dart2java: Running Dart in Java-based Environments

ICOOOLPS 2017

Matthias Springer*, Andrew Krieger+, Stanislav Manilov#, Hidehiko Masuhara*

* Tokyo Institute of Technology + UC Los Angeles # University of Edinburgh

Thanks to Vijay Menon, Jennifer Messerly and Leaf Peterson

Overview

1. **Introduction**
2. Dart Language Features and Implementation
3. Compilation Process
4. Generics
5. Language Interoperability
6. Conclusion

Introduction

- Motivation:
  - Migration path from Java env. to Dart env.
  - Investigate if Dart is suitable for execution on JVM
  - Support Dart on many platforms (where JVM runs)
- What is dart2java?
  - Analyzer/Kernel frontend for static type checking
  - Compiler from Dart code to Java code
  - (Partial) Implementation of Dart SDK

Dart Type System

- Various type checking modes:
  Unchecked mode, checked mode, strong mode
- dart2java is based on strong mode
  - Many static type gurantees
  - Runtime type checks required for:
    - Type casts
    - Implicit downcasts
    - Generic assignments

Overview

1. Introduction
2. **Dart Language Features and Implementation**
3. Compilation Process
4. Generics
5. Language Interoperability
6. Conclusion

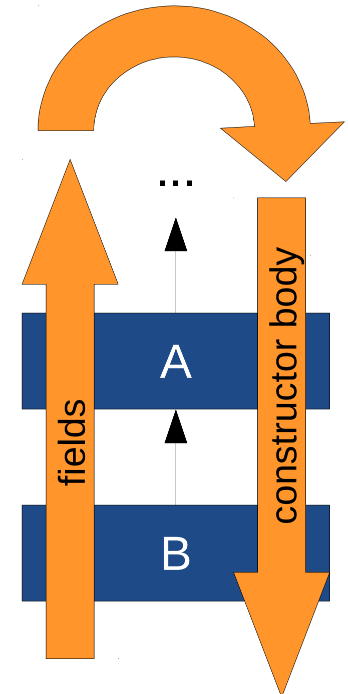
Dart Features

Constructor Semantics	Instance method for constructor body
Dynamic Type	Java Reflection/Method Handles API
Factory Constructors	Factory method is entry point for constructor
Getters / Setters	Java method prefixed with get / set
Generic Reification	First method/constr. arg.: class<C> object
Generic Covariance	Type safety ensured by runtime type system
Implicit Interfaces	Generate Java interface for Dart class
Keyword Parameters	Implicit Map object as last argument
Lambda Functions	Not supported yet
List / Map Literals	Special List / Map constructor with varargs
Mixins	Insert copy of mixin in hierarchy (fut. work)
noSuchMethod	Run handler if Java Reflection lookup fails
Operators	Ordinary Java method with name mangling
Optional Parameters	Automatically-generated method overloads
Synchronization	async / await are not supported yet
Top-level Members	Special __TopLevel class
Type Casts	Runtime type system check (if necessary) and Java type cast

Dart Features: Constructors

Constructor Semantics	Instance method for constructor body
Dynamic [illegible]e	Java Reflection/Method Handles API
Fact[illegible]uctors	Factory method is entry point for constructor
[illegible]	[illegible]fixed with get / set
[illegible]	[illegible]nstr. arg.: class<C> object

```
int method => 25;

class A {
  int a;
  A() : this.a = method() { ... }
}

class B extends A {
  int b;
  B() : this.b = method(), super() { ... }
}
```



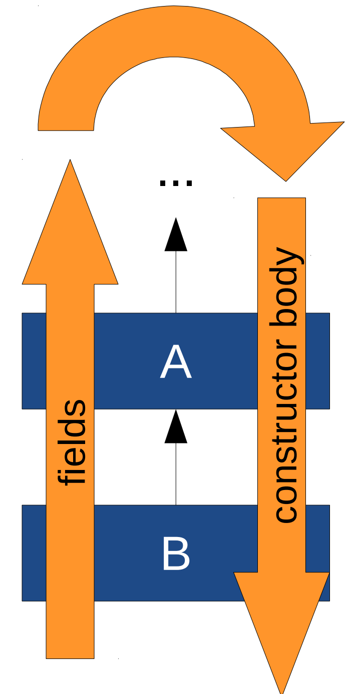

```
class __TopLevel { public static int method() { return 25; } }

class B {
  public static B_IF _new_() {
    B_IF instance = new B();
    instance._constructor();
    return instance;
  }

  void _constructor() {
    this.b = __TopLevel.method();
    super._constructor();
  }
}
```

