Protocol Buffers Doc from Google

Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.

**From Google Developers**

<https://developers.google.com/protocol-buffers/>

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code samples are licensed under the Apache 2.0 License. For details, see our Site Policies.

Last updated May 27, 2015.

**Creative Commons Attribution 3.0 License**

<https://creativecommons.org/licenses/by/3.0/>

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

# HOME

## What are protocol buffers?

Protocol buffers are Google's language-neutral, platform-neutral, extensible 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.

LEARN MORE

## Pick your favourite language

Protocol buffers currently supports generated code in Java, Python, and C++. With our new proto3 language version, you can also work with Go, JavaNano, and Ruby, with more languages to come.

C++ JAVA PYTHON

## How do I start?

Download and install the protocol buffer compiler.

Read the overview.

Try the tutorial for your chosen language.

## example

message Person {

required string name = 1;

required int32 id = 2;

optional string email = 3;

}

Person john = Person.newBuilder()

.setId(1234)

.setName("John Doe")

.setEmail("jdoe@example.com")

.build();

output = new FileOutputStream(args[0]);

john.writeTo(output);

Person john;

fstream input(argv[1],

ios::in | ios::binary);

john.ParseFromIstream(&input);

id = john.id();

name = john.name();

email = john.email();

# GUIDES

## Overview

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 name() andset\_name()) 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](https://developers.google.com/protocol-buffers/docs/downloads.html) – 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.

### Introducing proto3

Our most recent version 3 [alpha release](https://github.com/google/protobuf/releases) introduces a new language version - Protocol Buffers language version 3 (aka proto3), as well as some new features in our existing language version (aka proto2). Proto3 simplifies the protocol buffer language, both for ease of use and to make it available in a wider range of programming languages: our current alpha release lets you generate protocol buffer code in Java, C++, Python, JavaNano, and Ruby, with [some limitations](https://github.com/google/protobuf/releases). In addition you can generate proto3 code for Go using the latest Go protoc plugin, available from the [golang/protobuf](https://github.com/golang/protobuf) Github repository. More languages are in the pipeline.

We currently recommend trying proto3 only:

* if you want try using protocol buffers in one of our newly-supported languages.
* If you you want to try our new open-source RPC implementation [gRPC](http://github.com/grpc/grpc-common) (currently also in alpha release) – we recommend using proto3 for all new gRPC servers and clients as it avoids compatibility issues.

Note that the two language version APIs are not completely compatible. To avoid inconvenience to existing users, we will continue to support the previous language version in new protocol buffers releases.

You can see the major differences from the current default version in the [release notes](https://github.com/google/protobuf/releases) and learn about proto3 syntax in the [Proto3 Language Guide](https://developers.google.com/protocol-buffers/docs/proto3). Full documentation for proto3 is coming soon!

(If the names proto2 and proto3 seem a little confusing, it's because when we originally open-sourced protocol buffers it was actually Google's second version of the language – also known as proto2. This is also why our open source version number started from v2.0.0).

### 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.

## Developer Guide

### Language Guide (proto2)

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. It covers the **proto2**version of the protocol buffers language: for information on the newer **proto3** syntax, see the [Proto3 Language Guide](https://developers.google.com/protocol-buffers/docs/proto3).

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 throughFieldDescriptor::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 yourSearchResponse 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 specialBuilder 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 .protofile, 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 | bool |
| 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 typeCorpus:

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 theallow\_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 .protoand 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.

##### Using proto3 Message Types

It's possible to import [proto3](https://developers.google.com/protocol-buffers/docs/proto3) message types and use them in your proto2 messages, and vice versa. However, proto2 enums cannot be used in proto3 syntax.

#### Nested Types

You can define and use message types inside other message types, as in the following example – here theResult 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 aSearchResponse 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 SearchResponseabove, 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.
* 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 .protofiles 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, but cannot be oneofs or maps.

##### 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 throughFieldDescriptor::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.

#### Oneof

If you have a message with many optional fields and where at most one field will be set at the same time, you can enforce this behavior and save memory by using the oneof feature.

Oneof fields are like optional fields except all the fields in a oneof share memory, and at most one field can be set at the same time. Setting any member of the oneof automatically clears all the other members. You can check which value in a oneof is set (if any) using a special case() or WhichOneof() method, depending on your chosen language.

##### Using Oneof

To define a oneof in your .proto you use the oneof keyword followed by your oneof name, in this casetest\_oneof:

message SampleMessage {  
  oneof test\_oneof {  
     string name = 4;  
     SubMessage sub\_message = 9;  
  }  
}

You then add your oneof fields to the oneof definition. You can add fields of any type, but cannot use therequired, optional, or repeated keywords.

In your generated code, oneof fields have the same getters and setters as regular optional methods. You also get a special method for checking which value (if any) in the oneof is set. You can find out more about the oneof API for your chosen language in the relevant [API reference](https://developers.google.com/protocol-buffers/docs/reference/overview).

##### Oneof Features

* Setting a oneof field will automatically clear all other members of the oneof. So if you set several oneof fields, only the *last* field you set will still have a value.

SampleMessage message;  
message.set\_name("name");  
CHECK(message.has\_name());  
message.mutable\_sub\_message();   // Will clear name field.  
CHECK(!message.has\_name());

* If the parser encounters multiple members of the same oneof on the wire, only the last member seen is used in the parsed message.
* Extensions are not supported for oneof.
* A oneof cannot be repeated.
* Reflection APIs work for oneof fields.
* If you're using C++, make sure your code doesn't cause memory crashes. The following sample code will crash because sub\_message was already deleted by calling the set\_name() method.

SampleMessage message;  
SubMessage\* sub\_message = message.mutable\_sub\_message();  
message.set\_name("name");      // Will delete sub\_message  
sub\_message->set\_...            // Crashes here

* Again in C++, if you Swap() two messages with oneofs, each message will end up with the other’s oneof case: in the example below, msg1 will have a sub\_message and msg2 will have a name.

SampleMessage msg1;  
msg1.set\_name("name");  
SampleMessage msg2;  
msg2.mutable\_sub\_message();  
msg1.swap(&msg2);  
CHECK(msg1.has\_sub\_message());  
CHECK(msg2.has\_name());

##### Backwards-compatibility issues

Be careful when adding or removing oneof fields. If checking the value of a oneof returns None/NOT\_SET, it could mean that the oneof has not been set or it has been set to a field in a different version of the oneof. There is no way to tell the difference, since there's no way to know if an unknown field on the wire is a member of the oneof.

###### Tag Reuse Issues

* **Move optional fields into or out of a oneof**: You may lose some of your information (some fields will be cleared) after the message is serialized and parsed.
* **Delete a oneof field and add it back**: This may clear your currently set oneof field after the message is serialized and parsed.
* **Split or merge oneof**: This has similar issues to moving regular optional fields.

#### Maps

If you want to create an associative map as part of your data definition, protocol buffers provides a handy shortcut syntax:

map<key\_type, value\_type> map\_field = N;

...where the key\_type can be any integral or string type (so, any [scalar](https://developers.google.com/protocol-buffers/docs/proto#scalar) type except for floating point types andbytes). The value\_type can be any type.

So, for example, if you wanted to create a map of projects where each Project message is associated with a string key, you could define it like this:

map<string, Project> projects = 3;

The generated map API is currently available **only for C++ and Java**. You can find out more about the map API for your chosen language in the relevant [API reference](https://developers.google.com/protocol-buffers/docs/reference/overview).

##### Maps Features

* Extensions are not supported for maps.
* Maps cannot be repeated, optional, or required.
* Wire format ordering and map iteration ordering of map values is undefined, so you cannot rely on your map items being in a particular order.
* In text format, maps are sorted by key.

##### Backwards compatibility

The map syntax is equivalent to the following on the wire, so protocol buffers implementations that do not support maps can still handle your data:

message MapFieldEntry {  
  key\_type key = 1;  
  value\_type value = 2;  
}  
  
repeated MapFieldEntry map\_field = N;

#### 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\_packagein 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 yourSearchRequest and returns a SearchResponse, you can define it in your .proto file as follows:

service SearchService {

rpc Search (SearchRequest) returns (SearchResponse);

}

By default, 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 anRpcChannel 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;  
}

If you don't want to plug in your own existing RPC system, you can now use [gRPC](https://github.com/grpc/grpc-common): a language- and platform-neutral open source RPC system developed at Google. gRPC works particularly well with protocol buffers and lets you generate the relevant RPC code directly from your .proto files using a special protocol buffer compiler plugin. However, as there are potential compatibility issues between clients and servers generated with proto2 and proto3, we recommend that you use proto3 for defining gRPC services. You can find out more about proto3 syntax in the [Proto3 Language Guide](https://developers.google.com/protocol-buffers/docs/proto3). If you do want to use proto2 with gRPC, you need to use version 3.0.0 or higher of the protocol buffers compiler and libraries.

In addition to gRPC, there are also 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](https://github.com/google/protobuf/wiki/Third-Party-Add-ons).

#### 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 explicitjava\_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.protobecomes 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 SPEEDmode. 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;

* cc\_enable\_arenas (file option): Enables [arena allocation](https://developers.google.com/protocol-buffers/docs/reference/arenas) for C++ generated code.
* 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 @Deprecatedannotation. 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 ingoogle/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 forMyMessage. 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](https://developers.google.com/protocol-buffers/docs/downloads.html) 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 theIMPORT\_PATHs so that the compiler can determine its canonical name.

### Language Guide (proto3)

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. It covers the **proto3**version of the protocol buffers language: for information on the older **proto2** syntax, see the [Proto2 Language Guide](https://developers.google.com/protocol-buffers/docs/proto).

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 (currently proto2 only; more proto3 documentation is coming soon).

#### 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.

syntax = "proto3";

message SearchRequest {

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

}

* The first line of the file specifies that you're using proto3 syntax: if you don't do this the protocol buffer compiler will assume you are using [proto2](https://developers.google.com/protocol-buffers/docs/proto). This must be the first non-empty, non-comment line of the file.
* 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/proto3#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/proto3#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 throughFieldDescriptor::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

Message fields can be one of the following:

* singular: 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];

##### 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 yourSearchResponse message type, you could add it to the same .proto:

message SearchRequest {

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

}

message SearchResponse {

...

}

##### Adding Comments

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

message SearchRequest {

string query = 1;

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

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/proto3#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 specialBuilder 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.
* For **Go**, the compiler generates a .pb.go file with a type for each message type in your file.
* For **Ruby**, the compiler generates a .rb file with a Ruby module containing your message types.
* For **JavaNano**, the compiler output is similar to Java but there are no Builder classes.

You can find out more about using the APIs for each language by following the tutorial for your chosen language (proto3 versions coming soon). For even more API details, see the relevant [API reference](https://developers.google.com/protocol-buffers/docs/reference/overview) (proto3 versions also coming soon).

#### Scalar Value Types

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| .proto Type | Notes | C++  Type | Java  Type | Python  Type[2] | Go  Type | Ruby  Type |
| double |  | double | double | float | float64 | Float |
| float |  | float | float | float | float32 | 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 | int32 | Fixnum or Bignum (as required) |
| 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] | int64 | Bignum |
| uint32 | Uses variable-length encoding. | uint32 | int[1] | int/  long[3] | uint32 | Fixnum or Bignum (as required) |
| uint64 | Uses variable-length encoding. | uint64 | long[1] | int/  long[3] | uint64 | Bignum |
| sint32 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s. | int32 | int | int | int32 | Fixnum or Bignum (as required) |
| sint64 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s. | int64 | long | int/  long[3] | int64 | Bignum |
| fixed32 | Always four bytes. More efficient than uint32 if values are often greater than 228. | uint32 | int[1] | int | uint32 | Fixnum or Bignum (as required) |
| fixed64 | Always eight bytes. More efficient than uint64 if values are often greater than 256. | uint64 | long[1] | int/  long[3] | uint64 | Bignum |
| sfixed32 | Always four bytes. | int32 | int | int | int32 | Fixnum or Bignum (as required) |
| sfixed64 | Always eight bytes. | int64 | long | int/  long[3] | int64 | Bignum |
| bool |  | bool | boolean | boolean | bool | TrueClass/FalseClass |
| string | A string must always contain UTF-8 encoded or 7-bit ASCII text. | string | String | str/  Unicode[4] | string | String (UTF-8) |
| bytes | May contain any arbitrary sequence of bytes. | string | ByteString | str | []byte | String (ASCII-8BIT) |

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).

#### Default Values

When a message is parsed, if the encoded message does not contain a particular singular element, the corresponding field in the parsed object is set to the default value for that field. These defaults are type-specific:

* For strings, the default value is the empty string.
* For bytes, the default value is empty bytes.
* For bools, the default value is false.
* For numeric types, the default value is zero.
* For [enums](https://developers.google.com/protocol-buffers/docs/proto3#enum), the default value is the **first defined enum value**, which must be 0.
* For message fields, the default value is null.

The default value for repeated fields is empty (generally an empty list in the appropriate language).

Note that for scalar message fields, once a message is parsed there's no way of telling whether a field was explicitly set to the default value (for example whether a boolean was set to false) or just not set at all: you should bear this in mind when defining your message types. For example, don't have a boolean that switches on some behaviour when set to false if you don't want that behaviour to also happen by default. Also note that if a scalar message field **is** set to its default, the value will not be serialized on the wire.

#### 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 with a constant for each possible value.

In the following example we've added an enum called Corpus with all the possible values, and a field of typeCorpus:

message SearchRequest {

string query = 1;

int32 page\_number = 2;

int32 result\_per\_page = 3;

enum Corpus {

UNIVERSAL = 0;

WEB = 1;

IMAGES = 2;

LOCAL = 3;

NEWS = 4;

PRODUCTS = 5;

VIDEO = 6;

}

Corpus corpus = 4;

}

As you can see, the Corpus enum's first constant maps to zero: every enum definition **must** contain a constant that maps to zero as its first element. This is because:

* There must be a zero value, so that we can use 0 as a numeric [default value](https://developers.google.com/protocol-buffers/docs/proto3#default).
* The zero value needs to be the first element, for compatibility with the [proto2](https://developers.google.com/protocol-buffers/docs/proto) semantics where the first enum value is always the default.

You can define aliases by assigning the same value to different enum constants. To do this you need to set theallow\_alias option to true, otherwise the 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++, a special EnumDescriptor class for Python that's used to create a set of symbolic constants with integer values in the runtime-generated class.

During deserialization, unrecognized enum values will be preserved in the message, though how this is represented when the message is deserialized is language-dependent. In languages that support open enum types with values outside the range of specified symbols, such as C++ and Go, the unknown enum value is simply stored as its underlying integer representation. In languages with closed enum types such as Java, a case in the enum is used to represent an unrecognized value, and the underlying integer can be accessed with special accessors. In either case, if the message is serialized the unrecognized value will still be serialized with the message.

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 .protoand then specify a field of type Result in SearchResponse:

message SearchResponse {

repeated Result result = 1;

}

message Result {

string url = 1;

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.

##### Using proto2 Message Types

It's possible to import [proto2](https://developers.google.com/protocol-buffers/docs/proto) message types and use them in your proto3 messages, and vice versa. However, proto2 enums cannot be used in proto3 syntax.

#### Nested Types

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

message SearchResponse {

message Result {

string url = 1;

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 {

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

int64 ival = 1;

bool booly = 2;

}

}

message MiddleBB { // Level 1

message Inner { // Level 2

int32 ival = 1;

bool booly = 2;

}

}

}

#### 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.
* If you add new fields, any messages serialized by code using your "old" message format can still be parsed by your new generated code. You should keep in mind the [default values](https://developers.google.com/protocol-buffers/docs/proto3#default) 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. Note that unknown fields are discarded when the message is deserialized, so if the message is passed on to new code, the new fields will not still be available (this is different behaviour to [proto2](https://developers.google.com/protocol-buffers/docs/proto), where unknown fields are serialized along with the message).
* 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).
* 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.

#### Any

The Any message type lets you use messages as embedded types without having their .proto definition. AnAny contains an arbitrary serialized message as bytes, along with a URL that acts as a globally unique identifier for and resolves to that message's type. To use the Any type, you need to [import](https://developers.google.com/protocol-buffers/docs/proto3#other)google/protobuf/any.proto.

import "google/protobuf/any.proto";  
  
message ErrorStatus {  
  string message = 1;  
  repeated Any details = 2;  
}

The default type URL for a given message type is type.googleapis.com/*packagename*.*messagename*.

Different language implementations will support runtime library helpers to pack and unpack Any values in a typesafe manner – for example, in Java, the Any type will have special pack() and unpack() accessors, while in C++ there are PackFrom() and PackTo() methods:

// Storing an arbitrary message type in Any.  
NetworkErrorDetails details = ...;  
ErrorStatus status;  
status.add\_details()->PackFrom(details);  
  
// Reading an arbitrary message from Any.  
ErrorStatus status = ...;  
for (const Any& detail : status.details()) {  
  if (detail.IsType<NetworkErrorDetails>()) {  
    NetworkErrorDetails network\_error;  
    detail.UnpackTo(&network\_error);  
    ... processing network\_error ...  
  }  
}

**Currently the runtime libraries for working with Any types are under development**.

If you are already familiar with [proto2 syntax](https://developers.google.com/protocol-buffers/docs/proto), the Any type replaces [extensions](https://developers.google.com/protocol-buffers/docs/proto#extensions).

