Java Generated Code

This page describes exactly what Java code the protocol buffer compiler generates for any given protocol definition. You should read the language guide before reading this document.

Compiler Invocation

The protocol buffer compiler produces Java output when invoked with the `--java_out=` command-line flag. The parameter to the `--java_out=` option is the directory where you want the compiler to write your Java output. The compiler creates a single `.java` for each `.proto` file input. This file contains a single outer class definition containing several nested classes and static fields based on the declarations in the `.proto` file.

The outer class's name is chosen as follows: If the `.proto` file contains a line like the following:

```
option java_outer_classname = "Foo";
```

Then the outer class name will be `Foo`. Otherwise, the outer class name is determined by converting the `.proto` file base name to camel case. For example, `foo_bar.proto` will become `FooBar`.

The Java package name is chosen as described under Packages, below.

The output file is chosen by concatenating the parameter to `--java_out=`, the package name (with `.`s replaced with `/`s), and the `.java` file name.

So, for example, let's say you invoke the compiler as follows:

```
protoc --proto_path=src --java_out=build/gen src/foo.proto
```

If `foo.proto`'s java package is `com.example` and its outer classname is `FooProtos`, then the protocol buffer compiler will generate the file `build/gen/com/example/FooProtos.java`. The protocol buffer compiler will automatically create the `build/gen/com` and `build/gen/com/example` directories if needed. However, it will not create `build/gen` or `build`; they must already exist. You can specify multiple `.proto` files in a single invocation; all output files will be generated at once.

> When outputting Java code, the protocol buffer compiler's ability to output directly to JAR archives is particularly convenient, as many Java tools are able to read source code directly from JAR files. To output to a JAR file, simply provide an output location ending in `.jar`. Note that only the Java source code is placed in the archive; you must still compile it separately to produce Java class files.

Packages

The generated class is placed in a Java package based on the `java_package` option. If the option is omitted, the `package` declaration is used instead.

For example, if the `.proto` file contains:

```
package foo.bar;
```

Then the resulting Java class will be placed in Java package `foo.bar`. However, if the `.proto` file also contains a `java_package` option, like so:

```
package foo.bar;
option java_package = "com.example.foo.bar";
```

Then the class is placed in the `com.example.foo.bar` package instead. The `java_package` option is provided because normal `.proto` `package` declarations are not expected to start with a backwards domain name.

Messages

Given a simple message declaration:

```
message Foo {}
```

The protocol buffer compiler generates a class called `Foo`, which implements the `Message` interface. The class is declared `final`; no further subclassing is allowed. `Foo` extends `GeneratedMessage`, but this should be considered an implementation detail. By default, `Foo` overrides many methods of `GeneratedMessage` with specialized versions for maximum speed. However, if the `.proto` file contains the line:

```
option optimize_for = CODE_SIZE;
```

then `Foo` will override only the minimum set of methods necessary to function and rely on `GeneratedMessage`'s reflection-based implementations of the rest. This significantly reduces the size of the generated code, but also reduces performance. Alternatively, if the `.proto` file contains:

```
option optimize_for = LITE_RUNTIME;
```

then `Foo` will include fast implementations of all methods, but will implement the `MessageLite` interface, which only contains a subset of the methods of `Message`. In particular, it does not support descriptors or reflection. However, in this mode, the generated code only needs to link against `libprotobuf-lite.jar` instead of `libprotobuf.jar`. The "lite" library is much smaller than the full library, and is more appropriate for resource-constrained systems such as mobile phones.

The `Message` interface defines methods that let you check, manipulate, read, or write the entire message. In addition to these methods, the `Foo` class defines the following static methods:

- `static Foo getDefaultInstance()`: Returns a singleton instance of `Foo`, which is identical to what you'd get if you called `Foo.newBuilder().build()` (so all singular fields are unset and all repeated fields are empty). Note that the default instance of a message can be used as a factory by calling its `newBuilderForType()` method.
- `static Descriptor getDescriptor()`: Returns the type's descriptor. This contains information about the type, including what fields it has and what their types are. This can be used with the reflection methods of the `Message`, such as `getField()`.
- `static Foo parseFrom(...)`: Parses a message of type `Foo` from the given source and returns it. There is one `parseFrom` method corresponding to each variant of `mergeFrom()` in the `Message.Builder` interface. Note that `parseFrom()` never throws `UninitializedMessageException`; it throws `InvalidProtocolBufferException` if the parsed message is missing required fields. This makes it subtly different from calling `Foo.newBuilder().mergeFrom(...).build()`.
- `Foo.Builder newBuilder()`: Creates a new builder (described below).
- `Foo.Builder newBuilder(Foo prototype)`: Creates a new builder with all fields initialized to the same values that they have in `prototype`. Since embedded message and string objects are immutable, they are shared between the original and the copy.

