**Language Guide (proto3)**

* [目录](https://developers.google.com/protocol-buffers/docs/proto3" \l "top_of_page)
* [Defining A Message Type](https://developers.google.com/protocol-buffers/docs/proto3" \l "simple)
  + [Specifying Field Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "specifying-field-types)
  + [Assigning Tags](https://developers.google.com/protocol-buffers/docs/proto3" \l "assigning-tags)
  + [Specifying Field Rules](https://developers.google.com/protocol-buffers/docs/proto3" \l "specifying-field-rules)
  + [Adding More Message Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "adding-more-message-types)
  + [Adding Comments](https://developers.google.com/protocol-buffers/docs/proto3" \l "adding-comments)
  + [Reserved Fields](https://developers.google.com/protocol-buffers/docs/proto3" \l "reserved)
  + [What's Generated From Your .proto?](https://developers.google.com/protocol-buffers/docs/proto3" \l "whats-generated-from-your-proto)
* [Scalar Value Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "scalar)
* [Default Values](https://developers.google.com/protocol-buffers/docs/proto3" \l "default)
* [Enumerations](https://developers.google.com/protocol-buffers/docs/proto3" \l "enum)
* [Using Other Message Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "other)
  + [Importing Definitions](https://developers.google.com/protocol-buffers/docs/proto3" \l "importing-definitions)
  + [Using proto2 Message Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "using-proto2-message-types)
* [Nested Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "nested)
* [Updating A Message Type](https://developers.google.com/protocol-buffers/docs/proto3" \l "updating)
* [Unknown Fields](https://developers.google.com/protocol-buffers/docs/proto3" \l "unknowns)
* [Any](https://developers.google.com/protocol-buffers/docs/proto3" \l "any)
* [Oneof](https://developers.google.com/protocol-buffers/docs/proto3" \l "oneof)
  + [Using Oneof](https://developers.google.com/protocol-buffers/docs/proto3" \l "using-oneof)
  + [Oneof Features](https://developers.google.com/protocol-buffers/docs/proto3" \l "oneof-features)
  + [Backwards-compatibility issues](https://developers.google.com/protocol-buffers/docs/proto3" \l "backwards-compatibility-issues)
* [Maps](https://developers.google.com/protocol-buffers/docs/proto3" \l "maps)
  + [Backwards compatibility](https://developers.google.com/protocol-buffers/docs/proto3" \l "backwards-compatibility)
* [Packages](https://developers.google.com/protocol-buffers/docs/proto3" \l "packages)
  + [Packages and Name Resolution](https://developers.google.com/protocol-buffers/docs/proto3" \l "packages-and-name-resolution)
* [Defining Services](https://developers.google.com/protocol-buffers/docs/proto3" \l "services)
* [JSON Mapping](https://developers.google.com/protocol-buffers/docs/proto3" \l "json)
* [Options](https://developers.google.com/protocol-buffers/docs/proto3" \l "options)
  + [Custom Options](https://developers.google.com/protocol-buffers/docs/proto3" \l "custom_options)
* [Generating Your Classes](https://developers.google.com/protocol-buffers/docs/proto3" \l "generating)
* [Defining A Message Type](https://developers.google.com/protocol-buffers/docs/proto3" \l "simple)
* [Scalar Value Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "scalar)
* [Default Values](https://developers.google.com/protocol-buffers/docs/proto3" \l "default)
* [Enumerations](https://developers.google.com/protocol-buffers/docs/proto3" \l "enum)
* [Using Other Message Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "other)
* [Nested Types](https://developers.google.com/protocol-buffers/docs/proto3" \l "nested)
* [Updating A Message Type](https://developers.google.com/protocol-buffers/docs/proto3" \l "updating)
* [Unknown Fields](https://developers.google.com/protocol-buffers/docs/proto3" \l "unknowns)
* [Any](https://developers.google.com/protocol-buffers/docs/proto3" \l "any)
* [Oneof](https://developers.google.com/protocol-buffers/docs/proto3" \l "oneof)
* [Maps](https://developers.google.com/protocol-buffers/docs/proto3" \l "maps)
* [Packages](https://developers.google.com/protocol-buffers/docs/proto3" \l "packages)
* [Defining Services](https://developers.google.com/protocol-buffers/docs/proto3" \l "services)
* [JSON Mapping](https://developers.google.com/protocol-buffers/docs/proto3" \l "json)
* [Options](https://developers.google.com/protocol-buffers/docs/proto3" \l "options)
* [Generating Your Classes](https://developers.google.com/protocol-buffers/docs/proto3" \l "generating)

This guide describes how to use the protocol buffer language to structure your protocol buffer data, including .proto file syntax and how to generate data access classes from your .proto files. 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" \l "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" \l "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" \l "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 through 19999 (FieldDescriptor::kFirstReservedNumber through FieldDescriptor::kLastReservedNumber), as they are reserved for the Protocol Buffers implementation - the protocol buffer compiler will complain if you use one of these reserved numbers in your .proto. Similarly, you cannot use any previously [reserved](https://developers.google.com/protocol-buffers/docs/proto3" \l "reserved) tags.

