dart2java: Running Dart in Java-based Environments

Matthias Springer
Tokyo Institute of Technology
matthias.springer@acm.org

Andrew Krieger
University of California, Los Angeles
akrieger@math.ucla.edu

Stanislav Manilov
University of Edinburgh
s.z.manilov@sms.ed.ac.uk

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

Abstract

We present the design and implementation of *dart2java*, an experimental Dart to Java compiler. It is implemented in Dart and currently supports many but not all Dart language constructs. dart2java is a playground to evaluate performance implications of running Dart code on the JVM and to investigate if it is possible to write Dart code in a largely Java-dominated environment.

This paper describes the architecture of dart2java, performance optimizations such as non-nullability of primitive types and generic specialization (and their implications), as well as ideas for language interoperability, i.e., calling Java code from Dart and vice versa.

Keywords Dart, Java, compiler, source code generation

ACM Reference format:
Matthias Springer, Andrew Krieger, Stanislav Manilov, and Hidehiko Masuhara. 2017. dart2java: Running Dart in Java-based Environments. In *Proceedings of ICOOOLPS'17, Barcelona , Spain, June 19, 2017,* 6 pages.
https://doi.org/10.1145/3098572.3098575

1 Introduction

Dart is an object-oriented, class-based programming language and was originally designed as a replacement for JavaScript as the scripting language of the web. Nowadays, Dart can be used for standalone applications running in the *Dart VM* and for web applications with Google's source-to-source compiler to JavaScript (*DDC*). This paper presents the design and implementation of dart2java, an experimental Dart to Java compiler, whose design is inspired by DDC.

Dart's syntax is similar to Java but it provides special notations for commonly used features such as list/map literals, getter/setter methods, or factory constructors. Therefore, we think that Dart is an interesting alternative for Java programmers. The motivation for dart2java is threefold: First, dart2java can provide a migration path from a mostly Java-dominated environment to a Dart environment, if Java code can be called from Dart (and vice versa). Second, dart2java is an experiment to see if Dart code is suitable for execution on the JVM. Finally, dart2java allows programmers to write Dart code for platforms where only a Java VM is available.

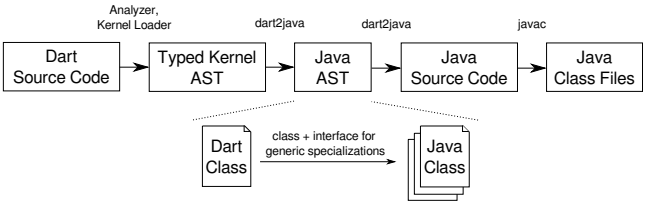

Figure 1. Dart Code Compilation Process

2 Architecture

Every Dart system has two components: A Dart implementation (compiler/interpreter; e.g., the Dart VM or dart2java) and the Dart SDK. The latter one contains Dart source code of core interfaces (such as `int` and `Object`) and standard library classes. It is meant to be shared by all Dart implementations and provides two kinds of variation points for Dart implementations. First, some methods are declared as external: It is up to the Dart implementation to provide an implementation for the method. And second, certain core classes are pure interfaces (all methods are abstract) and it is up to the Dart implementation to provide an implementation class; only the public SDK interface type will be exposed to programmers, not the implementation class type.

Compilation Process dart2java uses *Analyzer* and *Kernel*[1] to generate a typed abstract syntax tree of every Dart class (Figure 1). This Dart AST is then transformed into a Java AST, generating one Java interface and one Java class for every Dart class, along with generic specializations (Section 3.5). The Java interface is necessary because every Dart class implicitly defines an interface that can be implemented by any class. The resulting Java source code can be compiled and run with a copy of the compiled Dart SDK in the classpath.

[1] A front-end and an intermediate tree representation for Dart.

ICOOOLPS'17, June 19, 2017, Barcelona , Spain

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

Figure 2. Translation of Dart Language Features to Java

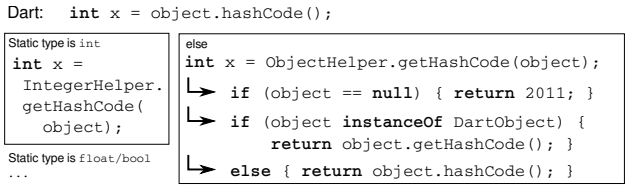

Figure 3. Method call with name defined in Dart `Object`

The compilation process for the Dart SDK is mostly identical but starts with *patching*: Some external methods are replaced with pure Dart implementations. For example, the external factory constructor for `Map` is replaced with a concrete one, creating an instance of `LinkedHashMap`, a class that is implemented in Dart and shipped together with dart2java. Patching allows us to implement external methods in Dart without having to change the Dart SDK itself.

