Language Guide (proto 3)

This topic covers how to use the version 3 of Protocol Buffers in your project.

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 **proto2** syntax, see the Proto2 Language Guide.

This is a reference guide – for a step by step example that uses many of the features described in this document, see the tutorial 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.

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_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. 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 earlier example, all the fields are scalar types: two integers (`page_number` and `results_per_page`) and a string (`query`). You can also specify enumerations and composite types like other message types for your field.

Assigning Field Numbers

You must give each field in your message definition a number between `1` and `536,870,911` with the following restrictions:

- The given number **must be unique** among all fields for that message.
- Field numbers `19,000` to `19,999` are reserved for the Protocol Buffers implementation. The protocol buffer compiler will complain if you use one of these reserved field numbers in your message.
- You cannot use any previously reserved field numbers or any field

This number **cannot be changed once your message type is in use** because it identifies the field in the message wire format. “Changing” a field number is equivalent to deleting that field and creating a new field with the same type but a new number. See Deleting Fields for how to do this properly.

Field numbers **should never be reused**. Never take a field number out of the reserved list for reuse with a new field definition. See Consequences of Reusing Field Numbers.

You should use the field numbers 1 through 15 for the most-frequently-set fields. Lower field number values take less space in the wire format. For example, field numbers in the range 1 through 15 take one byte to encode. Field numbers in the range 16 through 2047 take two bytes. You can find out more about this in Protocol Buffer Encoding.

Consequences of Reusing Field Numbers

Reusing a field number makes decoding wire-format messages ambiguous.

The protobuf wire format is lean and doesn’t provide a way to detect fields encoded using one definition and decoded using another.

Encoding a field using one definition and then decoding that same field with a different definition can lead to:

- Developer time lost to debugging
- A parse/merge error (best case scenario)
- Leaked PII/SPII
- Data corruption

Common causes of field number reuse:

- renumbering fields (sometimes done to achieve a more aesthetically pleasing number order for fields). Renumbering effectively deletes and re-adds all the fields involved in the renumbering, resulting in incompatible wire-format changes.
- deleting a field and not reserving the number to prevent future reuse.

The max field is 29 bits instead of the more-typical 32 bits because three lower bits are used for the wire format. For more on this, see the Encoding topic.

Specifying Field Labels

Message fields can be one of the following:

- `optional` : An `optional` field is in one of two possible states:
  - the field is set, and contains a value that was explicitly set or parsed from the wire. It will be serialized to the wire.
  - the field is unset, and will return the default value. It will not be serialized to the wire.

  You can check to see if the value was explicitly set.
- `repeated` : this field type can be repeated zero or more times in a well-formed message. The order of the repeated values will be preserved.
- `map` : this is a paired key/value field type. See Maps for more on this field type.
- If no explicit field label is applied, the default field label, called “implicit field presence,” is assumed. (You cannot explicitly set a field to this state.) A well-formed message can have zero or one of this field (but not more than one). You also cannot determine whether a field of this type

was parsed from the wire. An implicit presence field will be serialized to the wire unless it is the default value. For more on this subject, see Field Presence.

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.

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 results_per_page = 3;
}

message SearchResponse {
 ...
}
```

Adding Comments

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

```
/* SearchRequest represents a search query, with pagination options to
 * indicate which results to include in the response. */

message SearchRequest {
  string query = 1;
  int32 page_number = 2;  // Which page number do we want?
  int32 results_per_page = 3;  // Number of results to return per page.
}
```

Deleting Fields

Deleting fields can cause serious problems if not done properly.

When you no longer need a non-required field and all references have been deleted from client code, you may delete the field definition from the message. However, you **must** reserve the deleted field number. If you do not reserve the field number, it is possible for a developer to reuse that number in the future.

You should also reserve the field name to allow JSON and TextFormat encodings of your message to continue to parse.

Reserved Fields

If you update a message type by entirely deleting a field, or commenting it out, future developers can reuse the field number when making their own updates to the type. This can cause severe issues, as described in Consequences of Reusing Field Numbers.

To make sure this doesn't happen, add your deleted field number to the `reserved` list. To make sure JSON and TextFormat instances of your message can still be parsed, also add the deleted field name to a `reserved` list.

The protocol buffer compiler will complain if any future developers try to use these reserved field numbers or names.