#### Oneof

If you have a message with many fields and where at most one field will be set at the same time, you can enforce this behavior and save memory by using the oneof feature.

Oneof fields are like regular fields except all the fields in a oneof share memory, and at most one field can be set at the same time. Setting any member of the oneof automatically clears all the other members. You can check which value in a oneof is set (if any) using a special case() or WhichOneof() method, depending on your chosen language.

##### Using Oneof

To define a oneof in your .proto you use the oneof keyword followed by your oneof name, in this casetest\_oneof:

message SampleMessage {  
  oneof test\_oneof {  
    string name = 4;  
    SubMessage sub\_message = 9;  
  }  
}

You then add your oneof fields to the oneof definition. You can add fields of any type, but cannot use repeatedfields.

In your generated code, oneof fields have the same getters and setters as regular fields. You also get a special method for checking which value (if any) in the oneof is set. You can find out more about the oneof API for your chosen language in the relevant [API reference](https://developers.google.com/protocol-buffers/docs/reference/overview).

##### Oneof Features

* Setting a oneof field will automatically clear all other members of the oneof. So if you set several oneof fields, only the *last* field you set will still have a value.

SampleMessage message;  
message.set\_name("name");  
CHECK(message.has\_name());  
message.mutable\_sub\_message();   // Will clear name field.  
CHECK(!message.has\_name());

* If the parser encounters multiple members of the same oneof on the wire, only the last member seen is used in the parsed message.
* A oneof cannot be repeated.
* Reflection APIs work for oneof fields.
* If you're using C++, make sure your code doesn't cause memory crashes. The following sample code will crash because sub\_message was already deleted by calling the set\_name() method.

SampleMessage message;  
SubMessage\* sub\_message = message.mutable\_sub\_message();  
message.set\_name("name");      // Will delete sub\_message  
sub\_message->set\_...            // Crashes here

* Again in C++, if you Swap() two messages with oneofs, each message will end up with the other’s oneof case: in the example below, msg1 will have a sub\_message and msg2 will have a name.

SampleMessage msg1;  
msg1.set\_name("name");  
SampleMessage msg2;  
msg2.mutable\_sub\_message();  
msg1.swap(&msg2);  
CHECK(msg1.has\_sub\_message());  
CHECK(msg2.has\_name());

##### Backwards-compatibility issues

Be careful when adding or removing oneof fields. If checking the value of a oneof returns None/NOT\_SET, it could mean that the oneof has not been set or it has been set to a field in a different version of the oneof. There is no way to tell the difference, since there's no way to know if an unknown field on the wire is a member of the oneof.

###### Tag Reuse Issues

* **Move fields into or out of a oneof**: You may lose some of your information (some fields will be cleared) after the message is serialized and parsed.
* **Delete a oneof field and add it back**: This may clear your currently set oneof field after the message is serialized and parsed.
* **Split or merge oneof**: This has similar issues to moving regular fields.

#### Maps

If you want to create an associative map as part of your data definition, protocol buffers provides a handy shortcut syntax:

map<key\_type, value\_type> map\_field = N;

...where the key\_type can be any integral or string type (so, any [scalar](https://developers.google.com/protocol-buffers/docs/proto3#scalar) type except for floating point types andbytes). The value\_type can be any type.

So, for example, if you wanted to create a map of projects where each Project message is associated with a string key, you could define it like this:

map<string, Project> projects = 3;

Map fields cannot be repeated. Also note that wire format ordering and map iteration ordering of map values is undefined, so you cannot rely on your map items being in a particular order.

The generated map API is currently available for all proto3 supported languages **except Python**. You can find out more about the map API for your chosen language in the relevant [API reference](https://developers.google.com/protocol-buffers/docs/reference/overview).

##### Backwards compatibility

The map syntax is equivalent to the following on the wire, so protocol buffers implementations that do not support maps can still handle your data:

message MapFieldEntry {  
  key\_type key = 1;  
  value\_type value = 2;  
}  
  
repeated MapFieldEntry map\_field = N;

#### 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 {

...

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 an 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.
* In **Go**, the package is used as the Go package name, unless you explicitly provide an option go\_package in your .proto file.
* In **Ruby**, the generated classes are wrapped inside nested Ruby namespaces, converted to the required Ruby capitalization style (first letter capitalized; if the first character is not a letter, PB\_ is prepended). For example, Open would be in the namespace Foo::Bar.
* In **JavaNano** the package is used as the Java package, unless you explicitly provide an option java\_package in your .proto file.

##### 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 yourSearchRequest and returns a SearchResponse, you can define it in your .proto file as follows:

service SearchService {

rpc Search (SearchRequest) returns (SearchResponse);

}

The most straightforward RPC system to use with protocol buffers is [gRPC](https://github.com/grpc/grpc-common): a language- and platform-neutral open source RPC system developed at Google. gRPC works particularly well with protocol buffers and lets you generate the relevant RPC code directly from your .proto files using a special protocol buffer compiler plugin.

If you don't want to use gRPC, it's also possible to use protocol buffers with your own RPC implementation. You can find out more about this in the [Proto2 Language Guide](https://developers.google.com/protocol-buffers/docs/proto#services).

There are also 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](https://github.com/google/protobuf/wiki/Third-Party-Add-ons).

#### JSON Mapping

Proto3 supports a canonical encoding in JSON, making it easier to share data between systems. The encoding is described on a type-by-type basis in the table below.

If a value is missing in the JSON-encoded data or if its value is null, it will be interpreted as the appropriate[default value](https://developers.google.com/protocol-buffers/docs/proto3#default) when parsed into a protocol buffer. If a field has the default value in the protocol buffer, it will be omitted in the JSON-encoded data by default to save space. An implementation may provide options to emit fields with default values in the JSON-encoded output.

|  |  |  |  |
| --- | --- | --- | --- |
| proto3 | JSON | JSON example | Notes |
| message | object | {"fBar": v,  "g": null,…} | Generates JSON objects. Message field names are mapped to lowerCamelCase and become JSON object keys. null is accepted and treated as the default value of the corresponding field type. |
| enum | string | "FOO\_BAR" | The name of the enum value as specified in proto is used. |
| map<K,V> | object | {"k": v, …} | All keys are converted to strings. |
| repeated V | array | [v, …] | null is accepted as the empty list []. |
| bool | true, false | true, false |  |
| string | string | "Hello World!" |  |
| bytes | base64 string |  |  |
| int32, fixed32, uint32 | number | 1, -10, 0 | JSON value will be a decimal number. Either numbers or strings are accepted. |
| int64, fixed64, uint64 | string | "1", "-10" | JSON value will be a decimal string. Either numbers or strings are accepted. |
| float, double | number | 1.1, -10.0,  0, "NaN",  "Infinity" | JSON value will be a number or one of the special string values "NaN", "Infinity", and "-Infinity". Either numbers or strings are accepted. Exponent notation is also accepted. |
| Any | object | {"@type": "url",  "f": v, … } | If the Any contains a value that has a special JSON mapping, it will be converted as follows: {"@type": xxx, "value": yyy}. Otherwise, the value will be converted into a JSON object, and the "@type" field will be inserted to indicate the actual data type. |
| Timestamp | string | "1972-01-01T  10:00:20.021Z" | Uses RFC 3339, where generated output will always be Z-normalized and uses 3, 6 or 9 fractional digits. |
| Duration | string | "1.000340012s",  "1s" | Generated output always contains 3, 6, or 9 fractional digits, depending on required precision. Accepted are any fractional digits (also none) as long as they fit into nano-seconds precision. |
| Struct | object | { … } | Any JSON object. See struct.proto. |
| Wrapper types | various types | 2, "2","foo",  true,"true",null,  0, … | Wrappers use the same representation in JSON as the wrapped primitive type, except that null is allowed and preserved during data conversion and transfer. |
| FieldMask | string | "f.fooBar,h" | See fieldmask.proto. |
| ListValue | array | [foo, bar, ...] |  |
| Value | value |  | Any JSON value |
| NullValue | null |  | JSON null |

#### 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 explicitjava\_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.protobecomes 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 SPEEDmode. 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\_enable\_arenas (file option): Enables [arena allocation](https://developers.google.com/protocol-buffers/docs/reference/arenas) for C++ generated code.
* 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 @Deprecatedannotation. 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.

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

##### Custom Options

Protocol Buffers also allows you to define and use your own options. This is an **advanced feature** which most people don't need. If you do think you need to create your own options, see the [Proto2 Language Guide](https://developers.google.com/protocol-buffers/docs/proto.html#customoptions) for details. Note that creating custom options uses [extensions](https://developers.google.com/protocol-buffers/docs/proto.html#extensions), which are permitted only for custom options in proto3.

#### Generating Your Classes

To generate the Java, Python, C++, Go, Ruby, or JavaNano 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](https://developers.google.com/protocol-buffers/docs/downloads.html) and follow the instructions in the README. For Go, you also need to install a special code generator plugin for the compiler: you can find this and installation instructions in the[golang/protobuf](https://github.com/golang/protobuf/) repository on GitHub.

The Protocol Compiler is invoked as follows:

protoc --proto\_path=*IMPORT\_PATH* --cpp\_out=*DST\_DIR* --java\_out=*DST\_DIR* --python\_out=*DST\_DIR* --go\_out=*DST\_DIR* --ruby\_out=*DST\_DIR* --javanano\_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.
  + --go\_out generates Go code in DST\_DIR. Go generated code reference is coming soon!
  + --ruby\_out generates Ruby code in DST\_DIR. Ruby generated code reference is coming soon!
  + --javanano\_out generates JavaNano code in DST\_DIR. The JavaNano code generator has a number of options you can use to customize the generator output: you can find out more about these in the generator [README](https://github.com/google/protobuf/tree/master/javanano). JavaNano generated code reference is coming soon!

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 theIMPORT\_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

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 (ifn 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 orrequired 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.

### Techniques

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.

### Add-ons

Many open source projects seek to add useful functionality on top of Protocol Buffers. For a list of links to projects we know about, see the [third-party add-ons wiki page](https://github.com/google/protobuf/wiki/Third-Party-Add-ons).

## Tutorials

### Tutorials Overview

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.

### Basics: C++

This tutorial provides a basic C++ 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 C++ protocol buffer API to write and read messages.

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

* The raw in-memory data structures can be sent/saved in binary form. Over time, this is a fragile approach, as the receiving/reading code must be compiled with exactly the same memory layout, endianness, etc. Also, as files accumulate data in the raw format and copies of software that are wired for that format are spread around, it's very hard to extend the format.
* 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.](https://developers.google.com/protocol-buffers/docs/downloads.html)

#### 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;

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 C++, your generated classes will be placed in a namespace matching the package 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, andstring. 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 AddressBookmessage 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". If libprotobuf is compiled in debug mode, serializing an uninitialized message will cause an assertion failure. In optimized builds, the check is skipped and the message will be written anyway. However, parsing an uninitialized message will always fail (by returning false from the parse method). 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 writeAddressBook (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](https://developers.google.com/protocol-buffers/docs/downloads.html) 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=$SRC\_DIR --cpp\_out=$DST\_DIR $SRC\_DIR/addressbook.proto

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

This generates the following files in your specified destination directory:

* addressbook.pb.h, the header which declares your generated classes.
* addressbook.pb.cc, which contains the implementation of your classes.

#### The Protocol Buffer API

Let's look at some of the generated code and see what classes and functions the compiler has created for you. If you look in tutorial.pb.h, you can see that you have a class for each message you specified intutorial.proto. Looking closer at the Person class, you can see that the complier has generated accessors for each field. For example, for the name, id, email, and phone fields, you have these methods:

  // name  
  inline bool has\_name() const;  
  inline void clear\_name();  
  inline const ::std::string& name() const;  
  inline void set\_name(const ::std::string& value);  
  inline void set\_name(const char\* value);  
  inline ::std::string\* mutable\_name();  
  
  // id  
  inline bool has\_id() const;  
  inline void clear\_id();  
  inline int32\_t id() const;  
  inline void set\_id(int32\_t value);  
  
  // email  
  inline bool has\_email() const;  
  inline void clear\_email();  
  inline const ::std::string& email() const;  
  inline void set\_email(const ::std::string& value);  
  inline void set\_email(const char\* value);  
  inline ::std::string\* mutable\_email();  
  
  // phone  
  inline int phone\_size() const;  
  inline void clear\_phone();  
  inline const ::google::protobuf::[**RepeatedPtrField**](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field.html#RepeatedPtrField)< ::tutorial::Person\_PhoneNumber >& phone() const;  
  inline ::google::protobuf::[**RepeatedPtrField**](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field.html#RepeatedPtrField)< ::tutorial::Person\_PhoneNumber >\* mutable\_phone();  
  inline const ::tutorial::Person\_PhoneNumber& phone(int index) const;  
  inline ::tutorial::Person\_PhoneNumber\* mutable\_phone(int index);  
  inline ::tutorial::Person\_PhoneNumber\* add\_phone();

As you can see, the getters have exactly the name as the field in lowercase, and the setter methods begin withset\_. There are also has\_ methods for each singular (required or optional) 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.

While the numeric id field just has the basic accessor set described above, the name and email fields have a couple of extra methods because they're strings – a mutable\_ getter that lets you get a direct pointer to the string, and an extra setter. Note that you can call mutable\_email() even if email is not already set; it will be initialized to an empty string automatically. If you had a singular message field in this example, it would also have a mutable\_ method but not a set\_ method.

Repeated fields also have some special methods – if you look at the methods for the repeated phone field, you'll see that you can

* check the repeated field's \_size (in other words, how many phone numbers are associated with thisPerson).
* get a specified phone number using its index.
* update an existing phone number at the specified index.
* add another phone number to the message which you can then edit (repeated scalar types have an add\_that just lets you pass in the new value).

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

##### Enums and Nested Classes

The generated code includes a PhoneType enum that corresponds to your .proto enum. You can refer to this type as Person::PhoneType and its values as Person::MOBILE, Person::HOME, and Person::WORK(the implementation details are a little more complicated, but you don't need to understand them to use the enum).

The compiler has also generated a nested class for you called Person::PhoneNumber. If you look at the code, you can see that the "real" class is actually called Person\_PhoneNumber, but a typedef defined inside Personallows you to treat it as if it were a nested class. The only case where this makes a difference is if you want to forward-declare the class in another file – you cannot forward-declare nested types in C++, but you can forward-declare Person\_PhoneNumber.

##### Standard Message Methods

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

* bool IsInitialized() const;: checks if all the required fields have been set.
* string DebugString() const;: returns a human-readable representation of the message, particularly useful for debugging.
* void CopyFrom(const Person& from);: overwrites the message with the given message's values.
* void Clear();: clears all the elements back to the empty state.

These and the I/O methods described in the following section implement the Message interface shared by all C++ protocol buffer classes. For more info, see the [complete API documentation for Message](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#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:

* bool SerializeToString(string\* output) const;: serializes the message and stores the bytes in the given string. Note that the bytes are binary, not text; we only use the string class as a convenient container.
* bool ParseFromString(const string& data);: parses a message from the given string.
* bool SerializeToOstream(ostream\* output) const;: writes the message to the given C++ostream.
* bool ParseFromIstream(istream\* input);: parses a message from the given C++ istream.

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/cpp/google.protobuf.message.html#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.

#include <iostream>  
#include <fstream>  
#include <string>  
#include "addressbook.pb.h"  
using namespace std;  
  
// This function fills in a Person message based on user input.  
void PromptForAddress(tutorial::Person\* person) {  
  cout << "Enter person ID number: ";  
  int id;  
  cin >> id;  
  person->set\_id(id);  
  cin.ignore(256, '\n');  
  
  cout << "Enter name: ";  
  getline(cin, \*person->mutable\_name());  
  
  cout << "Enter email address (blank for none): ";  
  string email;  
  getline(cin, email);  
  if (!email.empty()) {  
    person->set\_email(email);  
  }  
  
  while (true) {  
    cout << "Enter a phone number (or leave blank to finish): ";  
    string number;  
    getline(cin, number);  
    if (number.empty()) {  
      break;  
    }  
  
    tutorial::Person::PhoneNumber\* phone\_number = person->add\_phone();  
    phone\_number->set\_number(number);  
  
    cout << "Is this a mobile, home, or work phone? ";  
    string type;  
    getline(cin, type);  
    if (type == "mobile") {  
      phone\_number->set\_type(tutorial::Person::MOBILE);  
    } else if (type == "home") {  
      phone\_number->set\_type(tutorial::Person::HOME);  
    } else if (type == "work") {  
      phone\_number->set\_type(tutorial::Person::WORK);  
    } else {  
      cout << "Unknown phone type.  Using default." << endl;  
    }  
  }  
}  
  
// 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.  
int main(int argc, char\* argv[]) {  
  // Verify that the version of the library that we linked against is  
  // compatible with the version of the headers we compiled against.  
  GOOGLE\_PROTOBUF\_VERIFY\_VERSION;  
  
  if (argc != 2) {  
    cerr << "Usage:  " << argv[0] << " ADDRESS\_BOOK\_FILE" << endl;  
    return -1;  
  }  
  
  tutorial::AddressBook address\_book;  
  
  {  
    // Read the existing address book.  
    fstream input(argv[1], ios::in | ios::binary);  
    if (!input) {  
      cout << argv[1] << ": File not found.  Creating a new file." << endl;  
    } else if (!address\_book.ParseFromIstream(&input)) {  
      cerr << "Failed to parse address book." << endl;  
      return -1;  
    }  
  }  
  
  // Add an address.  
  PromptForAddress(address\_book.add\_person());  
  
  {  
    // Write the new address book back to disk.  
    fstream output(argv[1], ios::out | ios::trunc | ios::binary);  
    if (!address\_book.SerializeToOstream(&output)) {  
      cerr << "Failed to write address book." << endl;  
      return -1;  
    }  
  }  
  
  // Optional:  Delete all global objects allocated by libprotobuf.  
  google::protobuf::ShutdownProtobufLibrary();  
  
  return 0;  
}

Notice the GOOGLE\_PROTOBUF\_VERIFY\_VERSION macro. It is good practice – though not strictly necessary – to execute this macro before using the C++ Protocol Buffer library. It verifies that you have not accidentally linked against a version of the library which is incompatible with the version of the headers you compiled with. If a version mismatch is detected, the program will abort. Note that every .pb.cc file automatically invokes this macro on startup.

