**Developer Guide**

Welcome to the developer documentation for protocol buffers – a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more.

This documentation is aimed at Java, C++, or Python developers who want to use protocol buffers in their applications. This overview introduces protocol buffers and tells you what you need to do to get started – you can then go on to follow the [tutorials](https://developers.google.com/protocol-buffers/docs/tutorials) or delve deeper into [protocol buffer encoding](https://developers.google.com/protocol-buffers/docs/encoding). API [reference documentation](https://developers.google.com/protocol-buffers/docs/reference/overview) is also provided for all three languages, as well as [language](https://developers.google.com/protocol-buffers/docs/proto) and [style](https://developers.google.com/protocol-buffers/docs/style) guides for writing .proto files.

**What are protocol buffers?**

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format.

**How do they work?**

You specify how you want the information you're serializing to be structured by defining protocol buffer message types in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Here's a very basic example of a .proto file that defines a message containing information about a person:

message Person {

required string name = 1;

required int32 id = 2;

optional string email = 3;

enum PhoneType {

MOBILE = 0;

HOME = 1;

WORK = 2;

}

message PhoneNumber {

required string number = 1;

optional PhoneType type = 2 [default = HOME];

}

repeated PhoneNumber phone = 4;

}

As you can see, the message format is simple – each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers (integer or floating-point), booleans, strings, raw bytes, or even (as in the example above) other protocol buffer message types, allowing you to structure your data hierarchically. You can specify optional fields, required fields, and repeated fields. You can find more information about writing .proto files in the [Protocol Buffer Language Guide](https://developers.google.com/protocol-buffers/docs/proto).

Once you've defined your messages, you run the protocol buffer compiler for your application's language on your .proto file to generate data access classes. These provide simple accessors for each field (like query() and set\_query()) as well as methods to serialize/parse the whole structure to/from raw bytes – so, for instance, if your chosen language is C++, running the compiler on the above example will generate a class called Person. You can then use this class in your application to populate, serialize, and retrieve Person protocol buffer messages. You might then write some code like this:

Person person;

person.set\_name("John Doe");

person.set\_id(1234);

person.set\_email("jdoe@example.com");

fstream output("myfile", ios::out | ios::binary);

person.SerializeToOstream(&output);

Then, later on, you could read your message back in:

fstream input("myfile", ios::in | ios::binary);

Person person;

person.ParseFromIstream(&input);

cout << "Name: " << person.name() << endl;

cout << "E-mail: " << person.email() << endl;

You can add new fields to your message formats without breaking backwards-compatibility; old binaries simply ignore the new field when parsing. So if you have a communications protocol that uses protocol buffers as its data format, you can extend your protocol without having to worry about breaking existing code.

You'll find a complete reference for using generated protocol buffer code in the [API Reference section](https://developers.google.com/protocol-buffers/docs/reference/overview), and you can find out more about how protocol buffer messages are encoded in [Protocol Buffer Encoding](https://developers.google.com/protocol-buffers/docs/encoding).

**Why not just use XML?**

Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

* are simpler
* are 3 to 10 times smaller
* are 20 to 100 times faster
* are less ambiguous
* generate data access classes that are easier to use programmatically

For example, let's say you want to model a person with a name and an email. In XML, you need to do:

<person>

<name>John Doe</name>

<email>jdoe@example.com</email>

</person>

while the corresponding protocol buffer message (in protocol buffer [text format](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.text_format)) is:

# Textual representation of a protocol buffer.

# This is \*not\* the binary format used on the wire.

person {

name: "John Doe"

email: "jdoe@example.com"

}

When this message is encoded to the protocol buffer [binary format](https://developers.google.com/protocol-buffers/docs/encoding) (the text format above is just a convenient human-readable representation for debugging and editing), it would probably be 28 bytes long and take around 100-200 nanoseconds to parse. The XML version is at least 69 bytes if you remove whitespace, and would take around 5,000-10,000 nanoseconds to parse.

Also, manipulating a protocol buffer is much easier:

cout << "Name: " << person.name() << endl;

cout << "E-mail: " << person.email() << endl;

Whereas with XML you would have to do something like:

cout << "Name: "

<< person.getElementsByTagName("name")->item(0)->innerText()

<< endl;

cout << "E-mail: "

<< person.getElementsByTagName("email")->item(0)->innerText()

<< endl;

However, protocol buffers are not always a better solution than XML – for instance, protocol buffers would not be a good way to model a text-based document with markup (e.g. HTML), since you cannot easily interleave structure with text. In addition, XML is human-readable and human-editable; protocol buffers, at least in their native format, are not. XML is also – to some extent – self-describing. A protocol buffer is only meaningful if you have the message definition (the .proto file).

**Sounds like the solution for me! How do I get started?**

[Download the package](http://code.google.com/p/protobuf/downloads/) – this contains the complete source code for the Java, Python, and C++ protocol buffer compilers, as well as the classes you need for I/O and testing. To build and install your compiler, follow the instructions in the README.

Once you're all set, try following the [tutorial](https://developers.google.com/protocol-buffers/docs/tutorials) for your chosen language – this will step you through creating a simple application that uses protocol buffers.

**A bit of history**

Protocol buffers were initially developed at Google to deal with an index server request/response protocol. Prior to protocol buffers, there was a format for requests and responses that used hand marshalling/unmarshalling of requests and responses, and that supported a number of versions of the protocol. This resulted in some very ugly code, like:

if (version == 3) {

...

} else if (version > 4) {

if (version == 5) {

...

}

...

}

Explicitly formatted protocols also complicated the rollout of new protocol versions, because developers had to make sure that all servers between the originator of the request and the actual server handling the request understood the new protocol before they could flip a switch to start using the new protocol.

Protocol buffers were designed to solve many of these problems:

* New fields could be easily introduced, and intermediate servers that didn't need to inspect the data could simply parse it and pass through the data without needing to know about all the fields.
* Formats were more self-describing, and could be dealt with from a variety of languages (C++, Java, etc.)

However, users still needed to hand-write their own parsing code.

As the system evolved, it acquired a number of other features and uses:

* Automatically-generated serialization and deserialization code avoided the need for hand parsing.
* In addition to being used for short-lived RPC (Remote Procedure Call) requests, people started to use protocol buffers as a handy self-describing format for storing data persistently (for example, in Bigtable).
* Server RPC interfaces started to be declared as part of protocol files, with the protocol compiler generating stub classes that users could override with actual implementations of the server's interface.

Protocol buffers are now Google's *lingua franca* for data – at time of writing, there are 48,162 different message types defined in the Google code tree across 12,183 .proto files. They're used both in RPC systems and for persistent storage of data in a variety of storage systems.

# Language Guide

* [Defining A Message Type](https://developers.google.com/protocol-buffers/docs/proto#simple)
* [Scalar Value Types](https://developers.google.com/protocol-buffers/docs/proto#scalar)
* [Optional And Default Values](https://developers.google.com/protocol-buffers/docs/proto#optional)
* [Enumerations](https://developers.google.com/protocol-buffers/docs/proto#enum)
* [Using Other Message Types](https://developers.google.com/protocol-buffers/docs/proto#other)
* [Nested Types](https://developers.google.com/protocol-buffers/docs/proto#nested)
* [Updating A Message Type](https://developers.google.com/protocol-buffers/docs/proto#updating)
* [Extensions](https://developers.google.com/protocol-buffers/docs/proto#extensions)
* [Packages](https://developers.google.com/protocol-buffers/docs/proto#packages)
* [Defining Services](https://developers.google.com/protocol-buffers/docs/proto#services)
* [Options](https://developers.google.com/protocol-buffers/docs/proto#options)
* [Generating Your Classes](https://developers.google.com/protocol-buffers/docs/proto#generating)

This guide describes how to use the protocol buffer language to structure your protocol buffer data, including .proto file syntax and how to generate data access classes from your .proto files.

This is a reference guide – for a step by step example that uses many of the features described in this document, see the [tutorial](https://developers.google.com/protocol-buffers/docs/tutorials) for your chosen language.

## Defining A Message Type

First let's look at a very simple example. Let's say you want to define a search request message format, where each search request has a query string, the particular page of results you are interested in, and a number of results per page. Here's the .proto file you use to define the message type.

message SearchRequest {

required string query = 1;

optional int32 page\_number = 2;

optional int32 result\_per\_page = 3;

}

The SearchRequest message definition specifies three fields (name/value pairs), one for each piece of data that you want to include in this type of message. Each field has a name and a type.

### Specifying Field Types

In the above example, all the fields are [scalar types](https://developers.google.com/protocol-buffers/docs/proto#scalar): two integers (page\_number and result\_per\_page) and a string (query). However, you can also specify composite types for your fields, including [enumerations](https://developers.google.com/protocol-buffers/docs/proto#enum) and other message types.

### Assigning Tags

As you can see, each field in the message definition has a **unique numbered tag**. These tags are used to identify your fields in the [message binary format](https://developers.google.com/protocol-buffers/docs/encoding), and should not be changed once your message type is in use. Note that tags with values in the range 1 through 15 take one byte to encode, including the identifying number and the field's type (you can find out more about this in [Protocol Buffer Encoding](https://developers.google.com/protocol-buffers/docs/encoding.html#structure)). Tags in the range 16 through 2047 take two bytes. So you should reserve the tags 1 through 15 for very frequently occurring message elements. Remember to leave some room for frequently occurring elements that might be added in the future.

The smallest tag number you can specify is 1, and the largest is 229 - 1, or 536,870,911. You also cannot use the numbers 19000 though 19999 (FieldDescriptor::kFirstReservedNumber through FieldDescriptor::kLastReservedNumber), as they are reserved for the Protocol Buffers implementation - the protocol buffer compiler will complain if you use one of these reserved numbers in your .proto.