```
message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

Reserved field number ranges are inclusive (`9 to 11` is the same as `9, 10, 11`). Note that you can't mix field names and field numbers in the same `reserved` statement.

What's Generated From Your `.proto`?

When you run the protocol buffer compiler 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` class for creating message class instances.
- For **Kotlin**, in addition to the Java generated code, the compiler generates a `.kt` file for each message type, containing a DSL which can be used to simplify creating message 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 **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.
- For **Dart**, the compiler generates a `.pb.dart` file with a class for each message type 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 (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

.proto Type	Notes	C++ Type	Java/Kotlin Type[1]	Python Type[3]	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[4]	int64	Bignum
uint32	Uses variable-length encoding.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)
uint64	Uses variable-length encoding.	uint64	long[2]	int/long[4]	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[4]	int64	Bignum
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long[2]	int/long[4]	uint64	Bignum
sfixed32	Always four bytes.	int32	int	int	int32	Fixnum or Bignum (as required)

sfixed64	Always eight bytes.	int64	long	int/long[4]	int64	Bignum
bool		bool	boolean	bool	bool	TrueClass/FalseCla
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 2^{32}.	string	String	str/unicode[5]	string	String (UTF-8)
bytes	May contain any arbitrary sequence of bytes no longer than 2^{32}.	string	ByteString	str (Python 2) bytes (Python 3)	[]byte	String (ASCII-8BIT)

You can find out more about how these types are encoded when you serialize your message in Protocol Buffer Encoding.

[1] Kotlin uses the corresponding types from Java, even for unsigned types, to ensure compatibility in mixed Java/Kotlin codebases.

[2] 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.

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

[4] 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].

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

[6] 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 enums, 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 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 behavior when set to `false` if you don't want that behavior 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 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`:

```
enum Corpus {
  CORPUS_UNSPECIFIED = 0;
  CORPUS_UNIVERSAL = 1;
  CORPUS_WEB = 2;
  CORPUS_IMAGES = 3;
  CORPUS_LOCAL = 4;
  CORPUS_NEWS = 5;
  CORPUS_PRODUCTS = 6;
  CORPUS_VIDEO = 7;
}

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
  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.
- The zero value needs to be the first element, for compatibility with the proto2 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 generates a warning message when aliases

```proto
enum EnumAllowingAlias {
  option allow_alias = true;
  EAA_UNSPECIFIED = 0;
  EAA_STARTED = 1;
  EAA_RUNNING = 1;
  EAA_FINISHED = 2;
}

enum EnumNotAllowingAlias {
  ENAA_UNSPECIFIED = 0;
  ENAA_STARTED = 1;
  // ENAA_RUNNING = 1;  // Uncommenting this line will cause a warning me
  ENAA_FINISHED = 2;
}
```

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 `enum`s within a message definition, as in the above example, or outside – these `enum`s 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, Kotlin, 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.

> **Important**
>
> The generated code may be subject to language-specific limitations on the number of enumerators (low thousands for one language). Review the limitations for the languages you plan to use.

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.

> **Important**
>
> For information on how enums should work contrasted with how they currently work in different languages, see Enum Behavior.

For more information about how to work with message `enum`s in your applications, see the generated code guide for your chosen language.

Reserved Values

If you update an enum type by entirely removing an enum entry, or commenting it out, future users can reuse the numeric value 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

numeric values (and/or names, which can also cause issues for JSON serialization) of your deleted entries are `reserved`. The protocol buffer compiler will complain if any future users try to use these identifiers. You can specify that your reserved numeric value range goes up to the maximum possible value using the `max` keyword.

```
enum Foo {
  reserved 2, 15, 9 to 11, 40 to max;
  reserved "FOO", "BAR";
}
```

Note that you can't mix field names and numeric values in the same `reserved` statement.

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 use definitions only 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, you can put a placeholder `.proto` file in the old location to forward all the imports to the new location using the `import public` notion.

Note that the public import functionality is not available in Java.

`import public` dependencies can be transitively relied upon by any code importing the proto containing the `import public` statement. For example:

```
// new.proto
```

```
// 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 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 when you use the binary wire format.

