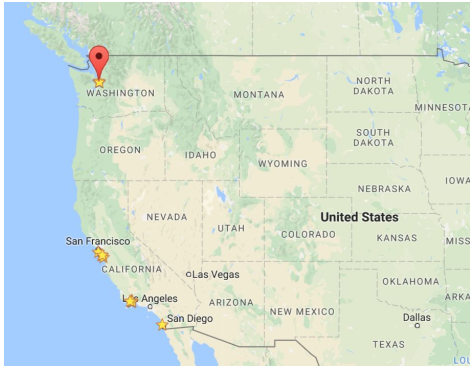
dart2java

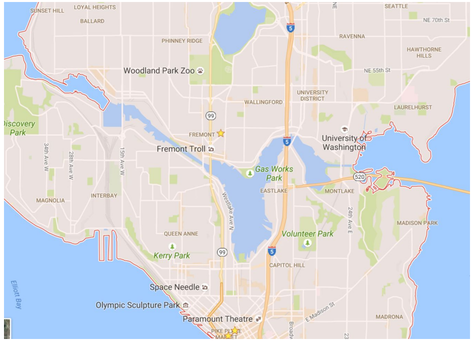
Internship Contributions

Matthias Springer

June 27, 2016 - October 7, 2016

Internship at Google Seattle





- 4 buildings in Fremont, ~100 interns in Seattle + Kirkland offices
- Working in the **Dart** team (Host: Vijay Menon)
- Dart Sub-Teams in Seattle, Portland (OR), Mountain View, Aarhus (Denmark)
- *Hierarchy*: Dart team (still) “belongs” to the Chrome team
- Group project with 2 other PhD interns (Andrew Krieger, Stan Manilov)

Why another Programming Language?

- An experiment, playground (optional typing, “isolates”, “mirrors”)
- Most of Google’s codebase is in Java, but Google is not in control of the language and its development
- *Previous:* built-in support in Chrome (“Dartium”), as a replacement for JavaScript
- Patent issues (Oracle)... Moving towards another programming language (also on Android, Web, ...)

Why compile Dart to Java?

- Android applications are written in Java
- Explore if Dart is suitable for AOT compilation (think of iOS)
- Most of Google’s codebase is in Java (interoperability with legacy code)

Dart Programming Language

- Object-oriented programming language with Java-like syntax
- Supports classes, single inheritance, mixins, optional typing, dynamic typing
- Supports generic classes (reified and covariant)
- No explicit interfaces, but abstract classes and classes can be implemented

- **Dart SDK:** Defines core classes/interfaces
  - Core interfaces: `dart:core.int`, `dart:core.num`, `dart:core:String`, `dart:core.List`, ...
  - Core classes: `dart:core.Stopwatch` (may have `external` functions)
- **Dart VM:** Written in C++, available for various operating systems
- **Dev Compiler:** Experimental Dart-to-JavaScript compiler
- **Analyzer:** Performs type inference, type checking of Dart code, provides (typed) AST representation
- **Kernel (AST):** Tree-based intermediate representation of Dart code (new)
- **Flutter:** Framework for writing Android and iOS applications in Dart
- **Dartino:** Dart for embedded devices (discontinued)
- **dart2java:** Dart-to-Java compiler (my project)

Dart Example Source Code

```
class A {
  A(this.foo);                             // constructor
  int foo;

  dynamic method(int a) => a + foo;        // base method
}

class B<T> implements A {
  int method(dynamic a) {                  // overridden method
    return super.method(10) as int + 10;
  }

  T get bar {                              // getter
    if (foo is T) { … }                    // generic type check
    return null;
  }
}
```

Dart Programming Language

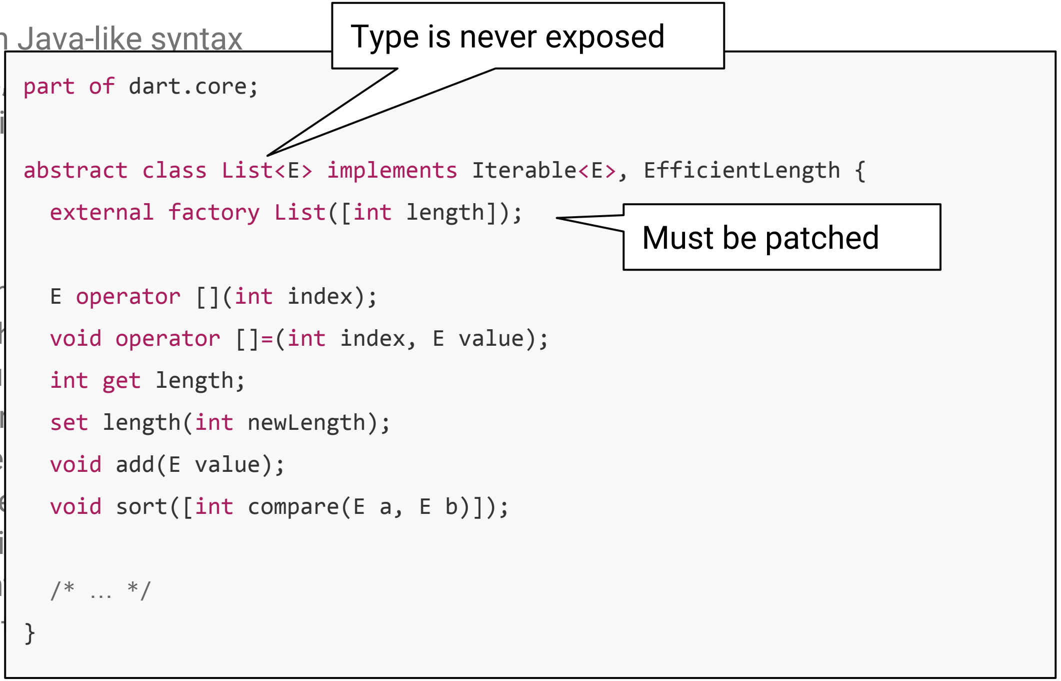
- Object-oriented programming language with Java-like syntax
- Supports classes, single inheritance, mixins, optional typing, dynamic typing
- Supports generic classes (reified and covariant)
- No explicit interfaces, but abstract classes and classes can be implemented

- **Dart SDK:** Defines core classes/interfaces
  - Core interfaces: `dart:core.int`, `dart:core.num`, `dart:core:String`, `dart:core.List`, ...
  - Core classes: `dart:core.Stopwatch` (may have `external` functions)
- **Dart VM:** Written in C++, available for various operating systems
- **Dev Compiler:** Experimental Dart-to-JavaScript compiler
- **Analyzer:** Performs type inference, type checking of Dart code, provides (typed) AST representation
- **Kernel (AST):** Tree-based intermediate representation of Dart code (new)
- **Flutter:** Framework for writing Android and iOS applications in Dart
- **Dartino:** Dart for embedded devices (discontinued)
- **dart2java:** Dart-to-Java compiler (my project)

Dart Programming Language

- Object-oriented programming language with Java-like syntax
- Supports classes, single inheritance, mixins,
- Supports generic classes (reified and covari
- No explicit interfaces, but abstract classes

- **Dart SDK:** Defines core classes/interfaces
  - Core interfaces: `dart:core.int`, dar
  - Core classes: `dart:core.Stopwatch`
- **Dart VM:** Written in C++, available for variou
- **Dev Compiler:** Experimental Dart-to-JavaScr
- **Analyzer:** Performs type inference, type che
- **Kernel (AST):** Tree-based intermediate repre
- **Flutter:** Framework for writing Android and i
- **Dartino:** Dart for embedded devices (discon
- **dart2java:** Dart-to-Java compiler (my projec



Type is never exposed

Must be patched

```
part of dart.core;

abstract class List<E> implements Iterable<E>, EfficientLength {
  external factory List([int length]);

  E operator [](int index);
  void operator []=(int index, E value);
  int get length;
  set length(int newLength);
  void add(E value);
  void sort([int compare(E a, E b)]);

  /* … */
}
```

Dart Programming Language

- Object-oriented programming language with Java-like syn
- Supports classes, single inheritance, mixins, optional typir
- Supports generic classes (reified and covariant)
- No explicit interfaces, but abstract classes and classes c

- **Dart SDK:** Defines core classes/interfaces
  - Core interfaces: `dart:core.int`, `dart:core.num`,
  - Core classes: `dart:core.Stopwatch` (may have e
- **Dart VM:** Written in C++, available for various operating sy
- **Dev Compiler:** Experimental Dart-to-JavaScript compiler
- **Analyzer:** Performs type inference, type checking of Dart
- **Kernel (AST):** Tree-based intermediate representation of
- **Flutter:** Framework for writing Android and iOS applicatio
- **Dartino:** Dart for embedded devices (discontinued)
- **dart2java:** Dart-to-Java compiler (my project)

```
part of dart.core;

class Stopwatch {
  void start() {
     if (_stop != null) {
       _start += _now() - _stop;
       _stop = null;
    }
  }

  void stop() {
    _stop ??= _now();
  }

  external static int _now();
}
```

Must be patched

Dart Programming Language

- Object-oriented programming language with Java-like syntax
- Supports classes, single inheritance, mixins, optional typing, dynamic typing
- Supports generic classes (reified and covariant)
- No explicit interfaces, but abstract classes and classes can be implemented

- **Dart SDK:** Defines core classes/interfaces
  - Core interfaces: `dart:core.int`, `dart:core.num`, `dart:core:String`, `dart:core.List`, ...
  - Core classes: `dart:core.Stopwatch` (may have `external` functions)
- **Dart VM:** Written in C++, available for various operating systems
- **Dev Compiler:** Experimental Dart-to-JavaScript compiler
- **Analyzer:** Performs type inference, type checking of Dart code, provides (typed) AST representation
- **Kernel (AST):** Tree-based intermediate representation of Dart code (new)
- **Flutter:** Framework for writing Android and iOS applications in Dart
- **Dartino:** Dart for embedded devices (discontinued)
- **dart2java:** Dart-to-Java compiler (my project)

Dart Types

program semantics well-defined
regardless of static type errors

we are using this one

- Typing is optional… Not so much anymore…

Unchecked Mode:

```
int foo = “Hello World”;
```

Checked Mode:

```
int foo = “Hello World”;
// Fails at runtime, but can be
// detected with Analyzer
```

Strong Mode:

```
// Similar to checked mode but
// more strict, so we can
// detect more errors statically
```

- Strong mode has additional type guarantees (some examples)
  - *Checked:* `List<int>` <: `List` <: `List<String>` <: `List` <: `List<int>`
    Strong: `List<int>` <: `List`
  - Automatic downcasts still possible (e.g., `Object a = 123; String b = a;`)
  - Types of variables declared with **`var`** are inferred statically instead of using **`dynamic`**
- *Optimistic type checking:* Assume code is valid unless statically sure that it is not.