### Specifying Field Rules

You specify that message fields are one of the following:

* required: a well-formed message must have exactly one of this field.
* optional: a well-formed message can have zero or one of this field (but not more than one).
* repeated: this field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

For historical reasons, repeated fields of basic numeric types aren't encoded as efficiently as they could be. New code should use the special option [packed=true] to get a more efficient encoding. For example:

repeated int32 samples = 4 [packed=true];

**Required Is Forever** You should be very careful about marking fields as required. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field – old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Some engineers at Google have come to the conclusion that using required does more harm than good; they prefer to use only optional and repeated. However, this view is not universal.

### Adding More Message Types

Multiple message types can be defined in a single .proto file. This is useful if you are defining multiple related messages – so, for example, if you wanted to define the reply message format that corresponds to your SearchResponse message type, you could add it to the same .proto:

message SearchRequest {

required string query = 1;

optional int32 page\_number = 2;

optional int32 result\_per\_page = 3;

}

message SearchResponse {

...

}

### Adding Comments

To add comments to your .proto files, use C/C++-style // syntax.

message SearchRequest {

required string query = 1;

optional int32 page\_number = 2;// Which page number do we want?

optional int32 result\_per\_page = 3;// Number of results to return per page.

}

### What's Generated From Your .proto?

When you run the [protocol buffer compiler](https://developers.google.com/protocol-buffers/docs/proto#generating) on a .proto, the compiler generates the code in your chosen language you'll need to work with the message types you've described in the file, including getting and setting field values, serializing your messages to an output stream, and parsing your messages from an input stream.

For **C++**, the compiler generates a .h and .cc file from each .proto, with a class for each message type described in your file.

For **Java**, the compiler generates a .java file with a class for each message type, as well as a special Builder classes for creating message class instances.

**Python** is a little different – the Python compiler generates a module with a static descriptor of each message type in your .proto, which is then used with a metaclass to create the necessary Python data access class at runtime.

You can find out more about using the APIs for each language by following the tutorial for your chosen language. For even more API details, see the relevant [API reference](https://developers.google.com/protocol-buffers/docs/reference/overview).

## Scalar Value Types

A scalar message field can have one of the following types – the table shows the type specified in the .proto file, and the corresponding type in the automatically generated class:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **.proto Type** | **Notes** | **C++ Type** | **Java Type** | **Python Type[2]** |
| double |  | double | double | float |
| float |  | float | float | float |
| int32 | Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead. | int32 | int | int |
| int64 | Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead. | int64 | long | int/long[3] |
| uint32 | Uses variable-length encoding. | uint32 | int[1] | int/long[3] |
| uint64 | Uses variable-length encoding. | uint64 | long[1] | int/long[3] |
| sint32 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s. | int32 | int | int |
| sint64 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s. | int64 | long | int/long[3] |
| fixed32 | Always four bytes. More efficient than uint32 if values are often greater than 228. | uint32 | int[1] | int |
| fixed64 | Always eight bytes. More efficient than uint64 if values are often greater than 256. | uint64 | long[1] | int/long[3] |
| sfixed32 | Always four bytes. | int32 | int | int |
| sfixed64 | Always eight bytes. | int64 | long | int/long[3] |
| bool |  | bool | boolean | boolean |
| string | A string must always contain UTF-8 encoded or 7-bit ASCII text. | string | String | str/unicode[4] |
| bytes | May contain any arbitrary sequence of bytes. | string | ByteString | str |

You can find out more about how these types are encoded when you serialize your message in [Protocol Buffer Encoding](https://developers.google.com/protocol-buffers/docs/encoding).

[1] In Java, unsigned 32-bit and 64-bit integers are represented using their signed counterparts, with the top bit simply being stored in the sign bit.

[2] In all cases, setting values to a field will perform type checking to make sure it is valid.

[3] 64-bit or unsigned 32-bit integers are always represented as long when decoded, but can be an int if an int is given when setting the field. In all cases, the value must fit in the type represented when set. See [2].

[4] Python strings are represented as unicode on decode but can be str if an ASCII string is given (this is subject to change).

## Optional Fields And Default Values

As mentioned above, elements in a message description can be labeled optional. A well-formed message may or may not contain an optional element. When a message is parsed, if it does not contain an optional element, the corresponding field in the parsed object is set to the default value for that field. The default value can be specified as part of the message description. For example, let's say you want to provide a default value of 10 for a SearchRequest's result\_per\_page value.

optional int32 result\_per\_page = 3 [default = 10];

If the default value is not specified for an optional element, a type-specific default value is used instead: for strings, the default value is the empty string. For bools, the default value is false. For numeric types, the default value is zero. For enums, the default value is the first value listed in the enum's type definition.

## Enumerations

When you're defining a message type, you might want one of its fields to only have one of a pre-defined list of values. For example, let's say you want to add a corpus field for each SearchRequest, where the corpus can be UNIVERSAL, WEB, IMAGES, LOCAL, NEWS, PRODUCTS or VIDEO. You can do this very simply by adding an enum to your message definition - a field with an enum type can only have one of a specified set of constants as its value (if you try to provide a different value, the parser will treat it like an unknown field). In the following example we've added an enum called Corpus with all the possible values, and a field of type Corpus:

message SearchRequest {

required string query = 1;

optional int32 page\_number = 2;

optional int32 result\_per\_page = 3 [default = 10];

enum Corpus {

UNIVERSAL = 0;

WEB = 1;

IMAGES = 2;

LOCAL = 3;

NEWS = 4;

PRODUCTS = 5;

VIDEO = 6;

}

optional Corpus corpus = 4 [default = UNIVERSAL];

}

You can define aliases by assigning the same value to different enum constants. To do this you need to set the allow\_alias option to true, otherwise protocol compiler will generate an error message when aliases are found.

enum EnumAllowingAlias {

option allow\_alias = true;

UNKNOWN = 0;

STARTED = 1;

RUNNING = 1;

}

enum EnumNotAllowingAlias {

UNKNOWN = 0;

STARTED = 1;

// RUNNING = 1; // Uncommenting this line will cause a compile error inside Google and a warning message outside.

}

Enumerator constants must be in the range of a 32-bit integer. Since enum values use varint encoding on the wire, negative values are inefficient and thus not recommended. You can define enums within a message definition, as in the above example, or outside – these enums can be reused in any message definition in your .proto file. You can also use an enum type declared in one message as the type of a field in a different message, using the syntax MessageType.EnumType.

When you run the protocol buffer compiler on a .proto that uses an enum, the generated code will have a corresponding enum for Java or C++, or a special EnumDescriptor class for Python that's used to create a set of symbolic constants with integer values in the runtime-generated class.

For more information about how to work with message enums in your applications, see the [generated code guide](https://developers.google.com/protocol-buffers/docs/reference/overview) for your chosen language.

## Using Other Message Types

You can use other message types as field types. For example, let's say you wanted to include Result messages in each SearchResponse message – to do this, you can define a Result message type in the same .proto and then specify a field of type Result in SearchResponse:

message SearchResponse {

repeated Result result = 1;

}

message Result {

required string url = 1;

optional string title = 2;

repeated string snippets = 3;

}

### Importing Definitions

In the above example, the Result message type is defined in the same file as SearchResponse – what if the message type you want to use as a field type is already defined in another .proto file?

You can use definitions from other .proto files by importing them. To import another .proto's definitions, you add an import statement to the top of your file:

import "myproject/other\_protos.proto";

By default you can only use definitions from directly imported .proto files. However, sometimes you may need to move a .proto file to a new location. Instead of moving the .proto file directly and updating all the call sites in a single change, now you can put a dummy .proto file in the old location to forward all the imports to the new location using the import public notion. import public dependencies can be transitively relied upon by anyone importing the proto contaning the import public statement. For example:

// new.proto

// All definitions are moved here

// old.proto

// This is the proto that all clients are importing.

import public "new.proto";

import "other.proto";

// client.proto

import "old.proto";

// You use definitions from old.proto and new.proto, but not other.proto

The protocol compiler searches for imported files in a set of directories specified on the protocol compiler command line using the -I/--proto\_path flag. If no flag was given, it looks in the directory in which the compiler was invoked. In general you should set the --proto\_path flag to the root of your project and use fully qualified names for all imports.

## Nested Types

You can define and use message types inside other message types, as in the following example – here the Result message is defined inside the SearchResponse message:

message SearchResponse {

message Result {

required string url = 1;

optional string title = 2;

repeated string snippets = 3;

}

repeated Result result = 1;

}

If you want to reuse this message type outside its parent message type, you refer to it as Parent.Type:

message SomeOtherMessage {

optional SearchResponse.Result result = 1;

}

You can nest messages as deeply as you like:

message Outer { // Level 0

message MiddleAA { // Level 1

message Inner { // Level 2

required int64 ival = 1;

optional bool booly = 2;

}

}

message MiddleBB { // Level 1

message Inner { // Level 2

required int32 ival = 1;

optional bool booly = 2;

}

}

}

### Groups

**Note that this feature is deprecated and should not be used when creating new message types – use nested message types instead.**

Groups are another way to nest information in your message definitions. For example, another way to specify a SearchResponse containing a number of Results is as follows:

message SearchResponse {

repeated group Result = 1 {

required string url = 2;

optional string title = 3;

repeated string snippets = 4;

}

}

A group simply combines a nested message type and a field into a single declaration. In your code, you can treat this message just as if it had a Result type field called result (the latter name is converted to lower-case so that it does not conflict with the former). Therefore, this example is exactly equivalent to the SearchResponse above, except that the message has a different [wire format](https://developers.google.com/protocol-buffers/docs/encoding).