3 Code Generation

This section describes the Java code generation process (see Figure 2 for an overview). The Dart source code is already parsed and types are fully inferred at this point of time.

3.1 Datatypes

There are a number of type checking modes in Dart. In *unchecked mode*, all type names are treated as mere comments and actual types are inferred. dart2java is based on *strong mode* which provides many type guarantees at compile time. Types can either be specified by the programmer or are inferred. Runtime type checking is required only for type casts, implicit downcasts and when assigning (or passing as an argument) an object whose type is generic.

Java provides boxed and unboxed versions for the primitive types *boolean*, *float*, *double*, and *integer*. dart2java always uses unboxed types, because they are significantly faster in numeric applications[2]. Method calls on such objects are translated to invocations of static methods of *helper classes* (e.g., `IntegerHelper.gcd` for Dart `int.gcd`). Calls to methods defined in Dart `Object` are dispatched to a primitive helper or to `ObjectHelper`, which can handle Dart objects, `null`, and ordinary Java objects outside of Dart's class hierarchy (Figure 3). The generated Java code never uses boxed types except for the following cases: (a) Assignment (including passing arguments during method calls) to a variable (parameter) of type `Object` or `num`, and (b) Generic classes with more than two type parameters (Section 3.5).

Every Dart class implicitly defines an interface that can be implemented by any other class. dart2java creates a class and an interface (ending with `_IF`) for every Dart class and always uses the interface type during code generation except for instantiation of classes and invocation of static methods.

Class Hierarchy Dart supports single inheritance, interfaces, and mixins. If no explicit superclass is specified, a Dart class is a subclass of `Object` (a core interface defined in the SDK). This interface provides the usual methods (e.g., for equality checking) and is implemented by the hand-written Java class `DartObject`, provided by dart2java. Every generated Java class implements exactly one interface: the interface that is generated for that class. If the corresponding Dart class implements other interfaces, the generated Java interface extends those interfaces. Mixins are future work.

Constructors Dart constructors are more powerful than Java constructors: There are ordinary constructors and *factory constructors*, which are also invoked with the `new` keyword but are actually static methods (and can return existing objects) with a return type that can be any subtype of the class. Constructors can have names and classes can have multiple constructors. In addition, there is special syntax for field initialization (similar to C++), and a `super` constructor can be invoked at any point of time. For all constructors, the entry point in the generated Java code is a static method and the constructor body is in an instance method because Java constructors allow `super` calls only as the first statement.

3.2 Non-nullability of Primitive Types

Due to using exclusively unboxed types, `null` can no longer be assigned to lvalues of primitive type. dart2java ships with a modified version of *Analyzer* (the component performing type inference) to ensure that such variables are explicitly initialized with a non-null value in the Dart code. Casting a null value at runtime results in a null pointer exception. Moreover, dart2java's implementation of `Map` throws an exception when looking up a non-existent key, instead of returning `null` (a change of language semantics).

[2]Integers bigger than `Integer.MAX_VALUE` are not supported.

3.3 Reified Generics

In contrast to Java generics [2], Dart generics are reified, i.e., an object knows the binding of its generic type variables at runtime (Java erases types). dart2java provides a runtime type system for checking types during assignments. Every Dart object has a field containing an object representing its (fully reified) type. Type checks are inserted when assigning/passing objects of generic type.

```
class A<T> {
  boolean isString() => T == String;
  void printMe(T item) { print(item); }
}
print(new A<String>().isString());
```

In the above example, the generated Java class will have a `type` variable containing the fully reified type of `A`, which will be used for type checking. Furthermore, `printMe` will check if `item` is actually a string.

```
static final Type STR = /* ... */
static final Type A_STR = /* ... */
class A<T> implements A_IF<T> {
  private Type type;
  public A(Type type) {
    this.type = type; this._constructor();
  }
  boolean isString() {
    return type.typeParams[0] == STR;
  }
  void printMe(T item) {
    TypeSystem.check(item, type.typeParams[0]);
    dart.core.__TopLevel.print(item);
  }
}
dart.core.__TopLevel.print(new A<String>(A_STR)↙
    ↳ .isString());
```

Type checks for arguments of generic type have to be inserted due to generic covariance, as shown in the following listing. The Java type system alone cannot detect the type violation at runtime, because type information is erased.

```
A<Object> a = new A<String>();
a.printMe(10);  // runtime exception
```

Generic methods are also reified: An additional type object per type parameter will be passed as an argument.

