming
 - New language
 - New tools
 - New libraries
- Open source as of early October 2011
- Available at <http://dartlang.org>



What Is Dart?

- A simple, unsurprising OO language
- Class based, single inheritance with interfaces
- Optional static typing
- Real lexical scoping and closures
- Single threaded
- Familiar syntax (to the Java[Script] developer)



Hello World

```
#import('dart:html');
```

```
void main() {  
  new Hello().doStuff();  
}
```

```
class Hello {  
  void doStuff() {  
    var message = "Hello World";  
    document.query('#status').innerHTML = message;  
  }  
}
```

Libraries



more at

<http://dartlang.org/language-tour/>



Hello World

```
#import('dart:html');
```

```
void main() {  
  new Hello().doStuff();  
}
```

Functions



```
class Hello {  
  void doStuff() {  
    var message = "Hello World";  
    document.query('#status').innerHTML = message;  
  }  
}
```

more at

<http://dartlang.org/language-tour/>



Hello World

```
#import('dart:html');
```

```
void main() {  
  new Hello().doStuff();  
}
```

Functions



```
class Hello {  
  void doStuff() {  
    var message = "Hello World";  
    document.query('#status').innerHTML = message;  
  }  
}
```

more at

<http://dartlang.org/language-tour/>



Hello World

```
#import('dart:html');
```

```
void main() {  
  new Hello().doStuff();  
}
```

Classes

```
class Hello {  
  void doStuff() {  
    var message = "Hello World";  
    document.query('#status').innerHTML = message;  
  }  
}
```

more at

<http://dartlang.org/language-tour/>



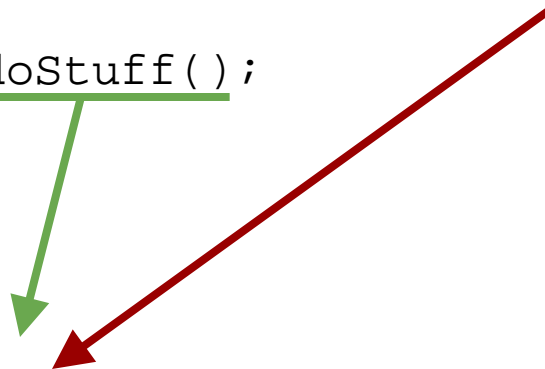
Hello World

```
#import('dart:html');
```

```
void main() {  
  new Hello().doStuff();  
}
```

```
class Hello {  
  void doStuff() {  
    var message = "Hello World";  
    document.query('#status').innerHTML = message;  
  }  
}
```

Methods



more at

<http://dartlang.org/language-tour/>



Hello World

```
#import('dart:html');
```

```
void main() {  
  new Hello().doStuff();  
}
```

```
class Hello {  
  void doStuff() {  
    var message = "Hello World";  
    document.query('#status').innerHTML = message;  
  }  
}
```

Optional Types

more at

<http://dartlang.org/language-tour/>



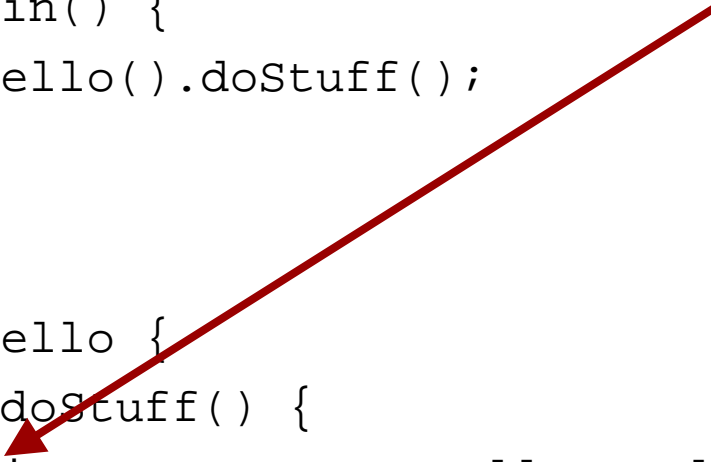
Hello World

```
#import('dart:html');
```

```
void main() {  
  new Hello().doStuff();  
}
```

```
class Hello {  
  void doStuff() {  
    String message = "Hello World";  
    document.query('#status').innerHTML = message;  
  }  
}
```

Optional Types



more at

<http://dartlang.org/language-tour/>




Optional Static Types...

- Are a low friction tool for developers to make their intent more clear to humans and machines
- Let you scale up your program, moving from untyped functions as you experiment to typed classes and interfaces as you get more complex
- Help you get your job done faster thanks to the Dart Editor



Dart Types at Runtime

- Developers may check types at runtime

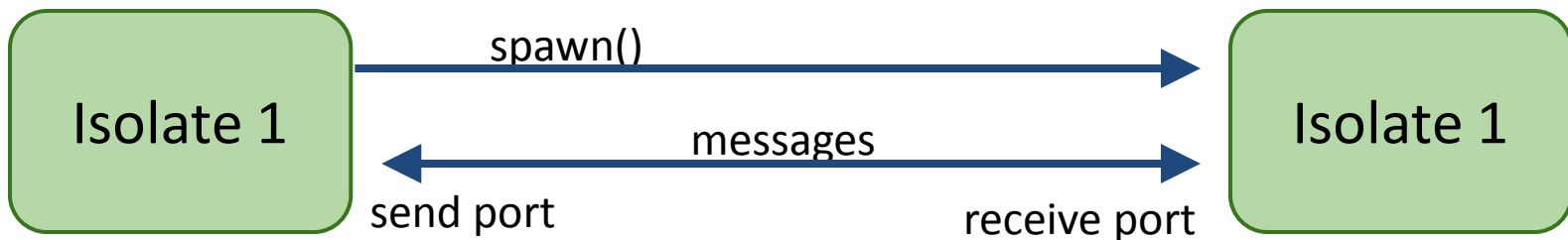
`T x=o`  `assert (x === null || (o is T))`

- By default, type annotations have:
 - No effect
 - No cost



Isolates

- Inspired by Erlang, Dart has isolates
- Lightweight units of execution
 - Each isolate is conceptually a process
 - Nothing shared
 - All communication via message passing
- Isolates support concurrent execution





Isolates

- Can be shared or separate processes
 - Lightweight isolates on UI thread
 - Heavyweight isolates map to their own OS thread
 - When compiled to JS, web workers
- Many potential uses for isolates
 - Isolation of 3rd-party code
 - Security
 - JavaScript interop
 - Uniform model for:
 - Client-server
 - Intra-client