Also notice the call to ShutdownProtobufLibrary() at the end of the program. All this does is delete any global objects that were allocated by the Protocol Buffer library. This is unnecessary for most programs, since the process is just going to exit anyway and the OS will take care of reclaiming all of its memory. However, if you use a memory leak checker that requires that every last object be freed, or if you are writing a library which may be loaded and unloaded multiple times by a single process, then you may want to force Protocol Buffers to clean up everything.

#### 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.

#include <iostream>  
#include <fstream>  
#include <string>  
#include "addressbook.pb.h"  
using namespace std;  
  
// Iterates though all people in the AddressBook and prints info about them.  
void ListPeople(const tutorial::AddressBook& address\_book) {  
  for (int i = 0; i < address\_book.person\_size(); i++) {  
    const tutorial::Person& person = address\_book.person(i);  
  
    cout << "Person ID: " << person.id() << endl;  
    cout << "  Name: " << person.name() << endl;  
    if (person.has\_email()) {  
      cout << "  E-mail address: " << person.email() << endl;  
    }  
  
    for (int j = 0; j < person.phone\_size(); j++) {  
      const tutorial::Person::PhoneNumber& phone\_number = person.phone(j);  
  
      switch (phone\_number.type()) {  
        case tutorial::Person::MOBILE:  
          cout << "  Mobile phone #: ";  
          break;  
        case tutorial::Person::HOME:  
          cout << "  Home phone #: ";  
          break;  
        case tutorial::Person::WORK:  
          cout << "  Work phone #: ";  
          break;  
      }  
      cout << phone\_number.number() << endl;  
    }  
  }  
}  
  
// Main function:  Reads the entire address book from a file and prints all  
//   the information inside.  
int main(int argc, char\* argv[]) {  
  // Verify that the version of the library that we linked against is  
  // compatible with the version of the headers we compiled against.  
  GOOGLE\_PROTOBUF\_VERIFY\_VERSION;  
  
  if (argc != 2) {  
    cerr << "Usage:  " << argv[0] << " ADDRESS\_BOOK\_FILE" << endl;  
    return -1;  
  }  
  
  tutorial::AddressBook address\_book;  
  
  {  
    // Read the existing address book.  
    fstream input(argv[1], ios::in | ios::binary);  
    if (!address\_book.ParseFromIstream(&input)) {  
      cerr << "Failed to parse address book." << endl;  
      return -1;  
    }  
  }  
  
  ListPeople(address\_book);  
  
  // Optional:  Delete all global objects allocated by libprotobuf.  
  google::protobuf::ShutdownProtobufLibrary();  
  
  return 0;  
}

#### 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.

#### Optimization Tips

The C++ Protocol Buffers library is extremely heavily optimized. However, proper usage can improve performance even more. Here are some tips for squeezing every last drop of speed out of the library:

* Reuse message objects when possible. Messages try to keep around any memory they allocate for reuse, even when they are cleared. Thus, if you are handling many messages with the same type and similar structure in succession, it is a good idea to reuse the same message object each time to take load off the memory allocator. However, objects can become bloated over time, especially if your messages vary in "shape" or if you occasionally construct a message that is much larger than usual. You should monitor the sizes of your message objects by calling the [SpaceUsed](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.SpaceUsed.details) method and delete them once they get too big.
* Your system's memory allocator may not be well-optimized for allocating lots of small objects from multiple threads. Try using [Google's tcmalloc](http://code.google.com/p/google-perftools/) instead.

#### Advanced Usage

Protocol buffers have uses that go beyond simple accessors and serialization. Be sure to explore the [C++ API reference](https://developers.google.com/protocol-buffers/docs/reference/cpp/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 by the [Message::Reflection interface](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.Reflection).

### 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.](https://developers.google.com/protocol-buffers/docs/downloads.html)

#### 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 ajava\_package, as we have here. Even if you do provide a java\_package, you should still define a normalpackage 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 andjava\_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 packagedeclaration, 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, andstring. 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 AddressBookmessage 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 writeAddressBook (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](https://developers.google.com/protocol-buffers/docs/downloads.html) 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 ownBuilder 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 Personclass (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 anOutputStream.
* static Person parseFrom(InputStream input);: reads and parses a message from anInputStream.

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.

### Basics: Python

This tutorial provides a basic Python 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 Python protocol buffer API to write and read messages.

This isn't a comprehensive guide to using protocol buffers in Python. For more detailed reference information, see the [Protocol Buffer Language Guide](https://developers.google.com/protocol-buffers/docs/proto), the [Python API Reference](https://developers.google.com/protocol-buffers/docs/reference/python/index.html), the [Python Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/python-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 Python pickling. This is the default approach since it's built into the language, but it doesn't deal well with schema evolution, and also doesn't work very well if you need to share data with application written in C++ or Java.
* 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.](https://developers.google.com/protocol-buffers/docs/downloads.html)

#### 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;

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 Python, packages are normally determined by directory structure, so the package you define in your.proto file will have no effect on the generated code. However, you should still declare one to avoid name collisions in the Protocol Buffers name space as well as in non-Python languages.

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, andstring. 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 AddressBookmessage 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". Serializing an uninitialized message will raise an exception. Parsing an uninitialized message will fail. 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 writeAddressBook (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](https://developers.google.com/protocol-buffers/docs/downloads.html) 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=$SRC\_DIR --python\_out=$DST\_DIR $SRC\_DIR/addressbook.proto

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

This generates addressbook\_pb2.py in your specified destination directory.

#### The Protocol Buffer API

Unlike when you generate Java and C++ protocol buffer code, the Python protocol buffer compiler doesn't generate your data access code for you directly. Instead (as you'll see if you look at addressbook\_pb2.py) it generates special descriptors for all your messages, enums, and fields, and some mysteriously empty classes, one for each message type:

class Person(message.Message):

\_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType

class PhoneNumber(message.Message):

\_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType

DESCRIPTOR = \_PERSON\_PHONENUMBER

DESCRIPTOR = \_PERSON

class AddressBook(message.Message):

\_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType

DESCRIPTOR = \_ADDRESSBOOK

The important line in each class is \_\_metaclass\_\_ = reflection.GeneratedProtocolMessageType. While the details of how Python metaclasses work is beyond the scope of this tutorial, you can think of them as like a template for creating classes. At load time, the GeneratedProtocolMessageType metaclass uses the specified descriptors to create all the Python methods you need to work with each message type and adds them to the relevant classes. You can then use the fully-populated classes in your code.

The end effect of all this is that you can use the Person class as if it defined each field of the Message base class as a regular field. For example, you could write:

import addressbook\_pb2

person = addressbook\_pb2.Person()

person.id = 1234

person.name = "John Doe"

person.email = "jdoe@example.com"

phone = person.phone.add()

phone.number = "555-4321"

phone.type = addressbook\_pb2.Person.HOME

Note that these assignments are not just adding arbitrary new fields to a generic Python object. If you were to try to assign a field that isn't defined in the .proto file, an AttributeError would be raised. If you assign a field to a value of the wrong type, a TypeError will be raised. Also, reading the value of a field before it has been set returns the default value.

person.no\_such\_field = 1 # raises AttributeError

person.id = "1234" # raises TypeError

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

##### Enums

Enums are expanded by the metaclass into a set of symbolic constants with integer values. So, for example, the constant addressbook\_pb2.Person.WORK has the value 2.

##### Standard Message Methods

Each message 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.
* \_\_str\_\_(): returns a human-readable representation of the message, particularly useful for debugging. (Usually invoked as str(message) or print message.)
* CopyFrom(other\_msg): overwrites the message with the given message's values.
* Clear(): clears all the elements back to the empty state.

These methods implement the Message interface. For more information, see the [complete API documentation for Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class).

##### 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:

* SerializeToString(): serializes the message and returns it as a string. Note that the bytes are binary, not text; we only use the str type as a convenient container.
* ParseFromString(data): parses a message from the given string.

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/python/google.protobuf.message.Message-class) 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.

#! /usr/bin/python

import addressbook\_pb2

import sys

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

def PromptForAddress(person):

person.id = int(raw\_input("Enter person ID number: "))

person.name = raw\_input("Enter name: ")

email = raw\_input("Enter email address (blank for none): ")

if email != "":

person.email = email

while True:

number = raw\_input("Enter a phone number (or leave blank to finish): ")

if number == "":

break

phone\_number = person.phone.add()

phone\_number.number = number

type = raw\_input("Is this a mobile, home, or work phone? ")

if type == "mobile":

phone\_number.type = addressbook\_pb2.Person.MOBILE

elif type == "home":

phone\_number.type = addressbook\_pb2.Person.HOME

elif type == "work":

phone\_number.type = addressbook\_pb2.Person.WORK

else:

print "Unknown phone type; leaving as default value."

# Main procedure: Reads the entire address book from a file,

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

# file.

if len(sys.argv) != 2:

print "Usage:", sys.argv[0], "ADDRESS\_BOOK\_FILE"

sys.exit(-1)

address\_book = addressbook\_pb2.AddressBook()

# Read the existing address book.

try:

f = open(sys.argv[1], "rb")

address\_book.ParseFromString(f.read())

f.close()

except IOError:

print sys.argv[1] + ": Could not open file. Creating a new one."

# Add an address.

PromptForAddress(address\_book.person.add())

# Write the new address book back to disk.

f = open(sys.argv[1], "wb")

f.write(address\_book.SerializeToString())

f.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.

#! /usr/bin/python

import addressbook\_pb2

import sys

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

def ListPeople(address\_book):

for person in address\_book.person:

print "Person ID:", person.id

print " Name:", person.name

if person.HasField('email'):

print " E-mail address:", person.email

for phone\_number in person.phone:

if phone\_number.type == addressbook\_pb2.Person.MOBILE:

print " Mobile phone #: ",

elif phone\_number.type == addressbook\_pb2.Person.HOME:

print " Home phone #: ",

elif phone\_number.type == addressbook\_pb2.Person.WORK:

print " Work phone #: ",

print phone\_number.number

# Main procedure: Reads the entire address book from a file and prints all

# the information inside.

if len(sys.argv) != 2:

print "Usage:", sys.argv[0], "ADDRESS\_BOOK\_FILE"

sys.exit(-1)

address\_book = addressbook\_pb2.AddressBook()

# Read the existing address book.

f = open(sys.argv[1], "rb")

address\_book.ParseFromString(f.read())

f.close()

ListPeople(address\_book)

#### 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 [Python API reference](https://developers.google.com/protocol-buffers/docs/reference/python/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 interface](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class).

# REFERENCE

This section contains reference documentation for working with protocol buffer classes in C++, Java, and Python, as well as some reference documentation for Protocol Buffers itself. The documentation for each language includes:

* A reference guide to the code generated by the protocol buffer compiler from your .proto files.
* Generated API documentation for the provided source code.

Note that there are APIs for several more languages in the pipeline – for details, see the [other languages wiki page](https://github.com/google/protobuf/wiki/Third-Party-Add-ons).

### C++ Reference

#### [C++ Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated)

This page describes exactly what C++ 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 C++ output when invoked with the --cpp\_out= command-line flag. The parameter to the --cpp\_out= option is the directory where you want the compiler to write your C++ output. The compiler creates a header file and an implementation file for each .proto file input. The names of the output files are computed by taking the name of the .proto file and making two changes:

* The extension (.proto) is replaced with either .pb.h or .pb.cc for the header or implementation file, respectively.
* The proto path (specified with the --proto\_path= or -I command-line flag) is replaced with the output path (specified with the --cpp\_out= flag).

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

protoc --proto\_path=src --cpp\_out=build/gen src/foo.proto src/bar/baz.proto

The compiler will read the files src/foo.proto and src/bar/baz.proto and produce four output files:build/gen/foo.pb.h, build/gen/foo.pb.cc, build/gen/bar/baz.pb.h,build/gen/bar/baz.pb.cc. The compiler will automatically create the directory build/gen/bar if necessary, but it will *not* create build or build/gen; they must already exist.

##### Packages

If a .proto file contains a package declaration, the entire contents of the file will be placed in a corresponding C++ namespace. For example, given the package declaration:

package foo.bar;

All declarations in the file will reside in the foo::bar namespace.

##### Messages

Given a simple message declaration:

message Foo {}

The protocol buffer compiler generates a class called Foo, which publicly derives from[google::protobuf::Message](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message). The class is a concrete class; no pure-virtual methods are left unimplemented. Methods that are virtual in [Message](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message) but not pure-virtual may or may not be overridden by Foo, depending on the optimization mode. By default, Foo implements specialized versions of all methods 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 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[google::protobuf::MessageLite](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message_lite.html#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.so (libprotobuf-lite.lib on Windows) instead of libprotobuf.so(libprotobuf.lib). The "lite" library is much smaller than the full library, and is more appropriate for resource-constrained systems such as mobile phones.

You should *not* create your own Foo subclasses. If you subclass this class and override a virtual method, the override may be ignored, as many generated method calls are de-virtualized to improve performance.

The [Message](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message) interface defines methods that let you check, manipulate, read, or write the entire message, including parsing from and serializing to binary strings. In addition to these methods, the Foo class defines the following methods:

* Foo(): Default constructor.
* ~Foo(): Default destructor.
* Foo(const Foo& other): Copy constructor.
* Foo& operator=(const Foo& other): Assignment operator.
* void Swap(Foo\* other): Swap content with another message.
* const [UnknownFieldSet](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set.html#UnknownFieldSet)& unknown\_fields() const: Returns the set of unknown fields encountered while parsing this message.
* [UnknownFieldSet](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set.html#UnknownFieldSet)\* mutable\_unknown\_fields(): Returns a pointer to the mutable set of unknown fields encountered while parsing this message.

The class also defines the following static methods:

* static const [Descriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor)& descriptor(): 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 [reflection](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.Reflection) to inspect fields programmatically.
* static const Foo& default\_instance(): Returns a const singleton instance of Foo which is identical to a newly-constructed instance of Foo (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 [New()](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message.html#Message.New) method.

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

In this case, the compiler generates two classes: Foo and Foo\_Bar. In addition, the compiler generates a typedef inside Foo as follows:

typedef Foo\_Bar Bar;

This means that you can use the nested type's class as if it was the nested class Foo::Bar. However, note that C++ does not allow nested types to be forward-declared. If you want to forward-declare Bar in another file and use that declaration, you must identify it as Foo\_Bar.

##### 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.

As well as accessor methods, the compiler generates an integer constant for each field containing its field number. The constant name is the letter k, followed by the field name converted to camel-case, followed byFieldNumber. For example, given the field optional int32 foo\_bar = 5;, the compiler will generate the constant static const int kFooBarFieldNumber = 5;.

###### Singular Numeric Fields

For either of these field definitions:

optional int32 foo = 1;

required int32 foo = 1;

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if the field is set.
* int32 foo() const: Returns the current value of the field. If the field is not set, returns the default value.
* void set\_foo(int32 value): Sets the value of the field. After calling this, has\_foo() will returntrue and foo() will return value.
* void clear\_foo(): Clears the value of the field. After calling this, has\_foo() will return false andfoo() will return the default value.

For other numeric field types (including bool), int32 is replaced with the corresponding C++ type according to the [scalar value types table](https://developers.google.com/protocol-buffers/docs/proto.html#scalar).

###### Singular String Fields

For any of these field definitions:

optional string foo = 1;

required string foo = 1;

optional bytes foo = 1;

required bytes foo = 1;

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if the field is set.
* const string& foo() const: Returns the current value of the field. If the field is not set, returns the default value.
* void set\_foo(const string& value): Sets the value of the field. After calling this, has\_foo()will return true and foo() will return a copy of value.
* void set\_foo(const char\* value): Sets the value of the field using a C-style null-terminated string. After calling this, has\_foo() will return true and foo() will return a copy of value.
* void set\_foo(const char\* value, int size): Like above, but the string size is given explicitly rather than determined by looking for a null-terminator byte.
* string\* mutable\_foo(): Returns a pointer to the mutable string object that stores the field's value. If the field was not set prior to the call, then the returned string will be empty (*not* the default value). After calling this, has\_foo() will return true and foo() will return whatever value is written into the given string. The pointer is invalidated by a call to Clear() or clear\_foo().
* void clear\_foo(): Clears the value of the field. After calling this, has\_foo() will return false andfoo() will return the default value.
* void set\_allocated\_foo(string\* value): Sets the string object to the field and frees the previous field value if it exists. If the string pointer is not NULL, the message takes ownership of the allocated string object and has\_foo() will return true. Otherwise, if the value is NULL, the behavior is the same as calling clear\_foo().
* string\* release\_foo(): Releases the ownership of the field and returns the pointer of the stringobject. After calling this, caller takes the ownership of the allocated string object, has\_foo() will return false, and foo() will return the default value.