Builders

Message objects – such as instances of the `Foo` class described above – are immutable, just like a Java `String`. To construct a message object, you need to use a *builder*. Each message class has its own builder class – so in our `Foo` example, the protocol buffer compiler generates a nested class `Foo.Builder` which can be used to build a `Foo`. `Foo.Builder` implements the `Message.Builder` interface. It extends the `GeneratedMessage.Builder` class, but, again, this should be considered an implementation detail. Like `Foo`, `Foo.Builder` may rely on generic method implementations in `GeneratedMessage.Builder` or, when the `optimize_for` option is used, generated custom code that is much faster.

`Foo.Builder` does not define any static methods. Its interface is exactly as defined by the `Message.Builder` interface, with the exception that return types are more specific: methods of `Foo.Builder` that modify the builder return type `Foo.Builder`, and `build()` returns type `Foo`.

Methods that modify the contents of a builder – including field setters – always return a reference to the builder (i.e. they "`return this;`"). This allows multiple method calls to be chained together in one line. For example: `builder.mergeFrom(obj).setFoo(1).setBar("abc").clearBaz();`

Sub Builders

For messages containing sub-messages, the compiler also generates sub builders. This allows you to repeatedly modify deep-nested sub-messages without rebuilding them. For example:

```
message Foo {
  optional int32 val = 1;
  // some other fields.
}

message Bar {
  optional Foo foo = 1;
  // some other fields.
}

message Baz {
```

```
  optional Bar bar = 1;
  // some other fields.
}
```

If you have a `Baz` message already, and want to change the deeply nested `val` in `Foo`. Instead of:

```
baz = baz.toBuilder().setBar(
    baz.getBar().toBuilder().setFoo(
        baz.getBar().getFoo().toBuilder().setVal(10).build()
    ).build()).build();
```

You can write:

```
Baz.Builder builder = baz.toBuilder();
builder.getBarBuilder().getFooBuilder().setVal(10);
baz = builder.build();
```

Nested Types

A message can be declared inside another message. For example: `message Foo { message Bar { } }`

In this case, the compiler simply generates `Bar` as an inner class nested inside `Foo`.

Fields

In addition to the methods described in the previous section, the protocol buffer compiler generates a set of accessor methods for each field defined within the message in the `.proto` file. The methods that read the field value are defined both in the message class and its corresponding builder; the methods that modify the value are only defined in the builder only.

Note that method names always use camel-case naming, even if the field name in the `.proto` file uses lower-case with underscores (as it should). The case-conversion works as follows:

1. For each underscore in the name, the underscore is removed, and the following letter is capitalized.
2. If the name will have a prefix attached (e.g. "get"), the first letter is capitalized. Otherwise, it is lower-cased.

Thus, the field `foo_bar_baz` becomes `fooBarBaz`. If prefixed with `get`, it would be `getFooBarBaz`.

As well as accessor methods, the compiler generates an integer constant for each field containing its field number. The constant name is the field name converted to upper-case followed by `_FIELD_NUMBER`. For example, given the field `optional int32 foo_bar = 5;`, the compiler will generate the constant `public static final int FOO_BAR_FIELD_NUMBER = 5;`.

Singular Fields

For either of these field definitions:

```
optional int32 foo = 1;
required int32 foo = 1;
```

The compiler will generate the following accessor methods in both the message class and its builder:

- `boolean hasFoo()`: Returns `true` if the field is set.
- `int getFoo()`: Returns the current value of the field. If the field is not set, returns the default value.

The compiler will generate the following methods only in the message's builder:

- `Builder setFoo(int value)`: Sets the value of the field. After calling this, `hasFoo()` will return `true` and `getFoo()` will return `value`.
- `Builder clearFoo()`: Clears the value of the field. After calling this, `hasFoo()` will return `false` and `getFoo()` will return the default value.