Initialization order:
1) Fields of B
2) Fields of A
3) Constructor Body of A
4) Constructor Body of B

Dart Features: Constructors

Constructor Semantics	Instance method for constructor body
Dynamic ...e	Java Reflection/Method Handles API
Fact...uctors	Factory method is entry point for constructor
	...fixed with get / set
	...str. arg.: class<C> object

```
int method => 25;

class A {
  int a;
  A() : this.a = method() { ... }
}

class B extends A {
  int b;
  B() : this.b = method(), super() { ... }
}
```



Initialization order:
1) Fields of B
2) Fields of A
3) Constructor Body of A
4) Constructor Body of B

suffix for named constructor

```
class __TopLevel { public static int method() { return 25; } }

class B {
  public static B_IF _new_() {
    B_IF instance = new B();
    instance._constructor();
    return instance;
  }
  void _constructor() {
    this.b = __TopLevel.method();
    super._constructor();
  }
}
```

static method can be a factory constructor

Dart Features: Implicit Interfaces

東京工業大学 Tokyo Institute of Technology

Generic Reification	First method/constr. arg.: `class<C>` object
Generic Covariance	Type safety ensured by runtime type system
Implicit Interfaces	Generate Java interface for Dart class
Keywo… …ameters	Implicit `Map` object as last argument

```
class A {
  void method(int a) { ... }
}

class B extends C implements A {
  // Must provide method(int)
  void method(int a) { ... }
}

A variable = new B();
```

```
interface A_IF { ... }
class A implements A_IF { ... }

interface B_IF extends C_IF { ... }
class B extends C implements B_IF { ... }
```