3.4 Covariant Generics

In contrast to Java generics, Dart generics are covariant. Intuitively, covariance [4] means that subtyping takes into account the binding of generic type variables and requires them to be in a subtyping relationship as well. For example, `List<int>` is a subtype of `List<Object>` in Dart. Generated Java classes do not have to be generic. For example, a generic Dart class `A<T>` could be translated to a non-generic Java class `A` where all occurrences of `T` are replaced with `Object`. However, generated Java classes are generic so that the return values of their methods are easy to use from Java code (without type casts). Type casts to raw types make generated Java code compliant with the Java type system while preserving covariance. E.g., the following code defines a list of `A` and casts it to a list of `Object`, then to a list of `String`. This should trigger an exception at runtime.

```
List<A> a = new List<A>();
List<Object> o = a;
// The following line causes a runtime error
List<String> s = o as List<String>;
// Analyzer reports a (compile-time) error
List<String> s = a as List<String>;
```

The generated Java code utilizes the runtime type system to perform type checks for downcasts.

```
List_IF<A_IF> a = new List<A_IF>();
List_IF<Object_IF> o = (List_IF<Object_IF>)(↙
    ↳ List_IF) a;
// The following line causes a runtime error
List_IF<String> o = (List_IF<String>) (List_IF)
    TypeSystem.check(a, LIST_STRING_TYPE);
```

3.5 Generic Specializations

Using only unboxed versions of primitive types requires additional work when combined with generics, because Java allows only their boxed counterparts when used as type arguments. dart2java could fall back to boxed types in such a case. However, since Dart does not have an array datatype and primitive types are widely used as generic types in many applications (typically with lists or maps), including in the benchmarks we used, dart2java generates specialized versions of generic classes for all primitive types.

For a generic class `A<T>`, dart2java generates four classes and interfaces. One class and interface is generated for each of *bool* (`A__bool`, `A__bool_IF`), *double* and *int*. The fourth one (`A<T>`, `A_IF<T>`) is a generic class/interface for all other types.

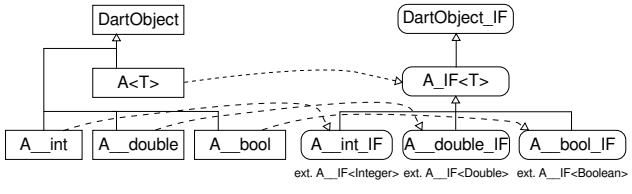

Figure 4. Generated Java Classes/Interfaces for `A<T>`

Figure 4 illustrates the subclass relationship between those classes and interfaces. All generated classes are subclasses of `DartObject` (generated Java class for the Dart class `Object`) if no other superclass was specified. However, the

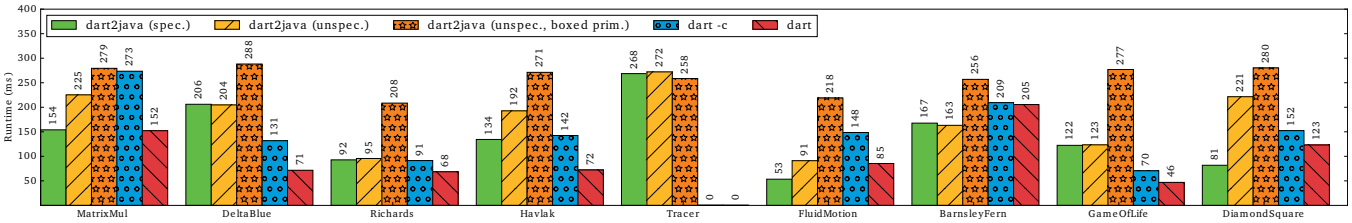

Figure 5. Benchmarks comparing dart2java (with/without specializations, with only boxed primitives) and Dart VM (checked/unchecked). All benchmarks ran repeatedly for 10 seconds after 7.5 seconds of warmup. Numbers are average running times for a single run.

specialized interfaces are subinterfaces of `A_IF<T>`, where `T` is bound to the corresponding boxed type. This is necessary because of covariance (e.g., `A<int>` is a subtype of `A<Object>` in Dart). Inside a specialized class, all occurrences of `T` are replaced with the unboxed specialized type.