T assignable to S if
T <:S or S <: T

“The lack of static or runtime errors in the Dart specification's type rules is not an oversight; it is by design. It provides developers a mechanism to circumvent or ignore types when convenient, but it comes at cost.” [1]

[1] https://github.com/dart-lang/dev_compiler/blob/master/STRONG_MODE.md

Table of Contents

Slide 12 - 24 Overview of Compiler Infrastructure
class generation, method calls, bootstrapping

Slide 25 - 30 Benchmark Results

Slide 31 - 57 Dart Generics
covariance, reification, specialization

01 Overview of Compiler Infrastructure

Current State of Implementation

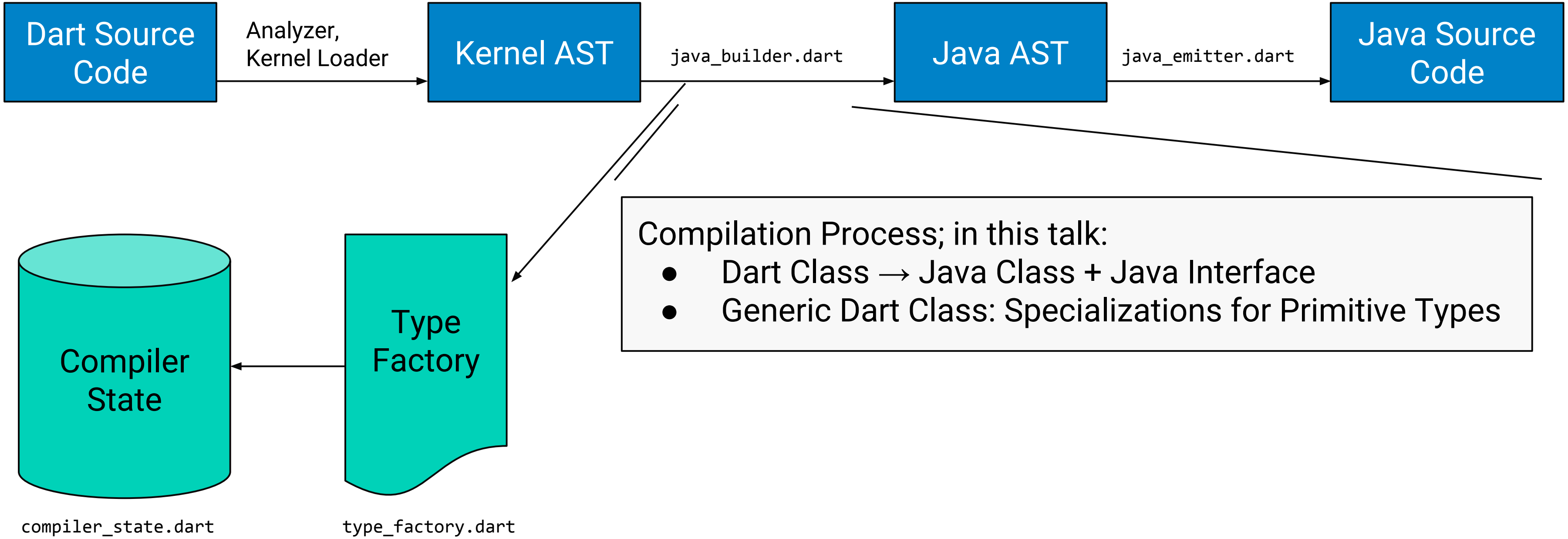
- Can compile **lots of (strong mode-compliant) Dart code**.
  No support for named parameters, exception, large parts of the SDK, anonymous functions (lambdas), mixins.
- ~25 unit test suites, various codegen test cases, benchmarks: 5 from `ton80` + various rendering benchmarks
- Support for **generic classes**. Generic methods partly supported (generic factory constructors are working).
  Generics are reified, covariant, and specialized for primitive type parameters.
- Java is statically typed: Use specified types / types inferred by Kernel (and `java.lang.Object` for `dynamic`).
- Working **run-time type system** (sometimes overly-conservative) performing type checks.
- Source code available on GitHub: https://github.com/google/dart2java

```
void testTypeCheckFails() {
  Map<String, String> mapStringString = new Map<String, String>();
  Map<Object, Object> mapObjectObject = mapStringString;
  mapObjectObject["this should fail at runtime"] = new List<int>();
}
```

Design Decisions

- **Maximize Usage of Primitive Types:** Use unboxed types wherever possible (`int, double, boolean`)
  (*Exception:* classes with >2 generic parameters)
- **Reuse Java Classes/Interfaces:** Use Java primitive types and collection interfaces for performance and interop.
  (→ Use Java generics together with our type system for reified types)
- Rely on **Java runtime type checks** whenever safe (performance)

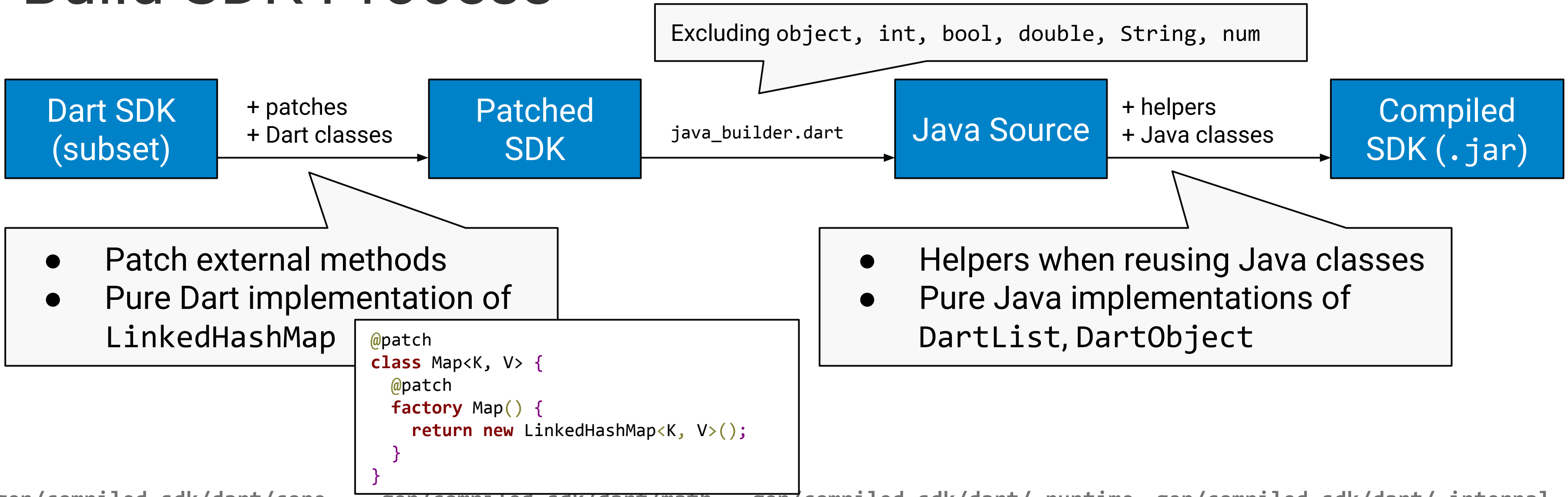
High-level Overview



Compilation Process; in this talk:

- Dart Class → Java Class + Java Interface
- Generic Dart Class: Specializations for Primitive Types

Build SDK Process



- Patch external methods
- Pure Dart implementation of `LinkedHashMap`

```
@patch
class Map<K, V> {
  @patch
  factory Map() {
    return new LinkedHashMap<K, V>();
  }
}
```

- Helpers when reusing Java classes
- Pure Java implementations of `DartList`, `DartObject`

```
gen/compiled_sdk/dart/core
Stopwatch.java
Stopwatch_interface.java
Map.java
Map_interface.java
Map__int_int.java
Map_interface__int_int.java
...
```

```
gen/compiled_sdk/dart/math
__TopLevel.java
Random.java
Random_interface.java
```

```
gen/compiled_sdk/dart/_runtime
DartObject.java
DartObject_interface.java
DartList.java
DartList__int.java
```

```
gen/compiled_sdk/dart/_internal
LinkedHashMap.java
LinkedHashMap_interface.java
LinkedHashMap__int_int.java
LinkedHashMap_interface__int...
...
```

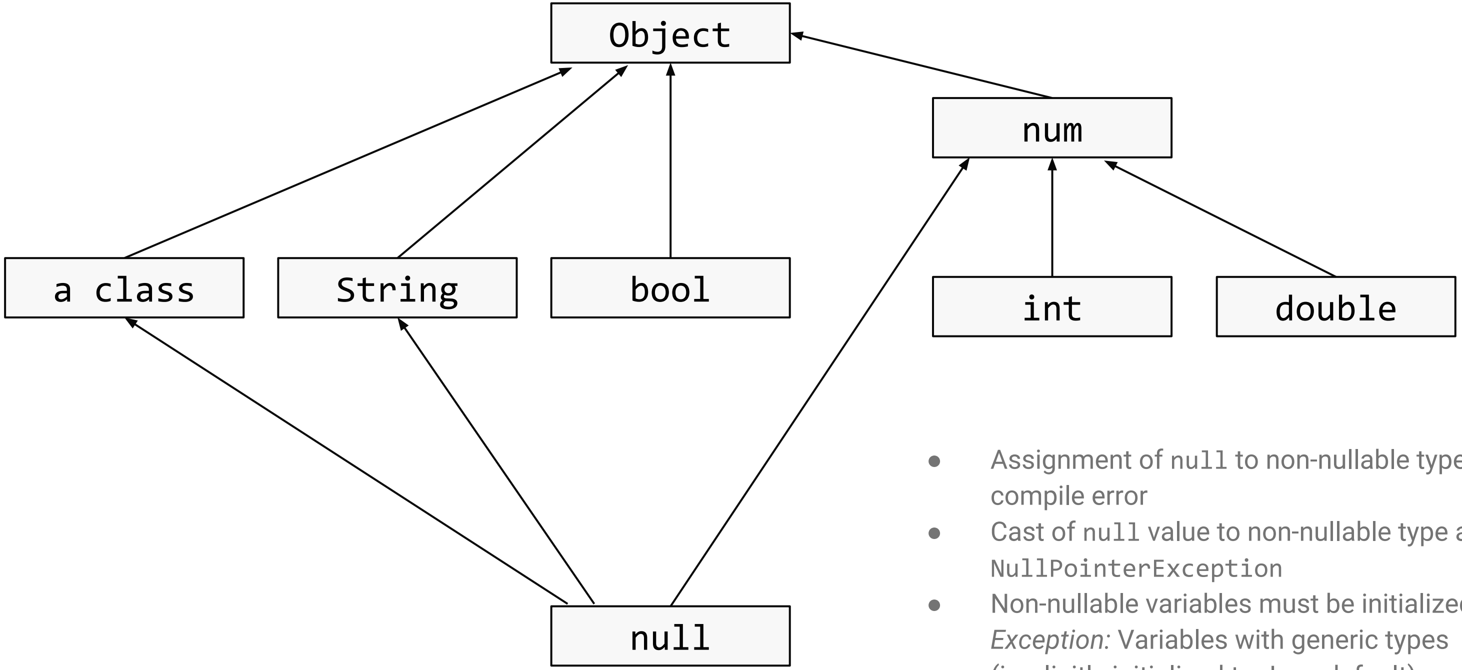
Example: `List` SDK Core Class

```
part of dart.core;

abstract class List<E> implements Iterable<E>, EfficientLength {

  external factory List([int length]);

  E operator [](int index);

  void operator []=(int index, E value);

  int get length;

  /* … */

}
```

patched

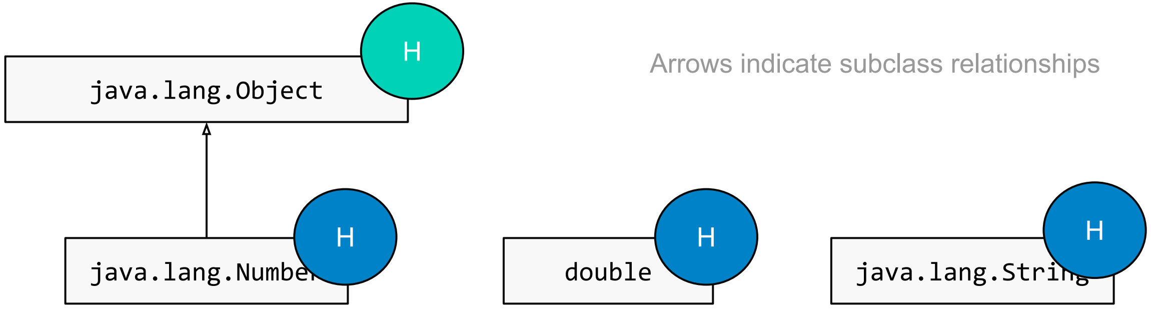
```
@patch
class List<E> {
 @patch
 @JavaCall("dart._runtime.base.DartList.<E>factory\$newInstance")
 external factory List([int length = 0]);
}
```

Type System (Primitive Types)

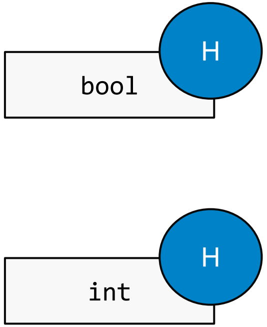


- Assignment of `null` to non-nullable type at compile type: compile error
- Cast of `null` value to non-nullable type at run time: `NullPointerException`
- Non-nullable variables must be initialized explicitly
  Exception: Variables with generic types
  (implicitly initialized to Java default)

Object Model



Arrows indicate subclass relationships



“special” classes

In most cases: Classes that have a Java implementation.