## Updating A Message Type

If an existing message type no longer meets all your needs – for example, you'd like the message format to have an extra field – but you'd still like to use code created with the old format, don't worry! It's very simple to update message types without breaking any of your existing code. Just remember the following rules:

* Don't change the numeric tags for any existing fields.
* Any new fields that you add should be optional or repeated. This means that any messages serialized by code using your "old" message format can be parsed by your new generated code, as they won't be missing any required elements. You should set up sensible [default values](https://developers.google.com/protocol-buffers/docs/proto#optional) for these elements so that new code can properly interact with messages generated by old code. Similarly, messages created by your new code can be parsed by your old code: old binaries simply ignore the new field when parsing. However, the unknown fields are not discarded, and if the message is later serialized, the unknown fields are serialized along with it – so if the message is passed on to new code, the new fields are still available. Note that preservation of unknown fields is currently not available for Python.
* Non-required fields can be removed, as long as the tag number is not used again in your updated message type (it may be better to rename the field instead, perhaps adding the prefix "OBSOLETE\_", so that future users of your .proto can't accidentally reuse the number).
* A non-required field can be converted to an [extension](https://developers.google.com/protocol-buffers/docs/proto#extensions) and vice versa, as long as the type and number stay the same.
* int32, uint32, int64, uint64, and bool are all compatible – this means you can change a field from one of these types to another without breaking forwards- or backwards-compatibility. If a number is parsed from the wire which doesn't fit in the corresponding type, you will get the same effect as if you had cast the number to that type in C++ (e.g. if a 64-bit number is read as an int32, it will be truncated to 32 bits).
* sint32 and sint64 are compatible with each other but are *not* compatible with the other integer types.
* string and bytes are compatible as long as the bytes are valid UTF-8.
* Embedded messages are compatible with bytes if the bytes contain an encoded version of the message.
* fixed32 is compatible with sfixed32, and fixed64 with sfixed64.
* optional is compatible with repeated. Given serialized data of a repeated field as input, clients that expect this field to be optional will take the last input value if it's a primitive type field or merge all input elements if it's a message type field.
* Changing a default value is generally OK, as long as you remember that default values are never sent over the wire. Thus, if a program receives a message in which a particular field isn't set, the program will see the default value as it was defined in that program's version of the protocol. It will NOT see the default value that was defined in the sender's code.

## Extensions

Extensions let you declare that a range of field numbers in a message are available for third-party extensions. Other people can then declare new fields for your message type with those numeric tags in their own .proto files without having to edit the original file. Let's look at an example:

message Foo {

// ...

extensions 100 to 199;

}

This says that the range of field numbers [100, 199] in Foo is reserved for extensions. Other users can now add new fields to Foo in their own .proto files that import your .proto, using tags within your specified range – for example:

extend Foo {

optional int32 bar = 126;

}

This says that Foo now has an optional int32 field called bar.

When your user's Foo messages are encoded, the wire format is exactly the same as if the user defined the new field inside Foo. However, the way you access extension fields in your application code is slightly different to accessing regular fields – your generated data access code has special accessors for working with extensions. So, for example, here's how you set the value of bar in C++:

Foo foo;

foo.SetExtension(bar, 15);

Similarly, the Foo class defines templated accessors HasExtension(), ClearExtension(), GetExtension(), MutableExtension(), and AddExtension(). All have semantics matching the corresponding generated accessors for a normal field. For more information about working with extensions, see the generated code reference for your chosen language.

Note that extensions can be of any field type, including message types.

### Nested Extensions

You can declare extensions in the scope of another type:

message Baz {

extend Foo {

optional int32 bar = 126;

}

...

}

In this case, the C++ code to access this extension is:

Foo foo;

foo.SetExtension(Baz::bar, 15);

In other words, the only effect is that bar is defined within the scope of Baz.

This is a common source of confusion: Declaring an extend block nested inside a message type does not imply any relationship between the outer type and the extended type. In particular, the above example does not mean that Baz is any sort of subclass of Foo. All it means is that the symbol bar is declared inside the scope of Baz; it's simply a static member.

A common pattern is to define extensions inside the scope of the extension's field type – for example, here's an extension to Foo of type Baz, where the extension is defined as part of Baz:

message Baz {

extend Foo {

optional Baz foo\_ext = 127;

}

...

}

However, there is no requirement that an extension with a message type be defined inside that type. You can also do this:

message Baz {

...

}

// This can even be in a different file.

extend Foo {

optional Baz foo\_baz\_ext = 127;

}

In fact, this syntax may be preferred to avoid confusion. As mentioned above, the nested syntax is often mistaken for subclassing by users who are not already familiar with extensions.

### Choosing Extension Numbers

It's very important to make sure that two users don't add extensions to the same message type using the same numeric tag – data corruption can result if an extension is accidentally interpreted as the wrong type. You may want to consider defining an extension numbering convention for your project to prevent this happening.

If your numbering convention might involve extensions having very large numbers as tags, you can specify that your extension range goes up to the maximum possible field number using the max keyword:

message Foo {

extensions 1000 to max;

}

max is 229 - 1, or 536,870,911.

As when choosing tag numbers in general, your numbering convention also needs to avoid field numbers 19000 though 19999 (FieldDescriptor::kFirstReservedNumber through FieldDescriptor::kLastReservedNumber), as they are reserved for the Protocol Buffers implementation. You can define an extension range that includes this range, but the protocol compiler will not allow you to define actual extensions with these numbers.

## Packages

You can add an optional package specifier to a .proto file to prevent name clashes between protocol message types.

package foo.bar;

message Open { ... }

You can then use the package specifier when defining fields of your message type:

message Foo {

...

required foo.bar.Open open = 1;

...

}

The way a package specifier affects the generated code depends on your chosen language:

* In **C++** the generated classes are wrapped inside a C++ namespace. For example, Open would be in the namespace foo::bar.
* In **Java**, the package is used as the Java package, unless you explicitly provide a option java\_package in your .proto file.
* In **Python**, the package directive is ignored, since Python modules are organized according to their location in the file system.

### Packages and Name Resolution

Type name resolution in the protocol buffer language works like C++: first the innermost scope is searched, then the next-innermost, and so on, with each package considered to be "inner" to its parent package. A leading '.' (for example, .foo.bar.Baz) means to start from the outermost scope instead.

The protocol buffer compiler resolves all type names by parsing the imported .proto files. The code generator for each language knows how to refer to each type in that language, even if it has different scoping rules.

## Defining Services

If you want to use your message types with an RPC (Remote Procedure Call) system, you can define an RPC service interface in a .proto file and the protocol buffer compiler will generate service interface code and stubs in your chosen language. So, for example, if you want to define an RPC service with a method that takes your SearchRequest and returns a SearchResponse, you can define it in your .proto file as follows:

service SearchService {

rpc Search (SearchRequest) returns (SearchResponse);

}

The protocol compiler will then generate an abstract interface called SearchService and a corresponding "stub" implementation. The stub forwards all calls to an RpcChannel, which in turn is an abstract interface that you must define yourself in terms of your own RPC system. For example, you might implement an RpcChannel which serializes the message and sends it to a server via HTTP. In other words, the generated stub provides a type-safe interface for making protocol-buffer-based RPC calls, without locking you into any particular RPC implementation. So, in C++, you might end up with code like this:

using google::protobuf;

protobuf::RpcChannel\* channel;

protobuf::RpcController\* controller;

SearchService\* service;

SearchRequest request;

SearchResponse response;

void DoSearch() {

// You provide classes MyRpcChannel and MyRpcController, which implement

// the abstract interfaces protobuf::RpcChannel and protobuf::RpcController.

channel = new MyRpcChannel("somehost.example.com:1234");

controller = new MyRpcController;

// The protocol compiler generates the SearchService class based on the

// definition given above.

service = new SearchService::Stub(channel);

// Set up the request.

request.set\_query("protocol buffers");

// Execute the RPC.

service->Search(controller, request, response, protobuf::NewCallback(&Done));

}

void Done() {

delete service;

delete channel;

delete controller;

}

All service classes also implement the Service interface, which provides a way to call specific methods without knowing the method name or its input and output types at compile time. On the server side, this can be used to implement an RPC server with which you could register services.

using google::protobuf;

class ExampleSearchService : public SearchService {

public:

void Search(protobuf::RpcController\* controller,

const SearchRequest\* request,

SearchResponse\* response,

protobuf::Closure\* done) {

if (request->query() == "google") {

response->add\_result()->set\_url("http://www.google.com");

} else if (request->query() == "protocol buffers") {

response->add\_result()->set\_url("http://protobuf.googlecode.com");

}

done->Run();

}

};

int main() {

// You provide class MyRpcServer. It does not have to implement any

// particular interface; this is just an example.

MyRpcServer server;

protobuf::Service\* service = new ExampleSearchService;

server.ExportOnPort(1234, service);

server.Run();

delete service;

return 0;

}

There are a number of ongoing third-party projects to develop RPC implementations for Protocol Buffers. For a list of links to projects we know about, see the [third-party add-ons wiki page](http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns).

## Options

Individual declarations in a .proto file can be annotated with a number of options. Options do not change the overall meaning of a declaration, but may affect the way it is handled in a particular context. The complete list of available options is defined in google/protobuf/descriptor.proto.

Some options are file-level options, meaning they should be written at the top-level scope, not inside any message, enum, or service definition. Some options are message-level options, meaning they should be written inside message definitions. Some options are field-level options, meaning they should be written inside field definitions. Options can also be written on enum types, enum values, service types, and service methods; however, no useful options currently exist for any of these.