**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.

In proto3, repeated fields of scalar numeric types use packed encoding by default.

You can find out more about packed encoding in [Protocol Buffer Encoding](https://developers.google.com/protocol-buffers/docs/encoding.html" \l "packed).

**Adding More Message Types**

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

message SearchRequest {

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.

}

**Reserved Fields**

If you [update](https://developers.google.com/protocol-buffers/docs/proto3" \l "updating) a message type by entirely removing a field, or commenting it out, future users can reuse the tag number when making their own updates to the type. This can cause severe issues if they later load old versions of the same .proto, including data corruption, privacy bugs, and so on. One way to make sure this doesn't happen is to specify that the field tags (and/or names, which can also cause issues for JSON serialization) of your deleted fields are reserved. The protocol buffer compiler will complain if any future users try to use these field identifiers.

message Foo {

reserved 2, 15, 9 to 11;

reserved "foo", "bar";

}

Note that you can't mix field names and tag numbers in the same reserved statement.

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

When you run the [protocol buffer compiler](https://developers.google.com/protocol-buffers/docs/proto3" \l "generating) on a .proto, the compiler generates the code in your chosen language you'll need to work with the message types you've described in the file, including getting and setting field values, serializing your messages to an output stream, and parsing your messages from an input stream.

* For **C++**, the compiler generates a .h and .cc file from each .proto, with a class for each message type described in your file.
* For **Java**, the compiler generates a .java file with a class for each message type, as well as a special Builder classes for creating message class instances.
* **Python** is a little different – the Python compiler generates a module with a static descriptor of each message type in your .proto, which is then used with a *metaclass* to create the necessary Python data access class at runtime.
* 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.
* For **Objective-C**, the compiler generates a pbobjc.h and pbobjc.m file from each .proto, with a class for each message type described in your file.
* For **C#**, the compiler generates a .cs file from each .proto, with a class for each message type described in your file.

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 .proto file, and the corresponding type in the automatically generated class:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **.proto Type** | **Notes** | **C++ Type** | **Java Type** | **Python Type[2]** | **Go Type** | **Ruby Type** | **C# Type** | **PHP Type** |
| double |  | double | double | float | float64 | Float | double | float |
| float |  | float | float | float | float32 | 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 | int32 | Fixnum or Bignum (as required) | int | integer |
| 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 | long | integer/string[5] |
| uint32 | Uses variable-length encoding. | uint32 | int[1] | int/long[3] | uint32 | Fixnum or Bignum (as required) | uint | integer |
| uint64 | Uses variable-length encoding. | uint64 | long[1] | int/long[3] | uint64 | Bignum | ulong | integer/string[5] |
| 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) | int | integer |
| sint64 | Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s. | int64 | long | int/long[3] | int64 | Bignum | long | integer/string[5] |
| 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) | uint | integer |
| fixed64 | Always eight bytes. More efficient than uint64 if values are often greater than 256. | uint64 | long[1] | int/long[3] | uint64 | Bignum | ulong | integer/string[5] |
| sfixed32 | Always four bytes. | int32 | int | int | int32 | Fixnum or Bignum (as required) | int | integer |
| sfixed64 | Always eight bytes. | int64 | long | int/long[3] | int64 | Bignum | long | integer/string[5] |
| bool |  | bool | boolean | bool | bool | TrueClass/FalseClass | bool | boolean |
| string | A string must always contain UTF-8 encoded or 7-bit ASCII text. | string | String | str/unicode[4] | string | String (UTF-8) | string | string |
| bytes | May contain any arbitrary sequence of bytes. | string | ByteString | str | []byte | String (ASCII-8BIT) | ByteString | string |

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).

[5] Integer is used on 64-bit machines and string is used on 32-bit machines.

**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" \l "enum), the default value is the **first defined enum value**, which must be 0.
* For message fields, the field is not set. Its exact value is language-dependent. See the [generated code guide](https://developers.google.com/protocol-buffers/docs/reference/overview) for details.

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.

See the [generated code guide](https://developers.google.com/protocol-buffers/docs/reference/overview) for your chosen language for more details about how defaults work in generated code.

**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 type Corpus:

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" \l "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 the allow\_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](https://developers.google.com/protocol-buffers/docs/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 .proto and then specify a field of type Result in SearchResponse:

message SearchResponse {

repeated Result results = 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 containing 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 directly in proto3 syntax (it's okay if an imported proto2 message uses them).

**Nested Types**

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

message SearchResponse {

message Result {

string url = 1;

string title = 2;

repeated string snippets = 3;

}

repeated Result results = 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" \l "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. See the [Unknown Fields](https://developers.google.com/protocol-buffers/docs/proto3" \l "unknowns) section for details.
* Fields can be removed, as long as the tag number is not used again in your updated message type. You may want to rename the field instead, perhaps adding the prefix "OBSOLETE\_", or make the tag [reserved](https://developers.google.com/protocol-buffers/docs/proto3" \l "reserved), 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.
* enum is compatible with int32, uint32, int64, and uint64 in terms of wire format (note that values will be truncated if they don't fit). However be aware that client code may treat them differently when the message is deserialized: for example, unrecognized proto3 enum types will be preserved in the message, but how this is represented when the message is deserialized is language-dependent. Int fields always just preserve their value.