> ### Note
>
> If you use JSON or proto text format to store your protocol buffer messages, the changes that you can make in your proto definition are different.

Check Proto Best Practices and the following rules:

- Don't change the field numbers for any existing fields. "Changing" the field number is equivalent to deleting the field and adding a new field with the same type. If you want to renumber a field, see the instructions for deleting a field.
- 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 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 section for details.
- Fields can be removed, as long as the field 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 field number 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++ (for example, 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

- `fixed32` is compatible with `sfixed32`, and `fixed64` with `sfixed64`.
- For `string`, `bytes`, and message fields, `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. Note that this is **not** generally safe for numeric types, including bools and enums. Repeated fields of numeric types can be serialized in the packed format, which will not be parsed correctly when an `optional` field is expected.
- `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.
- Changing a single `optional` field or extension into a member of a **new** `oneof` is binary compatible, however for some languages (notably, Go) the generated code's API will change in incompatible ways. For this reason, Google does not make such changes in its public APIs, as documented in AIP-180. With the same caveat about source-compatibility, moving multiple fields into a new `oneof` may be safe if you are sure that no code sets more than one at a time. Moving fields into an existing `oneof` is not safe. Likewise, changing a single field `oneof` to an `optional` field or extension is safe.

Unknown Fields

Unknown fields are well-formed protocol buffer 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.

Originally, proto3 messages always discarded unknown fields during parsing, but in version 3.5 we reintroduced the preservation of unknown fields to match the proto2 behavior. In versions 3.5 and later, unknown fields are retained during parsing and included in the serialized output.

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 `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

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 google::protobuf::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, the `Any` can hold arbitrary proto3 messages, similar to proto2 messages which can allow 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.

Note that if *multiple values are set, the last set value as determined by the order in the proto will overwrite all previous ones*.

Field numbers for oneof fields must be unique within the enclosing message.

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, except `map` fields and `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.

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_EQ(message.name(), "name");
// Calling mutable_sub_message() will clear the name field and will
// sub_message to a new instance of SubMessage with none of its fie
message.mutable_sub_message();
CHECK(message.name().empty());
```

- 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 set a oneof field to the default value (such as setting an int32 oneof field to 0), the "case" of that oneof field will be set, and the value will be serialized on the wire.
- 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_EQ(msg2.name(), "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

and parsed. However, you can safely move a single field into a **new** oneof and may be able to move multiple fields if it is known that only one is ever set. See Updating A Message Type for further details.

- **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 type except for floating point types and `bytes`). Note that enum is not a valid `key_type`. 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 are 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.
- If you provide a key but no value for a map field, the behavior when the field is serialized is language-dependent. In C++, Java, Kotlin, and Python the default value for the type is serialized, while in other languages nothing is serialized.

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.

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;
```

Any protocol buffers implementation that supports maps must both produce

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** and **Kotlin**, 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 **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

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

The most straightforward RPC system to use with protocol buffers is gRPC: 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.

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.

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.

When parsing JSON-encoded data into a protocol buffer, if a value is missing or if its value is `null`, it will be interpreted as the corresponding default value.

When generating JSON-encoded output from a protocol buffer, if a protobuf field has the default value and if the field doesn't support field presence, it will be omitted from the output by default. An implementation may provide options to include fields with default values in the output.

A proto3 field that is defined with the `optional` keyword supports field presence. Fields that have a value set and that support field presence always include the field value in the JSON-encoded output, even if it is the default value.

proto3	JSON	JSON example	Notes
message	object	`{"fooBar": v, "g": null, ...}`	Generates JSON objects. Message field names are mapped to lowerCamelCase and become JSON object keys. If the `json_name` field option is specified, the specified value will be used as the key instead. Parsers accept both the lowerCamelCase name (or the one specified by the `json_name` option) and the original proto field name. `null` is an accepted value for all field types and treated as the default value of the corresponding field type. However, `null` cannot be used for the `json_name` value. For more on why, see Stricter validation for json_name.

enum	string	`"FOO_BAR"`	The name of the enum value as specified in proto is used. Parsers accept both enum names and integer values.
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. -0 is considered equivalent to 0.
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. Offsets other than "Z" are also accepted.
Duration	string	`"1.000340012s", "1s"`	Generated output always contains 0, 3, 6, or 9 fractional digits, depending on required precision, followed by the suffix "s". Accepted are any fractional digits (also none) as long as they fit into nano-seconds precision and the suffix "s" is required.