For classes with two generic type variables, all combinations of specializations are generated (16 classes and interfaces). No specializations are generated for classes with more than two type variables because of the combinatorial explosion of classes, but annotations could be used to specify which types to specialize for [5].

dart2java's implementation class for the SDK interface `List<T>` is special: All of its specializations are hand-written and backed by a Java array. The implementation of `Map<K, V>` is written in Dart and backed by two Dart lists.

3.6 Delegator Methods

Unboxed primitive types are not subtypes of `Object` in Java. Therefore, overriding a method that consumes primitive types with a method consuming non-primitive types (or vice versa) requires additional effort. The problem is shown in its most basic form in the following Dart source code snippet.

```
class A { void f(int a); }
class B extends A { void f(Object a); }
(new B()).f(10);
```

The generated Java source code would be mostly identical. However, `B.f` does not override `A.f`; instead, it defines a method overload. The method call invokes the method defined in `A`. To solve this problem, dart2java adds *delegator methods* for every method that consumes a primitive type but has a subtype where the method is overridden with one consuming a non-primitive type. Delegator methods perform a type cast and call the actual method:

```
void B.f(int a) { f((Integer) a); }
```

Delegator methods are also required for generic specializations. For example, for the method `List<T>.add(T obj)`, dart2java generates a delegator method `List__int.add(Integer obj)` (delegating to `add(int obj)`).

The rule for generating delegator methods is as follows. When translating `A.method`, let S be the set of all supertypes of `A`, i.e., super classes, interfaces, super interfaces and/or super types for generic type parameter bindings[3]. E.g., for `List<int>`, S = {`List<Object>`, `Object`}. For every $s \in S$, if `method` is defined in s and `A.method` is not a valid Java override of s`.method`, generate a delegator method in `A` with the signature of s`.method`[4]. A method is not a valid Java override if the types of the parameters are not identical/super types.

3.7 Benchmarks

Figure 5 shows the runtime of 9 benchmarks, comparing dart2java with the Dart VM. We ran the Dart VM in both checked mode and unchecked mode (the latter treats types as mere comments). dart2java is based on strong mode with additional compile-time type guarantees which is not available in the Dart VM yet, allowing us to avoid some runtime type checks (see Dart Language Guide). The Dart VM is the standard VM for Dart; ideally, dart2java should deliver performance comparable to the Dart VM. We ran all benchmarks on a system with an Intel Core i7-6820HQ CPU, 32 GB RAM, Ubuntu 16.04.1, OpenJDK 1.8.0_111 and Dart VM 1.22.1.

Five benchmarks are taken from the "ton80" benchmark suite. Havlak uses `Map<int, BasicBlock>`, so the specialized version is faster here. The speedup is even better in FluidMotion, MatrixMul, DiamondSquare, which use `L<double>`, `L<L<int>>`, `L<L<int>>`, respectively (`L` means `List`). These three benchmarks along with BarnsleyFern (a highly numeric benchmark) are an indicator that the generated Java code with unboxed primitive types can be executed faster on the JVM than on the Dart VM. However, some other benchmarks are slower in dart2java, mostly due to the overhead of instance creation and our runtime type system, which is not optimized for performance yet. In Tracer, all variables have type *dynamic*. dart2java generates code that uses the Java Reflection API and the Java Method Handles API. It does not cache method lookup results or perform any other optimizations [9]. The Dart VM can execute the same workload in less than one millisecond. Overall, the performance of dart2java is comparable to the Dart VM, even though the runtime type system is still unoptimized.

[3] For `Map<int, int>`, this includes `Map<Object, int>`, `Map<int, Object>` and `Map<Object, Object>`.

[4] Our current implementation uses an unnecessarily more complex approach with name mangling, resulting in more delegator methods. However, the number of type casts and runtime method calls is identical.

4 Language Interoperability

This section is mostly about future work. Some parts of our current implementation were designed with interoperability in mind, but more work has to be done.