For other simple field types, the corresponding Java type is chosen according to the scalar value types table. For message and enum types, the value type is replaced with the message or enum class.

Embedded Message Fields

For message types, `setFoo()` also accepts an instance of the message's builder type as the parameter. This is just a shortcut which is equivalent to calling `.build()` on the builder and passing the result to the method.

If the field is not set, `getFoo()` will return a Foo instance with none of its fields set (possiblly the instance returned by `Foo.getDefaultInstance()`).

In addition, the compiler generates the following additional accessor methods in both the message class and its builder for message types, allowing you to access the relevant subbuilders:

- `Builder getFooBuilder()`: Returns the builder for the field.
- `FooOrBuilder getFooOrBuilder()`: Returns the builder for the field, if it already exists, or the message if not.

Repeated Fields

For this field definition:

```
repeated int32 foo = 1;
```

The compiler will generate the following accessor methods in both the message class and its builder:

- `int getFooCount()`: Returns the number of elements currently in the field.
- `int getFoo(int index)`: Returns the element at the given zero-based index.
- `List<Integer> getFooList()`: Returns the entire field as an immutable list. If the field is not set, returns an empty list.

The compiler will generate the following methods only in the message's builder:

- `Builder setFoo(int index, int value)`: Sets the value of the element at the given zero-based index.
- `Builder addFoo(int value)`: Appends a new element to the field with the given value.
- `Builder addAllFoo(List<Integer> value)`: Appends all elements in the given list to the field.
- `Builder clearFoo()`: Removes all elements from the field. After calling this, `getFooCount()` will return zero.

For other simple field types, the corresponding Java type is chosen according to the scalar value types table. For message and enum types, the type is the message or enum class.

Repeated Embedded Message Fields

For message types, `setFoo()` and `addFoo()` also accept an instance of the message's builder type as the parameter. This is just a shortcut which is equivalent to calling `.build()` on the builder and passing the result to the method.

In addition, the compiler generates the following additional accessor methods in both the message class and its builder for message types, allowing you to access the relevant subbuilders:

- `FooOrBuilder getFooOrBuilder(int index)`: Returns the builder for the specified element, if it already exists, or the element if not. If this is called from a message class, it will always return a message rather than a builder.
- `List<FooOrBuilder> getFooOrBuilderList()`: Returns the entire field as a list of builders (if available) or messages if not. If this is called from a message class, it will always return messages rather than builders.

The compiler will generate the following methods only in the message's builder:

- `Builder getFooBuilder(int index)`: Returns the builder for the element at the specified index.
- `Builder addFooBuilder(int index)`: Appends and returns a builder for a default message instance at the specified index.
- `Builder addFooBuilder()`: Appends and returns a builder for a default message instance.
- `List<FooOrBuilder> getFooBuilderList()`: Returns the entire field as a list of builders.

Enumerations

Given an enum definition like:

```
enum Foo {
  VALUE_A = 1;
  VALUE_B = 5;
  VALUE_C = 1234;
}
```

The protocol buffer compiler will generate a Java enum type called `Foo` with the same set of values. Additionally, the values of this enum type have the following special methods:

- `int getNumber()`: Returns the object's numeric value as defined in the `.proto` file.
- `EnumValueDescriptor getValueDescriptor()`: Returns the value's descriptor, which contains information about the value's name, number, and type.
- `EnumDescriptor getDescriptorForType()`: Returns the enum type's descriptor, which contains e.g. information about each defined value.

Additionally, the `Foo` enum type contains the following static methods:

- `static Foo valueOf(int value)`: Returns the enum object corresponding to the given numeric value.
- `static Foo valueOf(EnumValueDescriptor descriptor)`: Returns the enum object corresponding to the given value descriptor. May be faster than `valueOf(int)`.
- `EnumDescriptor getDescriptor()`: Returns the enum type's descriptor, which contains e.g. information about each defined value. (This differs from `getDescriptorForType()` only in that it is a static method.)

An integer constant is also generated with the suffix _VALUE for each enum value.