Here are a few of the most commonly used options:

* java\_package (file option): The package you want to use for your generated Java classes. If no explicit java\_package option is given in the .proto file, then by default the proto package (specified using the "package" keyword in the .proto file) will be used. However, proto packages generally do not make good Java packages since proto packages are not expected to start with reverse domain names. If not generating Java code, this option has no effect.

option java\_package = "com.example.foo";

* java\_outer\_classname (file option): The class name for the outermost Java class (and hence the file name) you want to generate. If no explicit java\_outer\_classname is specified in the .proto file, the class name will be constructed by converting the .proto file name to camel-case (so foo\_bar.proto becomes FooBar.java). If not generating Java code, this option has no effect.

option java\_outer\_classname = "Ponycopter";

* optimize\_for (file option): Can be set to SPEED, CODE\_SIZE, or LITE\_RUNTIME. This affects the C++ and Java code generators (and possibly third-party generators) in the following ways:
  + SPEED (default): The protocol buffer compiler will generate code for serializing, parsing, and performing other common operations on your message types. This code is extremely highly optimized.
  + CODE\_SIZE: The protocol buffer compiler will generate minimal classes and will rely on shared, reflection-based code to implement serialialization, parsing, and various other operations. The generated code will thus be much smaller than with SPEED, but operations will be slower. Classes will still implement exactly the same public API as they do in SPEED mode. This mode is most useful in apps that contain a very large number .proto files and do not need all of them to be blindingly fast.
  + LITE\_RUNTIME: The protocol buffer compiler will generate classes that depend only on the "lite" runtime library (libprotobuf-lite instead of libprotobuf). The lite runtime is much smaller than the full library (around an order of magnitude smaller) but omits certain features like descriptors and reflection. This is particularly useful for apps running on constrained platforms like mobile phones. The compiler will still generate fast implementations of all methods as it does in SPEED mode. Generated classes will only implement the MessageLite interface in each language, which provides only a subset of the methods of the full Message interface.

option optimize\_for = CODE\_SIZE;

* cc\_generic\_services, java\_generic\_services, py\_generic\_services (file options): Whether or not the protocol buffer compiler should generate abstract service code based on [services definitions](https://developers.google.com/protocol-buffers/docs/proto#services) in C++, Java, and Python, respectively. For legacy reasons, these default to true. However, as of version 2.3.0 (January 2010), it is considered preferrable for RPC implementations to provide [code generator plugins](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb) to generate code more specific to each system, rather than rely on the "abstract" services.
* // This file relies on plugins to generate service code.
* option cc\_generic\_services = false;
* option java\_generic\_services = false;

option py\_generic\_services = false;

* message\_set\_wire\_format (message option): If set to true, the message uses a different binary format intended to be compatible with an old format used inside Google called MessageSet. Users outside Google will probably never need to use this option. The message must be declared exactly as follows:
* message Foo {
* option message\_set\_wire\_format = true;
* extensions 4 to max;
* }
* packed (field option): If set to true on a repeated field of a basic integer type, a more compact encoding will be used. There is no downside to using this option. However, note that prior to version 2.3.0, parsers that received packed data when not expected would ignore it. Therefore, it was not possible to change an existing field to packed format without breaking wire compatibility. In 2.3.0 and later, this change is safe, as parsers for packable fields will always accept both formats, but be careful if you have to deal with old programs using old protobuf versions.

repeated int32 samples = 4 [packed=true];

* deprecated (field option): If set to true, indicates that the field is deprecated and should not be used by new code. In most languages this has no actual effect. In Java, this becomes a @Deprecated annotation. In the future, other language-specific code generators may generate deprecation annotations on the field's accessors, which will in turn cause a warning to be emitted when compiling code which attempts to use the field.

optional int32 old\_field = 6 [deprecated=true];

### Custom Options

Protocol Buffers even allow you to define and use your own options. Note that this is an **advanced feature** which most people don't need. Since options are defined by the messages defined in google/protobuf/descriptor.proto (like FileOptions or FieldOptions), defining your own options is simply a matter of [extending](https://developers.google.com/protocol-buffers/docs/proto#extensions) those messages. For example:

import "google/protobuf/descriptor.proto";

extend google.protobuf.MessageOptions {

optional string my\_option = 51234;

}

message MyMessage {

option (my\_option) = "Hello world!";

}

Here we have defined a new message-level option by extending MessageOptions. When we then use the option, the option name must be enclosed in parentheses to indicate that it is an extension. We can now read the value of my\_option in C++ like so:

string value = MyMessage::descriptor()->options().GetExtension(my\_option);

Here, MyMessage::descriptor()->options() returns the MessageOptions protocol message for MyMessage. Reading custom options from it is just like reading any other [extension](https://developers.google.com/protocol-buffers/docs/proto#extensions).

Similarly, in Java we would write:

String value = MyProtoFile.MyMessage.getDescriptor().getOptions()

.getExtension(MyProtoFile.myOption);

In Python it would be:

value = my\_proto\_file\_pb2.MyMessage.DESCRIPTOR.GetOptions()

.Extensions[my\_proto\_file\_pb2.my\_option]

Custom options can be defined for every kind of construct in the Protocol Buffers language. Here is an example that uses every kind of option:

import "google/protobuf/descriptor.proto";

extend google.protobuf.FileOptions {

optional string my\_file\_option = 50000;

}

extend google.protobuf.MessageOptions {

optional int32 my\_message\_option = 50001;

}

extend google.protobuf.FieldOptions {

optional float my\_field\_option = 50002;

}

extend google.protobuf.EnumOptions {

optional bool my\_enum\_option = 50003;

}

extend google.protobuf.EnumValueOptions {

optional uint32 my\_enum\_value\_option = 50004;

}

extend google.protobuf.ServiceOptions {

optional MyEnum my\_service\_option = 50005;

}

extend google.protobuf.MethodOptions {

optional MyMessage my\_method\_option = 50006;

}

option (my\_file\_option) = "Hello world!";

message MyMessage {

option (my\_message\_option) = 1234;

optional int32 foo = 1 [(my\_field\_option) = 4.5];

optional string bar = 2;

}

enum MyEnum {

option (my\_enum\_option) = true;

FOO = 1 [(my\_enum\_value\_option) = 321];

BAR = 2;

}

message RequestType {}

message ResponseType {}

service MyService {

option (my\_service\_option) = FOO;

rpc MyMethod(RequestType) returns(ResponseType) {

// Note: my\_method\_option has type MyMessage. We can set each field

// within it using a separate "option" line.

option (my\_method\_option).foo = 567;

option (my\_method\_option).bar = "Some string";

}

}

Note that if you want to use a custom option in a package other than the one in which it was defined, you must prefix the option name with the package name, just as you would for type names. For example:

// foo.proto

import "google/protobuf/descriptor.proto";

package foo;

extend google.protobuf.MessageOptions {

optional string my\_option = 51234;

}

// bar.proto

import "foo.proto";

package bar;

message MyMessage {

option (foo.my\_option) = "Hello world!";

}

One last thing: Since custom options are extensions, they must be assigned field numbers like any other field or extension. In the examples above, we have used field numbers in the range 50000-99999. This range is reserved for internal use within individual organizations, so you can use numbers in this range freely for in-house applications. If you intend to use custom options in public applications, however, then it is important that you make sure that your field numbers are globally unique. To obtain globally unique field numbers, please send a request to [protobuf-global-extension-registry@google.com](mailto:protobuf-global-extension-registry@google.com). Simply provide your project name (e.g. Object-C plugin) and your project website (if available). Usually you only need one extension number. You can declare multiple options with only one extension number by putting them in a sub-message:

message FooOptions {

optional int32 opt1 = 1;

optional string opt2 = 2;

}

extend google.protobuf.FieldOptions {

optional FooOptions foo\_options = 1234;

}

// usage:

message Bar {

optional int32 a = 1 [(foo\_options).opt1 = 123, (foo\_options).opt2 = "baz"];

// alternative aggregate syntax (uses TextFormat):

optional int32 b = 2 [(foo\_options) = { opt1: 123 opt2: "baz" }];

}

Also, note that each option type (file-level, message-level, field-level, etc.) has its own number space, so e.g. you could declare extensions of FieldOptions and MessageOptions with the same number.

## Generating Your Classes

To generate the Java, Python, or C++ code you need to work with the message types defined in a .proto file, you need to run the protocol buffer compiler protoc on the .proto. If you haven't installed the compiler, [download the package](http://code.google.com/p/protobuf/downloads/) and follow the instructions in the README.

The Protocol Compiler is invoked as follows:

protoc --proto\_path=IMPORT\_PATH --cpp\_out=DST\_DIR --java\_out=DST\_DIR --python\_out=DST\_DIR path/to/file.proto

* IMPORT\_PATH specifies a directory in which to look for .proto files when resolving import directives. If omitted, the current directory is used. Multiple import directories can be specified by passing the --proto\_path option multiple times; they will be searched in order. -I=IMPORT\_PATH can be used as a short form of --proto\_path.
* You can provide one or more output directives:
  + --cpp\_out generates C++ code in DST\_DIR. See the [C++ generated code reference](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated) for more.
  + --java\_out generates Java code in DST\_DIR. See the [Java generated code reference](https://developers.google.com/protocol-buffers/docs/reference/java-generated) for more.
  + --python\_out generates Python code in DST\_DIR. See the [Python generated code reference](https://developers.google.com/protocol-buffers/docs/reference/python-generated) for more.

As an extra convenience, if the DST\_DIR ends in .zip or .jar, the compiler will write the output to a single ZIP-format archive file with the given name. .jar outputs will also be given a manifest file as required by the Java JAR specification. Note that if the output archive already exists, it will be overwritten; the compiler is not smart enough to add files to an existing archive.

* You must provide one or more .proto files as input. Multiple .proto files can be specified at once. Although the files are named relative to the current directory, each file must reside in one of the IMPORT\_PATHs so that the compiler can determine its canonical name.

# Style Guide

This document provides a style guide for .proto files. By following these conventions, you'll make your protocol buffer message definitions and their corresponding classes consistent and easy to read.

