Language Guide

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

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

Defining A Message Type

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

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

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

Specifying Field Types

In the above example, all the fields are scalar types: two integers (`page_number` and `result_per_page`) and a string (`query`). However, you can also specify composite types for your fields, including enumerations 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, 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). 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 2^{29} - 1, or 536,870,911. You also cannot use the numbers 19000 though 19999 (`FieldDescriptor::kFirstReservedNumber` through `FieldDescriptor::kLastReservedNumber`), as they are reserved for the Protocol Buffers implementation - the protocol buffer compiler will complain if you use one of these reserved numbers in your `.proto`.

Specifying Field Rules

You specify that message fields are one of the following:

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

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

```
repeated int32 samples = 4 [packed=true];
```

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

Adding More Message Types

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

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}

message SearchResponse {
 ...
}
```

Adding Comments

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

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

What's Generated From Your `.proto`?

When you run the protocol buffer compiler on a `.proto`, the compiler generates the code in your chosen language you'll need to work with the message types you've described in the file, including getting and setting field values, serializing your messages to an output stream, and parsing your messages from an input stream.

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

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

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

You can find out more about using the APIs for each language by following the tutorial for your chosen language. For even more API details, see the relevant API reference.

Scalar Value Types

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

.proto Type	Notes	C++ Type	Java Type	Python Type[2]
double		double	double	float
float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long[3]
uint32	Uses variable-length encoding.	uint32	int[1]	int/long[3]
uint64	Uses variable-length encoding.	uint64	long[1]	int/long[3]
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int

sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long[3]
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int[1]	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long[1]	int/long[3]
sfixed32	Always four bytes.	int32	int	int
sfixed64	Always eight bytes.	int64	long	int/long[3]
bool		bool	boolean	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String	str/unicode[4]
bytes	May contain any arbitrary sequence of bytes.	string	ByteString	str

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

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

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

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

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

Optional Fields And Default Values

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

```
optional int32 result_per_page = 3 [default = 10];
```

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

Enumerations

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

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3 [default = 10];
  enum Corpus {
    UNIVERSAL = 0;
    WEB = 1;
    IMAGES = 2;
    LOCAL = 3;
    NEWS = 4;
    PRODUCTS = 5;
    VIDEO = 6;
  }
  optional Corpus corpus = 4 [default = UNIVERSAL];
}
```

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

```
enum EnumAllowingAlias {
  option allow_alias = true;
  UNKNOWN = 0;
  STARTED = 1;
  RUNNING = 1;
}
enum EnumNotAllowingAlias {
  UNKNOWN = 0;
  STARTED = 1;
  // RUNNING = 1;  // Uncommenting this line will cause a compile error inside Google and a warning messag
e outside.
}
```

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

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

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

Using Other Message Types

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

```
message SearchResponse {
  repeated Result result = 1;
}

message Result {
  required string url = 1;
  optional string title = 2;
  repeated string snippets = 3;
}
```

Importing Definitions

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

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

```
import "myproject/other_protos.proto";
```

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

```
// new.proto
// All definitions are moved here
```

```
// old.proto
// This is the proto that all clients are importing.
import public "new.proto";
import "other.proto";
```

```
// client.proto
import "old.proto";
// You use definitions from old.proto and new.proto, but not other.proto
```

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

Nested Types

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

```
message SearchResponse {
  message Result {
    required string url = 1;
    optional string title = 2;
    repeated string snippets = 3;
  }
  repeated Result result = 1;
}
```

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

```
message SomeOtherMessage {
  optional SearchResponse.Result result = 1;
}
```

You can nest messages as deeply as you like:

```
message Outer {                  // Level 0
  message MiddleAA {  // Level 1
    message Inner {   // Level 2
      required int64 ival = 1;
      optional bool  booly = 2;
    }
  }
  message MiddleBB {  // Level 1
    message Inner {   // Level 2
      required int32 ival = 1;
      optional bool  booly = 2;
    }
  }
}
```

Groups

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

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

```
message SearchResponse {
  repeated group Result = 1 {
    required string url = 2;
    optional string title = 3;
    repeated string snippets = 4;
  }
}
```

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

Updating A Message Type

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

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

Extensions

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

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

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

```
extend Foo {
  optional int32 bar = 126;
}
```

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

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

```
Foo foo;
foo.SetExtension(bar, 15);
```

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

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

Nested Extensions

You can declare extensions in the scope of another type:

```
message Baz {
  extend Foo {
    optional int32 bar = 126;
  }
  ...
}
```

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

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

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

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

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

```
message Baz {
  extend Foo {
    optional Baz foo_ext = 127;
  }
  ...
}
```

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

```
message Baz {
  ...
}

// This can even be in a different file.
extend Foo {
  optional Baz foo_baz_ext = 127;
}
```

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

Choosing Extension Numbers

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

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

```
message Foo {
  extensions 1000 to max;
}
```

`max` is 2^{29} - 1, or 536,870,911.