Note that the `.proto` language allows multiple enum symbols to have the same numeric value. Symbols with the same numeric value are synonyms. For example:

```
enum Foo {
  BAR = 1;
  BAZ = 1;
}
```

In this case, `BAZ` is a synonym for `BAR`. In Java, `BAZ` will be defined as a static final field like so:

```
static final Foo BAZ = BAR;
```

Thus, `BAR` and `BAZ` compare equal, and `BAZ` should never appear in switch statements. The compiler always chooses the first symbol defined with a given numeric value to be the "canonical" version of that symbol; all subsequent symbols with the same number are just aliases.

An enum can be defined nested within a message type. The compiler generates the Java enum definition nested within that message type's class.

Extensions

Given a message with an extension range:

```
message Foo {
  extensions 100 to 199;
}
```

The protocol buffer compiler will make `Foo` extend `GeneratedMessage.ExtendableMessage` instead of the usual `GeneratedMessage`. Similarly, `Foo`'s builder will extend `GeneratedMessage.ExtendableBuilder`. You should never refer to these base types by name (`GeneratedMessage` is considered an implementation detail). However, these superclasses define a number of additional methods that you can use to manipulate extensions.

In particular `Foo` and `Foo.Builder` will inherit the methods `hasExtension()`, `getExtension()`, and `getExtensionCount()`. Additionally, `Foo.Builder` will inherit methods `setExtension()` and `clearExtension()`. Each of these methods takes, as its first parameter, an extension identifier (described below), which identifies an extension field. The remaining parameters and the return value are exactly the same as those for the corresponding accessor methods that would be generated for a normal (non-extension) field of the same type as the extension identifier.

Given an extension definition:

```
extend Foo {
  optional int32 bar = 123;
}
```

The protocol buffer compiler generates an "extension identifier" called `bar`, which you can use with `Foo`'s extension accessors to access this extension, like so:

```
Foo foo =
  Foo.newBuilder()
     .setExtension(bar, 1)
     .build();
assert foo.hasExtension(bar);
assert foo.getExtension(bar) == 1;
```

(The exact implementation of extension identifiers is complicated and involves magical use of generics – however, you don't need to worry about how extension identifiers work to use them.)

Note that `bar` would be declared as a static field of the outer class for the `.proto` file, as described above; we have omitted the outer class name in the example.

Extensions can be declared nested inside of another type. For example, a common pattern is to do something like this:

```
message Baz {
  extend Foo {
    optional Baz foo_ext = 124;
  }
}
```

In this case, the extension identifier `foo_ext` is declared nested inside `Baz`. It can be used as follows:

```
Baz baz = createMyBaz();
Foo foo =
  Foo.newBuilder()
     .setExtension(Baz.fooExt, baz)
     .build();
```

When parsing a message that might have extensions, you must provide an ExtensionRegistry in which you have registered any extensions that you want to be able to parse. Otherwise, those extensions will just be treated like unknown fields. For example:

```
  ExtensionRegistry registry = ExtensionRegistry.newInstance();
  registry.add(Baz.fooExt);
  Foo foo = Foo.parseFrom(input, registry);
```

Services

If the `.proto` file contains the following line:

```
option java_generic_services = true;
```

Then the protocol buffer compiler will generate code based on the service definitions found in the file as described in this section. However, the generated code may be undesirable as it is not tied to any particular RPC system, and thus requires more levels of indirection that code tailored to one system. If you do NOT want this code to be generated, add this line to the file:

```
option java_generic_services = false;
```

If neither of the above lines are given, the option defaults to `false`, as generic services are deprecated. (Note that prior to 2.4.0, the option defaults to `true`)

RPC systems based on `.proto`-language service definitions should provide plugins to generate code approriate for the system. These plugins are likely to require that abstract services are disabled, so that they can generate their own classes of the same names. Plugins are new in version 2.3.0 (January 2010).

The remainder of this section describes what the protocol buffer compiler generates when abstract services are enabled.

Interface

Given a service definition:

```
service Foo {
  rpc Bar(FooRequest) returns(FooResponse);
}
```

The protocol buffer compiler will generate an abstract class `Foo` to represent this service. `Foo` will have an abstract method for each method defined in the service definition. In this case, the method `Bar` is defined as:

```
abstract void bar(RpcController controller, FooRequest request,
                  RpcCallback<FooResponse> done);
```

The parameters are equivalent to the parameters of `Service.CallMethod()`, except that the `method` argument is implied and `request` and `done` specify their exact type.