###### Singular Enum Fields

Given the enum type:

enum Bar {

BAR\_VALUE = 1;

}

For either of these field definitions:

optional Bar foo = 1;

required Bar foo = 1;

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if the field is set.
* Bar foo() const: Returns the current value of the field. If the field is not set, returns the default value.
* void set\_foo(Bar value): Sets the value of the field. After calling this, has\_foo() will returntrue and foo() will return value. In debug mode (i.e. NDEBUG is not defined), if value does not match any of the values defined for Bar, this method will abort the process.
* void clear\_foo(): Clears the value of the field. After calling this, has\_foo() will return false andfoo() will return the default value.

###### Singular Embedded Message Fields

Given the message type:

message Bar {}

For either of these field definitions:

optional Bar foo = 1;

required Bar foo = 1;

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if the field is set.
* const Bar& foo() const: Returns the current value of the field. If the field is not set, returns a Barwith none of its fields set (possibly Bar::default\_instance()).
* Bar\* mutable\_foo(): Returns a pointer to the mutable Bar object that stores the field's value. If the field was not set prior to the call, then the returned Bar will have none of its fields set (i.e. it will be identical to a newly-allocated Bar). After calling this, has\_foo() will return true and foo() will return a reference to the same instance of Bar. The pointer is invalidated by a call to Clear() orclear\_foo().
* void clear\_foo(): Clears the value of the field. After calling this, has\_foo() will return false andfoo() will return the default value.
* void set\_allocated\_foo(Bar\* bar): Sets the Bar object to the field and frees the previous field value if it exists. If the Bar pointer is not NULL, the message takes ownership of the allocated Barobject and has\_foo() will return true. Otherwise, if the Bar is NULL, the behavior is the same as calling clear\_foo().
* Bar\* release\_foo(): Releases the ownership of the field and returns the pointer of the Bar object. After calling this, caller takes the ownership of the allocated Bar object, has\_foo() will return false, and foo() will return the default value.

###### Repeated Numeric Fields

For this field definition:

repeated int32 foo = 1;

The compiler will generate the following accessor methods:

* int foo\_size() const: Returns the number of elements currently in the field.
* int32 foo(int index) const: Returns the element at the given zero-based index. Calling this method with index outside of [0, foo\_size()) yields undefined behavior.
* void set\_foo(int index, int32 value): Sets the value of the element at the given zero-based index.
* void add\_foo(int32 value): Appends a new element to the field with the given value.
* void clear\_foo(): Removes all elements from the field. After calling this, foo\_size() will return zero.
* const [RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<int32>& foo() const: Returns the underlying [RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.
* [RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<int32>\* mutable\_foo(): Returns a pointer to the underlying mutable[RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.

For other numeric field types (including bool), int32 is replaced with the corresponding C++ type according to the [scalar value types table](https://developers.google.com/protocol-buffers/docs/proto.html#scalar).

###### Repeated String Fields

For either of these field definitions:

repeated string foo = 1;

repeated bytes foo = 1;

The compiler will generate the following accessor methods:

* int foo\_size() const: Returns the number of elements currently in the field.
* const string& foo(int index) const: Returns the element at the given zero-based index. Calling this method with index outside of [0, foo\_size()) yields undefined behavior.
* void set\_foo(int index, const string& value): Sets the value of the element at the given zero-based index.
* void set\_foo(int index, const char\* value): Sets the value of the element at the given zero-based index using a C-style null-terminated string.
* void set\_foo(int index, const char\* value, int size): Like above, but the string size is given explicitly rather than determined by looking for a null-terminator byte.
* string\* mutable\_foo(int index): Returns a pointer to the mutable string object that stores the value of the element at the given zero-based index. The pointer is invalidated by a call to Clear() orclear\_foo(), or by manipulating the underlying [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) in a way that would remove this element.
* void add\_foo(const string& value): Appends a new element to the field with the given value.
* void add\_foo(const char\* value): Appends a new element to the field using a C-style null-terminated string.
* void add\_foo(const char\* value, int size): Like above, but the string size is given explicitly rather than determined by looking for a null-terminator byte.
* string\* add\_foo(): Adds a new empty string element and returns a pointer to it. The pointer is invalidated by a call to Clear() or clear\_foo(), or by manipulating the underlying[RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) in a way that would remove this element.
* void clear\_foo(): Removes all elements from the field. After calling this, foo\_size() will return zero.
* const [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<string>& foo() const: Returns the underlying [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)that stores the field's elements. This container class provides STL-like iterators and other methods.
* [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<string>\* mutable\_foo(): Returns a pointer to the underlying mutable[RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.

###### Repeated Enum Fields

Given the enum type:

enum Bar {

BAR\_VALUE = 1;

}

For this field definition:

repeated Bar foo = 1;

The compiler will generate the following accessor methods:

* int foo\_size() const: Returns the number of elements currently in the field.
* Bar foo(int index) const: Returns the element at the given zero-based index. Calling this method with index outside of [0, foo\_size()) yields undefined behavior.
* void set\_foo(int index, Bar value): Sets the value of the element at the given zero-based index. In debug mode (i.e. NDEBUG is not defined), if value does not match any of the values defined for Bar, this method will abort the process.
* void add\_foo(Bar value): Appends a new element to the field with the given value. In debug mode (i.e. NDEBUG is not defined), if value does not match any of the values defined for Bar, this method will abort the process.
* void clear\_foo(): Removes all elements from the field. After calling this, foo\_size() will return zero.
* const [RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<int>& foo() const: Returns the underlying [RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.
* [RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<int>\* mutable\_foo(): Returns a pointer to the underlying mutable[RepeatedField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.

###### Repeated Embedded Message Fields

Given the message type:

message Bar {}

For this field definitions:

repeated Bar foo = 1;

The compiler will generate the following accessor methods:

* int foo\_size() const: Returns the number of elements currently in the field.
* const Bar& foo(int index) const: Returns the element at the given zero-based index. Calling this method with index outside of [0, foo\_size()) yields undefined behavior.
* Bar\* mutable\_foo(int index): Returns a pointer to the mutable Bar object that stores the value of the element at the given zero-based index. The pointer is invalidated by a call to Clear() orclear\_foo(), or by manipulating the underlying [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) in a way that would remove this element.
* Bar\* add\_foo(): Adds a new element and returns a pointer to it. The returned Bar will have none of its fields set (i.e. it will be identical to a newly-allocated Bar). The pointer is invalidated by a call to Clear()or clear\_foo(), or by manipulating the underlying [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) in a way that would remove this element.
* void clear\_foo(): Removes all elements from the field. After calling this, foo\_size() will return zero.
* const [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<Bar>& foo() const: Returns the underlying [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.
* [RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)<Bar>\* mutable\_foo(): Returns a pointer to the underlying mutable[RepeatedPtrField](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) that stores the field's elements. This container class provides STL-like iterators and other methods.

###### Oneof Numeric Fields

For this [oneof](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated#oneof) field definition:

oneof oneof\_name {

int32 foo = 1;

...

}

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if oneof case is kFoo.
* int32 foo() const: Returns the current value of the field if oneof case is kFoo. Otherwise, returns the default value.
* void set\_foo(int32 value):
  + If any other oneof field in the same oneof is set, calls clear\_oneof\_name().
  + Sets the value of this field and sets the oneof case to kFoo.
  + has\_foo() will return true, foo() will return value, and oneof\_name\_case() will returnkFoo.
* void clear\_foo():
  + Nothing will be changed if oneof case is not kFoo.
  + If oneof case is kFoo, clears the value of the field and oneof case. has\_foo() will return false,foo() will return the default value and oneof\_name\_case() will return ONEOF\_NAME\_NOT\_SET.

For other numeric field types (including bool), int32 is replaced with the corresponding C++ type according to the [scalar value types table](https://developers.google.com/protocol-buffers/docs/reference/language.shtml#scalar).

###### Oneof String Fields

For any of these [oneof](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated#oneof) field definitions:

oneof oneof\_name {

string foo = 1;

…

}

oneof onef\_name {

bytes foo = 1;

….

}

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if the oneof case is kFoo.
* const string& foo() const: Returns the current value of the field if the oneof case is kFoo. Otherwise, returns the default value.
* void set\_foo(const string& value):
  + If any other oneof field in the same oneof is set, calls clear\_oneof\_name().
  + Sets the value of this field and sets the oneof case to kFoo.
  + has\_foo() will return true, foo() will return a copy of value and oneof\_name\_case() will return kFoo.
* void set\_foo(const char\* value):
  + If any other oneof field in the same oneof is set, calls clear\_oneof\_name().
  + Sets the value of the field using a C-style null-terminated string and set the oneof case to kFoo.
  + has\_foo() will return true, foo() will return a copy of value and oneof\_name\_case() will return kFoo.
* void set\_foo(const char\* value, int size): Like above, but the string size is given explicitly rather than determined by looking for a null-terminator byte.
* string\* mutable\_foo():
  + If any other oneof field in the same oneof is set, calls clear\_oneof\_name().
  + Sets the oneof case to kFoo and returns a pointer to the mutable string object that stores the field's value. If the oneof case was not kFoo prior to the call, then the returned string will be empty (not the default value).
  + has\_foo() will return true, foo() will return whatever value is written into the given string andoneof\_name\_case() will return kFoo. The pointer is invalidated by a call to Clear() or if the oneof case is changed: clear\_foo(), clear\_oneof\_name(), or setting another oneof field in the same oneof will all change the oneof case.
* void clear\_foo():
  + If the oneof case is not kFoo, nothing will be changed .
  + If the oneof case is kFoo, frees the field and clears the oneof case . has\_foo() will return false,foo() will return the default value, and oneof\_name\_case() will return ONEOF\_NAME\_NOT\_SET.
* void set\_allocated\_foo(string\* value):
  + Calls clear\_oneof\_name().
  + If the string pointer is not NULL: Sets the string object to the field and sets the oneof case to kFoo. The message takes ownership of the allocated string object, has\_foo() will return true andoneof\_name\_case() will return kFoo.
  + If the string pointer is NULL, has\_foo() will return false and oneof\_name\_case() will returnONEOF\_NAME\_NOT\_SET.
* string\* release\_foo():
  + Returns NULL if oneof case is not kFoo.
  + Clears the oneof case, releases the ownership of the field and returns the pointer of the string object. After calling this, caller takes the ownership of the allocated string object, has\_foo() will return false, foo() will return the default value, and oneof\_name\_case() will returnONEOF\_NAME\_NOT\_SET.

###### Oneof Enum Fields

Given the enum type:

enum Bar {

BAR\_VALUE = 1;

}

For the [oneof](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated#oneof) field definition:

oneof oneof\_name {

Bar foo = 1;

...

}

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if oneof case is kFoo.
* Bar foo() const: Returns the current value of the field if oneof case is kFoo. Otherwise, returns the default value.
* void set\_foo(Bar value):
  + If any other oneof field in the same oneof is set, calls clear\_oneof\_name().
  + Sets the value of this field and sets the oneof case to kFoo.
  + has\_foo() will return true, foo() will return value and oneof\_name\_case() will returnkFoo.
  + In debug mode (i.e. NDEBUG is not defined), if value does not match any of the values defined forBar, this method will abort the process.
* void clear\_foo():
  + Nothing will be changed if the oneof case is not kFoo.
  + If the oneof case is kFoo, clears the value of the field and the oneof case. has\_foo() will returnfalse, foo() will return the default value and oneof\_name\_case() will returnONEOF\_NAME\_NOT\_SET.

###### Oneof Embedded Message Fields

Given the message type:

message Bar {}

For the [oneof](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated#oneof) field definition:

oneof oneof\_name {

Bar foo = 1;

...

}

The compiler will generate the following accessor methods:

* bool has\_foo() const: Returns true if oneof case is kFoo.
* const Bar& foo() const: Returns the current value of the field if oneof case is kFoo. Otherwise, returns Bar::default\_instance().
* Bar\* mutable\_foo():
  + If any other oneof field in the same oneof is set, calls clear\_oneof\_name().
  + Sets the oneof case to kFoo and returns a pointer to the mutable Bar object that stores the field's value. If the oneof case was not kFoo prior to the call, then the returned Bar will have none of its fields set (i.e. it will be identical to a newly-allocated Bar).
  + After calling this, has\_foo() will return true, foo() will return a reference to the same instance of Bar and oneof\_name\_case() will return kFoo. The pointer is invalidated by a call toClear() or the oneof case is changed: clear\_foo(), clear\_oneof\_name(), or setting another oneof field in the same oneof will all change the oneof case.
* void clear\_foo():
  + Nothing will be changed if the oneof case is not kFoo.
  + If the oneof case equals kFoo, frees the field and clears the oneof case. has\_foo() will returnfalse, foo() will return the default value and oneof\_name\_case() will returnONEOF\_NAME\_NOT\_SET.
* void set\_allocated\_foo(Bar\* bar):
  + Calls clear\_oneof\_name().
  + If the Bar pointer is not NULL: Sets the Bar object to the field and sets the oneof case to kFoo. The message takes ownership of the allocated Bar object, has\_foo() will return true and oneof\_name\_case() will return kFoo.
  + If the pointer is NULL, has\_foo() will return false and oneof\_name\_case() will returnONEOF\_NAME\_NOT\_SET. (The behavior is like calling clear\_oneof\_name())
* Bar\* release\_foo():
  + Returns NULL if oneof case is not kFoo.
  + If the oneof case is kFoo, clears the oneof case, releases the ownership of the field and returns the pointer of the Bar object. After calling this, caller takes the ownership of the allocated Bar object,has\_foo() will return false, foo() will return the default value and oneof\_name\_case() will return ONEOF\_NAME\_NOT\_SET.

###### Map Fields

For this map field definition:

map<int32, int32> weight = 1;

The compiler will generate the following accessor methods:

* const proto2::Map<int32, int32>& weight();: Returns an immutable Map.
* proto2::Map<int32, int32>\* mutable\_weight();: Returns a mutable Map.

A proto2::Map is a special container type used in protocol buffers to store map fields. As you can see from its interface below, it uses a commonly-used subset of std::map and std::unordered\_map methods.

template<typename Key, typename T> {  
class Map {  
  // Member types  
  typedef Key key\_type;  
  typedef T mapped\_type;  
  typedef ... value\_type;  
  
  // Iterators  
  iterator begin();  
  const\_iterator begin() const;  
  const\_iterator cbegin() const;  
  iterator end();  
  const\_iterator end() const;  
  const\_iterator cend() const;  
  // Capacity  
  int size() const;  
  bool empty() const;  
  
  // Element access  
  T& operator[](const Key& key);  
  const T& at(const Key& key) const;  
  T& at(const Key& key);  
  
  // Lookup  
  int count(const Key& key) const;  
  const\_iterator find(const Key& key) const;  
  iterator find(const Key& key);  
  
  // Modifiers  
  pair<iterator, bool> insert(const value\_type& value);  
  template<class InputIt>  
  void insert(InputIt first, InputIt last);  
  size\_type erase(const Key& Key);  
  iterator erase(const\_iterator pos);  
  iterator erase(const\_iterator first, const\_iterator last);  
  void clear();  
  
  // Copy  
  Map(const Map& other);  
  Map& operator=(const Map& other);  
}

pair<iterator, bool> insert(const value\_type& value) will implicitly cause a deep copy of thevalue\_type instance. The most efficient way to insert a new value into a proto2::Map is as follows:

T& operator[](const Key& key): map[new\_key] = new\_mapped;

Using proto2::Map with standard maps

proto2::Map supports the same iterator API as std::map and std::unordered\_map. If you don't want to use proto2::Map directly, you can convert a proto2::Map to a standard map by doing the following:

std::map<int32, int32> standard\_map(message.weight().begin(),  
                                    message.weight().end());

Note that this will make a deep copy of the entire map.

Parsing unknown values

On the wire, a .proto map is equivalent to a map entry message for each key/value pair, while the map itself is a repeated field of map entries. Like ordinary message types, it's possible for a parsed map entry message to have unknown fields: for example a field of type int64 in a map defined as map<int32, string>.

If there are unknown fields in the wire format of a map entry message, they will be discarded.

If there is an unknown enum value in the wire format of a map entry message, it's handled differently in proto2 and proto3. In proto2, the whole map entry message is put into the unknown field set of the containing message. In proto3, it is put into a map field as if it is a known enum value.

##### Oneof

Given a oneof definition like this:

oneof oneof\_name {

int32 foo\_int = 4;

string foo\_string = 9;

...

}

The compiler will generate the following C++ enum type:

enum OneofNameCase {

kFooInt = 4,

kFooString = 9,

ONEOF\_NAME\_NOT\_SET = 0

}

In addition, it will generate this method:

* OneofNameCase oneof\_name\_case() const: Returns the enum indicating which field is set. ReturnsONEOF\_NAME\_NOT\_SET if none of them is set.