use interface type in most cases

```
A_IF variable = B._new_();
```

Dart Features

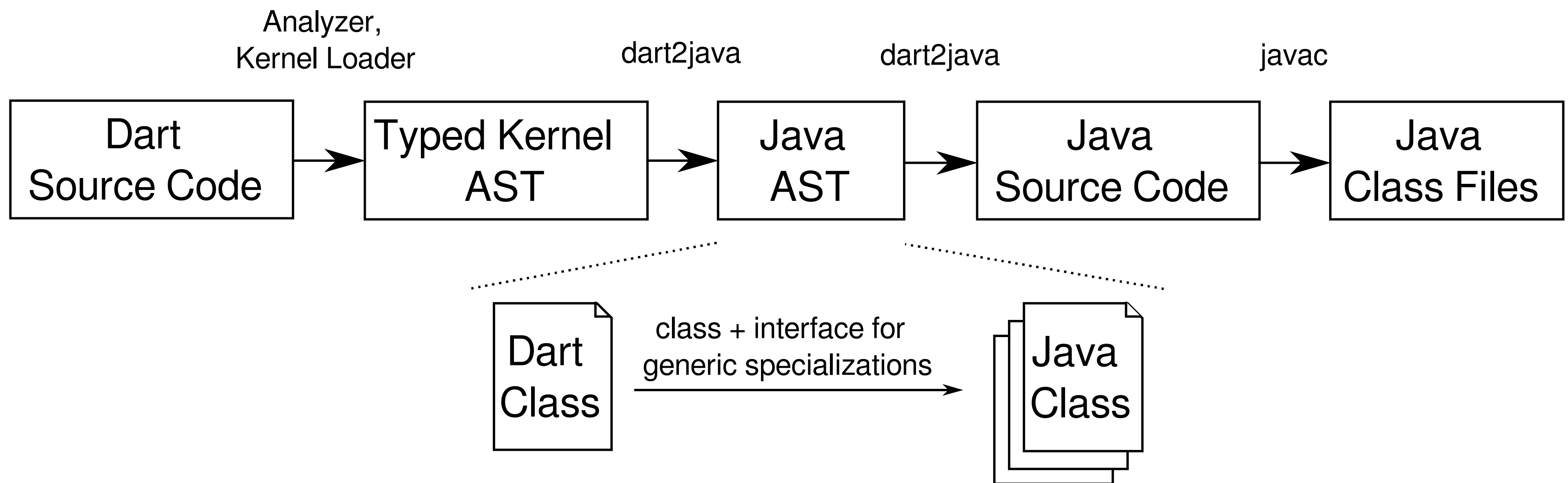
東京工業大学
Tokyo Institute of Technology

Constructor Semantics	Instance method for constructor body
Dynamic Type	Java Reflection/Method Handles API
Factory Constructors	Factory method is entry point for constructor
Getters / Setters	Java method prefixed with `get` / `set`
Generic Reification	First method/constr. arg.: `class<C>` object
Generic Covariance	Type safety ensured by runtime type system
Implicit Interfaces	Generate Java interface for Dart class
Keyword Parameters	Implicit `Map` object as last argument
Lambda Functions	Not supported yet
List / Map Literals	Special `List` / `Map` constructor with varargs
Mixins	Insert copy of mixin in hierarchy (fut. work)
`noSuchMethod`	Run handler if Java Reflection lookup fails
Operators	Ordinary Java method with name mangling
Optional Parameters	Automatically-generated method overloads
Synchronization	`async` / `await` are not supported yet
Top-level Members	Special `__TopLevel` class
Type Casts	Runtime type system check (if necessary) and Java type cast

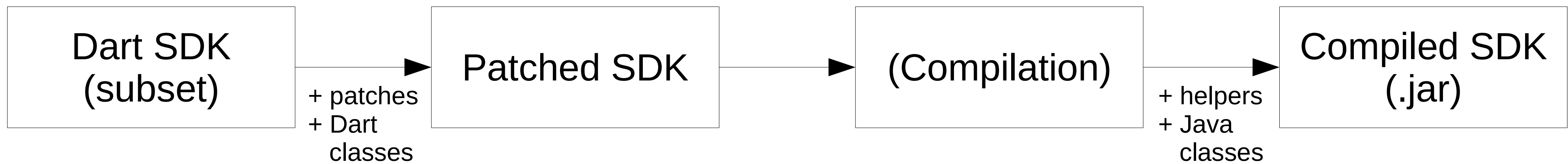
Overview

1. Introduction
2. Dart Language Features and Implementation
3. **Compilation Process**
4. Generics
5. Language Interoperability
6. Conclusion

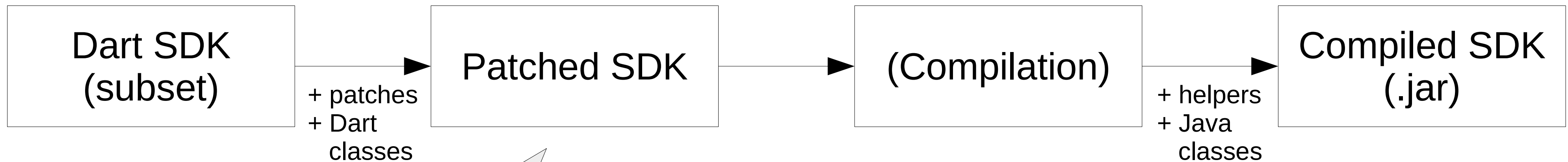
Compilation Process Overview



SDK Compilation



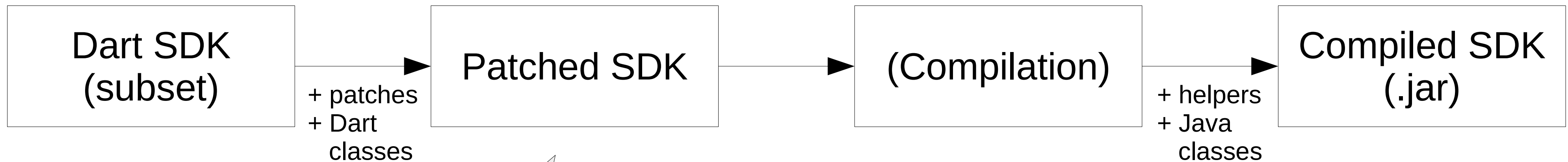
SDK Compilation: Patching




```
abstract class Map<K, V> {
  external factory Map();
  bool containsValue(Object value);
  bool containsKey(Object key);
  V operator [](Object key);
  void operator []=(K key, V value);
  V remove(Object key);
  void clear();
  Iterable<K> get keys;
  Iterable<V> get values;
  int get length;
  bool get isEmpty;
  bool get isNotEmpty;
}
```