Dart Board

<http://www.dartlang.org/>

Get started

Read a [technical overview](#), take a [language tour](#), or download [Dart Editor](#). Or play with Dart code right here in your browser.

```
Fibonacci
```

```
1 int fib(int n) {
2   if (n <= 1) return n;
3   return fib(n - 1) + fib(n - 2);
4 }
5
6 main() {
7   print('fib(20) = ${fib(20)}');
8 }
```

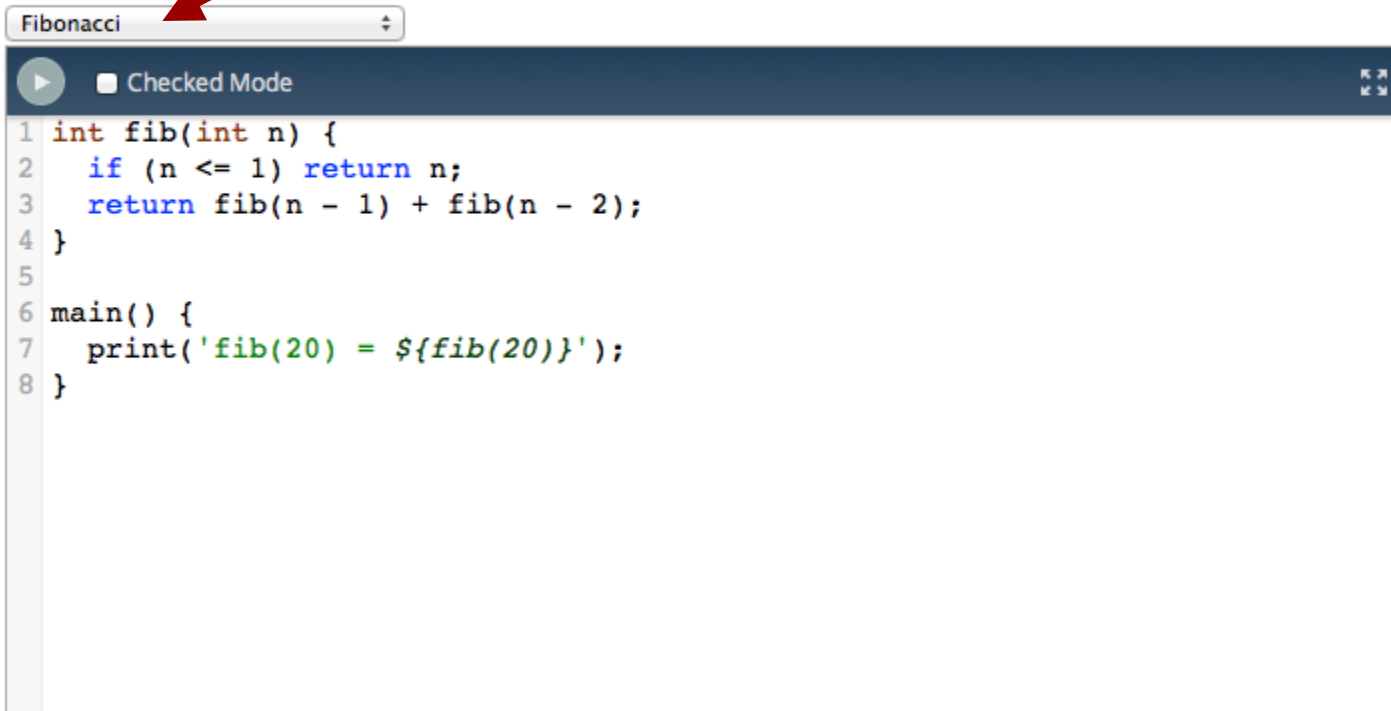


Dart Board

Select template

Get started

Read a [technical overview](#), take a [language tour](#), or download [Dart Editor](#). Or play with Dart code right here in your browser.



```
Fibonacci
```

```
1 int fib(int n) {
2   if (n <= 1) return n;
3   return fib(n - 1) + fib(n - 2);
4 }
5
6 main() {
7   print('fib(20) = ${fib(20)}');
8 }
```



Dart Board

Type stuff

Get started

Read a [technical overview](#), take a [language tour](#), or download [Dart Editor](#). Or play with Dart code right here in your browser.

```
Fibonacci
```

```
1 int fib(int n) {  
2   if (n <= 1) return n;  
3   return fib(n - 1) + fib(n - 2);  
4 }  
5  
6 main() {  
7   print('fib(20) = ${fib(20)}');  
8 }
```



Dart Board

Run program

Get started

Read a [technical overview](#), take a [language tour](#), or download [Dart Editor](#). Or play with Dart code right here in your browser.

```
Fibonacci
▶ Checked Mode
1 int fib(int n) {
2   if (n <= 1) return n;
3   return fib(n - 1) + fib(n - 2);
4 }
5
6 main() {
7   print('fib(20) = ${fib(20)}');
8 }
```