H *Helper Class:* Java class with static methods providing implementation of Dart methods

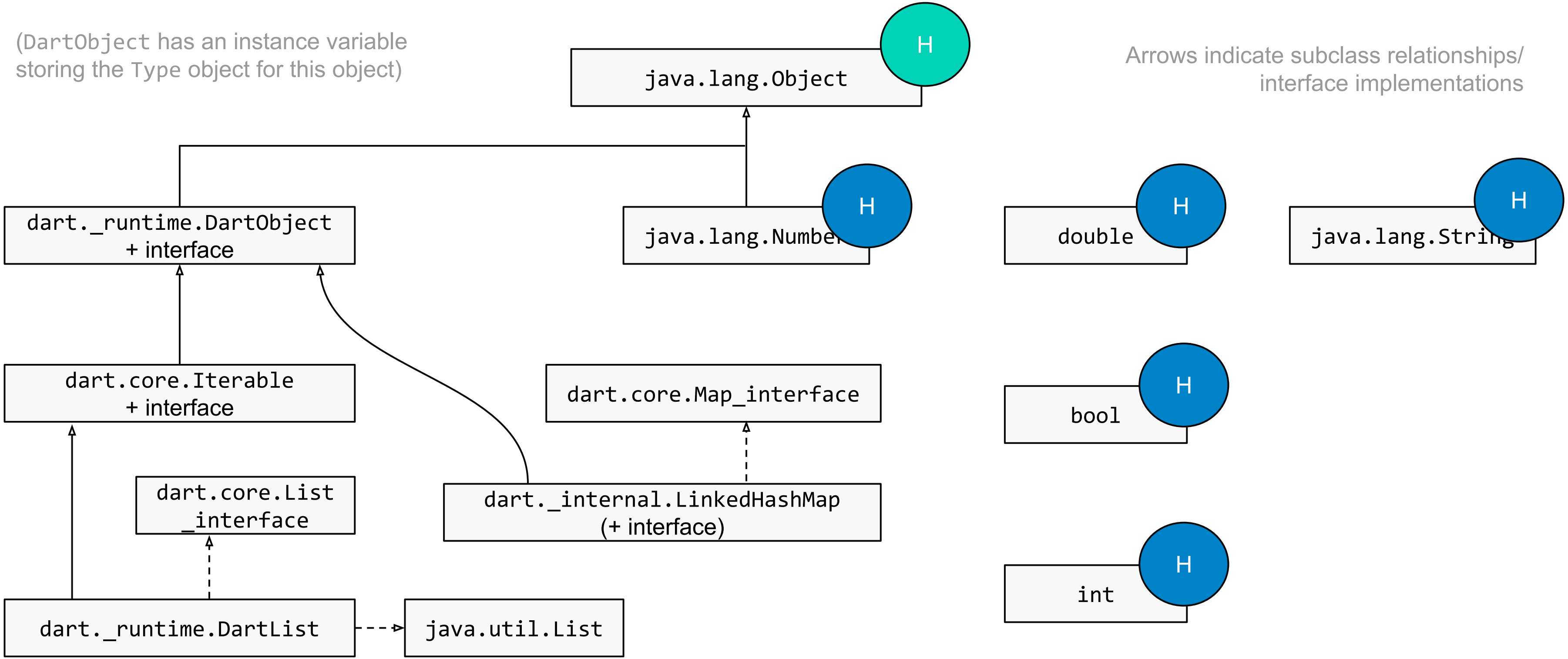
Helper Class Example

```java
package dart._runtime.helpers.IntegerHelper;

public static class IntegerHelper {
    public static int gcd(int self, int other) {
        if (b == 0) {
            return other;
        } else {
            return gcd(other, self % other);
        }
    }
}


// Dart: 10.gcd(5)
// Java: IntegerHelper.gcd(10, 5)
```

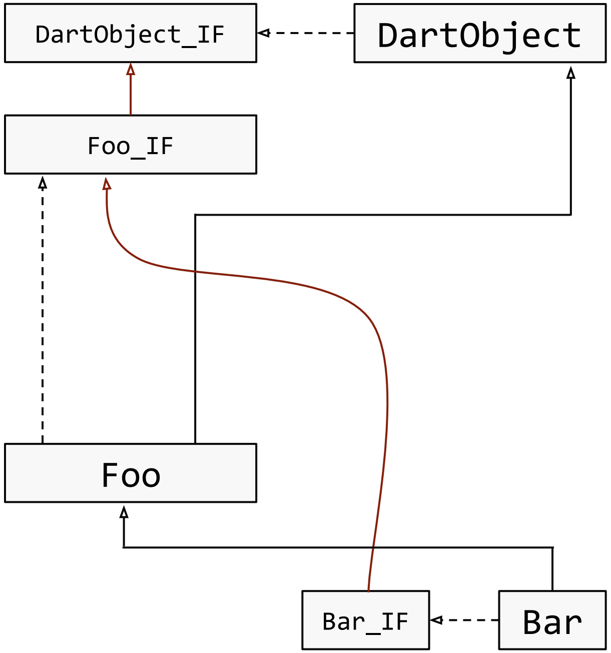
Object Model

(DartObject has an instance variable storing the Type object for this object)

Arrows indicate subclass relationships/
interface implementations



Example: Class Diagram (Dart → Java)



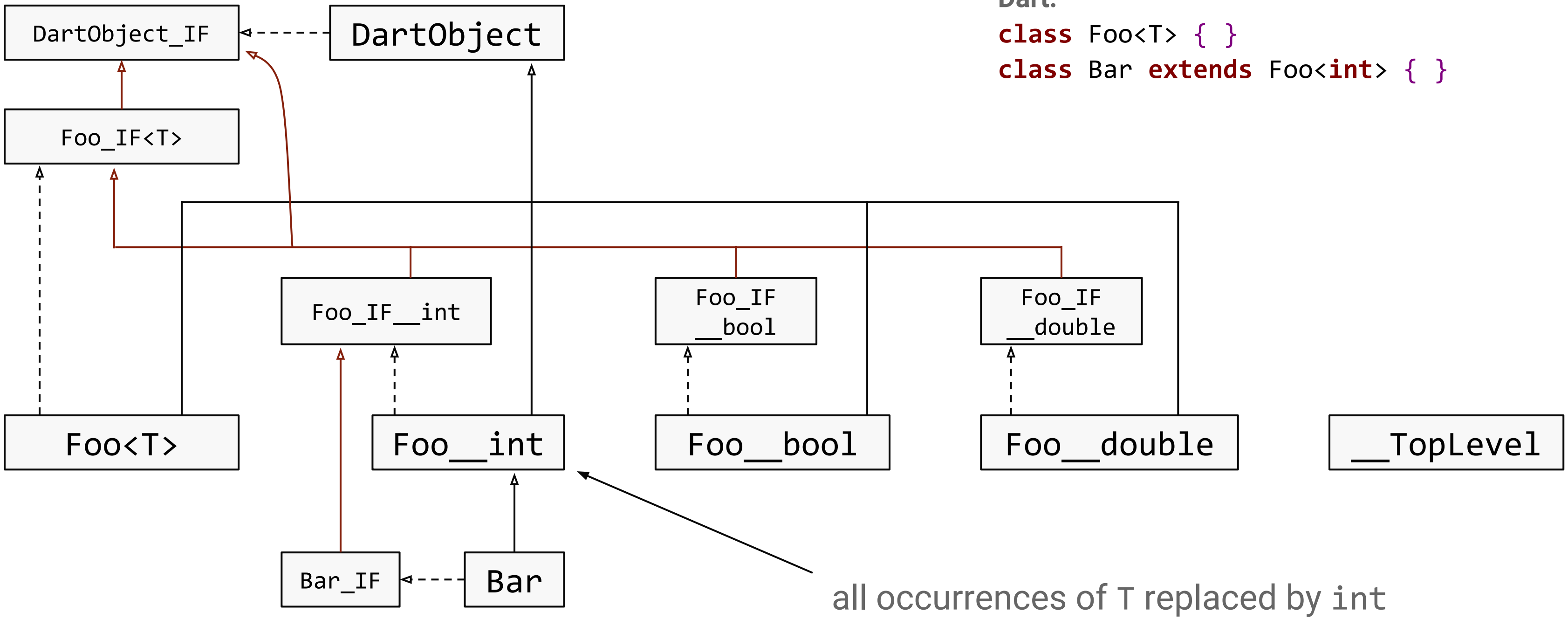
Dart:

```
class Foo { }
class Bar extends Foo { }
```

Java Generics

```
List<int> unboxedList;                    // Compile time error
List<Integer> boxedList;                  // OK

boxedList = new LinkedList<Integer>();
boxedList.add(10);                        // auto-boxing
```

Example: Class Diagram (Dart → Java)



Dart:

```
class Foo<T> { }
class Bar extends Foo<int> { }
```

all occurrences of T replaced by int

02 Benchmarks

Setting (Environment)

- Run on my workstation (Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz)
- 4 Configurations
  - **dart2java with generic specializations**
  - dart2java without generic specializations

    } Analyzer *Strong Mode*
  - Dart VM checked (1.18.0-dev.2.0)
  - **Dart VM unchecked (1.18.0-dev.2.0)**
- 1 second warmup, 10 seconds running
  (1 min. warumup results in minor speedup for `dart2java`)