Reusing Java Classes One main design decision is to reuse Java types as much as possible (Figure 6). For example, Dart `int` is mapped to Java `int`. To make Dart objects easier to use from Java code, Java translations of Dart SDK core interfaces should extend their Java counterparts. E.g., `dart.core.List_IF` should extend `java.util.List`. Methods should be defined in `List_IF` as default interface methods in terms of Dart methods, if they are not already provided by the Dart interface or have a different signature. If methods are incompatible (different semantics or return type) between Dart or Java, Dart methods can be name mangled (and mangled method names must be used whenever Dart code is translated). It seems that both languages are mostly compatible; the only conflict that we found was `List.add` which returns a boolean in Java but `void` in Dart. In our current implementation, only the handwritten Dart `List` implementation implements the Java `List` interface.

Using Java Classes in Dart There are three challenges in making Java classes/interfaces and libraries usable from Dart code. First, *Analyzer* requires some information about Java classes for type checking. For that purpose, Dart interfaces (*adapters* [7]) can be autogenerated for all Java classes of a library. For code generation, these interfaces must be mapped to the corresponding Java classes. E.g., references to the autogenerated interface `adapter:util.List` are translated to `java.util.List` in generated Java code. No Java interfaces will be generated for such adapters; they are only required for Analyzer. Second, certain Dart methods should be available on Java objects. If a Java interface has a corresponding Dart interface, then the adapter should extend it. E.g., `adapter:util.List` should extend `dart:core.List`. Missing or incompatible (different signature or semantics) methods should be dispatched to a handwritten helper class (e.g., `ListHelper`), which implements them in terms of existing Java methods. Third, the inheritance hierarchy of generated adapters should reflect the hierarchy of their respective Java classes. E.g., the adapter for `ArrayList` should extend the adapter for `List`.

dart:core.bool	boolean
dart:core.Comparable<T>	▷ java.lang.Comparable<T>
dart:core.double	double
dart:core.int	int
dart:core.Iterable<T>	▷ java.lang.Iterable<T>
dart:core.Iterator<T>	▷ java.lang.Iterator<T>
dart:core.List<T>	▷ java.util.List<T>
dart:core.Map<K, V>	▷ java.util.Map<K, V>
dart:core.Object	dart._runtime.base.DartObject ▷ java.lang.Object
dart:core.Set<T>	▷ java.util.Set<T>
dart:core.String	java.lang.String
dart:math.Random	dart._runtime.base.DartRandom
dynamic	java.lang.Object
adapter:util.List<T> ▷ dart:core.List<T>	java.util.List<T>
adapter:util.ArrayList<T> ▷ adapter:util.List<T>	java.util.ArrayList<T>

Figure 6. Partial Overview: Mapping between Dart Types and Java Types. ▷ denotes class/interface extension. If no name without ▷ is specified on the right side, a generated Java class/interface is used.

Generic Classes Generic classes in Java are not reified. Therefore, covariant downcasts on imported generic Java classes cannot be done in a type-safe way in Dart. This should trigger a warning or could possibly be handled with wrappers. If a Dart class is used from Java, the reified type must be the first argument to the constructor. Since generated Java classes also use Java generics, method return values of generic type have proper types (and not just `Object`).

5 Related Work

There are two main techniques for language interoperability: source-to-source compilation (e.g., dart2java, DDC, TypeScript) and multi-language virtual machines [3, 12]. If the languages are similar enough (like in dart2java), the methods and objects from the other language can directly be used without naming or semantics conflicts. Otherwise, proxy/adapter objects/interfaces [6], name mangling or different method calling notations [11] must be used.

dart2java generates all specialized versions for classes with up to two type parameters. Alternatively, specializations could be generated when used [8], possibly using JIT compilation, or specialization could be guided by annotations [5]. In general, code with reified generics can be executed more efficiently than code with unreified generics, because more information is known at runtime. For example, `L<String>[i]` is guaranteed to always return a string. In Java, a list of integers can masquerade as a list of strings using unsafe type casts. Therefore, Java has to add additional type casts [1, 10]. Because dart2java emits Java code (not bytecode), the resulting Java bytecode contains such (redundant) type checks, even though our runtime type system already ensures type safety (int list cannot masquerade as string list). C# has reified generics and avoids such type checks [8].