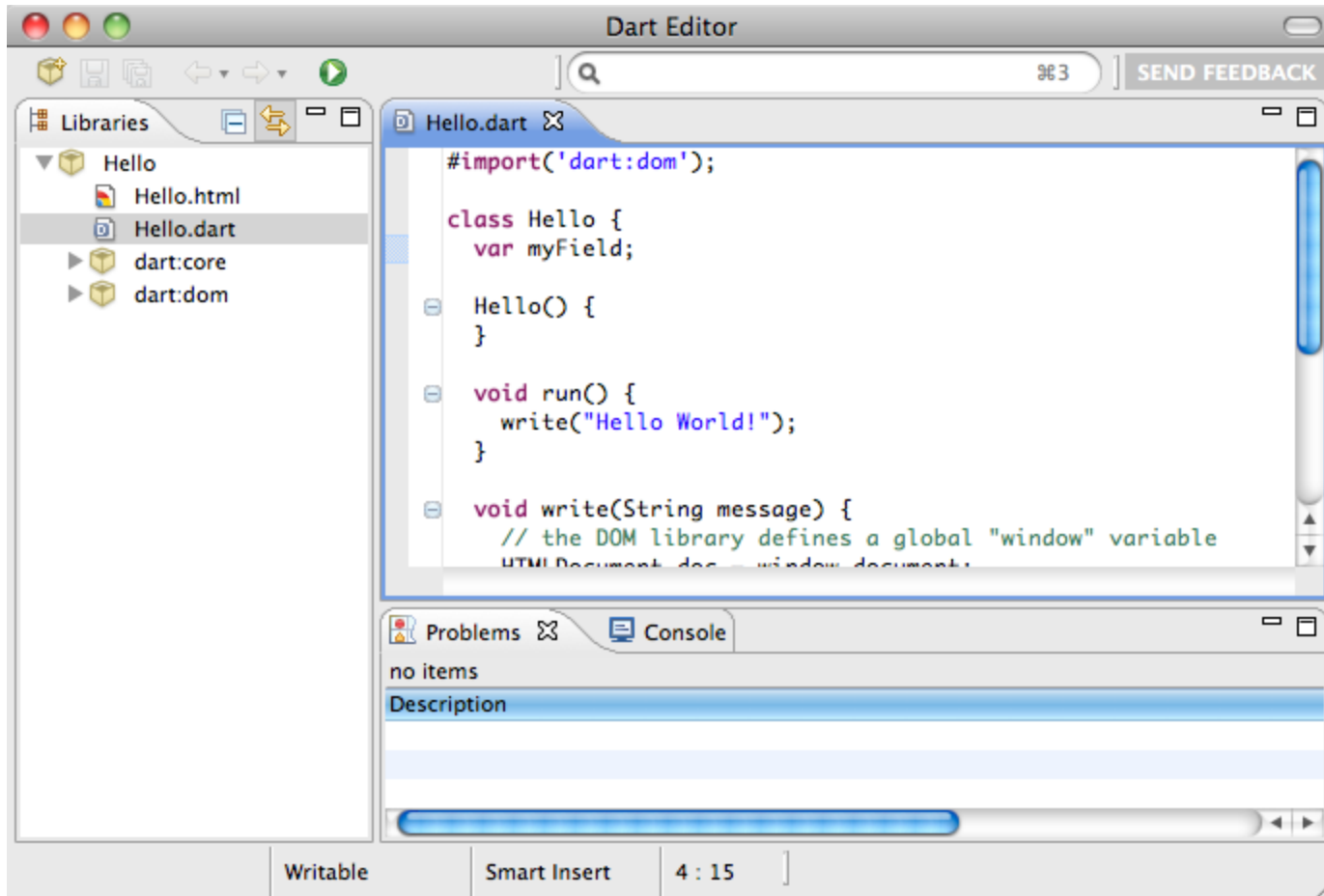


Dart Editor

- Editor for constructing and browsing Dart apps
- Built using Eclipse components
- But very small and very lightweight
- Supports code completion, etc.
- Available in open-source
(and as a [prebuilt binary](#))



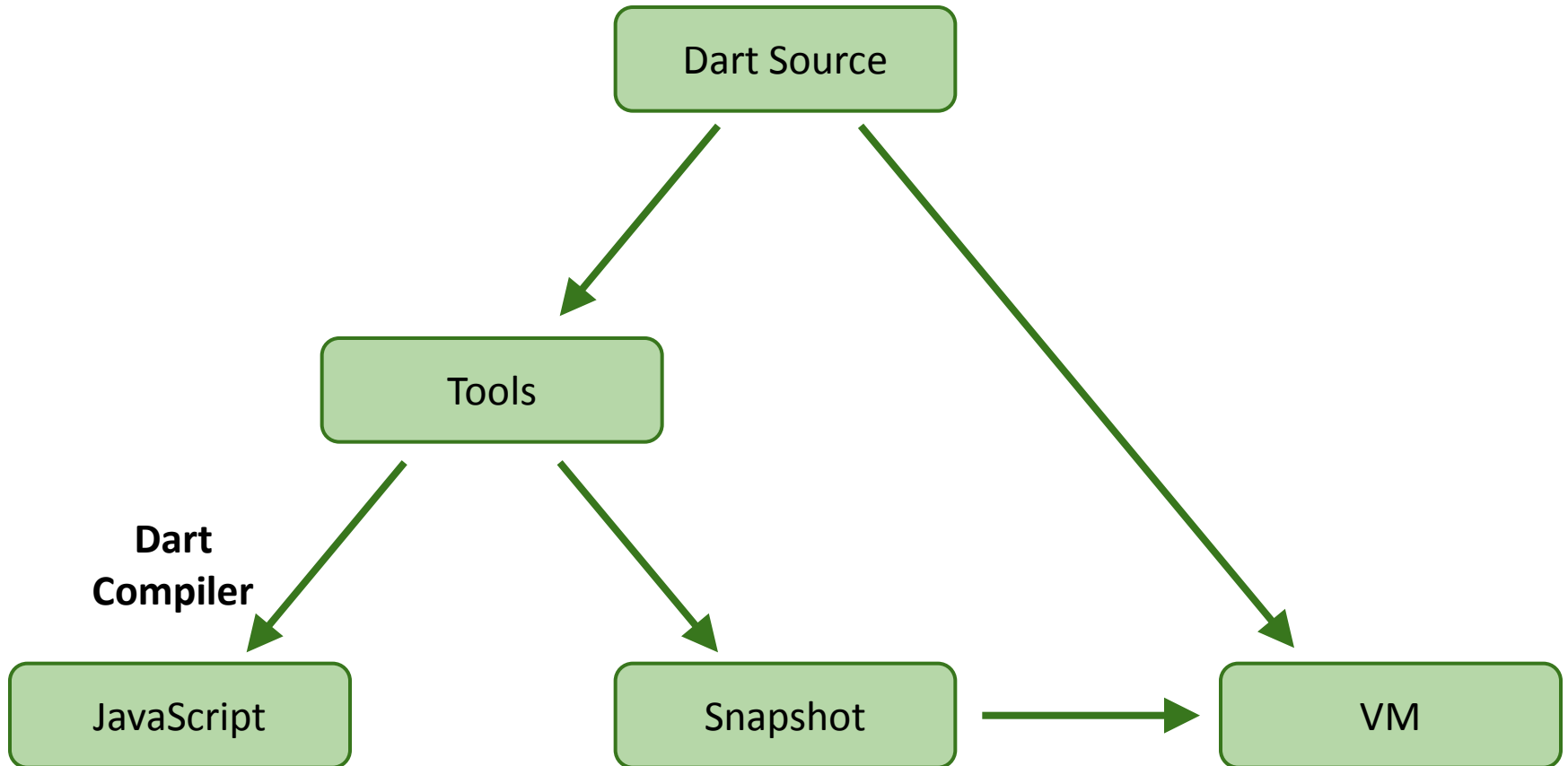
Dart Editor – Simple & Clean



More later, if we have time...