Unchecked Mode:

```
int foo = “Hello World”;
```

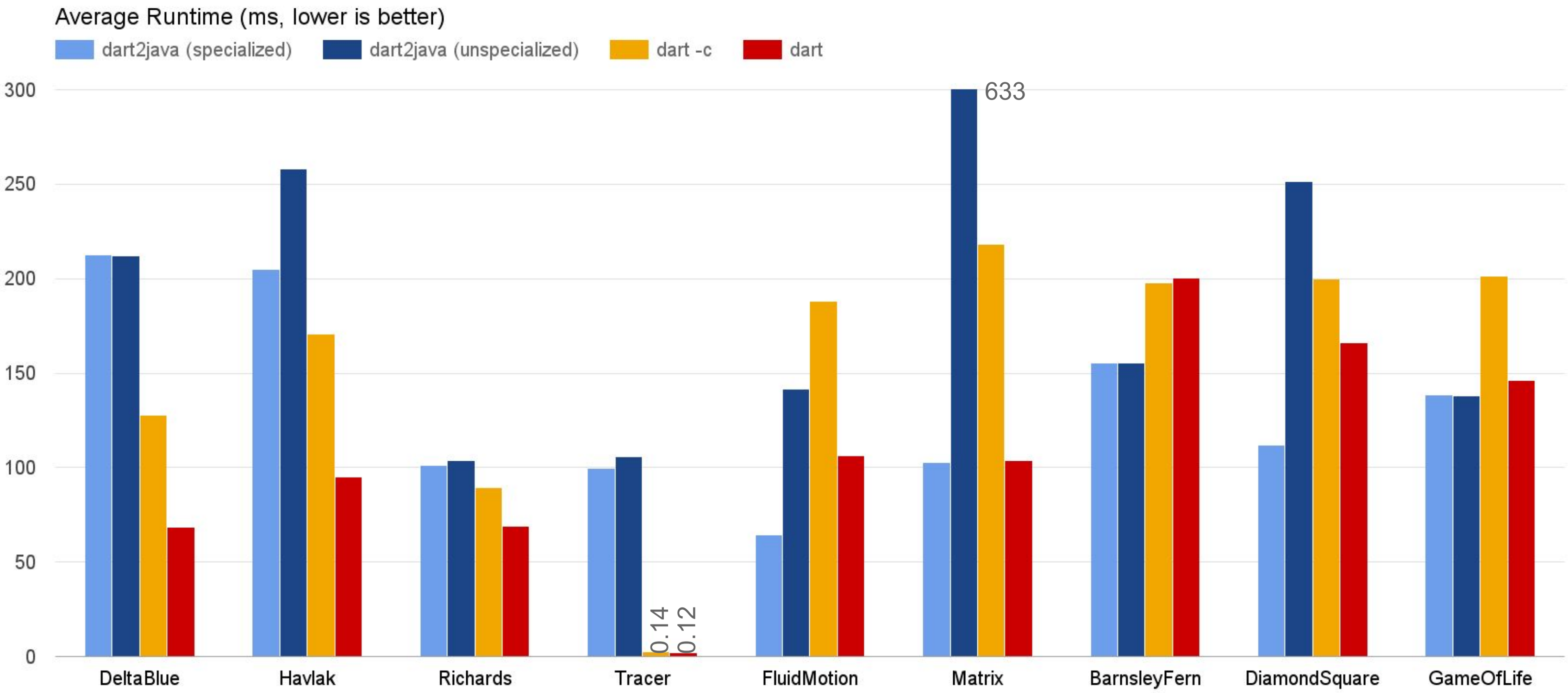
Checked Mode:

(more runtime checks)

Strong Mode:

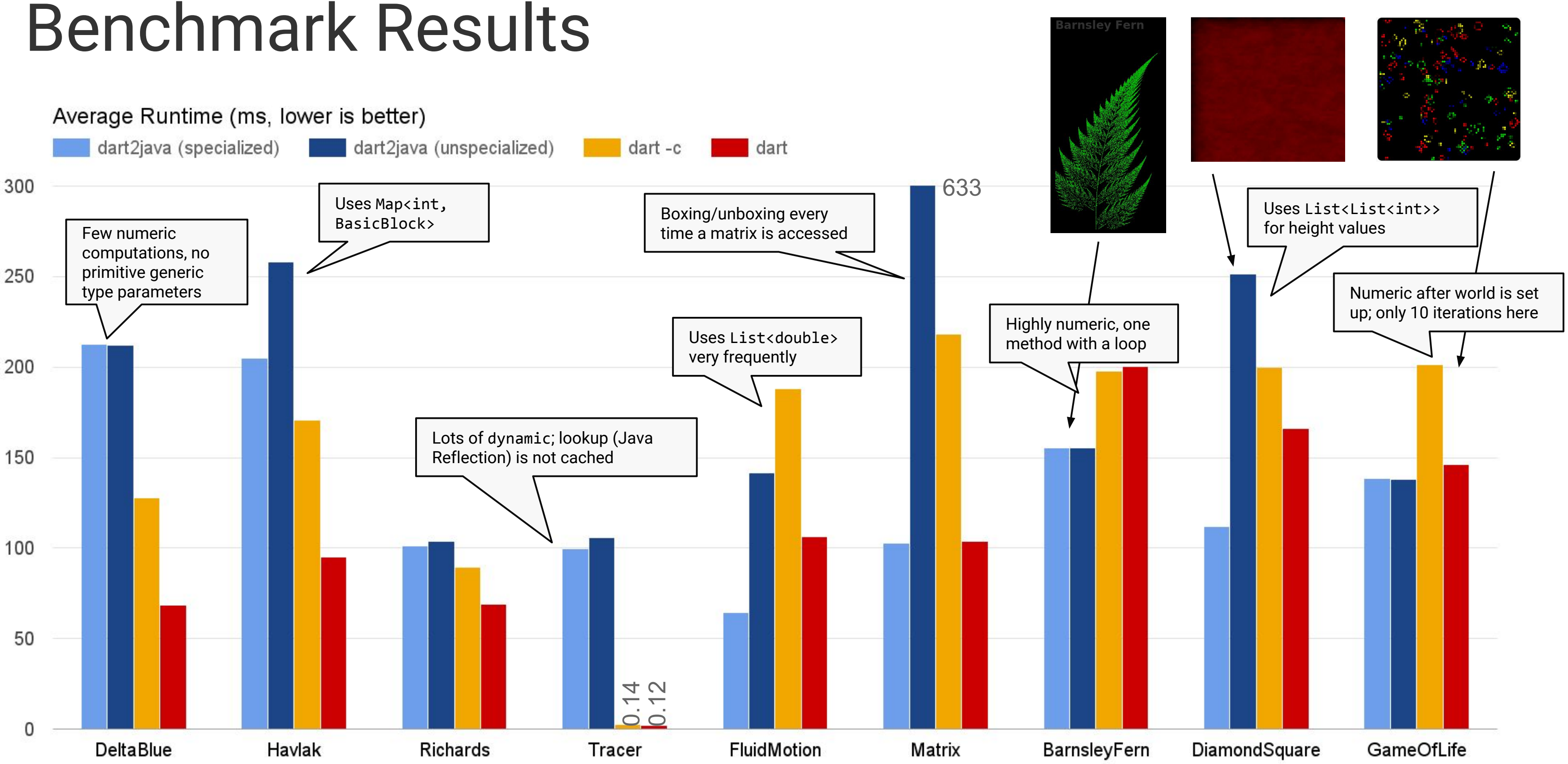
(strong type guarantees)

Benchmark Results



Benchmark Results

Last three examples taken from:
http://divingintodart.blogspot.com/ (Davy Mitchell)



Average Runtime (ms, lower is better)

Few numeric computations, no primitive generic type parameters

Uses `Map<int, BasicBlock>`

Lots of `dynamic`; lookup (Java Reflection) is not cached

Uses `List<double>` very frequently

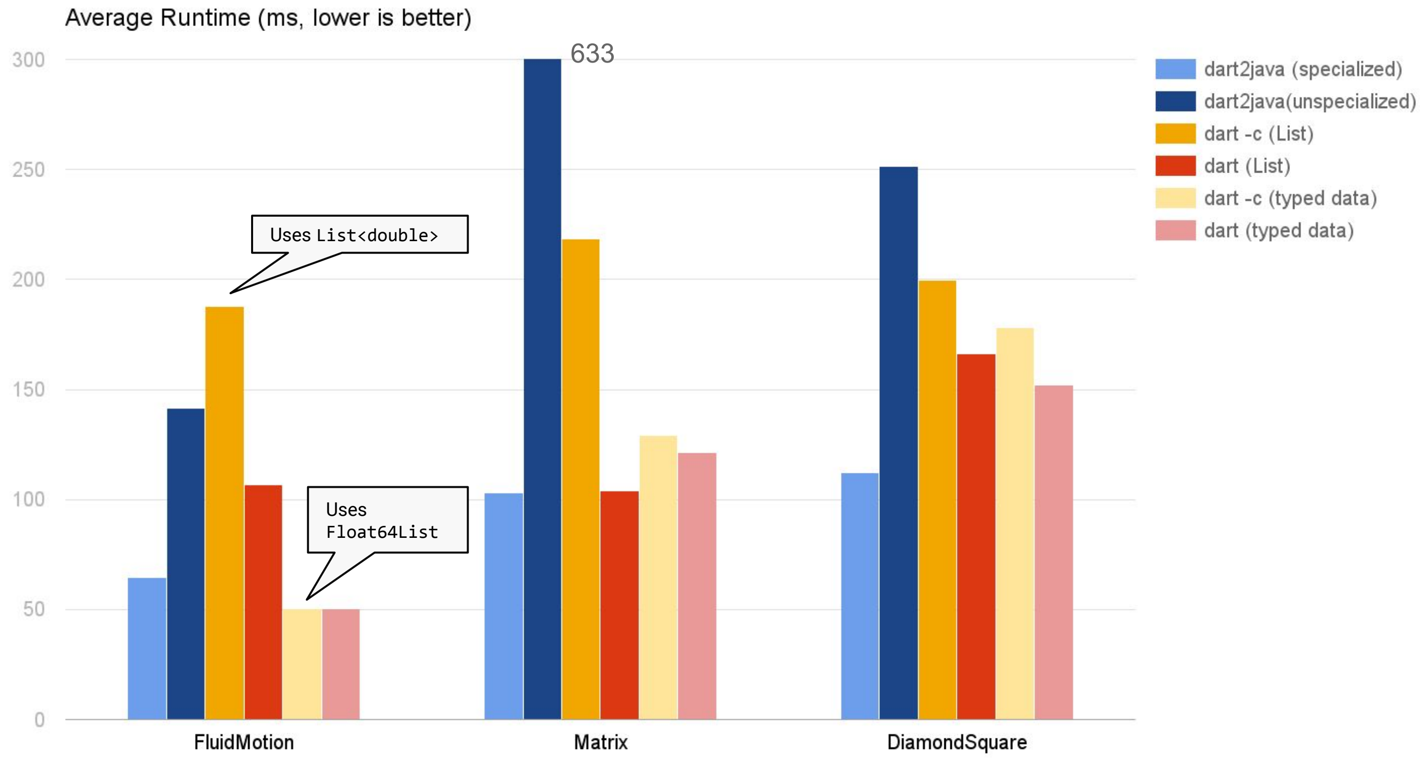
Boxing/unboxing every time a matrix is accessed

Highly numeric, one method with a loop

Uses `List<List<int>>` for height values

Numeric after world is set up; only 10 iterations here

Specialization vs. Typed Data List



Uses List<double>

Uses Float64List

Example: Barnsley Fern

```
int drawBarnsleyFern() {
    int checksum = 0;

    double x = 0.0;
    double y = 0.0;
    double nextx = 0.0;
    double nexty = 0.0;
    double plotDecider = 0.0;
    Random rng = new Random(1337);

    x = rng.nextDouble();
    y = rng.nextDouble();

    for (int i=0;i<50000;i++){

      plotDecider = rng.nextDouble();

      if (plotDecider<0.01)
      {
        x = 0.0;
        y = 0.16 * y;
      }
```