## Message And Field Names

Use CamelCase (with an initial capital) for message names – for example, SongServerRequest. Use underscore\_separated\_names for field names – for example, song\_name.

message SongServerRequest {

required string song\_name = 1;

}

Using this naming convention for field names gives you accessors like the following:

C++:

const string& song\_name() { ... }

void set\_song\_name(const string& x) { ... }

Java:

public String getSongName() { ... }

public Builder setSongName(String v) { ... }

## Enums

Use CamelCase (with an initial capital) for enum type names and CAPITALS\_WITH\_UNDERSCORES for value names:

enum Foo {

FIRST\_VALUE = 1;

SECOND\_VALUE = 2;

}

Each enum value should end with a semicolon, not a comma.

## Services

If your .proto defines an RPC service, you should use CamelCase (with an initial capital) for both the service name and any RPC method names:

service FooService {

rpc GetSomething(FooRequest) returns (FooResponse);

}

# Encoding

* [A Simple Message](https://developers.google.com/protocol-buffers/docs/encoding#simple)
* [Base 128 Varints](https://developers.google.com/protocol-buffers/docs/encoding#varints)
* [Message Structure](https://developers.google.com/protocol-buffers/docs/encoding#structure)
* [More Value Types](https://developers.google.com/protocol-buffers/docs/encoding#types)
* [Embedded Messages](https://developers.google.com/protocol-buffers/docs/encoding#embedded)
* [Optional And Repeated Elements](https://developers.google.com/protocol-buffers/docs/encoding#optional)
* [Field Order](https://developers.google.com/protocol-buffers/docs/encoding#order)

This document describes the binary wire format for protocol buffer messages. You don't need to understand this to use protocol buffers in your applications, but it can be very useful to know how different protocol buffer formats affect the size of your encoded messages.

## A Simple Message

Let's say you have the following very simple message definition:

message Test1 {

required int32 a = 1;

}

In an application, you create a Test1 message and set a to 150. You then serialize the message to an output stream. If you were able to examine the encoded message, you'd see three bytes:

08 96 01

So far, so small and numeric – but what does it mean? Read on...

## Base 128 Varints

To understand your simple protocol buffer encoding, you first need to understand varints. Varints are a method of serializing integers using one or more bytes. Smaller numbers take a smaller number of bytes.

Each byte in a varint, except the last byte, has the most significant bit (msb) set – this indicates that there are further bytes to come. The lower 7 bits of each byte are used to store the two's complement representation of the number in groups of 7 bits, **least significant group first**.

So, for example, here is the number 1 – it's a single byte, so the msb is not set:

0000 0001

And here is 300 – this is a bit more complicated:

1010 1100 0000 0010

How do you figure out that this is 300? First you drop the msb from each byte, as this is just there to tell us whether we've reached the end of the number (as you can see, it's set in the first byte as there is more than one byte in the varint):

1010 1100 0000 0010

→ 010 1100 000 0010

You reverse the two groups of 7 bits because, as you remember, varints store numbers with the least significant group first. Then you concatenate them to get your final value:

000 0010 010 1100

→ 000 0010 ++ 010 1100

→ 100101100

→ 256 + 32 + 8 + 4 = 300

## Message Structure

As you know, a protocol buffer message is a series of key-value pairs. The binary version of a message just uses the field's number as the key – the name and declared type for each field can only be determined on the decoding end by referencing the message type's definition (i.e. the .proto file).

When a message is encoded, the keys and values are concatenated into a byte stream. When the message is being decoded, the parser needs to be able to skip fields that it doesn't recognize. This way, new fields can be added to a message without breaking old programs that do not know about them. To this end, the "key" for each pair in a wire-format message is actually two values – the field number from your .proto file, plus a wire type that provides just enough information to find the length of the following value.

The available wire types are as follows:

|  |  |  |
| --- | --- | --- |
| **Type** | **Meaning** | **Used For** |
| 0 | Varint | int32, int64, uint32, uint64, sint32, sint64, bool, enum |
|  |  |  |
| 1 | 64-bit | fixed64, sfixed64, double |
|  |  |  |
| 2 | Length-delimited | string, bytes, embedded messages, packed repeated fields |
|  |  |  |
| 3 | Start group | groups (deprecated) |
|  |  |  |
| 4 | End group | groups (deprecated) |
|  |  |  |
| 5 | 32-bit | fixed32, sfixed32, float |
|  |  |  |

Each key in the streamed message is a varint with the value (field\_number << 3) | wire\_type – in other words, the last three bits of the number store the wire type.

Now let's look at our simple example again. You now know that the first number in the stream is always a varint key, and here it's 08, or (dropping the msb):

000 1000

You take the last three bits to get the wire type (0) and then right-shift by three to get the field number (1). So you now know that the tag is 1 and the following value is a varint. Using your varint-decoding knowledge from the previous section, you can see that the next two bytes store the value 150.

96 01 = 1001 0110 0000 0001

→ 000 0001 ++ 001 0110 (drop the msb and reverse the groups of 7 bits)

→ 10010110

→ 2 + 4 + 16 + 128 = 150

## More Value Types

### Signed Integers

As you saw in the previous section, all the protocol buffer types associated with wire type 0 are encoded as varints. However, there is an important difference between the signed int types (sint32 and sint64) and the "standard" int types (int32 and int64) when it comes to encoding negative numbers. If you use int32 or int64 as the type for a negative number, the resulting varint is always ten bytes long – it is, effectively, treated like a very large unsigned integer. If you use one of the signed types, the resulting varint uses ZigZag encoding, which is much more efficient.

ZigZag encoding maps signed integers to unsigned integers so that numbers with a small absolute value (for instance, -1) have a small varint encoded value too. It does this in a way that "zig-zags" back and forth through the positive and negative integers, so that -1 is encoded as 1, 1 is encoded as 2, -2 is encoded as 3, and so on, as you can see in the following table:

|  |  |
| --- | --- |
| **Signed Original** | **Encoded As** |
| 0 | 0 |
|  |  |
| -1 | 1 |
|  |  |
| 1 | 2 |
|  |  |
| -2 | 3 |
|  |  |
| 2147483647 | 4294967294 |
|  |  |
| -2147483648 | 4294967295 |
|  |  |

In other words, each value n is encoded using

(n << 1) ^ (n >> 31)

for sint32s, or

(n << 1) ^ (n >> 63)

for the 64-bit version.

Note that the second shift – the (n >> 31) part – is an arithmetic shift. So, in other words, the result of the shift is either a number that is all zero bits (if n is positive) or all one bits (if n is negative).

When the sint32 or sint64 is parsed, its value is decoded back to the original, signed version.

### Non-varint Numbers

Non-varint numeric types are simple – double and fixed64 have wire type 1, which tells the parser to expect a fixed 64-bit lump of data; similarly float and fixed32 have wire type 5, which tells it to expect 32 bits. In both cases the values are stored in little-endian byte order.

### Strings

A wire type of 2 (length-delimited) means that the value is a varint encoded length followed by the specified number of bytes of data.

message Test2 {

required string b = 2;

}

Setting the value of b to "testing" gives you:

12 07 74 65 73 74 69 6e 67

The red bytes are the UTF8 of "testing". The key here is 0x12 → tag = 2, type = 2. The length varint in the value is 7 and lo and behold, we find seven bytes following it – our string.

## Embedded Messages

Here's a message definition with an embedded message of our example type, Test1:

message Test3 {

required Test1 c = 3;

}

And here's the encoded version, again with the Test1's a field set to 150:

1a 03 08 96 01

As you can see, the last three bytes are exactly the same as our first example (08 96 01), and they're preceded by the number 3 – embedded messages are treated in exactly the same way as strings (wire type = 2).

## Optional And Repeated Elements

If your message definition has repeated elements (without the [packed=true] option), the encoded message has zero or more key-value pairs with the same tag number. These repeated values do not have to appear consecutively; they may be interleaved with other fields. The order of the elements with respect to each other is preserved when parsing, though the ordering with respect to other fields is lost.

If any of your elements are optional, the encoded message may or may not have a key-value pair with that tag number.

Normally, an encoded message would never have more than one instance of an optional or required field. However, parsers are expected to handle the case in which they do. For numeric types and strings, if the same value appears multiple times, the parser accepts the *last* value it sees. For embedded message fields, the parser merges multiple instances of the same field, as if with the Message::MergeFrom method – that is, all singular scalar fields in the latter instance replace those in the former, singular embedded messages are merged, and repeated fields are concatenated. The effect of these rules is that parsing the concatenation of two encoded messages produces exactly the same result as if you had parsed the two messages separately and merged the resulting objects. That is, this:

MyMessage message;

message.ParseFromString(str1 + str2);

is equivalent to this:

MyMessage message, message2;

message.ParseFromString(str1);

message2.ParseFromString(str2);

message.MergeFrom(message2);

This property is occasionally useful, as it allows you to merge two messages even if you do not know their types.

### Packed Repeated Fields

Version 2.1.0 introduced packed repeated fields, which are declared like repeated fields but with the special [packed=true] option. These function like repeated fields, but are encoded differently. A packed repeated field containing zero elements does not appear in the encoded message. Otherwise, all of the elements of the field are packed into a single key-value pair with wire type 2 (length-delimited). Each element is encoded the same way it would be normally, except without a tag preceding it.

For example, imagine you have the message type:

message Test4 {

repeated int32 d = 4 [packed=true];

}

Now let's say you construct a Test4, providing the values 3, 270, and 86942 for the repeated field d. Then, the encoded form would be:

22 // tag (field number 4, wire type 2)

06 // payload size (6 bytes)

03 // first element (varint 3)

8E 02 // second element (varint 270)

9E A7 05 // third element (varint 86942)

Only repeated fields of primitive numeric types (types which use the varint, 32-bit, or 64-bit wire types) can be declared "packed".