Dart Execution





Snapshotting in the Dart VM

- Snapshot
 - Heap serialized after application loaded
- Performance:
 - Un-snapshotted, 54000 lines of dart -> 640ms
 - Same application w/ snapshot -> 60ms
 - 10x faster startup
- Requires VM



Swarm - Sample Web Application

- Newsreader completely written in Dart
- Size:
 - App - 3210 lines
 - UI Library - 13200 lines
 - Animation: 30 fps
 - Compile time: 2.1s
 - Resulting JS: 539KB
- Available in [open-source project](#)



Fun code examples

- [Minimal](#)
- [Generics](#)
- [Lexical scoping](#)
- [Isolates](#)
- Point
- Rectangle
- Symbol
- Clock
- Sunflower



Smalltalk vs. Dart

- Basics
- Variables
- Collections
- Strings
- Booleans
- Methods/Blocks/
Functions
- Iterators
- Classes/Constructors
- Exceptions
- Math
- Runtime program
manipulation
- Processes



Basics



All values are objects
(nil, numbers, booleans)

Classes are first class objects.

nil
true
false
self/super



All values are objects
(null, numbers, booleans)

Classes are **not** first class objects.

null
true
false
this/super



Create and assign local variables



```
| myName |  
myName := 'Aaron'.
```

Default Value

```
| myName |  
" == nil"
```



Dart variables can be typed...

```
String myName = 'Aaron';
```

...but they don't need to be.

```
var myOtherName = 'Aaron';
```

Type information is advisory, producing warnings but still compiling and running.

```
int notReallyAnInt = 'Aaron';
```

```
var myName; // == null
```

```
int x; // == null
```




Create and assign instVars / fields



Smalltalk doesn't really have class definitions in the sense of having an actual syntax for them. Instance variables are created as part of a class definition which is itself a reflective message send.

```
Object subclass: #MyClass
  classInstanceVariableNames:
  instanceVariableNames: 'field1 field2'
  classVariableNames: ''
  poolDictionaries: ''
```

Instance variables are typically initialized in the #initialize method.

```
initialize
  field1 := 'foo'.
  field2 := 10.
```

The #initialize method is called from the #new method.

```
new
  ^super new initialize
```



Dart instance fields look just like local variables but are declared within a class rather than within a method or function.

```
String field1 = 'foo';
int field2 := 10;
```

Instance fields are typically initialized in a constructor.

```
MyClass(this.field1, int size) : field2 = size + 10;
```



Create and assign static variables



Static variables are created as part of a class definition.

```
Object subclass: #MyClass
  classInstanceVariableNames:
    'instanceCount'
  instanceVariableNames: ''
  classVariableNames: 'Option1 Option2'
  poolDictionaries: ''
```

Two forms:

- class variable: a single variable whose value is shared with subclasses
- class instance variable: an instance variable of the class object - presence of the variable shared with subclasses, but values differ.



Dart static fields look just like instance fields except that they have the keyword 'static'.

```
static String instanceCount = 0;
```

There is a single variable defined, but it is only visible in the scope of the declaring class. In subclasses it must be referenced as if it were a field of the class.

```
MyClass.instanceCount
```



Final Variables



Smalltalk has no support for final variables.



Final variables can't be changed after the initial assignment

```
final name = 'Bob';
```

You can combine types and final

```
final String name = 'Bob';
```

Trying to reassign a final variable raises an error

```
name = 'Alice';
```

```
// ERROR: cannot assign value to  
final variable
```



Privacy rules



Instance variables are per-object protected in Smalltalk while methods are public.



Dart privacy is library based

```
var thisIsPublic = 'foo';
```

The “_” prefix indicates private

```
var _thisIsPrivate = 'bar';
```



Arrays/Lists



```
| a |  
a := Array new.  
a := #().  
a := #(1 2 3).  
  
a := OrderedCollection new: 5.  
a := #(1 2 3) asOrderedCollection.  
a := #('apple' 'banana' 'cherry')  
  asOrderedCollection.  
a add: 'donut'. "= 'donut'"  
a size "= 4"  
a removeLast. "= 'donut'"  
Smalltalk collections are one-based.  
a at: 1 "= 'apple'
```



```
var a = new List();  
var a = [];  
var a = [1, 2, 3];  
  
var a = ['apple', 'banana', 'cherry'];  
a.add('donut'); // == null  
a.length // == 4  
a.removeLast() // == 'donut'
```

Dart collections are zero-based.

a[0] //== 'apple' or "apple"

Use single or double quotes to delimit strings.



Custom sort



| numbers |

```
numbers := #(42 2.1 5 0.1 391) copy.
```

```
numbers sort: [:a :b | a < b].
```

```
"#(0.1 2.1 5 42 391)"
```



```
var numbers = [42, 2.1, 5, 0.1, 391];
```

```
numbers.sort((a, b) => a - b);
```

```
// == [0.1, 2.1, 5, 42, 391]
```



Dictionaries/Maps



| periodic |

```
periodic := Dictionary new  
  at: #gold put: 'AU';  
  at: #silver put: 'AG';  
  yourself.
```



Keys in Dart map literals must be string literals

```
var periodic = {  
  'gold' : 'AU',  
  'silver' : 'AG'  
};
```

If you use a Map constructor, then you have more options: the key can be a string, a number, or any other object that implements the **Hashable** interface



Accessing values



periodic at: #gold.

```
" == 'AU'"
```

periodic at: #gold put: 'Glitter'.

```
" == 'Glitter'"
```



Values can only be get or set by using the square bracket notation.

```
periodic['gold'] // == 'AU'
```

```
periodic['gold'] = 'Glitter';
```




Interpolation & Concatenation



```
name := 'Aaron'.
greeting := 'My name is ', name.
greetingPolish :=
  'My Polish name would be ',
  name, 'ski'.

greetingPolish :=
  'My Polish name would be %1ski'
  bindWith: name.

element style top: (top+20) asString, 'px';
Use streams to concatenate multiple strings.
stream := String new writeStream.
stream
  nextPutAll: 'My Polish name would be ';
  nextPutAll: name.
```



```
var name = 'Aaron';
var greeting = 'My name is ' + name;
...or...
var greeting = "My name is $name.";
Use single or double quotes to delimitate
strings
var greetingPolish =
  'My Polish name would be ${name}ski.';
Calculations can be performed in string
interpolation.

element.style.top = '${top + 20}px';
```



Substring



```
'doghouses' copyFrom: 4 to: 9  
"==" 'houses'"
```

Smalltalk strings are one-based



```
'doghouses'.substring(3, 8);  
// == 'houses'
```

Dart strings are **zero**-based



Replace all occurrences



```
'doghouses' copyReplaceAll: 's' with: 'z'  
"==" 'doghouzez'"
```



```
'doghouses'.replaceAll('s','z');  
// == 'doghouzez'
```



Multi-line strings



string := 'This is a string that spans
many lines'.