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

SDK Compilation: Patching




```
abstract class List<E> implements ... {
  external factory List();
  E operator [](int index);
  void operator []=(int index, E value);
  int get length;
  /* … */
}
```

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

After patching: All *external* methods are gone or annotated with `@JavaCall`.

SDK Variation Points

- *External methods:* Must be patched or annotated with `@JavaCall`
- *Pure SDK Interfaces:* Implementation must be provided by execution environment
  - dart:core.bool → boolean
  - dart:core.double → double
  - dart:core:int → int
  - dart:core.Object → dart._runtime.base.DartObject

Design Decisions

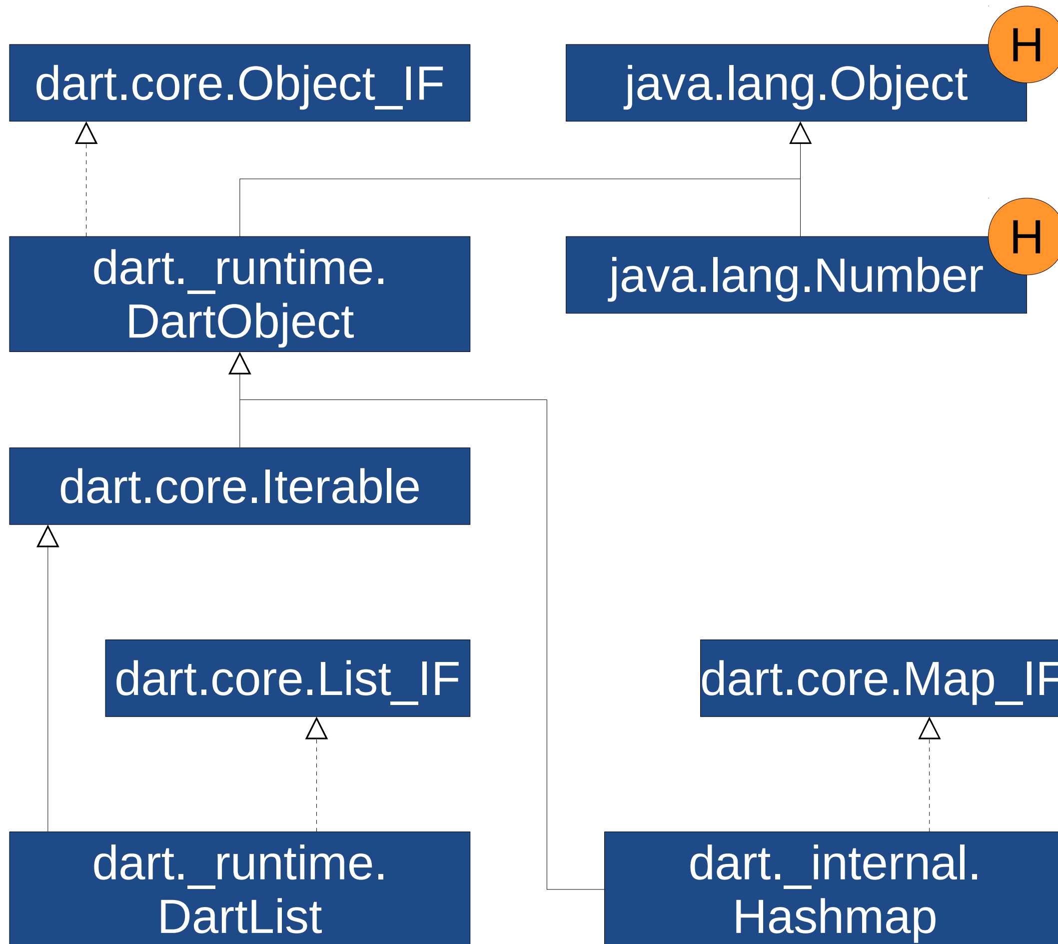
- Use only unboxed primitive types
- Do not allow assigning `null` to primitively-typed lvalues
- Generate specializations for generic classes where type parameter is a primitive type (later...)
- Reuse Java types as good possible (later...)