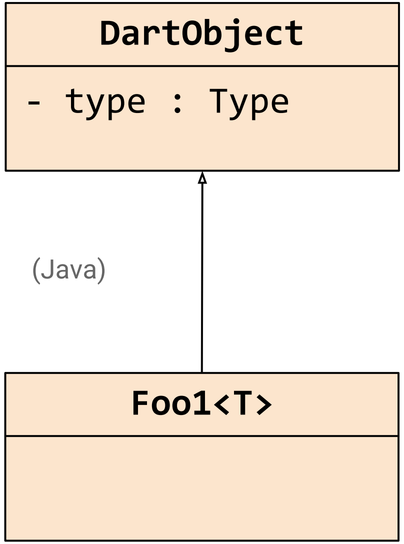
```
else if (plotDecider < 0.86)
{
  nextx = (0.85 * x) + (0.04 * y);
  nexty = (0.04 * x) + (0.85 * y) + 1.6;
  x = nextx;
  y = nexty;
}
else if (plotDecider < 0.92) {
  nextx = (0.2 * x) - (0.26 * y);
  nexty = (0.23 * x) + (0.22 * y) + 1.6;
  x = nextx;
  y = nexty;
}
else{
  nextx = (-0.15 * x) + (0.28 * y);
  nexty = (0.26 * x) + (0.24 * y) + 0.44;
  x = nextx;
  y = nexty;

}
```

```
    int col = 100 + rng.nextInt(143);
    // crc.fillStyle = "rgb(0,$col,00)";
    checksum += (100 + (x*50).toInt() +
       500 - (y*40).toInt()) % 9971;
  }

  return checksum;
}
```

https://github.com/daftspaniel/dartbarnsleyfern

03 Dart Generics

Reified Generics: Type Representation



How does an instance of `Foo1<T>` know what T is?

```
var fooObject = new Foo1<T>();
```

→ `fooObject.type` is “Foo<(whatever T is)> type”

```
var fooInt = new Foo1<int>();
```

→ `fooInt.type` is “Foo<int> type”

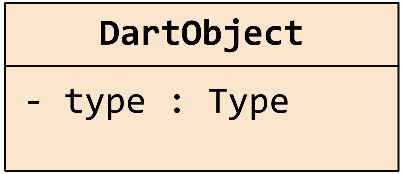
```
bool test = anObject is T;
```

Reified Generics

- Call Site:
  - **Constructor Invocation:** Retrieve `Type` from static variable (*hoisted*) and pass as first argument.
  - **Factory Invocation:** Build `TypeEnvironment` at call site and pass as first argument (if generic).
- Call Target:
  - **Constructor:** Store `Type` parameter in instance variable.
  - **Factory:** Regular translation process (static method), but never use any hoisted types, but build all types from scratch using passed `TypeEnvironment`.
    (Factory might call a constructor or another factory.)
- Dart Objects
  - `Type` instance variable, used for type checks, passing type variable around that is in scope.
  - `DartList`: `Type` variable + backed by reified generic array (`T[]`)

```
new Foo1<int>(42)
⇒ Foo1._new(dart2java$typeExpr_Foo1$ltint$0$gt, 42)
```

```
new Foo1<int>.aFactory(42)
⇒ Foo1.aFactory$factory(<T → int>, 42)

new Foo1<...>.aFactory(obj);
⇒ Foo1._new(<T → ...>, obj)
```



This slide is simplified: We hoist `TypeExpr` and not `Type` objects.

Java Generics for Interoperability

- Reified type information stored in Type instance variable
- For interoperability reasons: Use Java generics on top of that

Current interoperability for core SDK classes:

```
DartList<T> implements java.util.List<T>
```

What it should be like (if I had more time):

```
List_interface<T> extends java.util.List<T>
Map_interface<K, V> extends java.util.Map<K, V>
Iterable_interface<T> extends java.lang.Iterable<T>
Iterator_interface<T> extends java.lang.Iterator<T>
Comparable_interface<T> extends java.lang.Comparable<T>
```

Dart:

```
class Foo<A> {
      A variable;
}

var x = new Foo<String>();
```

Java:

```
class Foo<A> extends DartObject implements Foo_interface<A> {
    public static Foo _new(Type type) { … }

    A variable;

    public A getVariable() { return variable; }
    public A setVariable(A value) { … }
}

Foo<String> x = Foo._new(<type obj>);
```

Covariant Generics

- Comes (almost) for free when only using the run-time type system

Dart:

```
Foo<Object> o;
Foo<String> s;

o = s;
```

Java:

```
Foo o;
Foo s;

o = s;                                       // OK
```

Type checks inserted by run-time type system omitted.

Covariant Generics

- Comes (almost) for free when only using the run-time type system
- Requires additional casts when combined with Java generics

Dart:

```
Foo<Object> o;
Foo<String> s;

o = s;
```

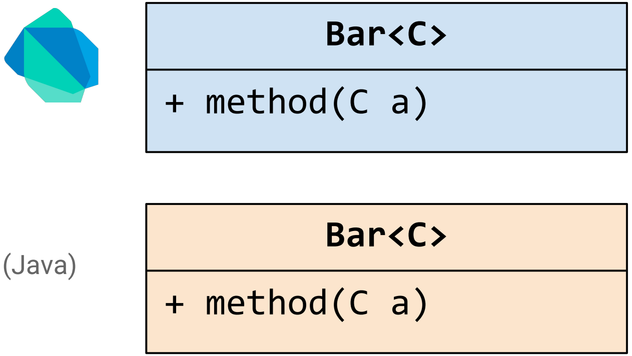
Java:

```
Foo<Object> o;
Foo<String> s;

o = s;                                   // Does not compile
o = (Foo<Object>) (Foo) s;               // OK
o = (Foo) s                              // OK (implicit cast)

List<? extends Object> o;
o = new List<String>();                  // Works, but cannot consume objects
```

Type checks inserted by run-time type system omitted.

See also: https://kotlinlang.org/docs/reference/generics.html

(Generic) Specialization: The Problem



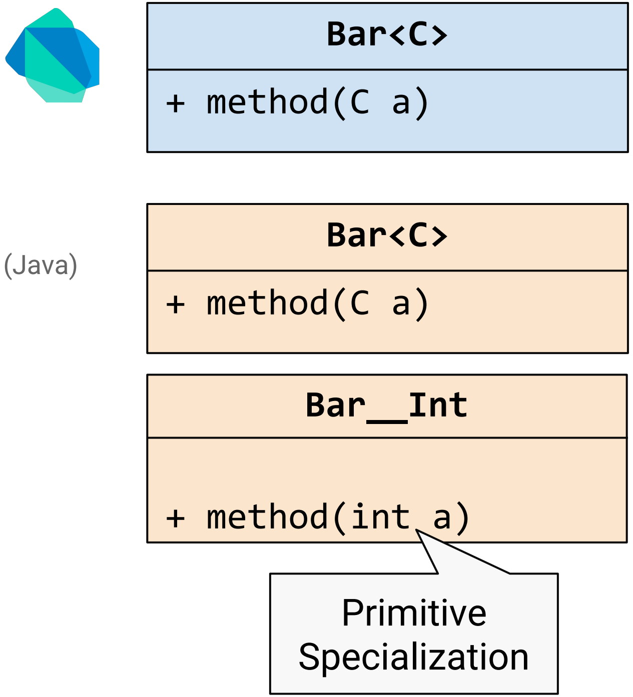
- **Goal:** Avoid boxing of primitive types
- **Bonus:** Get rid of some type checks

- Specialize for `bool, double, int`

```
Bar<int> object;
object.method(123);
⇒    Bar<Integer> object;
     object.method(123);                 (what we want)
     object.method(new Integer(123));    (what we get)
```

Implicit Boxing

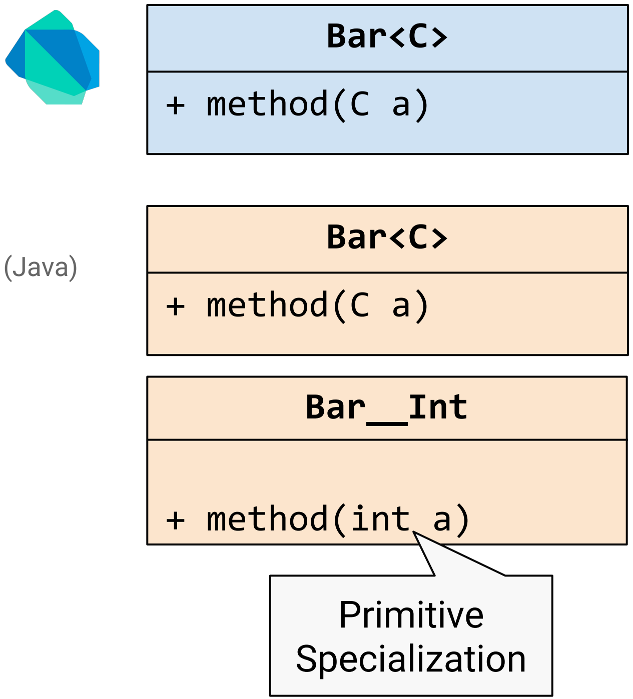
Specialization: Separate Implementations



- **Goal:** Avoid boxing of primitive types
- **Bonus:** Get rid of some type checks
- Create copies of generic classes with 1-2 type parameters (like C++ templates)
- Specialize for `bool, double, int`
- Invoke methods through specialized “unboxed” interface

```
Bar<int> object = new Bar<int>();
object.method(123);
⇒    Bar_IF__Int object = new Bar__Int();
     object.method(123);                       ✓
```

Specialization: Covariance Problem



```
Bar<C>
+ method(C a)
```

(Java)

```
Bar<C>
+ method(C a)
```

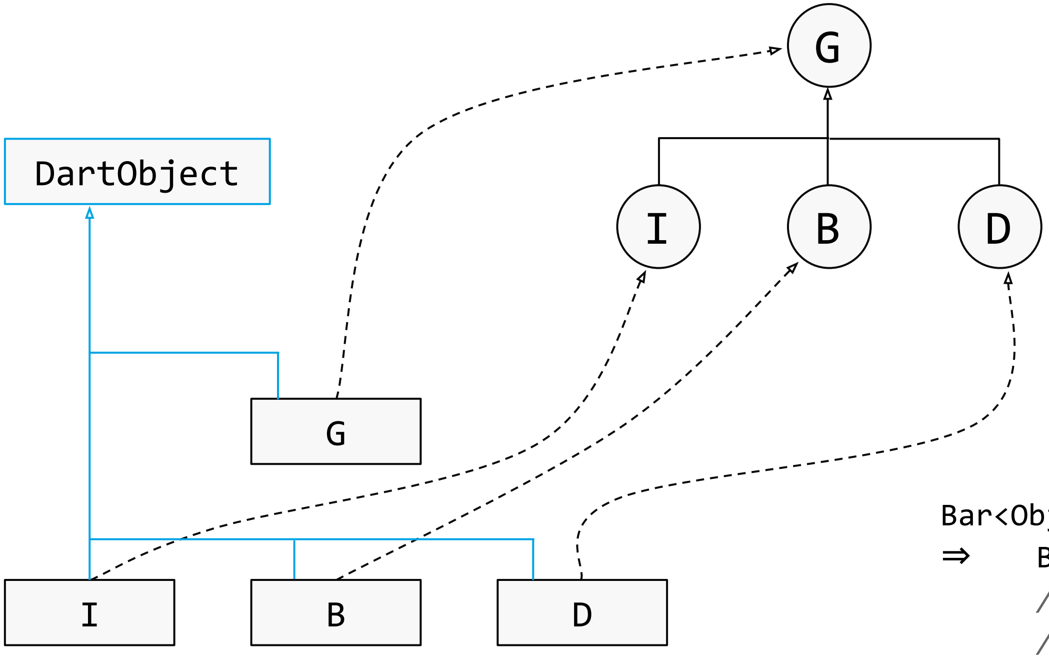
```
Bar__Int
+ method(int a)
```

Primitive Specialization

- **Goal:** Avoid boxing of primitive types
- **Bonus:** Get rid of some type checks
- Create copies of generic classes with 1-2 type parameters (like C++ templates)
- Specialize for `bool`, `double`, `int`
- Invoke methods through specialized “unboxed” interface

```
Bar<Object> object = new Bar<int>();
⇒    Bar_IF<Object> object = new Bar__Int();     ✗
     // compile error
```

Subtyping Relationship (1 Type Parameter)