Dart ignores the first new-line (if it is
directly after the quotes), but not the
last.

```
var string = """This is a string that  
spans many lines.
```

```
""";
```



Split into an array



| animals |

```
animals := 'dogs,cats,gophers,zebras'.
```

```
individualAnimals := animals split: $,.
```

```
"== #('dogs' 'cats' 'gophers' 'zebras')"
```

Smalltalk has Character objects



```
var animals = 'dogs,cats,gophers,zebras';
```

```
var individualAnimals = animals.split(',');
```

```
// == ['dogs', 'cats', 'gophers', 'zebras'];
```

Dart does not have character objects



Test if a string starts with a substring



'racecar' beginsWithSubCollection: 'race'

"== true"

'racecar' beginsWithSubCollection: 'pace'

"== false"



```
'racecar'.startsWith('race');
```

```
// == true
```

```
'racecar'.startsWith('pace');
```

```
// == false
```



If statements



```
| bugNumbers |  
bugNumbers := #(3234 4542 944 124).  
bugNumbers size > 0 ifTrue: [  
  Console log: 'Not ready for release']
```



```
var bugNumbers = [3234,4542,944,124];  
if (bugNumbers.length > 0) {  
  print('Not ready for release');  
}
```



Checking for empty string



```
| emptyString |  
emptyString := "".  
emptyString isEmpty  
ifTrue: [Console log: 'string is empty']
```



```
var emptyString = "";  
if (emptyString.isEmpty()) {  
  print('string is empty');  
}
```




Method definition



No return type is specified

```
fn
...
^true
```



Specifying the return type of the function is Dart in optional.

```
fn() {
...
return true;
}
```

...can also be written as...

```
bool fn() {
...
return true;
}
```



Anonymous functions



```
[true]
```

```
[:msg | msg toUpperCase]
```



```
() => true
```

```
(msg) => msg.toUpperCase();
```



Return value



fn

```
^'Hello'
```

self fn.

```
"== 'Hello'"
```

fn2

```
self fn.
```

```
"returns self"
```



Use the 'return' keyword in a function definition to return a value.

```
fn() { return 'Hello'; }
```

```
fn(); // == 'Hello'
```

A function with no return value returns null.

```
(({}))(); // == returns null
```

if the body of the function is returning a single expression, this is the short form:

```
fn() => true;
```



Non-local returns



```
detect: aBlock  
  self do: [:each |  
    (aBlock value: each) ifTrue: [^each]].  
  self error: 'Not found'].
```



Non-local returns **not** supported

```
detect( Function f ) {  
  forEach( smallerFunction (each) {  
    if (f(each)) return each;});  
}
```

Doesn't work, return only returns from "smallerFunction" that's being passed to forEach, not from the "detect" function.

```
detect (Function f) {  
  for (var each in this) {  
    if (f(each)) return each;};  
}
```

This does work, because **for each in** is a built-in construct, so return returns from the "detect" method. The **for..in** is implemented in terms of iterators, so can apply to any collection.



Accessors



```
Object subclass: #Name
  instanceVariableNames: 'first last'
  classVariableNames: ''
  poolDictionaries: ''
```

Fields are private to an object, so accessor methods are required.

```
first
  ^first
```

```
first: aString
  first := aString
```

```
last
  ^last
```

```
last: aString
  last := aString
```

```
| name fullName |
name := Name new.
name first: 'John'.
name last: 'Doe'
fullName = name first, ' ', name last
```



Fields are public and may be accessed directly.

```
String first, last;
```

```
var name = new Name().
name.first = 'John';
name.last = 'Doe';
var fullName = name.first + ' ' + name.last
```

Dart allows getters/setters to be defined that look like property accessors.

```
class Rectangle {
  num left, top, width, height;
  num get right() => left + width;
  set right(num value) => left = value - width;
  num get bottom() => top + height;
  set bottom(num value) => top = value - height;
  Rectangle(this.left, this.top, this.width, this.height);
}
```

```
var rect = new Rectangle(3, 4, 20, 15);
print(rect.left);
print(rect.bottom);
rect.top = 6;
rect.right = 12;
```



Assign a function to a variable



```
loudify := [:msg | msg toUpperCase].
```

```
loudify value: 'not gonna take it anymore'.
```

```
“ == ‘NOT GONNA TAKE IT ANYMORE’ ”
```



```
var loudify = (msg) => msg.toUpperCase();
```

```
loudify('not gonna take it anymore');
```

```
// ‘NOT GONNA TAKE IT ANYMORE’
```



Optional parameters



Smalltalk does not have support for optional parameters.

```
fn: a with: b with: c  
  ^c
```

```
fn: 1 "ERROR: MessageNotUnderstood"
```

```
fn: 1 with: 2 with: 3 "==" 3"
```



```
fn(a, b, c) => c;
```

```
fn(1); // ERROR: NoSuchMethodException
```

```
fn(1, 2, 3); // == 3
```

Dart specifies optional parameters with square braces

```
fn(a, [b, c]) => c;
```

```
fn('a'); // == null
```



Default parameters



Smalltalk does not have native support for default parameters.



```
send(msg, [rate='First Class']) {  
  return '${msg} was sent via ${rate}';  
}  
  
send('hello');  
// == 'hello was sent via First Class'  
  
send("I'm cheap", '4th class');  
// == "I'm cheap was sent via 4th class"
```




Named parameters



All keyword parameters in Smalltalk are named.

```
sendMsg: msg rate: rate  
  ^msg, 'was sent via ', rate.
```

```
self sendMsg: 'I'm cheap rate: '4th class'  
" == 'I'm cheap was sent via 4th class'"
```



```
send(msg, [rate='First Class']) {  
  return '${msg} was sent via ${rate}';  
}
```

Use named parameters if the argument is optional

```
send("I'm cheap", rate: '4th class');  
// == "I'm cheap was sent via 4th class "
```

Note that you could use named parameters to define very Smalltalk-ish idioms



Method cascades



Smalltalk method calls can cascade.

```
self address  
  street: "Elm" number: "13a";  
  city: "Carthage";  
  state: "Eurasia";  
  zip: 66666 extended: 6666.
```