The compiler also generates the following private method, which is used in oneof field accessors:

* void clear\_oneof\_name(): Frees the object if the oneof field set uses a pointer (Message or String), and sets the oneof case to ONEOF\_NAME\_NOT\_SET.

##### Enumerations

Given an enum definition like:

enum Foo {

VALUE\_A = 1;

VALUE\_B = 5;

VALUE\_C = 1234;

}

The protocol buffer compiler will generate a C++ enum type called Foo with the same set of values. In addition, the compiler will generate the following functions:

* const [EnumDescriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#EnumDescriptor)\* Foo\_descriptor(): Returns the type's descriptor, which contains information about what values this enum type defines.
* bool Foo\_IsValid(int value): Returns true if the given numeric value matches one of Foo's defined values. In the above example, it would return true if the input were 1, 5, or 1234.
* const string& Foo\_Name(int value): Returns the name for given numeric value. Returns an empty string if no such value exists. If multiple values have this number, the first one defined is returned. In the above example, Foo\_Name(5) would return "VALUE\_B".
* bool Foo\_Parse(const string& name, Foo\* value): If name is a valid value name for this enum, assigns that value into value and returns true. Otherwise returns false. In the above example,Foo\_Parse("VALUE\_C", &someFoo) would return true and set someFoo to 1234.
* const Foo Foo\_MIN: the smallest valid value of the enum (VALUE\_A in the example).
* const Foo Foo\_MAX: the largest valid value of the enum (VALUE\_C in the example).
* const Foo Foo\_ARRAYSIZE: always defined as Foo\_MAX + 1.

**Be careful when casting integers to enums.** If an integer is cast to an enum value, the integer *must* be one of the valid values for than enum, or the results may be undefined. If in doubt, use the generated **Foo\_IsValid()**function to test if the cast is valid. Setting an enum-typed field of a protocol message to an invalid value may cause an assertion failure. If an invalid enum value is read when parsing a message, it will be treated as an[unknown field](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set).

You can define an enum inside a message type. In this case, the protocol buffer compiler generates code that makes it appear that the enum type itself was declared nested inside the message's class. TheFoo\_descriptor() and Foo\_IsValid() functions are declared as static methods. In reality, the enum type itself and its values are declared at the global scope with mangled names, and are imported into the class's scope with a typedef and a series of constant definitions. This is done only to get around problems with declaration ordering. Do not depend on the mangled top-level names; pretend the enum really is nested in the message class.

##### Extensions

Given a message with an extension range:

message Foo {

extensions 100 to 199;

}

The protocol buffer compiler will generate some additional methods for Foo: HasExtension(),ExtensionSize(), ClearExtension(), GetExtension(), SetExtension(),MutableExtension(), AddExtension(), SetAllocatedExtension() and ReleaseExtension(). 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. (GetExtension() corresponds to the accessors with no special prefix.)

Given an extension definition:

extend Foo {

optional int32 bar = 1;

repeated int32 repeated\_bar = 2;

}

For the singular extension field bar, 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;  
assert(!foo.HasExtension(bar));  
foo.SetExtension(bar, 1);  
assert(foo.HasExtension(bar));  
assert(foo.GetExtension(bar) == 1);  
foo.ClearExtension(bar);  
assert(!foo.HasExtension(bar));

Similarly, for the repeated extension field repeated\_bar, the compiler generates an extension identifier calledrepeated\_bar, which you can also use with Foo's extension accessors:

Foo foo;  
for (int i = 0; i < kSize; ++i) {  
  foo.AddExtension(repeated\_bar, i)  
}  
assert(foo.ExtensionSize(repeated\_bar) == kSize)  
for (int i = 0; i < kSize; ++i) {  
  assert(foo.GetExtension(repeated\_bar, i) == i)  
}

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

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:

Foo foo;  
Baz\* baz = foo.MutableExtension(Baz::foo\_ext);  
FillInMyBaz(baz);

##### Arena Allocation

Arena allocation is a C++-only feature that helps you optimize your memory usage and improve performance when working with protocol buffers. Enabling arena allocation in your .proto adds additional code for working with arenas to your C++ generated code. You can find out more about the arena allocation API in the [Arena Allocation Guide](https://developers.google.com/protocol-buffers/docs/reference/arenas).

##### Services

If the .proto file contains the following line:

option cc\_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 cc\_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 a class Foo to represent this service. Foo will have a virtual method for each method defined in the service definition. In this case, the method Bar is defined as:

virtual void Bar([**RpcController**](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcController)\* controller, const FooRequest\* request,  
                 FooResponse\* response, [**Closure**](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.common.html#Closure)\* done);

The parameters are equivalent to the parameters of [Service::CallMethod()](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.CallMethod.details), except that the methodargument is implied and request and response specify their exact type.

These generated methods are virtual, but not pure-virtual. The default implementations simply callcontroller->[SetFailed()](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcController.SetFailed) with an error message indicating that the method is unimplemented, then invoke the done callback. When implementing your own service, you must subclass this generated service and implement its methods as appropriate.

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

* [GetDescriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.GetDescriptor): Returns the service's [ServiceDescriptor](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#ServiceDescriptor).
* [CallMethod](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.CallMethod): Determines which method is being called based on the provided method descriptor and calls it directly, down-casting the request and response messages objects to the correct types.
* [GetRequestPrototype](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.GetRequestPrototype) and [GetResponsePrototype](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.GetResponsePrototype): 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/cpp/google.protobuf.descriptor.html#ServiceDescriptor) descriptor(): Returns the type's descriptor, which contains information about what methods this service has and what their input and output types are.

###### 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 with nested message types, a typedef is used so that Foo\_Stubcan also be referred to as Foo::Stub.

Foo\_Stub is a subclass of Foo which also implements the following methods:

* Foo\_Stub([RpcChannel](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel)\* channel): Constructs a new stub which sends requests on the given channel.
* Foo\_Stub([RpcChannel](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel)\* channel, [ChannelOwnership](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#Service.ChannelOwnership) ownership): Constructs a new stub which sends requests on the given channel and possibly owns that channel. If ownership isService::STUB\_OWNS\_CHANNEL then when the stub object is deleted it will delete the channel as well.
* [RpcChannel](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel)\* channel(): 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/cpp/google.protobuf.service.html#RpcChannel.CallMethod).

The Protocol Buffer library does not include an RPC implementation. However, it includes all of the tools you need to hook up a generated service class to any arbitrary RPC implementation of your choice. You need only provide implementations of [RpcChannel](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcChannel) and [RpcController](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service.html#RpcController). See the documentation for [service.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service) for more information.

##### 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 C++ code generator may insert code of the following types using the given insertion point names. Each insertion point appears in both the .pb.cc file and the .pb.h file unless otherwise noted.

* includes: Include directives.
* namespace\_scope: Declarations that belong in the file's package/namespace, but not within any particular class. Appears after all other namespace-scope code.
* global\_scope: Declarations that belong at the top level, outside of the file's namespace. Appears at the very end of the file.
* class\_scope:TYPENAME: Member declarations that belong in a message class. TYPENAME is the full proto name, e.g. package.MessageType. Appears after all other public declarations in the class. This insertion point appears only in the .pb.h file.

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.

#### [C++ Arena Allocation Guide](https://developers.google.com/protocol-buffers/docs/reference/arenas)

Arena allocation is a C++-only feature that helps you optimize your memory usage and improve performance when working with protocol buffers. This page describes exactly what C++ code the protocol buffer compiler generates in addition to the code described in the [C++ Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated) when arena allocation is enabled. It assumes that you are familiar with the material in the [language guide](https://developers.google.com/protocol-buffers/docs/proto) and the [C++ Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated).

##### Why use arena allocation?

Memory allocation and deallocation constitutes a significant fraction of CPU time spent in protocol buffers code. By default, protocol buffers performs heap allocations for each message object, each of its subobjects, and several field types, such as strings. These allocations occur in bulk when parsing a message and when building new messages in memory, and associated deallocations happen when messages and their subobject trees are freed.

Arena-based allocation has been designed to reduce this performance cost. With arena allocation, new objects are allocated out of a large piece of preallocated memory called the arena. Objects can all be freed at once by discarding the entire arena, ideally without running destructors of any contained object (though an arena can still maintain a “destructor list” when required). This makes object allocation faster by reducing it to a simple pointer increment, and makes deallocation almost free. Arena allocation also provides greater cache efficiency: when messages are parsed, they are more likely to be allocated in continuous memory, which makes traversing messages more likely to hit hot cache lines.

To get these benefits you'll need to be aware of object lifetimes and find a suitable granularity at which to use arenas (for servers, this is often per-request). You can find out more about how to get the most from arena allocation in [Usage patterns and best practices](https://developers.google.com/protocol-buffers/docs/reference/arenas#usage).

##### Getting started

You enable arena allocation on a per-.proto basis. To do this, add the following option to your .proto file:

option cc\_enable\_arenas = true;

This tells the protocol buffer compiler to generate the additional code you need to use arena allocation for the messages in your file, as used in the following example.

#include "net/proto2/public/arena.h"  
{  
  proto2::Arena arena;  
  MyMessage\* message = proto2::Arena::CreateMessage<MyMessage>(&arena);  
  // ...  
}

The message object created by CreateMessage() will exist for as long as arena exists, and you should notdelete the returned message pointer. All of the message object's internal storage (with a few exceptions[1](https://developers.google.com/protocol-buffers/docs/reference/arenas#fn1)) and submessages (for example, submessages in a repeated field within MyMessage) are allocated on the arena as well.

For the most part, the rest of your code will be the same as if you weren't using arena allocation.

We'll look at the arena API in more detail in the following sections, and you can see a more extensive [example](https://developers.google.com/protocol-buffers/docs/reference/arenas#example) at the end of the document.

1. Currently, string fields store their data on the heap even when the containing message is on the arena. Unknown fields are also heap-allocated.[↩](https://developers.google.com/protocol-buffers/docs/reference/arenas#ref1)

##### Arena class API

You create message objects on the arena using the [proto2::Arena](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.arena.html) class. This class implements the following public methods.

###### Constructors

* Arena(): Creates a new arena with default parameters, tuned for average use cases.
* Arena(const ArenaOptions& options): Creates a new arena that uses the specified allocation options. The options available in [ArenaOptions](https://developers.google.com/protocol-buffers/docs/reference/arenas) include the ability to use an initial block of user-provided memory for allocations before resorting to the system allocator, control over the initial and maximum request sizes for blocks of memory, and allowing you to pass in custom block allocation and deallocation function pointers to build freelists and others on top of the blocks.

###### Allocation methods

* template<typename T> static T\* CreateMessage(Arena\* arena): Creates a new protocol buffer object of message type T on the arena. This message type must be defined in a .proto file withoption cc\_enable\_arenas = true; otherwise, a compile error will result.

If arena is not NULL, the returned message object is allocated on the arena, its internal storage and submessages (if any) will be allocated on the same arena, and its lifetime is the same as that of the arena. The object must not be deleted/freed manually: the arena owns the message object for lifetime purposes.

If arena is NULL, the returned message object is allocated on the heap, and the caller owns the object upon return.

* template<typename T> static T\* Create(Arena\* arena, args...): Similar toCreateMessage() but lets you create an object of any class on the arena, not just protocol buffer message types with option cc\_enable\_arenas = true;: you can use a protocol buffer message class from a file that doesn't have arena support enabled, or an arbitrary C++ class. For example, let's say you have this C++ class:

class MyCustomClass {  
    MyCustomClass(int arg1, int arg2);  
    // ...  
};

...you can create an instance of it on the arena like this:

void func() {  
    // ...  
    proto2::Arena arena;  
    MyCustomClass\* c = proto2::Arena::Create<MyCustomClass>(&arena, constructor\_arg1, constructor\_arg2);  
    // ...  
}

* template<typename T> static T\* CreateArray(Arena\* arena, size\_t n): If arena is not NULL, this method allocates raw storage for n elements of type T and returns it. The arena owns the returned memory and will free it on its own destruction. If arena is NULL, this method allocates storage on the heap and the caller receives ownership.

T must have a trivial constructor: constructors are not called when the array is created on the arena.

###### "Owned list" methods

The following methods let you specify that particular objects or destructors are "owned" by the arena, ensuring that they are deleted or called when the arena itself is deleted

* template<typename T> void Own(T\* object): Adds object to the arena's list of owned heap objects. When the arena is destroyed, it traverses this list and frees each object using operator delete, i.e., the system memory allocator. This method is useful in cases when an object's lifetime should be tied to the arena but, for whatever reason, the object itself cannot be or was not already allocated on the arena.
* template<typename T> void OwnDestructor(T\* object): Adds the destructor of object to the arena's list of destructors to call. When the arena is destroyed, it traverses this list and calls each destructor in turn. It does not attempt to free the underlying memory of object. This method is useful when an object is embedded in arena-allocated storage but its destructor will not otherwise be called, for example because its containing class is a protobuf message whose destructor won't be called, or because it was manually constructed in a block allocated by AllocateArray().

###### Other methods

 uint64 SpaceUsed() const: Returns the total size of the arena, which is the sum of the sizes of the underlying blocks. This method is thread-safe; however, if there are concurrent allocations from multiple threads this method's return value may not include the sizes of those new blocks.

 uint64 Reset(): Destroys the arena's storage, first calling all registered destructors and freeing all registered heap objects and then discarding all arena blocks. This teardown procedure is equivalent to that which occurs when the arena's destructor runs, except the arena is reusable for new allocations after this method returns. Returns the total size used by the arena: this information is useful for tuning performance.

 template<typename T> Arena\* GetArena(): Returns a pointer to this arena. Not directly very useful but allows Arena to be used in template instantiations that expect GetArena() methods to be present.

###### Thread safety

proto2::Arena's allocation methods are thread-safe, and the underlying implementation goes to some length to make multithreaded allocation fast. The Reset() method is *not* thread-safe: the thread performing the arena reset must synchronize with all threads performing allocations or using objects allocated from that arena first.

##### Generated message class

The following message class members are changed or added when you enable arena allocation.

###### Message class methods

* void Swap(Message\* other): If both messages to be swapped are not on arenas or are on the *same*arena, [Swap()](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated.html#message) behaves as it does without having arena allocation enabled: it efficiently swaps the message objects' contents, usually via cheap pointer swaps and avoiding copies at all costs. However, if only one message is on an arena, or the messages are on different arenas, Swap() performs *deep copies*of the underlying data. This new behavior is necessary because otherwise the swapped sub-objects could have differing lifetimes, leading potentially to use-after-free bugs.
* Message\* New(Arena\* arena): An alternate override for the standard New() method. It allows a new message object of this type to be created on the given arena. Its semantics are identical toArena::CreateMessage<T>(arena) if the concrete message type on which it is called is generated with arena allocation enabled. If the message type is not generated with arena allocation enabled, then it is equivalent to an ordinary allocation followed by arena->Own(message) if arena is not NULL.
* Arena\* GetArena(): Returns the arena on which this message object was allocated, if any.
* void UnsafeArenaSwap(Message\* other): Identical to Swap(), except it assumes both objects are on the same arena (or not on arenas at all) and always uses the efficient pointer-sawpping implementation of this operation. Using this method can improve performance as, unlike Swap(), it doesn't need to check which messages live on which arena before performing the swap. As the Unsafe prefix suggests, you should only use this method if you are sure the messages you want to swap aren't on different arenas; otherwise this method could have unpredictable results.

###### Embedded message fields

When you allocate a message object on an arena, its embedded message field objects (submessages) are automatically owned by the arena as well. How these message objects are allocated depends on where they are defined:

* If the message type is also defined in a .proto file with arena allocation enabled, the object is allocated on the arena directly.
* If the message type is from another .proto without arena allocation enabled, the object is heap-allocated but is "owned" by the parent message's arena. This means that when the arena is destroyed, the object will be freed along with the objects on the arena itself.

For either of these field definitions:

optional Bar foo = 1;  
required Bar foo = 1;

The following methods are added or have some special behavior when arena allocation is enabled. Otherwise, accessor methods just use the [default behavior](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated.html#embeddedmessage).

* Bar\* mutable\_foo(): Returns a mutable pointer to the submessage instance. If the parent object is on an arena then the returned object will be as well.
* void set\_allocated\_foo(Bar\* bar): Takes a new object and adopts it as the new value for the field. Arena support adds additional copying semantics to maintain proper ownership when objects cross arena/arena or arena/heap boundaries:
  + If the parent object is on the heap and bar is on the heap, or if the parent and message are on the same arena, this method's behavior is unchanged.
  + If the parent is on an arena and bar is on the heap, the parent message adds bar to its arena's ownership list with arena->Own().
  + If the parent is on an arena and bar is on a different arena, this method makes a copy of message and takes the copy as the new field value.
* Bar\* release\_foo(): Returns the existing submessage instance of the field, if set, or a NULL pointer if not set, releasing ownership of this instance to the caller and clearing the parent message's field. Arena support adds additional copying semantics to maintain the contract that the returned object is always *heap-allocated*:
  + If the parent message is on an arena, this method will make a copy of the submessage on the heap, clear the field value, and return the copy.
  + If the parent message is on the heap, the method behavior is unchanged.
* void unsafe\_arena\_set\_allocated\_foo(Bar\* bar): Identical to set\_allocated\_foo, but assumes both parent and submessage are on the same arena. Using this version of the method can improve performance as it doesn't need to check whether the messages are on a particular arena or the heap. This method should only be used when the parent message is on the arena and the submessage is on the same arena, or an arena with equivalent lifetime.
* Bar\* unsafe\_arena\_release\_foo(): Similar to release\_foo(), but assumes the parent message is on the arena, and returns an *arena-allocated* object that should not be deleted directly. This method should only be used when the parent message is on the arena.