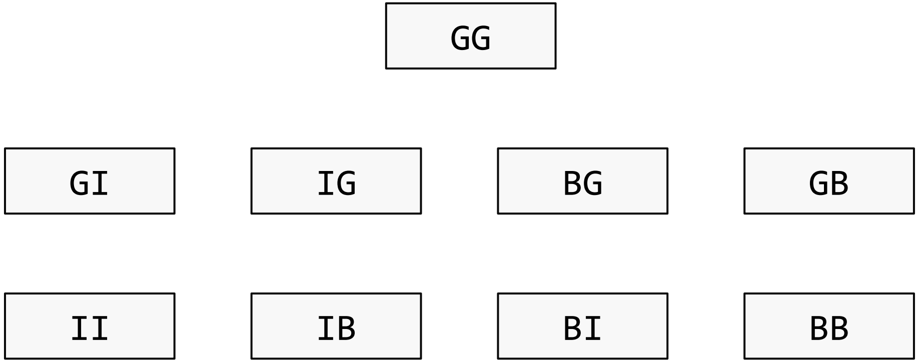


```
interface Bar_IF__int
      extends Bar_IF<Integer>
```

```
Bar<Object> object = new Bar<int>();
⇒    Bar_IF<Object> object = new Bar__Int();
     // ✓ OK
     // Bar__Int <: Bar_IF__Int
     // <: Bar_IF<Integer> <: Bar_IF<Object>
```

Subtyping Relationship (2 Type Parameters)

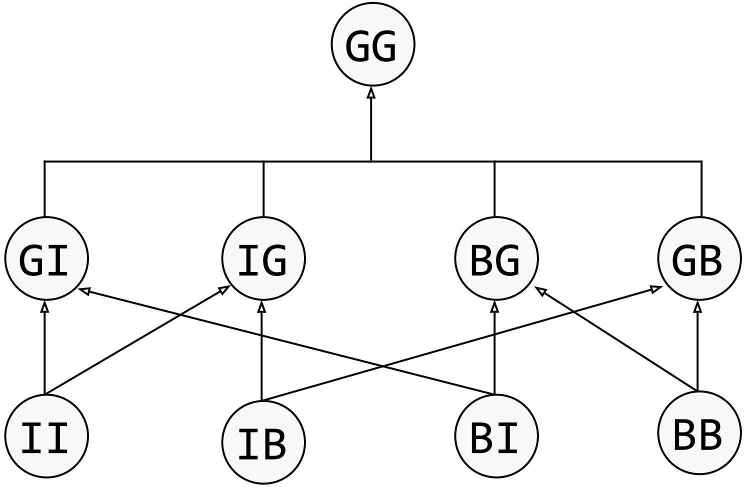
(only showing `int`, `bool` specializations)



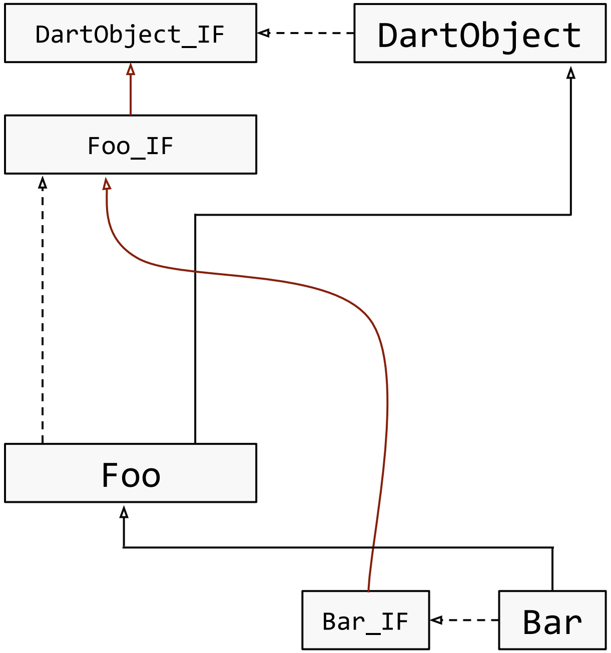
GG

GI IG BG GB

II IB BI BB



Example: Class Diagram (Dart → Java)

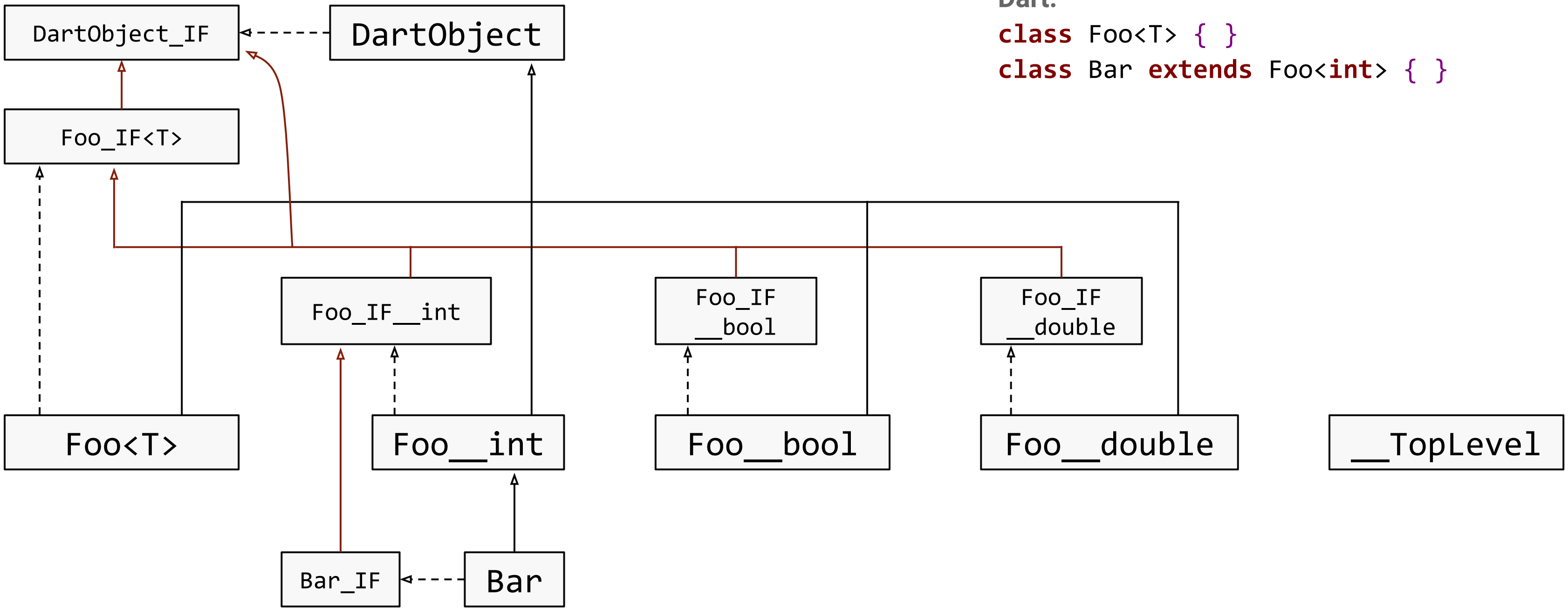


Dart:

```
class Foo { }
class Bar extends Foo { }
```

(same as on slide 12)

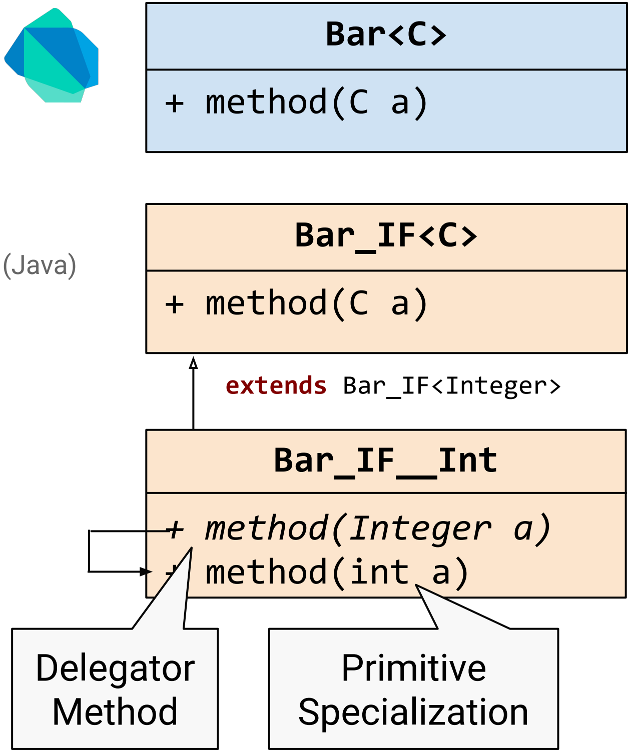
Example: Class Diagram (Dart → Java)



Dart:

```
class Foo<T> { }
class Bar extends Foo<int> { }
```

Specialization: Adding the Missing Overloadings



- **Goal:** Avoid boxing of primitive types
- **Bonus:** Get rid of some type checks
- Create copies of generic classes with 1-2 type parameters (like C++ templates)
- Specialize for `bool, double, int`
- Invoke methods through specialized “unboxed” interface

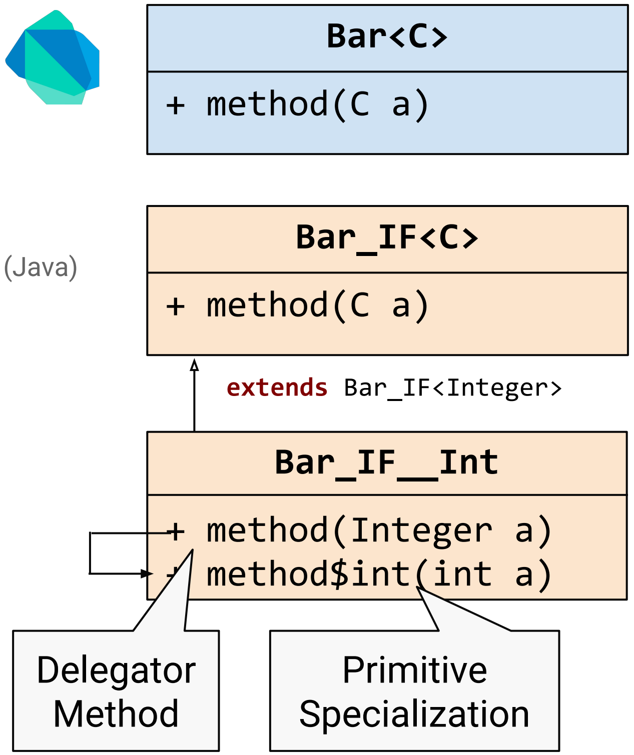
```
Bar<int> object;
object.method(123);
⇒    object.method(123);
```

```
Bar<Object> object;
object.method(123);
⇒    object.method(123);
```

Specialization: Name Mangling

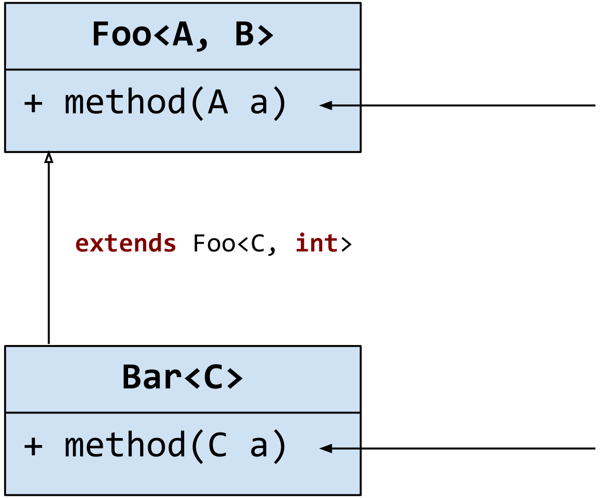
this is where
things went a
bit wrong

- **Goal:** Avoid boxing of prim
- **Bonus:** Get rid of some typ
- Create copies of generic classes th 1-2 type parameters (like C++ templates)
- Specialize for `bool, double, int`
- Invoke methods through specialized “unboxed” interface
- Encode generic parameter binding in method name



```
Bar<int> object;
object.method(123);
⇒    object.method$int(123);                              ✓

Bar<Object> object = new Bar<int>();
object.method(123);
⇒    object.method(new Integer(123));                     ✓
```