The expression resolves to the result of the last message send. To resolve to the receiver, many cascades end with a call to the #yourself method



Dart method cascade proposed using `..` syntax, but not yet available.

```
getAddress()  
  ..setStreet("Elm", "13a")  
  ..city = "Carthage"  
  ..state = "Eurasia"  
  ..zip(66666, extended: 6666);
```

The expression always resolves to the receiver



For loops for lists



```
| colors |  
colors := #('red' 'orange' 'green').  
1 to: colors size do: [:i |  
    Console log: (colors at: i).  
].
```

Note that lists are one-based



```
var colors = ['red', 'orange', 'green'];  
for (var i = 0; i < colors.length; i++) {  
    print(colors[i]);  
}
```

Note that lists are **zero-based**



For-in loops



```
| fruits |  
fruits := #('orange' 'apple' 'banana').  
fruits do: [:fruit |  
  Console log: fruit].
```



'in' notation in Dart returns the element of the list, not the index

```
var fruits = ['orange', 'apple', 'banana'];  
for (var fruit in fruits) {  
  print(fruit );  
}
```



For loops for object maps



```
data := Dictionary new ...  
data keysDo: [:key |  
  Console log: key, ' ', (data at: key)].
```

```
data keysAndValuesDo: [:key :value |  
  Console log: key, ' ', value].
```



```
var data = { ... };  
for (var key in data.getKeys()) {  
  print('$key, ${data[key]}');  
}
```

Alternatively, the **forEach** loop is a method on a Map in Dart.

```
data.forEach(  
  (key, value)=>print('${key}, ${value}')  
);
```



Closures and counters in loop



```
callbacks := OrderedCollection new.  
1 to: 2 do: [:i |  
  callbacks add:  
    [Console log: i asString]].
```

(callbacks at: 1) value “== ‘1’”

(callbacks at: 2) value “== ‘2’”



```
var callbacks = [];  
for (var i = 0; i < 2; i++) {  
  callbacks.add(() => print(i));  
}
```

callbacks[0]() // == 0

callbacks[1]() // == 1

In JavaScript, both would have returned 2



Filter a list



```
list := #( 1 2 3 4 5).
```

```
list select: [:x | x odd].
```

```
"#(1 3 5)"
```



```
var list = [1, 2, 3, 4, 5];
```

```
list.filter((x) => (x & 1) == 1);
```

```
// [1, 3, 5]
```

[update March 19, 2012]

the `isEven()` and `isOdd()` methods have been added to `int` so we can write the above as:

```
list.filter((x) => x.isOdd());
```



Transform a list



```
list := #(1 2 3 4 5).
```

```
list collect: [:x | x * 2].
```

```
"#( 2 4 6 8 10)"
```



```
var list = [1, 2, 3, 4, 5];
```

```
list.map((x) => x * 2);
```

```
// [2, 4, 6, 8, 10]
```




List reduction



```
#(1 3 5) inject: 10 into:  
  [ :sum :element | sum + element ]  
"19"
```



```
reduce([1,3,5],  
  (current, total) => current + total, 10)  
// 19  
  
Need to create the 'reduce' function  
  
reduce(source, callback, current) {  
  final i = source.iterator();  
  while (i.hasNext()) {  
    current = callback(current, i.next());  
  }  
  return current;  
}
```



Find 1st element matching a condition



```
#(1 2 3 4 5) detect: [:each | each > 3]
```

```
"4"
```



```
detect([1,2,3,4,5], (each) => each > 3)
```

```
// 4
```

Need to create the 'detect' function

```
detect(source, callback) {  
  final i = source.iterator();  
  while (i.hasNext()) {  
    var current = i.next();  
    if (callback(current)) {  
      return current;  
    }  
  }  
}
```



Define Classes



Smalltalk doesn't really have class definitions in the sense of having an actual syntax for them. Class creation is actually a reflective message send."

```
Object subclass: #Person
  instanceVariableNames: 'name'
  classVariableNames: ''
  poolDictionaries: ''
```

name

^name

name: aString

name := aString

great

^'Hello, ', name



Dart provides a specific syntax for defining classes (very similar to Java)

```
class Person {
  var name;
  greet() => 'Hello, $name';
}
```



Constructors / Instantiation



Objects are created by sending the #new message to a class object”

```
| person |
```

```
person := Person new.
```

```
person name: 'John Doe'.
```

```
person greet "== 'Hello, John Doe'"
```



Dart uses the “new” keyword to create instances

```
var person = new Person()
```

```
person.name = 'John Doe';
```

```
person.greet() // == 'Hello, John Doe'
```



Constructor with parameter



Constructors can be any class method

Object subclass: #Person

instanceVariableNames: 'name'

classVariableNames: ''

poolDictionaries: ''

“instance methods”

name: aString

name := aString

“class method”

new: aName

^super new name: aName

“use a better method name”

named: aName

^super new name: aName

| person |

person := Person named: 'John Doe'.

person greet "== 'Hello, John Doe'



```
class Person {  
  var name;  
  Person(name) {  
    this.name = name;  
  }  
}
```

Shorter alternative

```
class Person {  
  var name;  
  // parameters prefixed by 'this.' will assign to  
  // instance variables automatically  
  Person(this.name);  
}
```

```
var person = new Person('John Doe');  
person.greet // == 'Hello, John Doe'
```

Constructors can also be named

```
Person.named(this.name);
```



Reflection



```
| name |  
name := 'Bob'.  
name class " == String"  
name class class " == String class (a MetaClass)"  
name class class class " == MetaClass"  
  
name class methodDictionary keys  
" == #(name name:)"  
  
name class class methodDictionary keys  
" == #(new: named:)"
```

Dynamically add new instance variables to a class

```
Person addInstVarName: 'age'
```



There currently is no way to get the class of an object.
Reflection support coming soon (likely based on **Mirrors**).