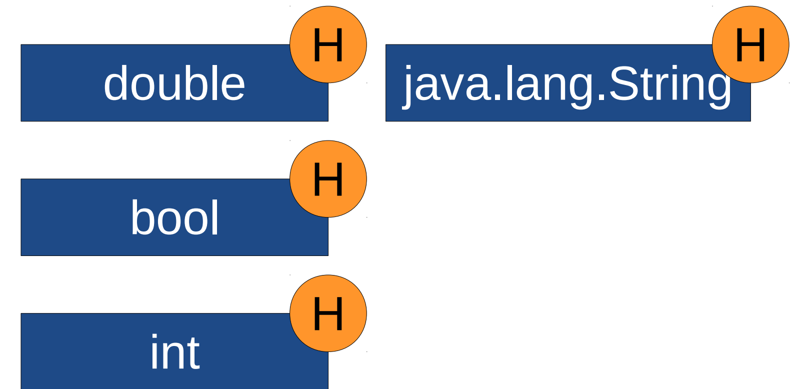
Design Decisions

- Use only unboxed primitive types
- Do not allow assigning `null` to primitively-typed lvalues
- Generate specia… classes where type parameter is a primitive type (later...)
- Reuse Java types as good possible (later...)

New semantics for `Map.[]`:
Throws exception if key not found

Object Model





H instance methods invoked through static helper class

e.g.: `15.gcd(3) => IntHelper.gcd(15, 3)`

Overview

1. Introduction
2. Dart Language Features and Implementation
3. Compilation Process
4. **Generics**
5. Language Interoperability
6. Conclusion

Reified Generics

東京工業大学 Tokyo Institute of Technology

- Objects know the binding of their type paramters at runtime (no type erasure)
- *Objects:* Store fully-reified type in `type` field
- *Generic Methods:* Pass fully-reified type arg.

```
class A<T> {
  factory A<S>() {
    if (S == int) {
      return new AInt.build();
    } else {
      return new A<S>.build();
    }
  }

  A.build() { ... }
}
```

not possible in Java due to type erasure

Covariant Generics

東京工業大学
Tokyo Institute of Technology

- Subtyping takes into account generic type arg.
- E.g.: List<String> is a subtype of List<Object>

```
List<int> i = new List<int>();
List<String> s = new List<String>();

List<Object> o = i;                              // Valid assignment
o.add(12);
o.add("A string");                               // Runtime exception

s = i as List<String>;                           // Static type error
s = o as List<String>;                           // Runtime exception
```

Insert runtime type check for every generic argument (use reified type information)

Insert runtime type check for assignments of generic objects

Generics: Code Example

```
class LinkedList<T> {
  Item<T> first;

  void add(T item) {
    if (first == null) {
      first = new Item<T>(item);
    } else {
      first.add(item);
    }
  }
}

class Item<T> {
  T value;
  Item<T> next;

  void add(T item) {
    if (next == null) {
      next = new Item<T>(item);
    } else {
      next.add(item);
    }
  }

  Item(this.value);
}
```

```
class LinkedList implements LinkedList_IF {
  Item first;
  Type type;

  void add(Object item) {
    type.typeParams[0].check(item);
    if (first == null) {
      first = Item._new_(
        Item.buildType$(type.typeParams[0]), item);
    } else {
      first.add(item);
    }
  }

  public static LinkedList_IF _new_(Type type) {
    LinkedList_IF result = new LinkedList_IF();
    result.type = type;
    result._constructor();
    return result;
  }
}

LinkedList_IF o = LinkedList._new_(
  LinkedList.buildType$(StringHelper.type));

LinkedList_IF i = LinkedList.buildType$(
  IntHelper.type).check(o);
```

```
LinkedList<Object> o = new LinkedList<String>();
LinkedList<int> i = o as LinkedList<int>;
```