`Foo` subclasses the `Service` interface. The protocol buffer compiler automatically generates implementations of the methods of `Service` as follows:

- `getDescriptorForType`: Returns the service's `ServiceDescriptor`.
- `callMethod`: Determines which method is being called based on the provided method descriptor and calls it directly, down-casting the request message and callback to the correct types.
- `getRequestPrototype` and `getRequestPrototype`: Returns the default instance of the request or response of the correct type for the given method.

The following static method is also generated:

- `static ServiceDescriptor getDescriptor()`: Returns the type's descriptor, which contains information about what methods this service has and what their input and output types are.

`Foo` will also contain a nested interface `Foo.Interface`. This is a pure interface that again contains methods corresponding to each method in your service definition. However, this interface does not extend the `Service` interface. This is a problem because RPC server implementations are usually written to use abstract `Service` objects, not your particular service. To solve this problem, if you have an object `impl` implementing `Foo.Interface`, you can call `Foo.newReflectiveService(impl)` to construct an instance of `Foo` that simply delegates to `impl`, and implements `Service`.

To recap, when implementing your own service, you have two options:

- Subclass `Foo` and implement its methods as appropriate, then hand instances of your subclass directly to the RPC server implementation. This is usually easiest, but some consider it less "pure".

- Implement `Foo.Interface` and use `Foo.newReflectiveService(Foo.Interface)` to construct a `Service` wrapping it, then pass the wrapper to your RPC implementation.

Stub

The protocol buffer compiler also generates a "stub" implementation of every service interface, which is used by clients wishing to send requests to servers implementing the service. For the `Foo` service (above), the stub implementation `Foo.Stub` will be defined as a nested class.

`Foo.Stub` is a subclass of `Foo` which also implements the following methods:

- `Foo.Stub(RpcChannel channel)`: Constructs a new stub which sends requests on the given channel.
- `RpcChannel getChannel()`: Returns this stub's channel, as passed to the constructor.

The stub additionally implements each of the service's methods as a wrapper around the channel. Calling one of the methods simply calls `channel.callMethod()`.

The Protocol Buffer library does not include an RPC implementation. However, it includes all of the tools you need to hook up a generated service class to any arbitrary RPC implementation of your choice. You need only provide implementations of `RpcChannel` and `RpcController`.

Blocking Interfaces

The RPC classes described above all have non-blocking semantics: when you call a method, you provide a callback object which will be invoked once the method completes. Often it is easier (though possibly less scalable) to write code using blocking semantics, where the method simply doesn't return until it is done. To accomodate this, the protocol buffer compiler also generates blocking versions of your service class. `Foo.BlockingInterface` is equivalent to `Foo.Interface` except that each method simply returns the result rather than call a callback. So, for example, `bar` is defined as:

```
abstract FooResponse bar(RpcController controller, FooRequest request)
                         throws ServiceException;
```

Analogous to non-blocking services, `Foo.newReflectiveBlockingService(Foo.BlockingInterface)` returns a `BlockingService` wrapping some `Foo.BlockingInterface`. Finally, `Foo.BlockingStub` returns a stub implementation of `Foo.BlockingInterface` that sends requests to a particular `BlockingRpcChannel`.

Plugin Insertion Points

Code generator plugins which want to extend the output of the Java code generator may insert code of the following types using the given insertion point names.

- `outer_class_scope`: Member declarations that belong in the file's outer class.
- `class_scope:TYPENAME`: Member declarations that belong in a message class. `TYPENAME` is the full proto name, e.g. `package.MessageType`.
- `builder_scope:TYPENAME`: Member declarations that belong in a message's builder class. `TYPENAME` is the full proto name, e.g. `package.MessageType`.
- `enum_scope:TYPENAME`: Member declarations that belong in an enum class. `TYPENAME` is the full proto enum name, e.g. `package.EnumType`.

Generated code cannot contain import statements, as these are prone to conflict with type names defined within the generated code itself. Instead, when referring to an external class, you must always use its fully-qualified name.

> The logic for determining output file names in the Java code generator is fairly complicated. You should probably look at the `protoc` source code, particularly `java_headers.cc`, to make sure you have covered all cases.

> Do not generate code which relies on private class members declared by the standard code generator, as these implementation details may change in future versions of Protocol Buffers.