Fields can't be added dynamically



Check the type



| name |

name := 'Bob'.

name isKindOf: String “== true”

name isKindOf: Integer “== false”



```
var name = 'Bob';
```

```
name is String // == true
```

```
name is int // == false
```

```
name is! int // == true
```

```
name is !int // == true
```



Subclass



```
Object subclass: #Person
  instanceVariableNames: 'name'
  classVariableNames: ''
  poolDictionaries: ''

Person subclass: #Employee
  instanceVariableNames: 'salary'
  classVariableNames: ''
  poolDictionaries: ''

salary: aNum
  salary := aNum

greet
  ^super greet, ', your salary is ', salary asString.
```



```
class Person {
  var name;
  Person(this.name);
  greet() => 'Hello, $name';
}

class Employee extends Person {
  var salary;
  Employee(name, this.salary) : super(name);
  greet() =>
    '${super.greet()}, your salary is $salary';
}
```




User Defined Operators



Most of the special (non-alphabetic) characters can be used as binary messages. These allow mathematical and logical operators to be written in their traditional form

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  poolDictionaries: ''
```

“class method assuming accessors for x & y”

```
x: xValue y: yValue
  ^self new x: xValue; y: yValue
```

“define a binary instance method”

```
+ aPoint
  ^self class x: x + aPoint x y: y + aPoint y
```

```
| p1 p2 p3 |
p1 := Point x: 1 y: 1.
p2 := Point x: 2 y: 2.
p3 := p1 + p2
```

Note that all binary messages are considered to have equal precedence and are evaluated simply from left to right. Thus:

3 + 4 * 5 = 35



Dart has only a fixed set of operators with fixed precedence, and some operators are not user-definable.

```
class Point {
  var x, y;
  Point(this.x, this.y);
  operator + (Point p) => new Point(x + p.x, y + p.y);
  toString() => 'x:$x, y:$y';
}
```

```
var p1 = new Point(1, 1);
var p2 = new Point(2, 2);
var p3 = p1 + p2;
print(p3)
```



Missing Methods



The Smalltalk Object class implements the `#doesNotUnderstand:` method which takes a Message instance as an argument.

```
doesNotUnderstand: aMessage
```

The Message instance contains the name of the method that was called and the arguments that we passed.

```
Object subclass: #Message  
  instanceVariableNames:  
    'selector arguments '  
  classVariableNames: ''  
  poolDictionaries: ''
```



The Dart Object class implements the `noSuchMethod()` method which takes the name of the function and the list of arguments as its arguments”

```
void noSuchMethod(  
  String function_name, List args)
```



Throw an exception



Exception signal: 'Intruder Alert!!'



```
throw new Exception("Intruder Alert!!");
```



Catch an exception



```
[[Math parseInt: 'three'  
  on: BadNumberFormatException  
  do: [:sig |  
    Console log: 'Ouch! Detected: bnfe']  
  on: Exception  
  do: [:sig |  
    Console log: 'If some other type']]  
ensure: [  
  Console log: 'This always runs']
```



```
try {  
  Math.parseInt("three");  
} catch(BadNumberFormatException be) {  
  print("Ouch! Detected: $be");  
} catch(var e) {  
  print("If some other type of exception");  
} finally {  
  print("This always runs");  
}
```



Resumable exceptions



Exceptions are resumable.
Important for the debugger.

```
[(3 / 0) + 5]
```

```
on: ZeroDivide
```

```
do: [:ex | ex resume: 1]
```

```
"== 6"
```



Not supported (can't be efficiently
compiled into Javascript)

Note that exceptions have an optional
stack trace parameter

```
catch (var e, StackTrace trace) { ... }
```

Optimization so that exceptions can be
caught without needing to generate an
expensive trace.



Math precedence



Strict left to right evaluation

$$3 + 4 * 5 = 35$$



Standard math evaluation

$$3 + 4 * 5 = 23$$



Math functions



-4 abs “== 4”

4.89 ceil “== 5”

4.89 floor “== 4”

Random new next

(Float pi / 2) sin “== 1”

Float pi cos “== -1”

‘3’ asNumber “== 3”

‘3.14’ asDecimal “== 3.14”



-4.abs() // == 4

4.89.ceil() // == 5

4.89.floor() // == 4

Math.random()

Math.sin(Math.PI/2) // == 1

Math.cos(Math.PI) // == -1

Math.parseInt('3') // == 3

Math.parseDouble('3.14') // == 3.14



Integers



Smalltalk has indefinite precision integers

```
123456789012345 * 123456789012345
```

```
"== 15241578753238669120562399025"
```



Dart has indefinite precision integers
in the VM

```
123456789012345 * 123456789012345
```

```
// == 15241578753238669120562399025
```

Large ints overflow to floats in JS

```
123456789012345 * 123456789012345
```

```
// == 1.5241578753238668e+28
```

Need to be careful here



Fractions



Smalltalk has first class fractions

```
1 / 3 "==" 1 / 3"
```

```
1 / 3 isKindOf: Fraction "==" true"
```



Dart does not have fractions

```
1 / 3 // == 0.3333333333333333
```

```
1 / 3 is double // == true
```



Runtime Evaluation



Compiler evaluate:

```
'Console log: "Hello!"'
```



Dart doesn't support `eval()`. Dart is designed for speed, so it does not include certain features of JS, like `eval()`, that inhibit optimizations.

Note that **Mirrors** will allow you to execute existing code dynamically, which is pretty close to an awkward `eval`.



Adding a method to a class



Compiler

compile:

 'fullName

 ^firstName, " ", lastName'

inClass: Name



Dart doesn't support changing a class after the program has been compiled

Note that it is expected that, at least in development, the VM with **Mirrors** will allow such things.



Processes



Smalltalk is multi-threaded and most implementations (not all) use preemptive green threads that are scheduled by the VM instead of natively by the underlying operating system.

Create a new process by sending the #fork message to a block.

```
process := [self doSomething] fork.  
process class "==" Process"
```

Processes can be suspended, resumed, terminated, scheduled, etc.

```
process  
  suspend;  
  resume;  
  terminate.
```



All code in Dart runs in the context of an isolate. Each isolate has its own heap, which means that all values in memory, including globals, are available only to that isolate. The only mechanism available to communicate between isolates is to pass messages.