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 `field_mask.proto`.
ListValue	array	`[foo, bar, ...]`	
Value	value		Any JSON value. Check google.protobuf.Value for details.
NullValue	null		JSON null
Empty	object	`{}`	An empty JSON object

JSON Options

A proto3 JSON implementation may provide the following options:

- **Emit fields with default values**: Fields with default values are omitted by default in proto3 JSON output. An implementation may provide an option to override this behavior and output fields with their default values.
- **Ignore unknown fields**: Proto3 JSON parser should reject unknown fields by default but may provide an option to ignore unknown fields in parsing.
- **Use proto field name instead of lowerCamelCase name**: By default proto3 JSON printer should convert the field name to lowerCamelCase and use that as the JSON name. An implementation may provide an option to use proto field name as the JSON name instead. Proto3 JSON parsers are required to accept both the converted lowerCamelCase name and the proto field name.
- **Emit enum values as integers instead of strings**: The name of an enum value is used by default in JSON output. An option may be provided to use the numeric value of the enum value instead.

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, oneof fields, 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

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 or Kotlin code, this option has no effect.

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

- `java_outer_classname` (file option): The class name (and hence the file name) for the wrapper Java class 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 the `java_multiple_files` option is disabled, then all other classes/enums/etc. generated for the `.proto` file will be generated *within* this outer wrapper Java class as nested classes/enums/etc. If not generating Java code, this option has no effect.

```
option java_outer_classname = "Ponycopter";
```

- `java_multiple_files` (file option): If false, only a single `.java` file will be generated for this `.proto` file, and all the Java classes/enums/etc. generated for the top-level messages, services, and enumerations will be nested inside of an outer class (see `java_outer_classname`). If true, separate `.java` files will be generated for each of the Java classes/enums/etc. generated for the top-level messages, services, and enumerations, and the wrapper Java class generated for this `.proto` file won’t contain any nested classes/enums/etc. This is a Boolean option which defaults to `false`. If not generating Java code, this option has no effect.

```
option java_multiple_files = true;
```

- `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 of `.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

```
option optimize_for = CODE_SIZE;
```

- `cc_enable_arenas` (file option): Enables arena allocation 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. 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. For C++, clang-tidy will generate warnings whenever deprecated fields are used. 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 statement.

```
int32 old_field = 6 [deprecated = true];
```

Enum Value Options

Enum value options are supported. You can use the `deprecated` option to indicate that a value shouldn't be used anymore. You can also create custom options using extensions.

The following example shows the syntax for adding these options:

```
import "google/protobuf/descriptor.proto";

extend google.protobuf.EnumValueOptions {
  optional string string_name = 123456789;
}

enum Data {
  DATA_UNKNOWN = 0;
  DATA_SEARCH = 1 [deprecated = true];
  DATA_DISPLAY = 2 [
    (string_name) = "display_value"
  ]
}
```

Continue to the next section, Custom Options to see how to apply custom options to enum values and to fields.

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 for details. Note that creating custom options uses extensions, which are permitted only for custom options in proto3.

Option Retention

Options have a notion of *retention*, which controls whether an option is retained in the generated code. Options have *runtime retention* by default, meaning that they are retained in the generated code and are thus visible at runtime in the generated descriptor pool. However, you can set `retention = RETENTION_SOURCE` to specify that an option (or field within an option) must not be retained at runtime. This is called *source retention*.

Option retention is an advanced feature that most users should not need to worry about, but it can be useful if you would like to use certain options without paying the code size cost of retaining them in your binaries. Options with source retention are still visible to `protoc` and `protoc` plugins, so code generators can use them to customize their behavior.