“Encoding Generic Types” is not Enough



For specialization Bar<int>:

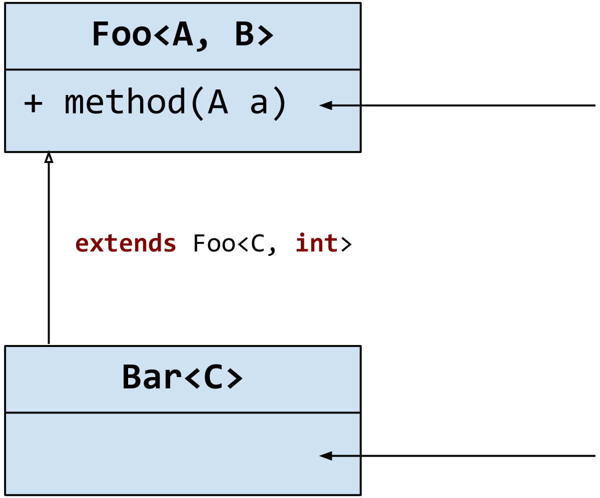
```
method$int_int
```

```
method$int
```

Obvious problems:

- Method overriding is broken

“Encoding Generic Types” is not Enough



For specialization `Bar<int>`:

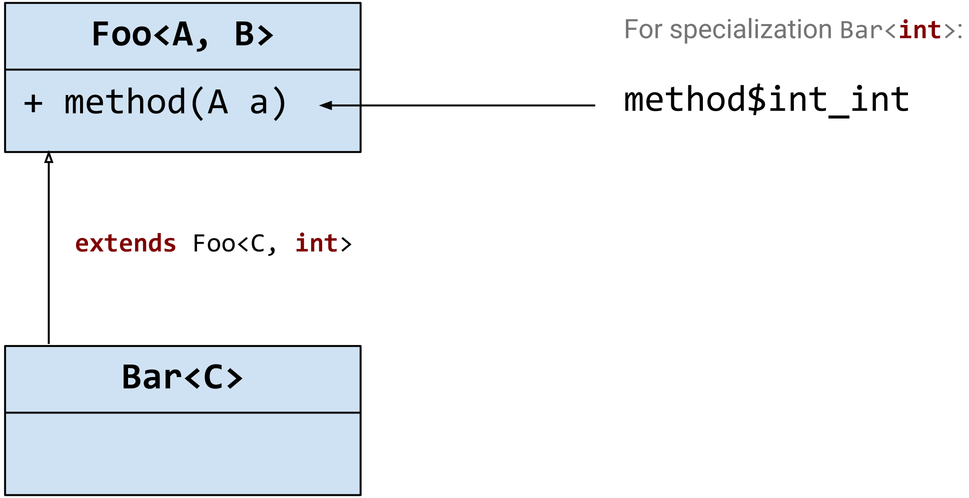
`method$int_int`

Calling `method$int` fails!

Obvious problems:

- Method overriding is broken
- If method is not overridden:
  Calling a method that does not exist

“Encoding Generic Types” is not Enough



Foo<A, B>

+ method(A a)

extends Foo<C, int>

Bar<C>

For specialization Bar<int>:

method$int_int

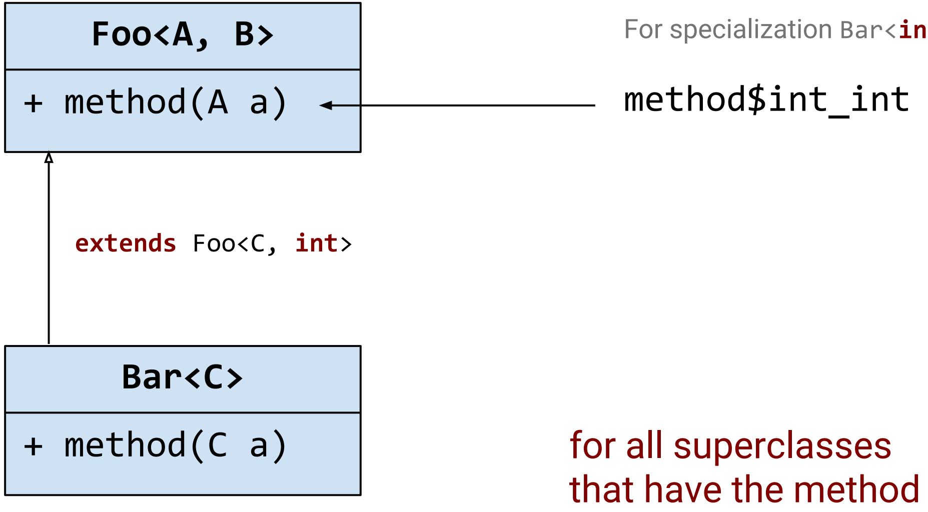
Obvious problems:

- Method overriding is broken
- If method is not overridden: Calling a method that does not exist

More serious problem:

- Foo and Bar have different type parameters
- Just because C is bool, it does not mean A or B are also bool

“Encoding Generic Types” is not Enough



For specialization Bar<**int**>:

`method$int_int`

for all superclasses that have the method

Obvious problems:

- Method overriding is broken
- If method is not overridden: Calling a method that does not exist

More serious problem:

- Foo and `Bar` have different type parameters
- Just because `C` is `bool`, it does not mean `A` or `B` are also bool

Solution:

- Make class name of type parameters part of the mangled method name:
  `method$Foo_int_int`
  `method$Bar_int`

Call Patterns involving Supertypes

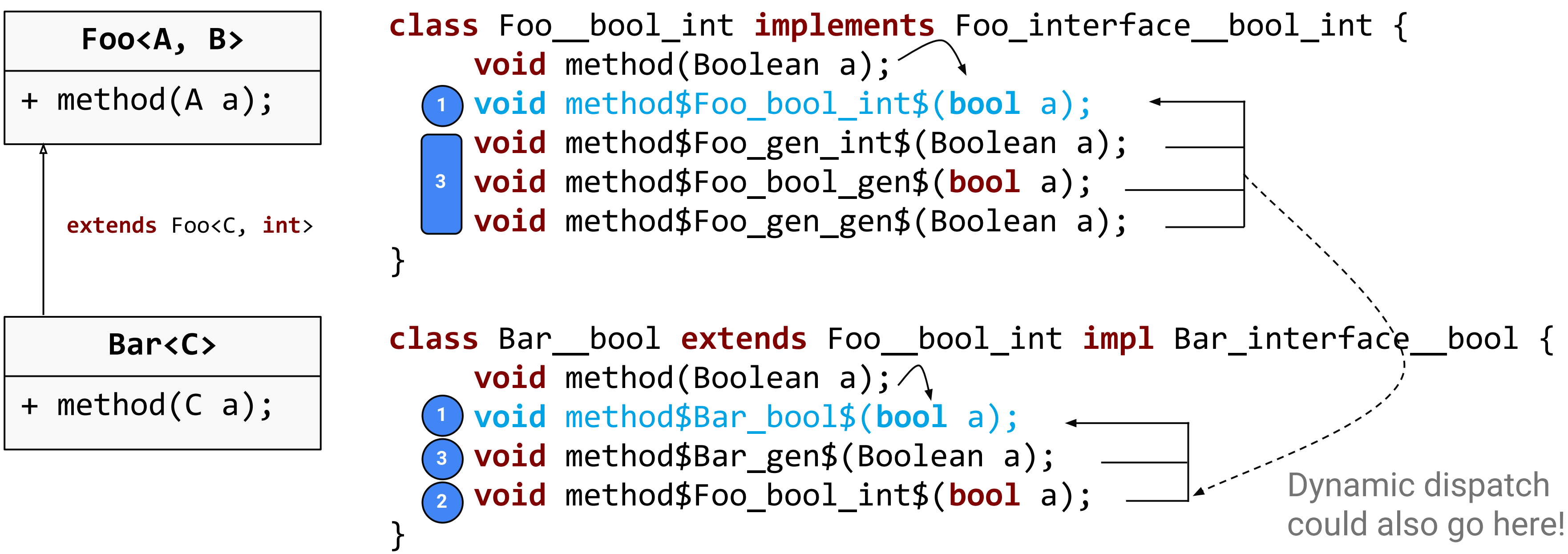
- Exact class and exact specialization 1
  `List<int> myList = new List<int>();`
- Superclass and (its) exact specialization 2 (*)
  `Iterable<int> myList = new List<int>();`
- Exact class and “super” specialization 3
  `List<Object> myList = new List<int>();`
- Superclass and “super” specialization 4 (*)
  `Iterable<Object> myList = new List<int>();`

```
myList.isNotEmpty;          ⇒ myList.getIsNotEmpty$Iterable_int
```

Encode in method name:

- Specialization (binding of type variables) of receiver
- Static type of receiver (to which the type variables belong)
  → required due to dynamic dispatch in (*)

Delegator Methods for Specializations



```
class Foo__bool_int implements Foo_interface__bool_int {
    void method(Boolean a);
    void method$Foo_bool_int$(bool a);
    void method$Foo_gen_int$(Boolean a);
    void method$Foo_bool_gen$(bool a);
    void method$Foo_gen_gen$(Boolean a);
}

class Bar__bool extends Foo__bool_int impl Bar_interface__bool {
    void method(Boolean a);
    void method$Bar_bool$(bool a);
    void method$Bar_gen$(Boolean a);
    void method$Foo_bool_int$(bool a);
}
```

Dynamic dispatch could also go here!

Optimization: No delegators are needed for subclasses: Determine call target statically and invoke method that is known to be defined.
This slide is simplified: Some delegator methods are default interface methods.