Generic Specializations

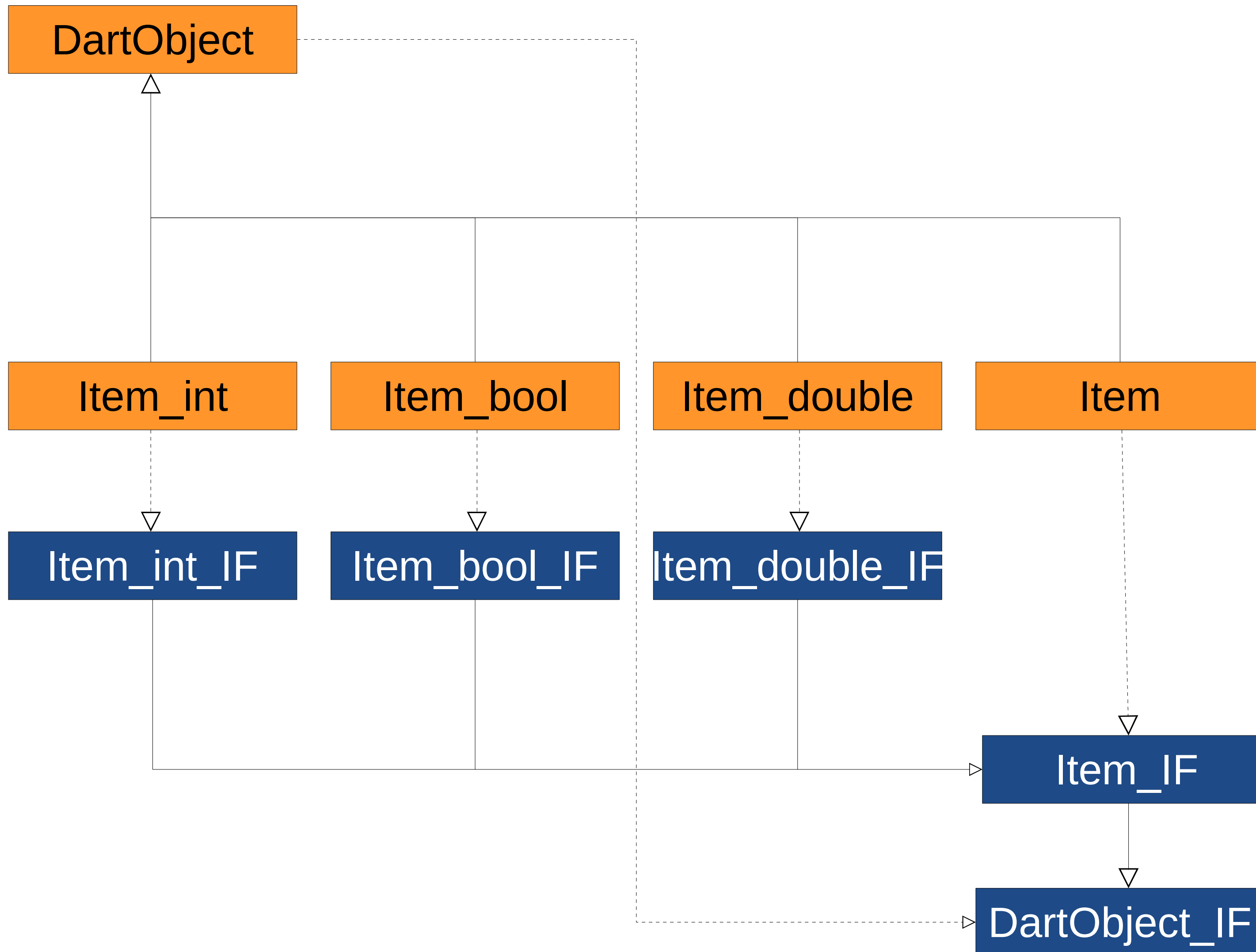
- Use only primitive types, even when used as generic type argument
- Generate special classes with int, bool, double instead of Object for type variables

```
interface Item_IF {
  void add(Object item);
}

class Item implements Item_IF {
  Object value;
  Item next;

  void add(Object item) { ... }
}
```