```
class Printer extends Isolate {  
  main() {  
    port.receive((message, replyTo) {  
      if (message == null) port.close();  
      else print(message);  
    });  
  }  
}  
  
main() {  
  new Printer().spawn().then((port) {  
    for (var message in ['Hello', 'from', 'other', 'isolate'])  
      port.send(message);  
  }  
  port.send(null);  
});  
}
```



Dart Is Not Done

- Reflection support?
- Rest arguments?
- enum?
- Pattern matching?
- More browser integration?



Dart Open Source Project

- Dart Web Site: www.dartlang.org
 - Dart language spec
 - Dart tutorials
 - Some prebuilt binaries
- Dart Project: dart.googlecode.com
 - Library and code samples
 - Dart virtual machine
 - Dart core libraries
 - Dart -> JavaScript compiler



Dart is Getting Ready

- "Batteries included"
 - language, libraries, editor, app stack
- Structured web programming
 - Compatible with today's web
- Please try it and give us feedback!
 - dartlang.org
- Let us know if Dart is your new 2nd favorite language 😊



Learn More

- Follow [+Dart Bits](#)
- Follow [#dartlang](#)
- Follow dartosphere.org
- try.dartlang.org in your browser
- Get the SDK
- Send feedback via dartbug.com



Appendixes

- Dart to JavaScript Example:
 - <https://gist.github.com/1385015>
- Dart Editor
 - Users
 - Goals
 - Strategy
 - Performance
 - Metrics



Users of Dart Editor

- Web programmers of varying backgrounds
 - Many languages - HTML, JS, Python, Java
 - Wide range of programming experience
- Primarily not Eclipse users



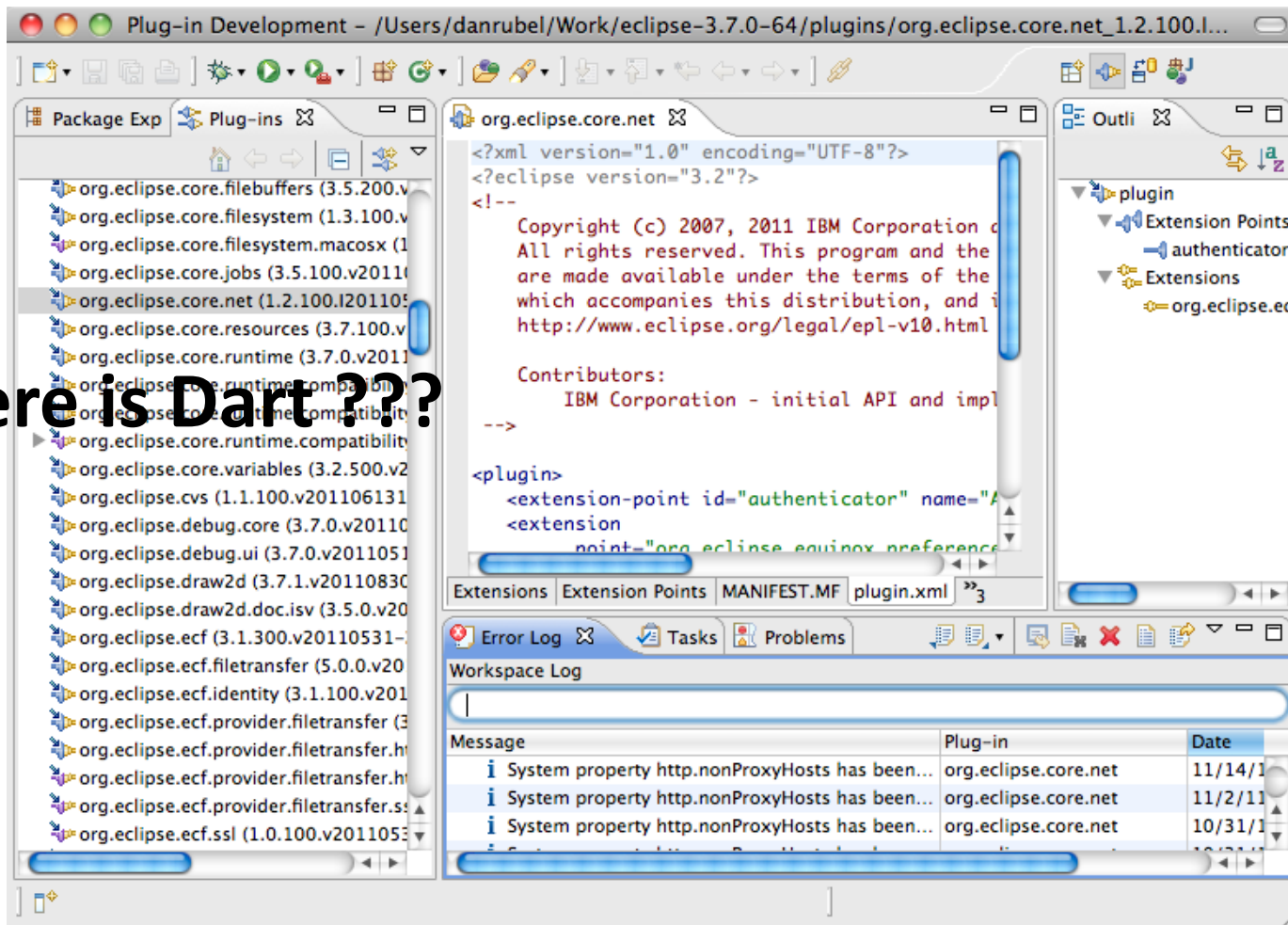
Goals of Dart Editor

- Introduce programmers to Dart
- Easy to understand
- Fast



Before - Cluttered UI

Where is Dart ???





Strategy

- Narrow the scope
 - Focus on doing a few things well
- Minimalist UI
 - Make it easy to understand
 - Reduce decision making



After - Easier to understand

The screenshot shows the Dart Editor interface. On the left is a 'Libraries' sidebar with a tree view containing 'Hello', 'Hello.html', 'Hello.dart', 'dart:core', and 'dart:dom'. The main editor window displays the code for 'Hello.dart':

```
#import('dart:dom');  
  
class Hello {  
  var myField;  
  
  Hello() {  
  }  
  
  void run() {  
    write("Hello World!");  
  }  
  
  void write(String message) {  
    // the DOM library defines a global "window" variable  
    HTMLDocument doc = window.document;
```

Below the code editor is a 'Problems' and 'Console' panel, which is currently empty and shows 'no items'. At the bottom of the editor, there is a status bar with 'Writable', 'Smart Insert', and '4 : 15'.



How?

- Single perspective
- Remove unnecessary plugins
- Redefine entire menu bar
- Use "activities" to suppress UI elements
- Key binding schema



Start-up Performance

- Remove unused plugins
- Modify plugins to remove dependencies
- Defer work until after UI appears
 - Early startup extension point
 - `Display.asyncExec(...)`
- Optimize load order
 - Record class load order
 - Reorder classes in plugin jar files



Application Performance

- Profile and optimize the code
 - Identify hotspots with VM profiler
 - Rewrite or eliminate slow code
- Defer work to background tasks



Performance-critical Areas

- Background indexing
- Code completion - how to make it fast
- Compilation time via incremental compilation



Metrics

First RCP build

65 MB

170 plugins

20s startup

Current build

37 MB

69 plugins

4s startup