**Unknown Fields**

Unknowns fields are well-formed protocol buffers serialized data representing fields that the parser does not recognize. For example, when an old binary parses data sent by a new binary with new fields, those new fields become unknown fields in the old binary.

Proto3 implementations can parse messages with unknown fields successfully, however, implementations may or may not support preserving those unknown fields. You should not rely on unknown fields being preserved or dropped. For most Google protocol buffers implementations, unknown fields are not accessible in proto3 via the corresponding proto runtimes, and are dropped and forgotten at deserialization time. This is different behaviour to [proto2](https://developers.google.com/protocol-buffers/docs/proto), where unknown fields are always preserved and serialized along with the message.

**Any**

The Any message type lets you use messages as embedded types without having their .proto definition. An Any 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" \l "other) google/protobuf/any.proto.

import "google/protobuf/any.proto";  
  
message ErrorStatus {  
  string message = 1;  
  repeated google.protobuf.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 UnpackTo() 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.Is<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" \l "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 case test\_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 repeated fields.

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" \l "scalar) type except for floating point types and bytes). The value\_type can be any type except another map.

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.
* 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.
* When generating text format for a .proto, maps are sorted by key. Numeric keys are sorted numerically.
* When parsing from the wire or when merging, if there are duplicate map keys the last key seen is used. When parsing a map from text format, parsing may fail if there are duplicate keys.

The generated map API is currently available for all proto3 supported languages. 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.
* In **C#** the package is used as the namespace after converting to PascalCase, unless you explicitly provide an option csharp\_namespace in your .proto file. For example, Open would be in the namespace Foo.Bar.

**Packages and Name Resolution**

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

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

**Defining Services**

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

service SearchService {

rpc Search (SearchRequest) returns (SearchResponse);

}

The 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" \l "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/blob/master/docs/third_party.md).

**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" \l "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 | "YWJjMTIzIT8kKiYoKSctPUB+" | JSON value will be the data encoded as a string using standard base64 encoding with paddings. Either standard or URL-safe base64 encoding with/without paddings are accepted. |
| 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 | "1972-01-01T10:00:20.021Z" | Uses RFC 3339, where generated output will always be Z-normalized and uses 0, 3, 6 or 9 fractional digits. |
| Duration | string | "1.000340012s", "1s" | Generated output always contains 0, 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 explicit java\_package option is given in the .proto file, then by default the proto package (specified using the "package" keyword in the .proto file) will be used. However, proto packages generally do not make good Java packages since proto packages are not expected to start with reverse domain names. If not generating Java code, this option has no effect.

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

* java\_multiple\_files (file option): Causes top-level messages, enums, and services to be defined at the package level, rather than inside an outer class named after the .proto file.

option java\_multiple\_files = true;

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

option java\_outer\_classname = "Ponycopter";

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

option optimize\_for = CODE\_SIZE;

* cc\_enable\_arenas (file option): Enables [arena allocation](https://developers.google.com/protocol-buffers/docs/reference/arenas) for C++ generated code.
* objc\_class\_prefix (file option): Sets the Objective-C class prefix which is prepended to all Objective-C generated classes and enums from this .proto. There is no default. You should use prefixes that are between 3-5 uppercase characters as [recommended by Apple](https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Conventions/Conventions.html" \l "//apple_ref/doc/uid/TP40011210-CH10-SW4). Note that all 2 letter prefixes are reserved by Apple.
* deprecated (field option): If set to true, indicates that the field is deprecated and should not be used by new code. In most languages this has no actual effect. In Java, this becomes a @Deprecated annotation. In the future, other language-specific code generators may generate deprecation annotations on the field's accessors, which will in turn cause a warning to be emitted when compiling code which attempts to use the field. If the field is not used by anyone and you want to prevent new users from using it, consider replacing the field declaration with a [reserved](https://developers.google.com/protocol-buffers/docs/proto3" \l "reserved) statement.

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" \l "customoptions) for details. Note that creating custom options uses [extensions](https://developers.google.com/protocol-buffers/docs/proto.html" \l "extensions), which are permitted only for custom options in proto3.

**Generating Your Classes**

To generate the Java, Python, C++, Go, Ruby, JavaNano, Objective-C, 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. 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* --objc\_out=*DST\_DIR* --csharp\_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. See the [Go generated code reference](https://developers.google.com/protocol-buffers/docs/reference/go-generated) for more.
  + --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!
  + --objc\_out generates Objective-C code in DST\_DIR. See the [Objective-C generated code reference](https://developers.google.com/protocol-buffers/docs/reference/objective-c-generated) for more.
  + --csharp\_out generates C# code in DST\_DIR. See the [C# generated code reference](https://developers.google.com/protocol-buffers/docs/reference/csharp-generated) for more.
  + --php\_out generates PHP code in DST\_DIR. See the [PHP generated code reference](https://developers.google.com/protocol-buffers/docs/reference/php-generated) for more.

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

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