Note that although there's usually no reason to encode more than one key-value pair for a packed repeated field, encoders must be prepared to accept multiple key-value pairs. In this case, the payloads should be concatenated. Each pair must contain a whole number of elements.

## Field Order

While you can use field numbers in any order in a .proto, when a message is serialized its known fields should be written sequentially by field number, as in the provided C++, Java, and Python serialization code. This allows parsing code to use optimizations that rely on field numbers being in sequence. However, protocol buffer parsers must be able to parse fields in any order, as not all messages are created by simply serializing an object – for instance, it's sometimes useful to merge two messages by simply concatenating them.

If a message has [unknown fields](https://developers.google.com/protocol-buffers/docs/proto.html#updating), the current Java and C++ implementations write them in arbitrary order after the sequentially-ordered known fields. The current Python implementation does not track unknown fields.

**Tutorials**

Each tutorial in this section shows you how to implement a simple application using protocol buffers in your favourite language, introducing you to the language's protocol buffer API as well as showing you the basics of creating and using [.proto files](https://developers.google.com/protocol-buffers/docs/proto). The complete sample code for each application is also provided.

The tutorials don't assume that you know anything about protocol buffers, but do assume that you are comfortable writing code in your chosen language, including using file I/O.

* [C++ Tutorial](https://developers.google.com/protocol-buffers/docs/cpptutorial)
* [Java Tutorial](https://developers.google.com/protocol-buffers/docs/javatutorial)
* [Python Tutorial](https://developers.google.com/protocol-buffers/docs/pythontutorial)

# Protocol Buffer Basics: Java

This tutorial provides a basic Java programmer's introduction to working with protocol buffers. By walking through creating a simple example application, it shows you how to

* Define message formats in a .proto file.
* Use the protocol buffer compiler.
* Use the Java protocol buffer API to write and read messages.

This isn't a comprehensive guide to using protocol buffers in Java. For more detailed reference information, see the [Protocol Buffer Language Guide](https://developers.google.com/protocol-buffers/docs/proto), the [Java API Reference](https://developers.google.com/protocol-buffers/docs/reference/java/index.html), the [Java Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/java-generated), and the [Encoding Reference](https://developers.google.com/protocol-buffers/docs/encoding).

## Why Use Protocol Buffers?

The example we're going to use is a very simple "address book" application that can read and write people's contact details to and from a file. Each person in the address book has a name, an ID, an email address, and a contact phone number.

How do you serialize and retrieve structured data like this? There are a few ways to solve this problem:

* Use Java Serialization. This is the default approach since it's built into the language, but it has a host of well-known problems (see Effective Java, by Josh Bloch pp. 213), and also doesn't work very well if you need to share data with applications written in C++ or Python.
* You can invent an ad-hoc way to encode the data items into a single string – such as encoding 4 ints as "12:3:-23:67". This is a simple and flexible approach, although it does require writing one-off encoding and parsing code, and the parsing imposes a small run-time cost. This works best for encoding very simple data.
* Serialize the data to XML. This approach can be very attractive since XML is (sort of) human readable and there are binding libraries for lots of languages. This can be a good choice if you want to share data with other applications/projects. However, XML is notoriously space intensive, and encoding/decoding it can impose a huge performance penalty on applications. Also, navigating an XML DOM tree is considerably more complicated than navigating simple fields in a class normally would be.

Protocol buffers are the flexible, efficient, automated solution to solve exactly this problem. With protocol buffers, you write a .proto description of the data structure you wish to store. From that, the protocol buffer compiler creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient binary format. The generated class provides getters and setters for the fields that make up a protocol buffer and takes care of the details of reading and writing the protocol buffer as a unit. Importantly, the protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format.

## Where to Find the Example Code

The example code is included in the source code package, under the "examples" directory. [Download it here.](http://code.google.com/p/protobuf/downloads/)

## Defining Your Protocol Format

To create your address book application, you'll need to start with a .proto file. The definitions in a .proto file are simple: you add a message for each data structure you want to serialize, then specify a name and a type for each field in the message. Here is the .proto file that defines your messages, addressbook.proto.

package tutorial;

option java\_package = "com.example.tutorial";

option java\_outer\_classname = "AddressBookProtos";

message Person {

required string name = 1;

required int32 id = 2;

optional string email = 3;

enum PhoneType {

MOBILE = 0;

HOME = 1;

WORK = 2;

}

message PhoneNumber {

required string number = 1;

optional PhoneType type = 2 [default = HOME];

}

repeated PhoneNumber phone = 4;

}

message AddressBook {

repeated Person person = 1;

}

As you can see, the syntax is similar to C++ or Java. Let's go through each part of the file and see what it does.

The .proto file starts with a package declaration, which helps to prevent naming conflicts between different projects. In Java, the package name is used as the Java package unless you have explicitly specified a java\_package, as we have here. Even if you do provide a java\_package, you should still define a normal package as well to avoid name collisions in the Protocol Buffers name space as well as in non-Java languages.

After the package declaration, you can see two options that are Java-specific: java\_package and java\_outer\_classname. java\_package specifies in what Java package name your generated classes should live. If you don't specify this explicitly, it simply matches the package name given by the package declaration, but these names usually aren't appropriate Java package names (since they usually don't start with a domain name). The java\_outer\_classname option defines the class name which should contain all of the classes in this file. If you don't give a java\_outer\_classname explicitly, it will be generated by converting the file name to camel case. For example, "my\_proto.proto" would, by default, use "MyProto" as the outer class name.

Next, you have your message definitions. A message is just an aggregate containing a set of typed fields. Many standard simple data types are available as field types, including bool, int32, float, double, and string. You can also add further structure to your messages by using other message types as field types – in the above example the Person message contains PhoneNumber messages, while the AddressBook message contains Person messages. You can even define message types nested inside other messages – as you can see, the PhoneNumber type is defined inside Person. You can also define enum types if you want one of your fields to have one of a predefined list of values – here you want to specify that a phone number can be one of MOBILE, HOME, or WORK.

The " = 1", " = 2" markers on each element identify the unique "tag" that field uses in the binary encoding. Tag numbers 1-15 require one less byte to encode than higher numbers, so as an optimization you can decide to use those tags for the commonly used or repeated elements, leaving tags 16 and higher for less-commonly used optional elements. Each element in a repeated field requires re-encoding the tag number, so repeated fields are particularly good candidates for this optimization.

Each field must be annotated with one of the following modifiers:

* required: a value for the field must be provided, otherwise the message will be considered "uninitialized". Trying to build an uninitialized message will throw a RuntimeException. Parsing an uninitialized message will throw an IOException. Other than this, a required field behaves exactly like an optional field.
* optional: the field may or may not be set. If an optional field value isn't set, a default value is used. For simple types, you can specify your own default value, as we've done for the phone number type in the example. Otherwise, a system default is used: zero for numeric types, the empty string for strings, false for bools. For embedded messages, the default value is always the "default instance" or "prototype" of the message, which has none of its fields set. Calling the accessor to get the value of an optional (or required) field which has not been explicitly set always returns that field's default value.
* repeated: the field may be repeated any number of times (including zero). The order of the repeated values will be preserved in the protocol buffer. Think of repeated fields as dynamically sized arrays.

**Required Is Forever** You should be very careful about marking fields as required. If at some point you wish to stop writing or sending a required field, it will be problematic to change the field to an optional field – old readers will consider messages without this field to be incomplete and may reject or drop them unintentionally. You should consider writing application-specific custom validation routines for your buffers instead. Some engineers at Google have come to the conclusion that using required does more harm than good; they prefer to use only optional and repeated. However, this view is not universal.

You'll find a complete guide to writing .proto files – including all the possible field types – in the [Protocol Buffer Language Guide](https://developers.google.com/protocol-buffers/docs/proto). Don't go looking for facilities similar to class inheritance, though – protocol buffers don't do that.

## Compiling Your Protocol Buffers

Now that you have a .proto, the next thing you need to do is generate the classes you'll need to read and write AddressBook (and hence Person and PhoneNumber) messages. To do this, you need to run the protocol buffer compiler protoc on your .proto:

1. If you haven't installed the compiler, [download the package](http://code.google.com/p/protobuf/downloads/) and follow the instructions in the README.
2. Now run the compiler, specifying the source directory (where your application's source code lives – the current directory is used if you don't provide a value), the destination directory (where you want the generated code to go; often the same as $SRC\_DIR), and the path to your .proto. In this case, you...:

protoc -I=$SRC\_DIR --java\_out=$DST\_DIR $SRC\_DIR/addressbook.proto

Because you want Java classes, you use the --java\_out option – similar options are provided for other supported languages.

This generates com/example/tutorial/AddressBookProtos.java in your specified destination directory.

## The Protocol Buffer API

Let's look at some of the generated code and see what classes and methods the compiler has created for you. If you look in AddressBookProtos.java, you can see that it defines a class called AddressBookProtos, nested within which is a class for each message you specified in addressbook.proto. Each class has its own Builder class that you use to create instances of that class. You can find out more about builders in the [Builders vs. Messages](https://developers.google.com/protocol-buffers/docs/javatutorial#builders) section below.