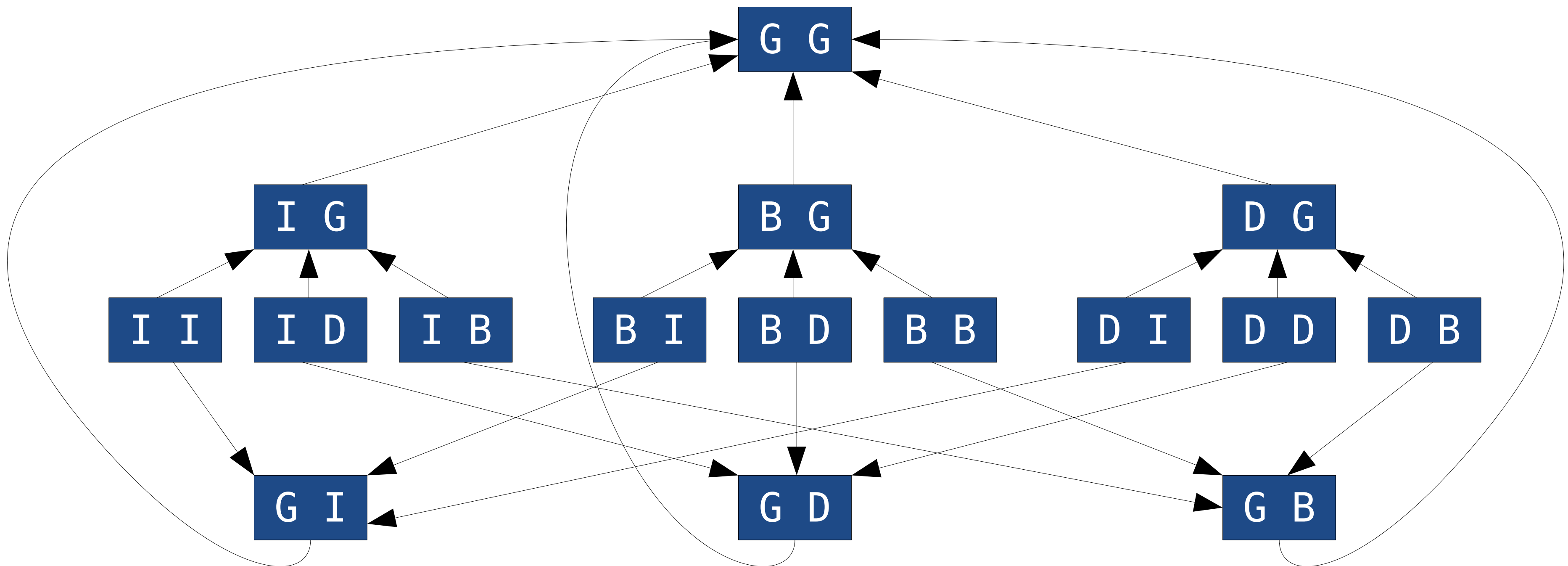
```
interface Item_int_IF extends Item_IF {
  void add(int item);
}

class Item_int implements Item_int_IF {
  int value;
  Item next;

  void add(int item) { ... }

  void add(Object item) {
    IntHelper.type.check(item);
    add((Integer) item);
  }
}
```