###### String fields

Currently, string fields store their data on the heap even when their parent message is on the arena. Because of this, string accessor methods largely use the [default behavior](https://developers.google.com/protocol-buffers/docs/reference/cpp-generated.html#string) even when arena allocation is enabled, with a couple of exceptions.

For any of these field definitions:

optional string foo = 1;  
required string foo = 1;  
optional bytes foo = 1;  
required bytes foo = 1;

The following methods are added or have some special/notable behavior when arena allocation is enabled.

 void set\_allocated\_foo(string\* value) and void unsafe\_arena\_set\_allocated\_foo(string\* value): Similar to the [embedded message field methods](https://developers.google.com/protocol-buffers/docs/reference/arenas#arenaembeddedmessage)described above where the child message is on the heap. As well as setting the field value to value, callingset\_allocated\_foo() when the parent is on the arena adds value to the parent message's arena's ownership list with arena->Own(). The unsafe\_arena version just sets the value without adding it to the ownership list.

 string\* release\_foo() and string\* unsafe\_arena\_release\_foo(): Again, these are similar to the [embedded message field methods](https://developers.google.com/protocol-buffers/docs/reference/arenas#arenaembeddedmessage) described above where the child message is on the heap. Callingrelease\_foo() when the parent is on the arena makes a copy of the string value, clears the field, and returns a pointer to the copy. Calling the unsafe\_arena version just releases the field and returns a pointer to the stillOwn()-ed string.

###### Repeated fields

Repeated fields allocate their internal array storage on the arena when the containing message is arena-allocated, and also allocate their elements on the arena when these elements are separate objects retained by pointer (messages or strings). At the message-class level, generated methods for repeated fields do not change. However, the RepeatedField and RepeatedPtrField objects that are returned by accessors do have new methods and modified semantics when arena support is enabled.

Repeated numeric fields

RepeatedField objects that contain [primitive types](https://developers.google.com/protocol-buffers/docs/reference/arenas#repeatednumeric) have the following new/changed methods when arena allocation is enabled:

* void UnsafeArenaSwap(RepeatedField\* other): Performs a swap of RepeatedField contents without validating that this repeated field and other are on the same arena. If they are not, the two repeated field objects must be on arenas with equivalent lifetimes. The case where one is on an arena and one is on the heap is checked and disallowed.
* void Swap(RepeatedField\* other): Checks each repeated field object's arena, and if one is on an arena while one is on the heap or if both are on arenas but on different ones, the underlying arrays are copied before the swap occurs. This means that after the swap, each repeated field object holds an array on its own arena or heap, as appropriate.

Repeated embedded message fields

RepeatedPtrField objects that contain [messages](https://developers.google.com/protocol-buffers/docs/reference/arenas#repeatedmessage) have the following new/changed methods when arena allocation is enabled.

* void UnsafeArenaSwap(RepeatedPtrField\* other): Performs a swap of RepeatedPtrFieldcontents without validating that this repeated field and other have the same arena pointer. If they do not, the two repeated field objects must have arena pointers with equivalent lifetimes. The case where one has a non-NULL arena pointer and one has a NULL arena pointer is checked and disallowed.
* void Swap(RepeatedPtrField\* other): Checks each repeated field object's arena pointer, and if one is non-NULL (contents on arena) while one is NULL (contents on heap) or if both are non-NULL but have different values, the underlying arrays and their pointed-to objects are copied before the swap occurs. This means that after the swap, each repeated field object holds an array on its own arena or on the heap, as appropriate.
* void AddAllocated(SubMessageType\* value): Checks that the provided message object is on the same arena as the repeated field's arena pointer. If it is on the same arena, then the object pointer is added directly to the underlying array. Otherwise, a copy is made, the original is freed if it was heap-allocated, and the copy is placed on the array. This maintains the invariant that all objects pointed to by a repeated field are in the same ownership domain (heap or specific arena) as indicated by the repeated field's arena pointer.
* SubMessageType\* ReleaseLast(): Returns a heap-allocated message equivalent to the last message in the repeated field, removing it from the repeated field. If the repeated field itself has a NULL arena pointer (and thus, all of its pointed-to messages are heap-allocated), then this method simply returns a pointer to the original object. Otherwise, if the repeated field has a non-NULL arena pointer, this method makes a copy that is heap-allocated and returns that copy. In both cases, the caller receives ownership of a heap-allocated object and is responsible for deleting the object.
* void UnsafeArenaAddAllocated(SubMessageType\* value): Like AddAllocated(), but does not perform heap/arena checks or any message copies. It adds the provided pointer directly to the internal array of pointers for this repeated field. The caller must guarantee that the provided object is heap-allocated if the repeated field has a NULL arena pointer, or arena-allocated (on the same arena or on one with identical lifetime) if the repeated field has a non-NULL arena pointer.
* SubMessageType\* UnsafeArenaReleaseLast(): Like ReleaseLast() but performs no copies, even if the repeated field has a non-NULL arena pointer. Instead, it directly returns the pointer to the object as it was in the repeated field. The returned object is thus on the heap if the repeated field's arena pointer is NULL and on an arena if the repeated field has a non-NULL arena pointer. The caller receives ownership if the object was heap-allocated. If the object was arena-allocated, the caller must not attempt to delete the returned object.
* void ExtractSubrange(int start, int num, SubMessageType\*\* elements): Removes numelements from the repeated field, starting from index start, and returns them in elements if it is not NULL. If the repeated field is on an arena, and elements are being returned, the elements are copied to the heap first. In both cases (arena or no arena), the caller owns the returned objects on the heap.
* void UnsafeArenaExtractSubrange(int start, int num, SubMessageType\*\* elements): Removes num elements from the repeated field, starting from index start, and returns them inelements if it is not NULL. Unlike ExtractSubrange(), this method never copies the extracted elements.

Repeated string fields

Repeated fields of strings have the same new methods and modified semantics as repeated fields of messages, because they also maintain their underlying objects (namely, strings) by pointer reference.

##### Usage patterns and best practices

When using arena-allocated messages, several usage patterns can result in unintended copies or other negative performance effects. You should be aware of the following common patterns that may need to be altered when adapting code for arenas. (Note that we have taken care in the API design to ensure that correct behavior still occurs — but higher-performance solutions may require some reworking.)

###### Unintended copies

Several methods that never create object copies when not using arena allocation may end up doing so when arena support is enabled. These unwanted copies can be avoided if you make sure that your objects are allocated appropriately and/or use provided arena-specific method versions, as described in more detail below.

Set Allocated/Add Allocated/Release

By default, the release\_*field*() and set\_allocated\_*field*() methods (for singular message and string fields), and the ReleaseLast() and AddAllocated()() methods (for repeated message and string fields) all allow user code to directly attach and detach sub-objects, passing ownership of pointers without copying any data.

However, when the parent message is on an arena, these methods now sometimes need to copy the passed in or returned object to maintain compatibility with existing ownership contracts. More specifically, methods that take ownership (set\_allocated\_*field*() and AddAllocated()) may copy data if the parent is on an arena and the new subobject is not, or vice versa, or they are on different arenas. Methods that release ownership (release\_*field*() and ReleaseLast()) may copy data if the parent is on the arena, because the returned object must be on the heap, by contract.

To avoid such copies, we have added corresponding "unsafe arena" versions of these methods where copies are**never performed**: unsafe\_arena\_set\_allocated\_*field*(), unsafe\_arena\_release\_*field*(),UnsafeArenaAddAllocated(), and UnsafeArenaRelease() for singular and repeated fields, respectively. These methods should be used only when you know they are safe to do so and that the parent and child objects are allocated as expected; otherwise, for example, you could end up with parent and child objects with differing lifetimes, leading potentially to use-after-free bugs.

Here's an example of how you can avoid unnecessary copies with these methods. Let's say you have created the following messages on an arena.

Arena\* arena = new proto2::Arena();  
MyFeatureMessage\* arena\_message\_1 =  
  proto2::Arena::CreateMessage<MyFeatureMessage>(arena);  
arena\_message\_1->mutable\_nested\_message()->set\_feature\_id(11);  
  
MyFeatureMessage\* arena\_message\_2 =  
  proto2::Arena::CreateMessage<MyFeatureMessage>(arena);

The following code makes inefficient usage of the release\_...() API:

arena\_message\_2->set\_allocated\_nested\_message(arena\_message\_1->release\_nested\_message());  
  
arena\_message\_1->release\_message(); // returns a copy of the underlying nested\_message and deletes underlying pointer

Using the "unsafe arena" version instead avoids the copy:

arena\_message\_2->set\_allocated\_nested\_message(  
   arena\_message\_1->unsafe\_arena\_release\_nested\_message());

You can find out more about these methods in the [Embedded message fields](https://developers.google.com/protocol-buffers/docs/reference/arenas#arenaembeddedmessage) and [String fields](https://developers.google.com/protocol-buffers/docs/reference/arenas#arenastring) sections above.

Swap

When two messages' contents are swapped with Swap(), the underlying subobjects may be copied if the two messages live on different arenas, or if one is on the arena and the other is on the heap. If you want to avoid this copy and either (i) know that the two messages are on the same arena or different arenas but the arenas have equivalent lifetimes, or (ii) know that the two messages are on the heap, you can use a new method,UnsafeArenaSwap(). This method both avoids the overhead of performing the arena check and avoids the copy if one would have occurred.

For example, the following code incurs a copy in the Swap() call:

MyFeatureMessage\* message\_1 =  
  proto2::Arena::CreateMessage<MyFeatureMessage>(arena);message\_1->mutable\_nested\_message()->set\_feature\_id(11);  
  
MyFeatureMessage\* message\_2 = new MyFeatureMessage;  
message\_2->mutable\_nested\_message()->set\_feature\_id(22);  
  
message\_1->Swap(message\_2); // Inefficient swap!

To avoid the copy in this code, you allocate message\_2 on the same arena as message\_1:

MyFeatureMessage\* message\_2 =          
   proto2::Arena::CreateMessage<MyFeatureMessage>(arena);

###### Embedded message fields and arena-enable options

Each .proto file has its own "feature switch" for arena support. If cc\_enable\_arenas is not set in a given.proto file, the types defined in that file will not be stored on the arena, even if some other type includes a submessage that has a type defined in that file. In other words, cc\_enable\_arenas is not transitive. Rather, submessages of an arena-capable message that do not themselves have arena support will always be stored on the heap, and will be added to the parent message's arena's Own() list so that their lifetimes are tied to the arena's lifetime.

The reason for this restriction is that adding arena support adds some overhead in the case that arenas are not used because of the extra generated code, so we choose (for now) not to enable arena support globally. Furthermore, for type- and API-compatibility reasons, we can have only one C++ generated class per proto message type, so we cannot generate with-arena and without-arena versions of a class. In the future, after further optimization, we may be able to lift this restriction and globally enable arena support. For now, though, it should be enabled for as many submessage types as possible to improve performance.

###### Granularity

We have found in most application server use cases that an "arena-per-request" model works well. You may be tempted to divide arena use further, either to reduce heap overhead (by destroying smaller arenas more often) or to reduce perceived thread-contention issues. However, the use of more fine-grained arenas may lead to unintended message copying, as we describe above. We have also spent effort to optimize the Arenaimplementation for the multithreaded use-case, so a single arena should be appropriate for use throughout a request lifetime even if multiple threads process that request.

##### Example

Here's a simple complete example demonstrating some of the features of the arena allocation API.

// my\_feature.proto  
  
syntax = "proto2";  
import “nested\_message.proto”;  
  
package feature\_package;  
  
option cc\_enable\_arenas = true;  
  
// NEXT Tag to use: 4  
message MyFeatureMessage {  
  optional string feature\_name = 1;  
  repeated int32 feature\_data = 2;  
  optional NestedMessage nested\_message = 3;  
};

// nested\_message.proto  
  
syntax = "proto2";  
  
package feature\_package;  
  
// add cc\_enable\_arenas on each submessage for  
// the best performance when using arenas.  
option cc\_enable\_arenas = true;  
  
// NEXT Tag to use: 2  
message NestedMessage {  
  optional int32 feature\_id = 1;  
};

Message construction and deallocation:

#include "net/proto2/public/arena.h"  
  
Arena arena = new proto2::Arena();  
  
MyFeatureMessage\* arena\_message =  
   proto2::Arena::CreateMessage<MyFeatureMessage>(&arena);  
  
arena\_message->set\_feature\_name(“Proto2 Arena”);  
arena\_message->mutable\_feature\_data()->Add(2);  
arena\_message->mutable\_feature\_data()->Add(4);  
arena\_message->mutable\_nested\_message()->set\_feature\_id(247);

#### [C++ API](https://developers.google.com/protocol-buffers/docs/reference/cpp/index)

|  |  |
| --- | --- |
| Packages | |
| [google::protobuf](https://developers.google.com/protocol-buffers/docs/reference/cpp/index#google.protobuf)  *Core components of the Protocol Buffers runtime library.* |  |
| [google::protobuf::io](https://developers.google.com/protocol-buffers/docs/reference/cpp/index#google.protobuf.io)  *Auxiliary classes used for I/O.* |  |
| [google::protobuf::compiler](https://developers.google.com/protocol-buffers/docs/reference/cpp/index#google.protobuf.compiler)  [*repeated\_field.h*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field) |  |

##### google::protobuf

Core components of the Protocol Buffers runtime library.

The files in this package represent the core of the Protocol Buffer system. All of them are part of the libprotobuf library.

A note on thread-safety:

Thread-safety in the Protocol Buffer library follows a simple rule: unless explicitly noted otherwise, it is always safe to use an object from multiple threads simultaneously as long as the object is declared const in all threads (or, it is only used in ways that would be allowed if it were declared const). However, if an object is accessed in one thread in a way that would not be allowed if it were const, then it is not safe to access that object in any other thread simultaneously.

Put simply, read-only access to an object can happen in multiple threads simultaneously, but write access can only happen in a single thread at a time.

The implementation does contain some "const" methods which actually modify the object behind the scenes – e.g., to cache results – but in these cases mutex locking is used to make the access thread-safe.

|  |  |
| --- | --- |
| Files | |
| [google/protobuf/descriptor.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor)  *This file contains classes which describe a type of protocol message.* |  |
| [google/protobuf/descriptor.pb.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.pb)  *Protocol buffer representations of descriptors.* |  |
| [google/protobuf/descriptor\_database.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor_database)  *Interface for manipulating databases of descriptors.* |  |
| [google/protobuf/dynamic\_message.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.dynamic_message)  *Defines an implementation of*[*Message*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message#Message)*which can emulate types which are not known at compile-time.* |  |
| [google/protobuf/message.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message)  *Defines*[*Message*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message#Message)*, the abstract interface implemented by non-lite protocol message objects.* |  |
| [google/protobuf/message\_lite.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message_lite)  *Defines*[*MessageLite*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.message_lite#MessageLite)*, the abstract interface implemented by all (lite and non-lite) protocol message objects.* |  |
| [google/protobuf/repeated\_field.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)  [*RepeatedField*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field#RepeatedField)*and*[*RepeatedPtrField*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field#RepeatedPtrField)*are used by generated protocol message classes to manipulate repeated fields.* |  |
| [google/protobuf/service.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.service)  *DEPRECATED: This module declares the abstract interfaces underlying proto2 RPC services.* |  |
| [google/protobuf/text\_format.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.text_format)  *Utilities for printing and parsing protocol messages in a human-readable, text-based format.* |  |
| [google/protobuf/unknown\_field\_set.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.unknown_field_set)  *Contains classes used to keep track of unrecognized fields seen while parsing a protocol message.* |  |
| [google/protobuf/stubs/common.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.common)  *Contains basic types and utilities used by the rest of the library.* |  |

##### google::protobuf::io

Auxiliary classes used for I/O.

The Protocol Buffer library uses the classes in this package to deal with I/O and encoding/decoding raw bytes. Most users will not need to deal with this package. However, users who want to adapt the system to work with their own I/O abstractions – e.g., to allow Protocol Buffers to be read from a different kind of input stream without the need for a temporary buffer – should take a closer look.

|  |  |
| --- | --- |
| Files | |
| [google/protobuf/io/coded\_stream.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.coded_stream)  *This file contains the*[*CodedInputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.coded_stream#CodedInputStream)*and*[*CodedOutputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.coded_stream#CodedOutputStream)*classes, which wrap a*[*ZeroCopyInputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream#ZeroCopyInputStream)*or*[*ZeroCopyOutputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream#ZeroCopyOutputStream)*, respectively, and allow you to read or write individual pieces of data in various formats.* |  |
| [google/protobuf/io/gzip\_stream.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.gzip_stream)  *This file contains the definition for classes*[*GzipInputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.gzip_stream#GzipInputStream)*and*[*GzipOutputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.gzip_stream#GzipOutputStream)*.* |  |
| [google/protobuf/io/printer.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.printer)  *Utility class for writing text to a*[*ZeroCopyOutputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream#ZeroCopyOutputStream)*.* |  |
| [google/protobuf/io/tokenizer.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.tokenizer)  *Class for parsing tokenized text from a*[*ZeroCopyInputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream#ZeroCopyInputStream)*.* |  |
| [google/protobuf/io/zero\_copy\_stream.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream)  *This file contains the*[*ZeroCopyInputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream#ZeroCopyInputStream)*and*[*ZeroCopyOutputStream*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream#ZeroCopyOutputStream)*interfaces, which represent abstract I/O streams to and from which protocol buffers can be read and written.* |  |
| [google/protobuf/io/zero\_copy\_stream\_impl.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream_impl)  *This file contains common implementations of the interfaces defined in*[*zero\_copy\_stream.h*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream)*which are only included in the full (non-lite) protobuf library.* |  |
| [google/protobuf/io/zero\_copy\_stream\_impl\_lite.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream_impl_lite)  *This file contains common implementations of the interfaces defined in*[*zero\_copy\_stream.h*](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.io.zero_copy_stream)*which are included in the "lite" protobuf library.* |  |

##### google::protobuf::compiler

[repeated\_field.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.repeated_field)

Implementation of the Protocol Buffer compiler.