Both messages and builders have auto-generated accessor methods for each field of the message; messages have only getters while builders have both getters and setters. Here are some of the accessors for the Person class (implementations omitted for brevity):

// required string name = 1;

public boolean hasName();

public String getName();

// required int32 id = 2;

public boolean hasId();

public int getId();

// optional string email = 3;

public boolean hasEmail();

public String getEmail();

// repeated .tutorial.Person.PhoneNumber phone = 4;

public List<PhoneNumber> getPhoneList();

public int getPhoneCount();

public PhoneNumber getPhone(int index);

Meanwhile, Person.Builder has the same getters plus setters:

// required string name = 1;

public boolean hasName();

public java.lang.String getName();

public Builder setName(String value);

public Builder clearName();

// required int32 id = 2;

public boolean hasId();

public int getId();

public Builder setId(int value);

public Builder clearId();

// optional string email = 3;

public boolean hasEmail();

public String getEmail();

public Builder setEmail(String value);

public Builder clearEmail();

// repeated .tutorial.Person.PhoneNumber phone = 4;

public List<PhoneNumber> getPhoneList();

public int getPhoneCount();

public PhoneNumber getPhone(int index);

public Builder setPhone(int index, PhoneNumber value);

public Builder addPhone(PhoneNumber value);

public Builder addAllPhone(Iterable<PhoneNumber> value);

public Builder clearPhone();

As you can see, there are simple JavaBeans-style getters and setters for each field. There are also has getters for each singular field which return true if that field has been set. Finally, each field has a clear method that un-sets the field back to its empty state.

Repeated fields have some extra methods – a Count method (which is just shorthand for the list's size), getters and setters which get or set a specific element of the list by index, an add method which appends a new element to the list, and an addAll method which adds an entire container full of elements to the list.

Notice how these accessor methods use camel-case naming, even though the .proto file uses lowercase-with-underscores. This transformation is done automatically by the protocol buffer compiler so that the generated classes match standard Java style conventions. You should always use lowercase-with-underscores for field names in your .proto files; this ensures good naming practice in all the generated languages. See the [style guide](https://developers.google.com/protocol-buffers/docs/style) for more on good .proto style.

For more information on exactly what members the protocol compiler generates for any particular field definition, see the [Java generated code reference](https://developers.google.com/protocol-buffers/docs/reference/java-generated).

### Enums and Nested Classes

The generated code includes a PhoneType Java 5 enum, nested within Person:

public static enum PhoneType {

MOBILE(0, 0),

HOME(1, 1),

WORK(2, 2),

;

...

}

The nested type Person.PhoneNumber is generated, as you'd expect, as a nested class within Person.

### Builders vs. Messages

The message classes generated by the protocol buffer compiler are all *immutable*. Once a message object is constructed, it cannot be modified, just like a Java String. To construct a message, you must first construct a builder, set any fields you want to set to your chosen values, then call the builder's build() method.

You may have noticed that each method of the builder which modifies the message returns another builder. The returned object is actually the same builder on which you called the method. It is returned for convenience so that you can string several setters together on a single line of code.

Here's an example of how you would create an instance of Person:

Person john =

Person.newBuilder()

.setId(1234)

.setName("John Doe")

.setEmail("jdoe@example.com")

.addPhone(

Person.PhoneNumber.newBuilder()

.setNumber("555-4321")

.setType(Person.PhoneType.HOME))

.build();

### Standard Message Methods

Each message and builder class also contains a number of other methods that let you check or manipulate the entire message, including:

* isInitialized(): checks if all the required fields have been set.
* toString(): returns a human-readable representation of the message, particularly useful for debugging.
* mergeFrom(Message other): (builder only) merges the contents of other into this message, overwriting singular fields and concatenating repeated ones.
* clear(): (builder only) clears all the fields back to the empty state.

These methods implement the Message and Message.Builder interfaces shared by all Java messages and builders. For more information, see the [complete API documentation for Message](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message).

### Parsing and Serialization

Finally, each protocol buffer class has methods for writing and reading messages of your chosen type using the protocol buffer [binary format](https://developers.google.com/protocol-buffers/docs/encoding). These include:

* byte[] toByteArray();: serializes the message and returns a byte array containing its raw bytes.
* static Person parseFrom(byte[] data);: parses a message from the given byte array.
* void writeTo(OutputStream output);: serializes the message and writes it to an OutputStream.
* static Person parseFrom(InputStream input);: reads and parses a message from an InputStream.

These are just a couple of the options provided for parsing and serialization. Again, see the [Message API reference](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message) for a complete list.

**Protocol Buffers and O-O Design** Protocol buffer classes are basically dumb data holders (like structs in C++); they don't make good first class citizens in an object model. If you want to add richer behaviour to a generated class, the best way to do this is to wrap the generated protocol buffer class in an application-specific class. Wrapping protocol buffers is also a good idea if you don't have control over the design of the .proto file (if, say, you're reusing one from another project). In that case, you can use the wrapper class to craft an interface better suited to the unique environment of your application: hiding some data and methods, exposing convenience functions, etc. **You should never add behaviour to the generated classes by inheriting from them**. This will break internal mechanisms and is not good object-oriented practice anyway.

## Writing A Message

Now let's try using your protocol buffer classes. The first thing you want your address book application to be able to do is write personal details to your address book file. To do this, you need to create and populate instances of your protocol buffer classes and then write them to an output stream.

Here is a program which reads an AddressBook from a file, adds one new Person to it based on user input, and writes the new AddressBook back out to the file again. The parts which directly call or reference code generated by the protocol compiler are highlighted.

import com.example.tutorial.AddressBookProtos.AddressBook;

import com.example.tutorial.AddressBookProtos.Person;

import java.io.BufferedReader;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.InputStreamReader;

import java.io.IOException;

import java.io.PrintStream;

class AddPerson {

// This function fills in a Person message based on user input.

static Person PromptForAddress(BufferedReader stdin,

PrintStream stdout) throws IOException {

Person.Builder person = Person.newBuilder();

stdout.print("Enter person ID: ");

person.setId(Integer.valueOf(stdin.readLine()));

stdout.print("Enter name: ");

person.setName(stdin.readLine());

stdout.print("Enter email address (blank for none): ");

String email = stdin.readLine();

if (email.length() > 0) {

person.setEmail(email);

}

while (true) {

stdout.print("Enter a phone number (or leave blank to finish): ");

String number = stdin.readLine();

if (number.length() == 0) {

break;

}

Person.PhoneNumber.Builder phoneNumber =

Person.PhoneNumber.newBuilder().setNumber(number);

stdout.print("Is this a mobile, home, or work phone? ");

String type = stdin.readLine();

if (type.equals("mobile")) {

phoneNumber.setType(Person.PhoneType.MOBILE);

} else if (type.equals("home")) {

phoneNumber.setType(Person.PhoneType.HOME);

} else if (type.equals("work")) {

phoneNumber.setType(Person.PhoneType.WORK);

} else {

stdout.println("Unknown phone type. Using default.");

}

person.addPhone(phoneNumber);

}

return person.build();

}

// Main function: Reads the entire address book from a file,

// adds one person based on user input, then writes it back out to the same

// file.

public static void main(String[] args) throws Exception {

if (args.length != 1) {

System.err.println("Usage: AddPerson ADDRESS\_BOOK\_FILE");

System.exit(-1);

}

AddressBook.Builder addressBook = AddressBook.newBuilder();

// Read the existing address book.

try {

addressBook.mergeFrom(new FileInputStream(args[0]));

} catch (FileNotFoundException e) {

System.out.println(args[0] + ": File not found. Creating a new file.");

}

// Add an address.

addressBook.addPerson(

PromptForAddress(new BufferedReader(new InputStreamReader(System.in)),

System.out));

// Write the new address book back to disk.

FileOutputStream output = new FileOutputStream(args[0]);

addressBook.build().writeTo(output);

output.close();

}

}

## Reading A Message

Of course, an address book wouldn't be much use if you couldn't get any information out of it! This example reads the file created by the above example and prints all the information in it.

import com.example.tutorial.AddressBookProtos.AddressBook;

import com.example.tutorial.AddressBookProtos.Person;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.PrintStream;

class ListPeople {

// Iterates though all people in the AddressBook and prints info about them.

static void Print(AddressBook addressBook) {

for (Person person: addressBook.getPersonList()) {

System.out.println("Person ID: " + person.getId());

System.out.println(" Name: " + person.getName());

if (person.hasEmail()) {

System.out.println(" E-mail address: " + person.getEmail());

}

for (Person.PhoneNumber phoneNumber : person.getPhoneList()) {

switch (phoneNumber.getType()) {

case MOBILE:

System.out.print(" Mobile phone #: ");

break;

case HOME:

System.out.print(" Home phone #: ");

break;

case WORK:

System.out.print(" Work phone #: ");

break;

}

System.out.println(phoneNumber.getNumber());

}

}

}

// Main function: Reads the entire address book from a file and prints all

// the information inside.

public static void main(String[] args) throws Exception {

if (args.length != 1) {

System.err.println("Usage: ListPeople ADDRESS\_BOOK\_FILE");

System.exit(-1);

}

// Read the existing address book.

AddressBook addressBook =

AddressBook.parseFrom(new FileInputStream(args[0]));

Print(addressBook);

}

}

## Extending a Protocol Buffer

Sooner or later after you release the code that uses your protocol buffer, you will undoubtedly want to "improve" the protocol buffer's definition. If you want your new buffers to be backwards-compatible, and your old buffers to be forward-compatible – and you almost certainly do want this – then there are some rules you need to follow. In the new version of the protocol buffer:

* you must not change the tag numbers of any existing fields.
* you must not add or delete any required fields.
* you may delete optional or repeated fields.
* you may add new optional or repeated fields but you must use fresh tag numbers (i.e. tag numbers that were never used in this protocol buffer, not even by deleted fields).

(There are [some exceptions](https://developers.google.com/protocol-buffers/docs/proto.html#updating) to these rules, but they are rarely used.)

If you follow these rules, old code will happily read new messages and simply ignore any new fields. To the old code, optional fields that were deleted will simply have their default value, and deleted repeated fields will be empty. New code will also transparently read old messages. However, keep in mind that new optional fields will not be present in old messages, so you will need to either check explicitly whether they're set with has\_, or provide a reasonable default value in your .proto file with [default = value] after the tag number. If the default value is not specified for an optional element, a type-specific default value is used instead: for strings, the default value is the empty string. For booleans, the default value is false. For numeric types, the default value is zero. Note also that if you added a new repeated field, your new code will not be able to tell whether it was left empty (by new code) or never set at all (by old code) since there is no has\_ flag for it.