Delegator method

Generic Specializations

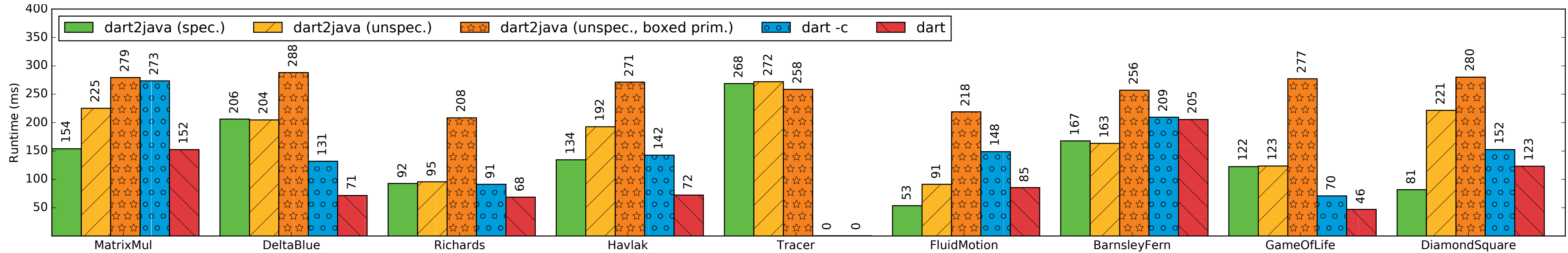


Generic Specializations



Benchmarks

東京工業大学
Tokyo Institute of Technology



MatrixMul: Implemented via List<List<int>>

DeltaBlue: few numeric computations, no primitive type arguments

Havlak: Uses Map<int, BasicBlock>

Tracer: Almost all method calls are on dynamic receivers

FluidMotion: Uses List<double>

BarnsleyFern: Highly numeric, only one method with a loop

DiamondSquare: Uses List<List<int>>

- Good performance for numerical code
- Generic specialization pays off
- Instance creation/runtime type system not fully optimized yet

Overview

1. Introduction
2. Dart Language Features and Implementation
3. Compilation Process
4. Generics
5. **Language Interoperability**
6. Conclusion

Using Dart Classes in Java

東京工業大学 Tokyo Institute of Technology

- Generated Java classes use Java generics
- Circumvent static Java type checking with unsafe type casts for covariance

```
class LinkedList<T> implements LinkedList_IF<T> {
  Item<T> first;
  Type type;

  void add(T item) {
    type.typeParams[0].check(item);
    ...
  }

  T getFirst() {
    return first.value;
  }
}
```

No type cast required when used from Java

```
// Dart:
LinkedList<Object> o;
LinkedList<String> s;
o = s;

// Java:
o = (LinkedList) s;
```

Using Java Classes in Dart

東京工業大学 Tokyo Institute of Technology

- Type information required for Analyzer frontend
- Provide adapter interfaces for Java classes
- Can be auto-generated for entire JARs

```
library adapter.java.util;

class ArrayList<E> implements dart.core.List<E> {
  bool add(E e);
  void add(int index, E element);
  /* ... */

  // Constructor
  @JavaCall("adapter.java.util.ArrayListHelper.instantiate")
  external factory ArrayList();

  // Dart List methods
  @JavaCall("adapter.java.util.ListHelper.operatorAt")
  external E operator [](int index);
}
```

```
class ArrayListHelper {
  static Object operatorAt(
    ArrayList self, int index) {
    return self.get(index);
  }
}
```

Java ArrayList should be useable like a Dart List

Type Safety

- Covariance breaks type safety for Java classes

```
import “adapter:java.util”

LinkedList<String> s = new LinkedList<String>();
LinkedList<Object> o = s;
LinkedList<int> i = o as LinkedList<int>;         // cannot be type checked

i.add(18);               // no exception
String str = s.get(0);   // cryptic runtime exception
```

Using Dart Classes in Java

- Special notation required for instantiation, getters / setters, generic classes / methods
- Generated Java interfaces should extend corresponding Java SDK interfaces and provide adapter methods (default interface methods)

```
package dart.core;

interface List_IF<E> extends java.util.List<E> {
  void add(E item);

  // Adapter methods
  public default int size() { return this.getLength(); }
}
```

Dart List should be useable like a Java List

Overview

1. Introduction
2. Dart Language Features and Implementation
3. Compilation Process
4. Generics
5. Language Interoperability
6. **Conclusion**

Future Work

- Anonymous functions / function types
- Full support for mixins
- Subclassing Java classes in Dart
- How to support assigning `null` to primitively-typed variables?

Conclusion

- Dart is similar to Java and an interesting alternative for Java programmers
- Suitable for execution the JVM (performant)
- Calling Java code from Dart is easy (for the programmer), the other direction not so much