This package contains code for parsing .proto files and generating code based on them. There are two reasons you might be interested in this package:

* You want to parse .proto files at runtime. In this case, you should look at [importer.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.importer). Since this functionality is widely useful, it is included in the libprotobuf base library; you do not have to link against libprotoc.
* You want to write a custom protocol compiler which generates different kinds of code, e.g. code in a different language which is not supported by the official compiler. For this purpose,[command\_line\_interface.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.command_line_interface) provides you with a complete compiler front-end, so all you need to do is write a custom implementation of [CodeGenerator](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.code_generator#CodeGenerator) and a trivial main() function. You can even make your compiler support the official languages in addition to your own. Since this functionality is only useful to those writing custom compilers, it is in a separate library called "libprotoc" which you will have to link against.

|  |  |
| --- | --- |
| Files | |
| [google/protobuf/compiler/code\_generator.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.code_generator)  *Defines the abstract interface implemented by each of the language-specific code generators.* |  |
| [google/protobuf/compiler/command\_line\_interface.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.command_line_interface)  *Implements the Protocol Compiler front-end such that it may be reused by custom compilers written to support other languages.* |  |
| [google/protobuf/compiler/importer.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.importer)  *This file is the public interface to the .proto file parser.* |  |
| [google/protobuf/compiler/parser.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.parser)  *Implements parsing of .proto files to FileDescriptorProtos.* |  |
| [google/protobuf/compiler/plugin.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin)  *Front-end for protoc code generator plugins written in C++.* |  |
| [google/protobuf/compiler/plugin.pb.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb)  *API for protoc plugins.* |  |
| [google/protobuf/compiler/cpp/cpp\_generator.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.cpp_generator)  *Generates C++ code for a given .proto file.* |  |
| [google/protobuf/compiler/java/java\_generator.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.java_generator)  *Generates Java code for a given .proto file.* |  |
| [google/protobuf/compiler/python/python\_generator.h](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.python_generator)  *Generates Python code for a given .proto file.* |  |

### Java Reference

#### [Java Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/java-generated)

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 .protofiles 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 onGeneratedMessage'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.jarinstead 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()) 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(com.google.protobuf.Descriptors.FieldDescriptor)).
* 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 callingFoo.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 JavaString. 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. LikeFoo, 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 ofFoo.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 returntrue and getFoo() will return value.
* Builder clearFoo(): Clears the value of the field. After calling this, hasFoo() will return false andgetFoo() 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 (possibly the instance returned by Foo.getDefaultInstance()).

In addition, the compiler generates two accessor methods that allow you to access the relevant subbuilders for message types. The following method is generated in both the message class and its builder:

* FooOrBuilder getFooOrBuilder(): Returns the builder for the field, if it already exists, or the message if not.

The compiler generates the following method only in the message's builder.

* Builder getFooBuilder(): Returns the builder for the field.

###### 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 removeFoo(int index): Removes the element at the given zero-based index.
* 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<Builder> getFooBuilderList(): Returns the entire field as a list of builders.

###### Oneof Fields

For this oneof field definition:

oneof oneof\_name {  
    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 oneof case is FOO.
* int getFoo(): Returns the current value of oneof\_name if the oneof case is FOO. Otherwise, returns the default value of this field.

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

* Builder setFoo(int value): Sets oneof\_name to this value and sets the oneof case to FOO. After calling this, hasFoo() will return true, getFoo() will return value and getOneofCase() will return FOO.
* Builder clearFoo():
  + Nothing will be changed if the oneof case is not FOO.
  + If the oneof case is FOO, sets oneof\_name to null and the oneof case to ONEOF\_NAME\_NOT\_SET. After calling this, hasFoo() will return false, getFoo() will return the default value andgetOneofCase() will return ONEOF\_NAME\_NOT\_SET.

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/reference/language.shtml#scalar). For message and enum types, the value type is replaced with the message or enum class.

###### Map Fields

For this map field definition:

map<int32, int32> weight = 1;

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

* Map<Integer, Integer> getWeight();: Returns an unmodifiable Map.

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

* Map<Integer, Integer> getMutableWeight();: Returns a mutable Map. Note that multiple calls to this method may return different map instances. The returned map reference may be invalidated by any subsequent method calls to the Builder.

##### Oneofs

Given a oneof definition like this:

oneof oneof\_name {  
    int32 foo\_int = 4;  
    string foo\_string = 9;  
    ...  
}

All the fields in the oneof\_name oneof will use the shared oneof\_name object for their value. In addition, the protocol buffer compiler will generate a Java enum type for the oneof case, as follows:

public enum OneofNameCase  
        implements com.google.protobuf.Internal.EnumLite {  
      FOO\_INT(4),  
      FOO\_STRING(9),  
      ...  
      ONEOF\_NAME\_NOT\_SET(0);  
      ...  
    };

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.
* static Foo valueOf(int value): Returns the enum object corresponding to the given numeric value (or null for other numeric values).

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

* OneofNameCase getOneofNameCase(): Returns the enum indicating which field is set. ReturnsONEOF\_NAME\_NOT\_SET if none of them is set.

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

* Builder clearOneofName(): Sets oneof\_name to null, and sets the oneof case toONEOF\_NAME\_NOT\_SET.

##### 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(), andgetExtensionCount(). Additionally, Foo.Builder will inherit methods setExtension() andclearExtension(). 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(com.google.protobuf.Descriptors.MethodDescriptor, com.google.protobuf.RpcController, com.google.protobuf.Message, com.google.protobuf.RpcCallback)), except that the methodargument 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()): 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(com.google.protobuf.Descriptors.MethodDescriptor, com.google.protobuf.RpcController, com.google.protobuf.Message, com.google.protobuf.RpcCallback)): 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(com.google.protobuf.Descriptors.MethodDescriptor)) and [getRequestPrototype](https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Service.html#getRequestPrototype(com.google.protobuf.Descriptors.MethodDescriptor)): 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 implementingFoo.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(com.google.protobuf.Descriptors.MethodDescriptor, com.google.protobuf.RpcController, com.google.protobuf.Message, com.google.protobuf.Message, com.google.protobuf.RpcCallback)).

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.BlockingStubreturns 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. TYPENAMEis 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.

#### [Java API (Javadoc)](https://developers.google.com/protocol-buffers/docs/reference/java/index)

See <https://developers.google.com/protocol-buffers/docs/reference/java/index> for detail.

### Python Reference

#### [Python Generated Code Guide](https://developers.google.com/protocol-buffers/docs/reference/python-generated)

This page describes exactly what Python definitions 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.

The Python Protocol Buffers implementation is a little different from C++ and Java. In Python, the compiler only outputs code to build descriptors for the generated classes, and a [Python metaclass](https://docs.python.org/2.7/reference/datamodel.html#metaclasses) does the real work. This document describes what you get *after* the metaclass has been applied.

##### Compiler Invocation

The protocol buffer compiler produces Python output when invoked with the --python\_out= command-line flag. The parameter to the --python\_out= option is the directory where you want the compiler to write your Python output. The compiler creates a .py file for each .proto file input. The names of the output files are computed by taking the name of the .proto file and making two changes:

* The extension (.proto) is replaced with \_pb2.py.
* The proto path (specified with the --proto\_path= or -I command-line flag) is replaced with the output path (specified with the --python\_out= flag).

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

protoc --proto\_path=src --python\_out=build/gen src/foo.proto src/bar/baz.proto

The compiler will read the files src/foo.proto and src/bar/baz.proto and produce two output files:build/gen/foo\_pb2.py and build/gen/bar/baz\_pb2.py. The compiler will automatically create the directory build/gen/bar if necessary, but it will *not* create build or build/gen; they must already exist.

Note that if the .proto file or its path contains any characters which cannot be used in Python module names (for example, hyphens), they will be replaced with underscores. So, the file foo-bar.proto becomes the Python file foo\_bar\_pb2.py.

When outputting Python code, the protocol buffer compiler's ability to output directly to ZIP archives is particularly convenient, as the Python interpreter is able to read directly from these archives if placed in the **PYTHONPATH**. To output to a ZIP file, simply provide an output location ending in **.zip**.

The number 2 in the extension **\_pb2.py** designates version 2 of Protocol Buffers. Version 1 was used primarily inside Google, though you might be able to find parts of it included in other Python code that was released before Protocol Buffers. Since version 2 of Python Protocol Buffers has a completely different interface, and since Python does not have compile-time type checking to catch mistakes, we chose to make the version number be a prominent part of generated Python file names.

##### Packages

The Python code generated by the protocol buffer compiler is completely unaffected by the package name defined in the .proto file. Instead, Python packages are identified by directory structure.

##### Messages

Given a simple message declaration:

message Foo {}

The protocol buffer compiler generates a class called Foo, which subclasses [google.protobuf.Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class). The class is a concrete class; no abstract methods are left unimplemented. Unlike C++ and Java, Python generated code is unaffected by the optimize\_for option in the .proto file; in effect, all Python code is optimized for code size.

You should *not* create your own Foo subclasses. Generated classes are not designed for subclassing and may lead to "fragile base class" problems. Besides, implementation inheritance is bad design.

Python message classes have no particular public members other than those defined by the [Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class) interface and those generated for nested fields, messages, and enum types (described below). Message provides methods you can use to check, manipulate, read, or write the entire message, including parsing from and serializing to binary strings. In addition to these methods, the Foo class defines the following static methods:

* FromString(s): Returns a new message instance deserialized from the given string.

Note that you can also use the [text\_format](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.text_format-module) module to work with protocol messages in text format: for example, the Merge() method lets you merge an ASCII representation of a message into an existing message.

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

In this case, the Bar class is declared as a static member of Foo, so you can refer to it as Foo.Bar.

##### Fields

For each field in a message type, the corresponding class has a member with the same name as the field. How you can manipulate the member depends on its type.

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 constantFOO\_BAR\_FIELD\_NUMBER = 5.

###### Singular Fields

If you have a singular (optional or required) field foo of any non-message type, you can manipulate the fieldfoo as if it were a regular field. For example, if foo's type is int32, you can say:

message.foo = 123

print message.foo

Note that setting foo to a value of the wrong type will raise a TypeError.

If foo is read when it is not set, its value is the default value for that field. To check if foo is set, or to clear the value of foo, you must call the HasField() or ClearField() methods of the [Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class) interface. For example:

assert not message.HasField("foo")

message.foo = 123

assert message.HasField("foo")

message.ClearField("foo")

assert not message.HasField("foo")

###### Singular Message Fields

Message types work slightly differently. You cannot assign a value to an embedded message field. Instead, assigning a value to any field within the child message implies setting the message field in the parent. So, for example, let's say you have the following .proto definition:

message Foo {

optional Bar bar = 1;

}

message Bar {

optional int32 i = 1;

}

You *cannot* do the following:

foo = Foo()

foo.bar = Bar() # WRONG!

Instead, to set bar, you simply assign a value directly to a field within bar, and - presto! - foo has a bar field:

foo = Foo()

assert not foo.HasField("bar")

foo.bar.i = 1

assert foo.HasField("bar")

assert foo.bar.i == 1

foo.ClearField("bar")

assert not foo.HasField("bar")

assert foo.bar.i == 0 # Default value

Similarly, you can set bar using the [Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class) interface's CopyFrom() method. This copies all the values from another message of the same type as bar.

foo.bar.CopyFrom(baz)

Note that simply reading a field inside bar does *not* set the field:

foo = Foo()

assert not foo.HasField("bar")

print foo.bar.i # Print i's default value

assert not foo.HasField("bar")

###### Repeated Fields

Repeated fields are represented as an object that acts like a Python sequence. As with embedded messages, you cannot assign the field directly, but you can manipulate it. For example, given this message definition:

message Foo {

repeated int32 nums = 1;

}

You can do the following:

foo = Foo()

foo.nums.append(15) # Appends one value

foo.nums.extend([32, 47]) # Appends an entire list

assert len(foo.nums) == 3

assert foo.nums[0] == 15

assert foo.nums[1] == 32

assert foo.nums == [15, 32, 47]

foo.nums[1] = 56 # Reassigns a value

assert foo.nums[1] == 56

for i in foo.nums: # Loops and print

print i

del foo.nums[:] # Clears list (works just like in a Python list)

The ClearField() method of the [Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class) interface works as well in addition to using Python del.

###### Repeated Message Fields

Repeated messages works similar to repeated scalar fields, except the corresponding Python object does not have an append() function. Instead, it has an add() function that creates a new message object, appends it to the list, and returns it for the caller to fill in. It also has an extend() function that appends an entire list of messages, but makes a **copy** of every message in the list. This is done so that messages are always owned by the parent message to avoid circular references and other confusion that can happen when a mutable data structure has multiple owners.

For example, given this message definition:

message Foo {

repeated Bar bars = 1;

}

message Bar {

optional int32 i = 1;

optional int32 j = 2;

}

You can do the following:

foo = Foo()

bar = foo.bars.add() # Adds a Bar then modify

bar.i = 15

foo.bars.add().i = 32 # Adds and modify at the same time

new\_bar = Bar()

new\_bar.i = 47

foo.bars.extend([new\_bar]) # Uses extend() to copy

assert len(foo.bars) = 3

assert foo.bars[0].i == 15

assert foo.bars[1].i == 32

assert foo.bars[2].i == 47

assert foo.bars[2] == new\_bar # The extended message is equal,

assert foo.bars[2] is not new\_bar # but it is a copy!

foo.bars[1].i = 56 # Modifies a single element

assert foo.bars[1].i == 56

for bar in foo.bars: # Loops and print

print bar.i

del foo.bars[:] # Clears list

# add() also forwards keyword arguments to the concrete class.

# For example, you can do:

foo.bars.add(i = 12, j = 13)

##### Enumerations

In Python, enums are just integers. A set of integral constants are defined corresponding to the enum's defined values. For example, given:

message Foo {

enum SomeEnum {

VALUE\_A = 1;

VALUE\_B = 5;

VALUE\_C = 1234;

}

optional SomeEnum bar = 1;

}

The constants VALUE\_A, VALUE\_B, and VALUE\_C are defined with values 1, 5, and 1234, respectively. No type corresponding to SomeEnum is defined. If an enum is defined in the outer scope, the values are module constants; if it is defined within a message (like above), they become static members of that message class.

An enum field works just like a scalar field. It does **not** do any type checking in the setter or getter.

foo = Foo()

foo.bar = Foo.VALUE\_A

assert foo.bar == 1

assert foo.bar == Foo.VALUE\_A

Note that in C++ and Java, an enum field cannot contain a numeric value other than those defined for the enum type. If an unknown enum value is encountered while parsing, the field will be treated as if its tag number were unknown. Therefore, you should never assign an enum field to an undefined value in Python, either. A future version of the library may explicitly disallow this.

Enums have a number of utility methods for getting field names from values and vice versa, lists of fields, and so on - these are defined in [enum\_type\_wrapper.py](https://github.com/google/protobuf/blob/master/python/google/protobuf/internal/enum_type_wrapper.py). So, for example, if you have the following standalone enum in myproto.proto:

enum SomeEnum {  
    VALUE\_A = 1;  
    VALUE\_B = 5;  
    VALUE\_C = 1234;  
}

..you can do this:

self.assertEqual('VALUE\_A', myproto\_pb2.SomeEnum.Name(myproto\_pb2.VALUE\_A))

##### Oneof

Given a message with a oneof:

message Foo {  
  oneof test\_oneof {  
     string name = 1;  
     int32 serial\_number = 2;  
  }  
}

The Python class corresponding to Foo will have members called name and serial\_number with accessor methods just like regular [fields](https://developers.google.com/protocol-buffers/docs/reference/python-generated#fields). However, unlike regular fields, at most one of the fields in a oneof can be set at a time, which is ensured by the runtime. For example:

message = Foo()

message.name = "Bender"

assert message.HasField("name")

message.serial\_number = 2716057

assert message.HasField("serial\_number")

assert not message.HasField("name")

The message class also has a WhichOneof method that lets you find out which field (if any) in the oneof has been set. This method returns the name of the field that is set, or None if nothing has been set:

assert message.WhichOneof("test\_oneof") is None

message.name = "Bender"

assert message.WhichOneof("test\_oneof") == "name"

HasField and ClearField also accept oneof names in addition to field names:

assert not message.HasField("test\_oneof")

message.name = "Bender"

assert message.HasField("test\_oneof")

message.serial\_number = 2716057

assert message.HasField("test\_oneof")

message.ClearField("test\_oneof")

assert not message.HasField("test\_oneof")

assert not message.HasField("serial\_number")

Note that calling ClearField on a oneof just clears the currently set field.