Future Work: Change Mangling Scheme

- `dart2java` currently mangles according to static type of receiver

  ```
  List<int> list; list.add(10);
  ⇒ List_IF__int list; list.add$List_int(10);
  ```
- Why not mangle according to parameter types?
- Java overloads could take care of that: Java compiler does the mangling (except for return type).
- Consequences
  - **No “super class/type” delegator methods**
  - All delegator methods (and the implementation method) have the same name
  - Generate a delegator method involving a specialization for a type variable `T` only if the signature of the method actually uses `T`

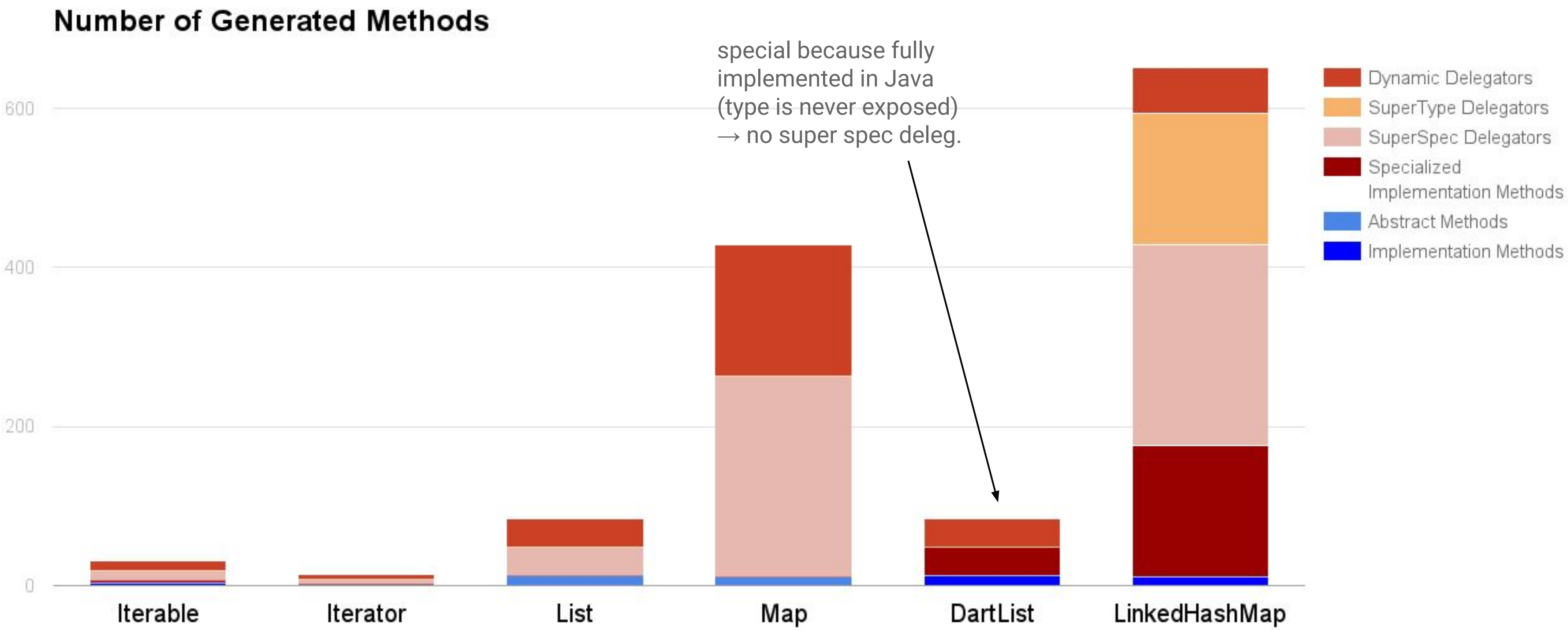
Specialization: Code Size Increase

- 1 Generic Parameter: 3 extra classes, 1 extra delegator method due to “super specialization”
- 2 Generic Parameters: 15 extra classes, 8 x 2 and 7 x 1 extra delegator methods due to “super specialization”
- Additional delegator methods due to “super class”:
  For every overriding method m: number of superclasses (+impl. interfaces) that also define a method m

Example: `LinkedHashMap<K, V> implements Map<K, V>`

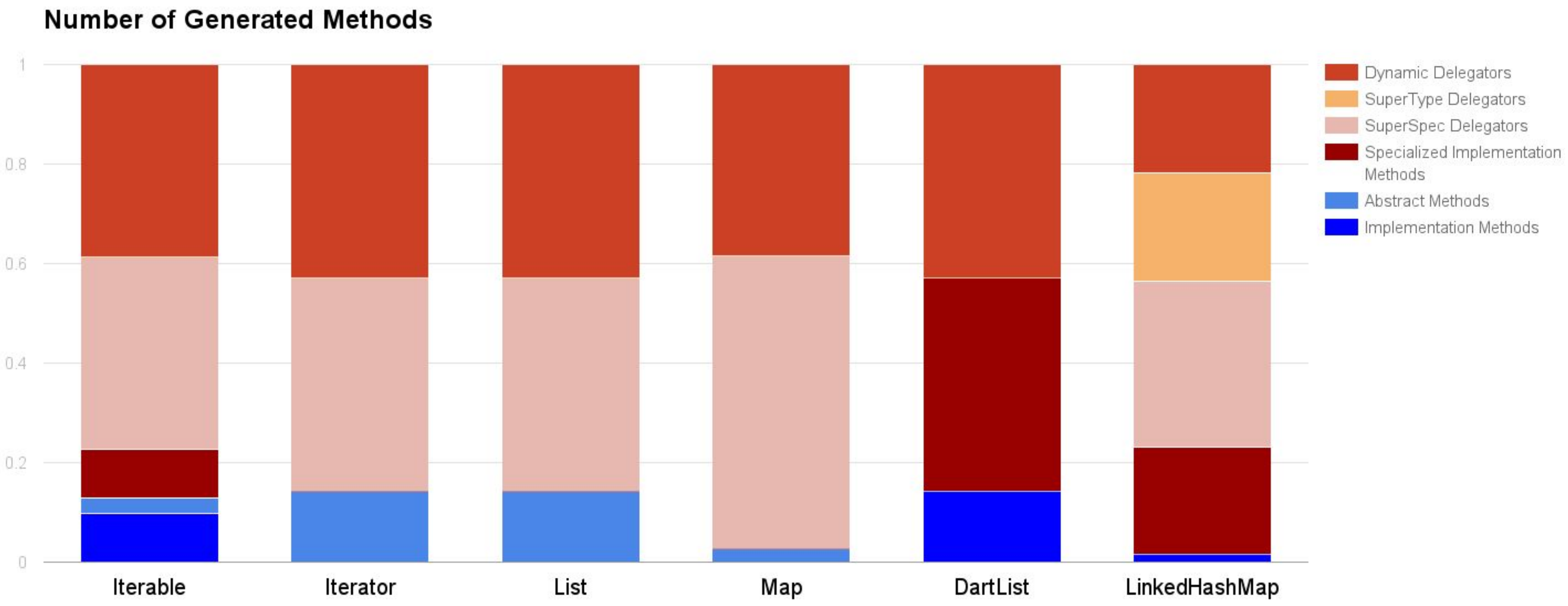
- 11 methods
- (8 * 2 + 7 * 1) * 11 = 253 delegator methods due to super specialization
- 15 * 11 = 165 delegator methods due to super class/implemented interfaces

Specialization: Code Size Increase

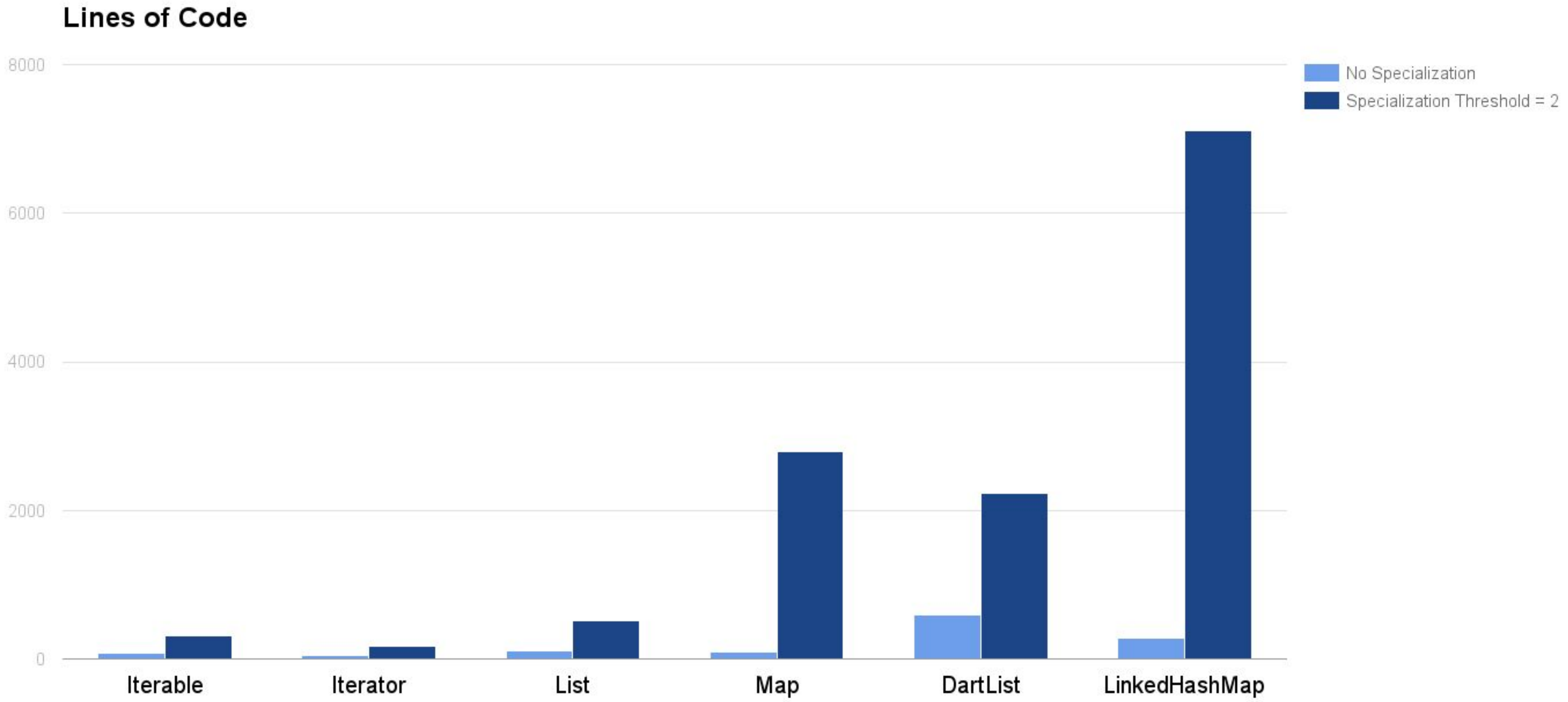


special because fully implemented in Java (type is never exposed) → no super spec deleg.

Specialization: Code Size Increase



Specialization: Code Size Increase



Summary

- *Question:* Is Dart suitable for execution on the JVM?
    - Many similarities between Java and Dart
    - Dart is very “static”, even more with *Strong Mode*:
      few dynamic invocations, fixed class hierarchy at runtime, no on-the-fly class definition
- *Question:* Is Dart suitable for an AOT optimization scheme?
    - Yes, if your device has enough memory
    - C++ approach might be better
      Generate specialized version upon first usage. However, user of library need access to its source code.
- Dart Infrastructure
    - Kernel AST 👍, even better with the latest version!

A Appendix

At Office...

In Seattle...

CENTURYLINK FIELD
BUD LIGHT
verizon
KICK OFF REDBOX MOVIE NIGHT!
HAWKS NEST
THOMAS III
29
SHERMAN
25
FAN
12

Constructors and Factory Constructors

```
class Foo {
   Foo.c1(int a) {
      // Like an instance method
   }

   factory Foo.c2(var b) {
      if (b) {
         return new SubFoo();
      } else {
         return new Foo.c1(42);
      }
   }
}

abstract class List {
   external factory List([int length]);
}
```

- Constructor: Returns new instance of specified class
- Factory Constructor: Returns instance of specific class or instance of subclass of specified class
  → Similar to a static method, but can be used with `new`
- (Factory) constructors can be named

Setting (Environment)

- Run on my workstation (Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz)
- 4 Configurations
  - ○ **dart2java with generic specializations**
  - ○ dart2java without generic specializations
  - ○ Dart VM checked (1.18.0-dev.2.0)
  - ○ **Dart VM unchecked (1.18.0-dev.2.0)**
- 1 second warmup, 10 seconds running
  (1 min. warumup results in minor speedup for `dart2java`)

} Analyzer *Strong Mode*

Unchecked Mode:

```
int foo = “Hello World”;
```

Types are “comments”

Checked Mode:

Rule of thumb: Type checks if there’s a case in which it would run

```
class A {
    int foo() { return 123; }
}

class B extends A {
   @Override Object foo() { return “Hello World”; }
}

A a = new B(); a.foo() + 10;
```

Strong Mode:

(strong type guarantees)

Do more checks at compile time, good for AOT compilation