## Advanced Usage

Protocol buffers have uses that go beyond simple accessors and serialization. Be sure to explore the [Java API reference](https://developers.google.com/protocol-buffers/docs/reference/java/index.html) to see what else you can do with them.

One key feature provided by protocol message classes is reflection. You can iterate over the fields of a message and manipulate their values without writing your code against any specific message type. One very useful way to use reflection is for converting protocol messages to and from other encodings, such as XML or JSON. A more advanced use of reflection might be to find differences between two messages of the same type, or to develop a sort of "regular expressions for protocol messages" in which you can write expressions that match certain message contents. If you use your imagination, it's possible to apply Protocol Buffers to a much wider range of problems than you might initially expect!

Reflection is provided as part of the [Message](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message) and [Message.Builder](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message.Builder) interfaces.

**Techniques**

* [Streaming Multiple Messages](https://developers.google.com/protocol-buffers/docs/techniques#streaming)
* [Large Data Sets](https://developers.google.com/protocol-buffers/docs/techniques#large-data)
* [Union Types](https://developers.google.com/protocol-buffers/docs/techniques#union)
* [Self-describing Messages](https://developers.google.com/protocol-buffers/docs/techniques#self-description)

This page describes some commonly-used design patterns for dealing with Protocol Buffers. You can also send design and usage questions to the [Protocol Buffers discussion group](http://groups.google.com/group/protobuf).

**Streaming Multiple Messages**

If you want to write multiple messages to a single file or stream, it is up to you to keep track of where one message ends and the next begins. The Protocol Buffer wire format is not self-delimiting, so protocol buffer parsers cannot determine where a message ends on their own. The easiest way to solve this problem is to write the size of each message before you write the message itself. When you read the messages back in, you read the size, then read the bytes into a separate buffer, then parse from that buffer. (If you want to avoid copying bytes to a separate buffer, check out the CodedInputStream class (in both C++ and Java) which can be told to limit reads to a certain number of bytes.)

**Large Data Sets**

Protocol Buffers are not designed to handle large messages. As a general rule of thumb, if you are dealing in messages larger than a megabyte each, it may be time to consider an alternate strategy.

That said, Protocol Buffers are great for handling individual messages *within* a large data set. Usually, large data sets are really just a collection of small pieces, where each small piece may be a structured piece of data. Even though Protocol Buffers cannot handle the entire set at once, using Protocol Buffers to encode each piece greatly simplifies your problem: now all you need is to handle a set of byte strings rather than a set of structures.

Protocol Buffers do not include any built-in support for large data sets because different situations call for different solutions. Sometimes a simple list of records will do while other times you may want something more like a database. Each solution should be developed as a separate library, so that only those who need it need to pay the costs.

**Union Types**

You may sometimes want to send a message that could be one of several different types. However, protocol buffer parsers cannot necessarily determine the type of a message based on the contents alone. So how do you make sure that the recipient application knows how to decode your message? One solution is to create a wrapper message that has one optional field for each possible message type.

For example, if you have message types Foo, Bar, and Baz, you can combine them with a type like:

message OneMessage {

// One of the following will be filled in.

optional Foo foo = 1;

optional Bar bar = 2;

optional Baz baz = 3;

}

You may also want to have an enum field that identifies which message is filled in, so that you can switch on it:

message OneMessage {

enum Type { FOO = 1; BAR = 2; BAZ = 3; }

// Identifies which field is filled in.

required Type type = 1;

// One of the following will be filled in.

optional Foo foo = 2;

optional Bar bar = 3;

optional Baz baz = 4;

}

If you have a very large number of possible types, listing every one of them in your container type may be unwieldy. Instead, you should consider using [extensions](https://developers.google.com/protocol-buffers/docs/proto.html#extensions):

message OneMessage {

extensions 100 to max;

}

// Elsewhere...

extend OneMessage {

optional Foo foo\_ext = 100;

optional Bar bar\_ext = 101;

optional Baz baz\_ext = 102;

}

Note that you can use the ListFields reflection method (in C++, Java, and Python) to get a list of all fields present in the message, including extensions. You might use this as part of a scheme for registering handlers for diverse message types.

**Self-describing Messages**

Protocol Buffers do not contain descriptions of their own types. Thus, given only a raw message without the corresponding .proto file defining its type, it is difficult to extract any useful data.

However, note that the contents of a .proto file can itself be represented using protocol buffers. The file src/google/protobuf/descriptor.proto in the source code package defines the message types involved. protoc can output a FileDescriptorSet – which represents a set of .proto files – using the --descriptor\_set\_out option. With this, you could define a self-describing protocol message like so:

message SelfDescribingMessage {

// Set of .proto files which define the type.

required FileDescriptorSet proto\_files = 1;

// Name of the message type. Must be defined by one of the files in

// proto\_files.

required string type\_name = 2;

// The message data.

required bytes message\_data = 3;

}

By using classes like DynamicMessage (available in C++ and Java), you can then write tools which can manipulate SelfDescribingMessages.

All that said, the reason that this functionality is not included in the Protocol Buffer library is because we have never had a use for it inside Google.

# Java Generated Code

* [Compiler Invocation](https://developers.google.com/protocol-buffers/docs/reference/java-generated#invocation)
* [Packages](https://developers.google.com/protocol-buffers/docs/reference/java-generated#package)
* [Messages](https://developers.google.com/protocol-buffers/docs/reference/java-generated#message)
* [Fields](https://developers.google.com/protocol-buffers/docs/reference/java-generated#fields)
* [Enumerations](https://developers.google.com/protocol-buffers/docs/reference/java-generated#enum)
* [Extensions](https://developers.google.com/protocol-buffers/docs/reference/java-generated#extension)
* [Services](https://developers.google.com/protocol-buffers/docs/reference/java-generated#service)
* [Plugin Insertion Points](https://developers.google.com/protocol-buffers/docs/reference/java-generated#plugins)

This page describes exactly what Java code the protocol buffer compiler generates for any given protocol definition. You should read the [language guide](https://developers.google.com/protocol-buffers/docs/proto) 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](https://developers.google.com/protocol-buffers/docs/reference/java-generated#package), 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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message) interface. The class is declared final; no further subclassing is allowed. Foo extends [GeneratedMessage](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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()](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message.html#newBuilderForType%28%29) method.
* static [Descriptor](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Descriptors.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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message), such as [getField()](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message.html#getField%28com.google.protobuf.Descriptors.FieldDescriptor%29).
* 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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message.Builder) interface. Note that parseFrom() never throws [UninitializedMessageException](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/UninitializedMessageException); it throws [InvalidProtocolBufferException](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Message.Builder) interface. It extends the [GeneratedMessage.Builder](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/style)). 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](https://developers.google.com/protocol-buffers/docs/proto.html#scalar). 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](https://developers.google.com/protocol-buffers/docs/proto.html#scalar). 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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Descriptors.EnumValueDescriptor) getValueDescriptor(): Returns the value's descriptor, which contains information about the value's name, number, and type.
* [EnumDescriptor](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Descriptors.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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Descriptors.EnumValueDescriptor) descriptor): Returns the enum object corresponding to the given value descriptor. May be faster than valueOf(int).
* [EnumDescriptor](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Descriptors.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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/GeneratedMessage.ExtendableMessage) instead of the usual [GeneratedMessage](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/GeneratedMessage). Similarly, Foo's builder will extend [GeneratedMessage.ExtendableBuilder](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java-generated#invocation); 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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb) 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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/RpcController) controller, FooRequest request,

[RpcCallback](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/RpcCallback)<FooResponse> done);

The parameters are equivalent to the parameters of [Service.CallMethod()](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service.html#callMethod%28com.google.protobuf.Descriptors.MethodDescriptor,%20com.google.protobuf.RpcController,%20com.google.protobuf.Message,%20com.google.protobuf.RpcCallback%29), except that the method argument is implied and request and done specify their exact type.

Foo subclasses the [Service](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service) interface. The protocol buffer compiler automatically generates implementations of the methods of Service as follows:

* [getDescriptorForType](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service.html#getDescriptorForType%28%29): Returns the service's [ServiceDescriptor](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Descriptors.ServiceDescriptor).
* [callMethod](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service.html#callMethod%28com.google.protobuf.Descriptors.MethodDescriptor,%20com.google.protobuf.RpcController,%20com.google.protobuf.Message,%20com.google.protobuf.RpcCallback%29): 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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service.html#getRequestPrototype%28com.google.protobuf.Descriptors.MethodDescriptor%29) and [getRequestPrototype](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service.html#getRequestPrototype%28com.google.protobuf.Descriptors.MethodDescriptor%29): 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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Descriptors.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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service) interface. This is a problem because RPC server implementations are usually written to use abstract [Service](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/RpcChannel) channel): Constructs a new stub which sends requests on the given channel.
* [RpcChannel](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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()](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/RpcChannel.html#callMethod%28com.google.protobuf.Descriptors.MethodDescriptor,%20com.google.protobuf.RpcController,%20com.google.protobuf.Message,%20com.google.protobuf.Message,%20com.google.protobuf.RpcCallback%29).

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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/RpcChannel) and [RpcController](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/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](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/RpcController) controller, FooRequest request)

throws [ServiceException](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/ServiceException);

Analogous to non-blocking services, Foo.newReflectiveBlockingService(Foo.BlockingInterface) returns a [BlockingService](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/BlockingService) wrapping some Foo.BlockingInterface. Finally, Foo.BlockingStub returns a stub implementation of Foo.BlockingInterface that sends requests to a particular [BlockingRpcChannel](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/BlockingRpcChannel).

## Plugin Insertion Points

[Code generator plugins](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb) 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.