##### Extensions

Given a message with an extension range:

message Foo {

extensions 100 to 199;

}

The Python class corresponding to Foo will have a member called Extensions, which is a dictionary mapping extension identifiers to their current values.

Given an extension definition:

extend Foo {

optional int32 bar = 123;

}

The protocol buffer compiler generates an "extension identifier" called bar. The identifier acts as a key to theExtensions dictionary. The result of looking up a value in this dictionary is exactly the same as if you accessed a normal field of the same type. So, given the above example, you could do:

foo = Foo()

foo.Extensions[proto\_file\_pb2.bar] = 2

assert foo.Extensions[proto\_file\_pb2.bar] == 2

Note that you need to specify the extension identifier constant, not just a string name: this is because it's possible for multiple extensions with the same name to be specified in different scopes.

Analogous to normal fields, Extensions[...] returns a message object for singular messages and a sequence for repeated fields.

The [Message](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.message.Message-class) interface's HasField() and ClearField() methods do not work with extensions; you must use HasExtension() and ClearExtension() instead.

##### Services

If the .proto file contains the following line:

option py\_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 py\_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 a class Foo to represent this service. Foo will have a method for each method defined in the service definition. In this case, the method Bar is defined as:

def Bar(self, rpc\_controller, request, done)

The parameters are equivalent to the parameters of [Service.CallMethod()](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.service.Service-class#CallMethod), except that themethod\_descriptor argument is implied.

These generated methods are intended to be overridden by subclasses. The default implementations simply callcontroller.[SetFailed()](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.service.RpcController-class#SetFailed) with an error message indicating that the method is unimplemented, then invoke the done callback. When implementing your own service, you must subclass this generated service and implement its methods as appropriate.

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

* GetDescriptor: Returns the service's [ServiceDescriptor](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.descriptor.ServiceDescriptor-class).
* CallMethod: Determines which method is being called based on the provided method descriptor and calls it directly.
* GetRequestClass and GetResponseClass: Returns the class of the request or response of the correct type for the given method.

###### 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.

Foo\_Stub is a subclass of Foo. Its constructor takes an [RpcChannel](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.service.RpcChannel-class) as a parameter. The stub then implements each of the service's methods by calling the channel's CallMethod() method.

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/python/google.protobuf.service.RpcChannel-class) and [RpcController](https://developers.google.com/protocol-buffers/docs/reference/python/google.protobuf.service.RpcController-class).

##### 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 Python code generator may insert code of the following types using the given insertion point names.

* imports: Import statements.
* module\_scope: Top-level declarations.
* class\_scope:TYPENAME: Member declarations that belong in a message class. TYPENAME is the full proto name, e.g. package.MessageType.

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.

##### C++ Implementation

There is also an experimental C++ implementation for Python messages via a Python extension for better performance. Implementation type is controlled by an environment variablePROTOCOL\_BUFFERS\_PYTHON\_IMPLEMENTATION (valid values: "cpp" and "python"). The default value is currently "python" but will be changed to "cpp" in future release.

Note that the environment variable needs to be set before installing the protobuf library, in order to build and install the python extension. The C++ implementation also requires CPython platforms. Seepython/INSTALL.txt for detailed install instructions.

#### [Python API (Epydoc)](https://developers.google.com/protocol-buffers/docs/reference/python/index)

See <https://developers.google.com/protocol-buffers/docs/reference/python/index> for detail.

### Protocol Buffers Reference

#### [Protocol Buffers Version 2 Language Specification](https://developers.google.com/protocol-buffers/docs/reference/proto2-spec)

This is a language specification reference for version 2 of the Protocol Buffers language (proto2). For more information about using proto2, see the the [language guide](https://developers.google.com/protocol-buffers/docs/proto).

##### Lexical elements

###### Letters and digits

letter = "A" … "Z" | "a" … "z"

capitalLetter = "A" … "Z"

decimalDigit = "0" … "9"

octalDigit = "0" … "7"

hexDigit = "0" … "9" | "A" … "F" | "a" … "f"

###### Identifiers

ident = letter { letter | unicodeDigit | "\_" }

fullIdent = ident {"." ident}

messageName = ident

enumName = ident

fieldName = ident

oneofName = ident

mapName = ident

serviceName = ident

rpcName = ident

streamName = ident

messageType = ["."] {ident "."} messageName

enumType = ["."] {ident "."} enumName

groupName = capital { letter | unicodeDigit | "\_" }

###### Integer literals

intLit = decimalLit | octalLit | hexLit

decimalLit = ( "1" … "9" ) { decimalDigit }

octalLit = "0" { octalDigit }

hexLit = "0" ( "x" | "X" ) hexDigit { hexDigit }

###### Floating-point literals

floatLit = decimals "." [ decimals ] [ exponent ] | decimals exponent | "."decimals [exponent ]

decimals = decimalDigit { decimalDigit }

exponent = ( "e" | "E" ) [ "+" | "-" ]decimals

###### Boolean

boolLit = "true" | "false"

###### String literals

strLit = ("`" { charValue } "`") | (`"` { charValue } `"`)

charVaue = hexEscape | octEscape | charEscape | /[^\0\n\\]/

hexEscape = `\` ("x" | "X") hexDigit hexDigit

octEscape = `\` octalDigit octalDigit octalDigit

charEscape = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` )

quote = "`"|`"`

###### EmptyStatement

emptyStatement = ";"

##### Syntax

The syntax statement is used to define the protobuf version.

syntax = "syntax" "=" quote "proto2" quote ";"

##### Import Statement

The import statement is used to import another .proto's definitions.

import = "import" [ "weak" | “public”] strLit ";"

Example:

import public “other.proto”;

##### Package

The package specifier can be used to prevent name clashes between protocol message types.

package = "package" fullIdent ";"

Example:

package foo.bar;

##### Option

Options can be used in proto files, messages, enums and services. An option can be a protobuf defined option or a custom option. For more information, see [Options](https://developers.google.com/protocol-buffers/docs/proto#options) in the language guide.

option = "option" optionName "=" constant ";"

optionName = (ident | "(" fullIdent ")") {"." ident}

For examples:

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

##### Fields

Fields are the basic elements of a protocol buffer message. Fields can be normal fields, group fields, oneof fields, or map fields. A field has a label, type and field number.

label = "required" | "optional" | "repeated"

type = "double" | "float" | "int32" | "int64" | "uint32" | "uint64"

| "sint32" | "sint64" | "fixed32" | "fixed64" | "sfixed32" | "sfixed64"

| "bool" | "string" | "bytes" | messageType | enumType

fieldNumber = intLit;

###### Normal field

Each field has label, type, name and field number. It may have field options.

field = label type fieldName "=" fieldNumber [ "[" fieldOptions "]" ] ";"

fieldOptions = fieldOption { "," fieldOption }

fieldOption = optionName "=" constant

Example:

optional foo.bar nested\_message = 2;  
repeated int32 samples = 4 [packed=true];

###### Group field

Groups are one way to nest information in message definitions. The group name must begin with capital letter.

group = label "group" groupName "=" fieldNumber messageBody

Example:

example:  
  repeated group Result = 1 {  
      required string url = 2;  
      optional string title = 3;  
      repeated string snippets = 4;  
    }

###### Oneof and oneof field

A oneof consists of oneof fields and a oneof name. Oneof fields do not have labels.

oneof = "oneof" oneofName "{" {oneofField | emptyStatement} "}"

oneofField = type fieldName "=" fieldNumber ["[" fieldOptions "]" ] ";"

Example:

oneof foo {  
    string name = 4;  
    SubMessage sub\_message = 9;  
}

###### Map field

A map field has a key type, value type, name, and field number. The key type can be any integral or string type.

mapField = "map" "<" keyType "," type ">" mapName "=" fieldNumber ";"

keyType = "int32" | "int64" | "uint32" | "uint64" | "sint32" | "sint64" |

"fixed32" | "fixed64" | "sfixed32" | "sfixed64" | "bool" | "string"

Example:

map<string, Project> projects = 3;

##### Extensions and Reserved

Extensions and reserved are message elements that declare a range of field numbers or field names.

###### Extensions

Extensions declare that a range of field numbers in a message are available for third-party extensions. Other people can declare new fields for your message type with those numeric tags in their own .proto files without having to edit the original file.

extensions = "extensions" ranges ";"

ranges = range { "," range }

range = intLit ["to" ( intLit | "max" )]

Examples:

examples:  
  extensions 100 to 199;  
  extensions 4, 20 to max;

###### Reserved

Reserved declares a range of field numbers or field names in a message that can not be used.reserved = "reserved" ( ranges | fieldNames) ";"fieldNames = fieldName { "," fieldName }

reserved 2, 15, 9 to 11;  
reserved "foo", "bar";

##### Top Level definitions

###### Enum definition

The enum definition consists of a name and an enum body. The enum body can have options and enum fields.

enum = "enum" enumName enumBody

enumBody = "{" { option | enumField | emptyStatement } "}"

enumField = ident "=" intLit [ "[" enumValueOption { "," enumValueOption } "]" ]";"

enumValueOption = optionName "=" constant

Example:

enum EnumAllowingAlias {  
  option allow\_alias = true;  
  UNKNOWN = 0;  
  STARTED = 1;  
  RUNNING = 2 [(custom\_option) = "hello world"];  
}

###### Message definition

A message consists of a message name and a message body. The message body can have fields, nested enum definitions, nested message definitions, extend statements, extensions, groups, options, oneofs, map fields, and reserved statements.

message = "message" messageName messageBody

messageBody = "{" { field | enum | message | extend | extensions | group |

option | oneof | mapField | reserved | emptyStatement } "}"

Example:

message Outer {  
  option (my\_option).a = true;  
  message Inner {   // Level 2  
    required int64 ival = 1;  
  }  
  map<int32, string> my\_map = 2;  
  extensions 20 to 30;  
}

###### Extend

If a message in the same or imported .proto file has reserved a range for extensions, the message can be extended.

extend = "extend" messageType "{" {field | group | emptyStatement} "}"

Example:

extend Foo {  
  optional int32 bar = 126;  
}

###### Service definition

service = "service" serviceName "{" { option | rpc | stream | emptyStatement } "}"

rpc = "rpc" rpcName "(" ["stream"] messageType ")" "returns" "(" ["stream"]

messageType ")" (("{" {option | emptyStatement } "}") | ";")

stream = "stream" streamName "(" messageType "," messageType ")" (("{"

{option | emptyStatement} "}") | ";")

Example:

service SearchService {  
  rpc Search (SearchRequest) returns (SearchResponse);  
}

##### Proto file

proto = syntax { import | package | option | topLevelDef | emptyStatement }

topLevelDef = message | enum | extend | service

An example .proto file:

syntax = “proto2”;  
import public “other.proto”;  
option java\_package = "com.example.foo";  
enum EnumAllowingAlias {  
  option allow\_alias = true;  
  UNKNOWN = 0;  
  STARTED = 1;  
  RUNNING = 2 [(custom\_option) = "hello world"];  
}  
message outer {  
  option (my\_option).a = true;  
  message inner {   // Level 2  
    required int64 ival = 1;  
  }  
  repeated inner inner\_message = 2;  
  optional EnumAllowingAlias enum\_field =3;  
  map<int32, string> my\_map = 4;  
  extensions 20 to 30;  
}  
message foo {  
  optional group GroupMessage {  
    optional a = 1;  
  }  
}

### Other Languages

While the current release just includes compilers and APIs for C++, Java, and Python, the compiler code is designed so that it's easy to add support for other languages. There are several ongoing projects to add new language implementations to Protocol Buffers, including C, C#, Haskell, Perl, Ruby, and more.

For a list of links to projects we know about, see the [third-party add-ons wiki page](https://github.com/google/protobuf/wiki/Third-Party-Add-ons).

#### Compiler Plugins

As of version 2.3.0 (January 2010), protoc, the Protocol Buffers Compiler, can be extended to support new languages via plugins. A plugin is just a program which reads a CodeGeneratorRequest protocol buffer from standard input and then writes a CodeGeneratorResponse protocol buffer to standard output. These message types are defined in [plugin.proto](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb). We recommend that all third-party code generators be written as plugins, as this allows all generators to provide a consistent interface and share a single parser implementation.

Additionally, plugins are able to insert code into the files generated by other code generators. See the comments about "insertion points" in [plugin.proto](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.plugin.pb) for more on this. This could be used, for example, to write a plugin which genreates RPC service code that is tailored for a particular RPC system. See the documentation for the generated code in each language to find out what insertion points they provide.

# SUPPORT

## FAQ

This document answers some frequently asked questions about the Protocol Buffers open source project. If you have a question that isn't answered here, join the [discussion group](http://groups.google.com/group/protobuf) and ask away!

### General

#### Why did you release protocol buffers?

There are several reasons:

* Protocol buffers are used by practically everyone inside Google. We have many other projects we would like to release as open source that use protocol buffers, so to do this, we needed to release protocol buffers first. In fact, bits of the technology have already found their way into the open – if you dig into the code for[Google AppEngine](https://developers.google.com/appengine/), you might find some of it.
* We would like to provide public APIs that accept protocol buffers as well as XML, both because it is more efficient and because we're just going to convert that XML to protocol buffers on our end anyway.
* We thought that people outside Google might find protocol buffers useful.
* Getting protocol buffers into a form we were happy to release was a fun [20% project](http://www.google.com/support/jobs/bin/static.py?page=about.html&about=eng).

#### Why is the first release version 2? What happened to version 1?

The initial version of protocol buffers (aka "Proto1") was developed in Google starting in early 2001, and evolved over the course of many years, sprouting new features whenever someone needed them and was willing to do the work themselves. Like anything created in such a way, it was a bit of a mess. We came to the conclusion that it would not be feasible to release the code as it was.

Version 2 ("Proto2") is a complete rewrite, though it keeps most of the design and uses many of the implementation ideas from Proto1. Some features have been added, some removed. Most importantly, though, the code is cleaned up and does not have any dependencies on Google libraries that have not yet been open-sourced.

#### Why the name "Protocol Buffers"?

The name originates from the early days of the format, before we had the protocol buffer compiler to generate classes for us. At the time, there was a class called ProtocolBuffer which actually acted as a buffer for an individual method. Users would add tag/value pairs to this buffer individually by calling methods likeAddValue(tag, value). The raw bytes were stored in a buffer which could then be written out once the message had been constructed.

Since that time, the "buffers" part of the name has lost its meaning, but it is still the name we use. Today, people usually use the term "protocol message" to refer to a message in an abstract sense, "protocol buffer" to refer to a serialized copy of a message, and "protocol message object" to refer to an in-memory object representing the parsed message.

#### Does Google have any patents on Protocol Buffers?

Google currently has no issued patents on Protocol Buffers, and we are happy to address any concerns around Protocol Buffers and patents that people may have.

### Similar Technologies

#### How do protocol buffers differ from XML?

See [the answer on the overview page](https://developers.google.com/protocol-buffers/docs/overview.html#whynotxml).

#### How do protocol buffers differ from ASN.1, COM, CORBA, Thrift, etc?

We think all of these systems have strengths and weaknesses. Google relies on protocol buffers internally and they are a vital component of our success, but that doesn't mean they are the ideal solution for every problem. You should evaluate each alternative in the context of your own project.

It is worth noting, though, that several of these technologies define both an interchange format and an RPC (remote procedure call) protocol. Protocol buffers are just an interchange format. They could easily be used for RPC – and, indeed, they do have limited support for defining [RPC services](https://developers.google.com/protocol-buffers/docs/proto.html#services) – but they are not tied to any one RPC implementation or protocol.

### Contributing

#### Can I add support for a new language to protocol buffers?

Yes! In fact, the protocol buffer compiler is designed such that it's easy to write your own compiler. Check out the[CommandLineInterface](https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.compiler.command_line_interface) class, which is available as part of the libprotoc library.

We encourage you to create code generators and runtime libraries for new languages. You should start your own, independent project for this – this way, you will have the freedom to manage your project as you see fit, and will not be held back by our release process. Please also join the [Protocol Buffers discussion group](http://groups.google.com/group/protobuf) and let us know about your project; we will be happy to link to it and help you out with design issues.

#### Can I contribute patches to protocol buffers?

Yes! Please join the [Protocol Buffers discussion group](http://groups.google.com/group/protobuf) and talk to us about it.

#### Can I add new features to protocol buffers?

Maybe. We always like suggestions, but we're very cautious about adding things. One thing we've learned over the years is that lots of people have interesting ideas for new features. Most of these features are very useful in specific cases, but if we accepted all of them, protocol buffers would become a bloated, confusing mess. So, we have to be very picky. When evaluating new features, we look for additions that are very widely useful or very simple – or hopefully both. We regularly turn down feature additions from Google employees. We even regularly turn down feature additions from our own team members.

That said, we'd still like to hear what you have in mind. Join the [Protocol Buffers discussion group](http://groups.google.com/group/protobuf) and let us know. We might be able to help you find a way to do what you want without changing the underlying library. Or, maybe we'll decide that your feature is so useful or so simple that it should be added.

## Forum

Check <https://groups.google.com/forum/#!forum/protobuf> for detail.