6 Conclusion

We presented dart2java, a source-to-source compiler which allows programmers to run Dart code on the JVM. Due to the similarities of Dart and Java, we think that Dart is a good fit for the JVM if performance considerations are taken into account. As our benchmarks show, dart2java's performance is comparable to the Dart VM's performance. Interoperability between Dart and Java is still limited at this point of time and subject to future work.

Acknowledgments

We would like to thank Vijay Menon, Jennifer Messerly and Leaf Peterson from Google Seattle for their guidance and mentorship while working on this project.

References

[1] Gilad Bracha. Generics in the Java programming language. *Sun Microsystems*, July 5, 2004.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. OOPSLA '98, pages 183–200. ACM.

[3] Thorsten Brunklaus and Leif Kornstaedt. A virtual machine for multi-language execution. Technical report, Saarland University, 2002.

[4] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, May 1995.

[5] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. ICOOOLPS '09, pages 42–47. ACM.

[6] Torbjörn Ekman, Peter Mechlenborg, and Ulrik Pagh Schultz. Flexible language interoperability. *Journal of Object Technology*, 6(8):95–116, 2007.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[8] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. PLDI '01, pages 1–12. ACM.

[9] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. PLDI '15, pages 545–554. ACM.

[10] Jaime Niño. The cost of erasure in Java generics type system. *Journal of Computing Sciences in Colleges*, 22(5):2–11, May 2007.

[11] Matthias Springer. Inter-language collaboration in an object-oriented virtual machine. *CoRR*, abs/1606.03644, 2016.

[12] Jan Vraný. *Supporting Multiple Languages in Virtual Machines.* PhD thesis, Czech Technical University, Prague, 2010.

A Links to Dart Components

- dart2java: https://github.com/google/dart2java
  This repository also contains all the benchmarks used in this paper and the generated Java source code.
- Dart Analyzer:
  https://pub.dartlang.org/packages/analyzer
- Kernel: https://github.com/dart-lang/kernel
- Dart to JavaScript Compiler (DDC): https://github.com/dart-lang/sdk/tree/master/sdk/lib/js/dart2js
- Dart Language Guide, strong mode:
  https://www.dartlang.org/guides/language/sound-dart

B Appendix: Full Example

The following two listing shows a generic class in Dart code along with the generated Java code (without generic specializations and interfaces). The static method `new_` is the Java entry point for the unnamed constructor. The other static method is the entry point for the named constructor. These methods create an instance of a specialized class or the generic class, based on the (fully reified) type parameter.

```
class MyMap<K, V> extends Map<K, V> {
  List<K> keys;
  List<V> values;

  int get size => keys.length;
  bool get isEmpty => keys.isEmpty;

  MyMap() {
    keys = new List<K>();
    values = new List<V>();
  }

  MyMap.fromList(List<K> keys, List<V> values)
    : keys = keys, values = values;

  V operator [](K key) {
    for (int i = 0; i < keys.length; i++) {
      if (keys[i] == key) {
        return values[i];
      }
    }

    throw "Key not found";
  }
}
```

```
class MyMap<K, V> extends DartObject implements ↙
    ↳ MyMap_IF<K, V> {
  dart.core.List_IF<K> keys;
  dart.core.List_IF<V> values;

  int getSize() { return keys.getLength(); }
  boolean getIsEmpty() { return keys.getIsEmpty(); }

  public MyMap(Type type) {
    this.type = type;
  }

  public static <K, V> MyMap_IF<K, V> new_(Type type){
    MyMap_IF<K, V> instance;

    if (type.typeParams[0] == INT_TYPE && type.↙
     ↳ typeParams[1] == INT_TYPE) {
      instance = new MyMap__int_int(type);
    } else if (/* ... */) {
      /* ... */
    } else {
      instance = new MyMap<K, V>(type);
    }

    instance._constructor();
    return instance;
  }

  public static <K, V> MyMap_IF<K, V> new_fromList(↙
     ↳ Type type, List_IF<K> keys, List_IF<V> values)↙
     ↳ {
    MyMap_IF instance;
    // (Initialize instance)

    instance._constructor_fromList(keys, values);
    return instance;
  }

  public V operatorAt_MyMap(K key) {
    for (int i = 0; i < getKeys().getLength(); i++) {
      if (dart._runtime.helpers.ObjectHelper.↙
     ↳ operatorEqual(getKeys().operatorAt(i), key)) {
        return getValues().operatorAt(i);
      }
    }

    throw new RuntimeException("Key not found");
  }
}
```