Retention can be set directly on an option, like this:

```
extend google.protobuf.FileOptions {
  optional int32 source_retention_option = 1234
      [retention = RETENTION_SOURCE];
}
```

It can also be set on a plain field, in which case it takes effect only when that field appears inside an option:

```
message OptionsMessage {
  int32 source_retention_field = 1 [retention = RETENTION_SOURCE];
}
```

You can set `retention = RETENTION_RUNTIME` if you like, but this has no effect since it is the default behavior. When a message field is marked `RETENTION_SOURCE`, its entire contents are dropped; fields inside it cannot override that by trying to set `RETENTION_RUNTIME`.

Note

As of Protocol Buffers 22.0, support for option retention is still in progress and only C++ and Java are supported. Go has support starting from 1.29.0. Python support is complete but has not made it into a release yet.

Option Targets

Fields have a `targets` option which controls the types of entities that the field may apply to when used as an option. For example, if a field has `targets = TARGET_TYPE_MESSAGE` then that field cannot be set in a custom option on an enum (or any other non-message entity). Protoc enforces this and will raise an error if there is a violation of the target constraints.

At first glance, this feature may seem unnecessary given that every custom option is an extension of the options message for a specific entity, which already constrains the option to that one entity. However, option targets are useful in the case where you have a shared options message applied to multiple entity types and you want to control the usage of individual fields in that message. For example:

```
message MyOptions {
  string file_only_option = 1 [targets = TARGET_TYPE_FILE];
  int32 message_and_enum_option = 2 [targets = TARGET_TYPE_MESSAGE,
                                     targets = TARGET_TYPE_ENUM];
}

extend google.protobuf.FileOptions {
  optional MyOptions file_options = 50000;
}

extend google.protobuf.MessageOptions {
  optional MyOptions message_options = 50000;
}

extend google.protobuf.EnumOptions {
  optional MyOptions enum_options = 50000;
}

// OK: this field is allowed on file options
option (file_options).file_only_option = "abc";

message MyMessage {
  // OK: this field is allowed on both message and enum options
  option (message_options).message_and_enum_option = 42;
}

enum MyEnum {
  MY_ENUM_UNSPECIFIED = 0;
  // Error: file_only_option cannot be set on an enum.
  option (enum_options).file_only_option = "xyz";
}
```

Generating Your Classes

To generate the Java, Kotlin, Python, C++, Go, Ruby, 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 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 repository on GitHub.

The Protocol Compiler is invoked as follows:

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --py
```

- `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 for more.
  - `--java_out` generates Java code in `DST_DIR`. See the Java generated code reference for more.
  - `--kotlin_out` generates additional Kotlin code in `DST_DIR`. See the

    - `--python_out` generates Python code in `DST_DIR`. See the Python generated code reference for more.
    - `--go_out` generates Go code in `DST_DIR`. See the Go generated code reference for more.
    - `--ruby_out` generates Ruby code in `DST_DIR`. See the Ruby generated code reference for more.
    - `--objc_out` generates Objective-C code in `DST_DIR`. See the Objective-C generated code reference for more.
    - `--csharp_out` generates C# code in `DST_DIR`. See the C# generated code reference for more.
    - `--php_out` generates PHP code in `DST_DIR`. See the PHP generated code reference 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_PATH`s so that the compiler can determine its canonical name.

File location

Prefer not to put `.proto` files in the same directory as other language sources. Consider creating a subpackage `proto` for `.proto` files, under the root package for your project.

Location Should be Language-agnostic

When working with Java code, it's handy to put related `.proto` files in the same directory as the Java source. However, if any non-Java code ever uses the same protos, the path prefix will no longer make sense. So in general, put the protos in a related language-agnostic directory such as `//myteam/mypackage`.

The exception to this rule is when it's clear that the protos will be used only in a Java context, such as for testing.

Supported Platforms

For information about:

- the operating systems, compilers, build systems, and C++ versions that are supported, see Foundational C++ Support Policy.
- the PHP versions that are supported, see Supported PHP versions.