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

Packages

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

```
package foo.bar;
message Open { ... }
```

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

```
message Foo {
  ...
  required foo.bar.Open open = 1;
  ...
}
```

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

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

Packages and Name Resolution

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

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

Defining Services

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

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

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

```
using google::protobuf;

protobuf::RpcChannel* channel;
protobuf::RpcController* controller;
SearchService* service;
SearchRequest request;
```

```
SearchResponse response;

void DoSearch() {
  // You provide classes MyRpcChannel and MyRpcController, which implement
  // the abstract interfaces protobuf::RpcChannel and protobuf::RpcController.
  channel = new MyRpcChannel("somehost.example.com:1234");
  controller = new MyRpcController;

  // The protocol compiler generates the SearchService class based on the
  // definition given above.
  service = new SearchService::Stub(channel);

  // Set up the request.
  request.set_query("protocol buffers");

  // Execute the RPC.
  service->Search(controller, request, response, protobuf::NewCallback(&Done));
}

void Done() {
  delete service;
  delete channel;
  delete controller;
}
```

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

```
using google::protobuf;

class ExampleSearchService : public SearchService {
 public:
  void Search(protobuf::RpcController* controller,
              const SearchRequest* request,
              SearchResponse* response,
              protobuf::Closure* done) {
    if (request->query() == "google") {
      response->add_result()->set_url("http://www.google.com");
    } else if (request->query() == "protocol buffers") {
      response->add_result()->set_url("http://protobuf.googlecode.com");
    }
    done->Run();
  }
};

int main() {
  // You provide class MyRpcServer.  It does not have to implement any
  // particular interface; this is just an example.
  MyRpcServer server;

  protobuf::Service* service = new ExampleSearchService;
  server.ExportOnPort(1234, service);
  server.Run();
  delete service;
  return 0;
}
```

There are a number of ongoing third-party projects to develop RPC implementations for Protocol Buffers. For a list of links to projects we know about, see the third-party add-ons wiki page.

Options

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

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

Here are a few of the most commonly used options:

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

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

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

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

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

implementations of all methods as it does in `SPEED` mode. Generated classes will only implement the `MessageLite` interface in each language, which provides only a subset of the methods of the full `Message` interface.

```
option optimize_for = CODE_SIZE;
```

- `cc_generic_services`, `java_generic_services`, `py_generic_services` (file options): Whether or not the protocol buffer compiler should generate abstract service code based on services definitions in C++, Java, and Python, respectively. For legacy reasons, these default to `true`. However, as of version 2.3.0 (January 2010), it is considered preferrable for RPC implementations to provide code generator plugins to generate code more specific to each system, rather than rely on the "abstract" services.

```
// This file relies on plugins to generate service code.
option cc_generic_services = false;
option java_generic_services = false;
option py_generic_services = false;
```

- `message_set_wire_format` (message option): If set to `true`, the message uses a different binary format intended to be compatible with an old format used inside Google called `MessageSet`. Users outside Google will probably never need to use this option. The message must be declared exactly as follows:

```
message Foo {
  option message_set_wire_format = true;
  extensions 4 to max;
}
```

- `packed` (field option): If set to `true` on a repeated field of a basic integer type, a more compact encoding will be used. There is no downside to using this option. However, note that prior to version 2.3.0, parsers that received packed data when not expected would ignore it. Therefore, it was not possible to change an existing field to packed format without breaking wire compatibility. In 2.3.0 and later, this change is safe, as parsers for packable fields will always accept both formats, but be careful if you have to deal with old programs using old protobuf versions.

```
repeated int32 samples = 4 [packed=true];
```

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

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

Custom Options

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

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

extend google.protobuf.MessageOptions {
  optional string my_option = 51234;
}

message MyMessage {
  option (my_option) = "Hello world!";
}
```

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

```
string value = MyMessage::descriptor()->options().GetExtension(my_option);
```

Here, `MyMessage::descriptor()->options()` returns the `MessageOptions` protocol message for `MyMessage`. Reading custom options from it is just like reading any other extension.

Similarly, in Java we would write:

```
String value = MyProtoFile.MyMessage.getDescriptor().getOptions()
  .getExtension(MyProtoFile.myOption);
```

In Python it would be:

```
value = my_proto_file_pb2.MyMessage.DESCRIPTOR.GetOptions()
  .Extensions[my_proto_file_pb2.my_option]
```

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

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

extend google.protobuf.FileOptions {
  optional string my_file_option = 50000;
}
extend google.protobuf.MessageOptions {
  optional int32 my_message_option = 50001;
}
extend google.protobuf.FieldOptions {
  optional float my_field_option = 50002;
}
extend google.protobuf.EnumOptions {
  optional bool my_enum_option = 50003;
}
extend google.protobuf.EnumValueOptions {
  optional uint32 my_enum_value_option = 50004;
}
extend google.protobuf.ServiceOptions {
  optional MyEnum my_service_option = 50005;
}
extend google.protobuf.MethodOptions {
  optional MyMessage my_method_option = 50006;
}
```

```
option (my_file_option) = "Hello world!";

message MyMessage {
  option (my_message_option) = 1234;

  optional int32 foo = 1 [(my_field_option) = 4.5];
  optional string bar = 2;
}

enum MyEnum {
  option (my_enum_option) = true;

  FOO = 1 [(my_enum_value_option) = 321];
  BAR = 2;
}

message RequestType {}
message ResponseType {}

service MyService {
  option (my_service_option) = FOO;

  rpc MyMethod(RequestType) returns(ResponseType) {
    // Note:  my_method_option has type MyMessage.  We can set each field
    //   within it using a separate "option" line.
    option (my_method_option).foo = 567;
    option (my_method_option).bar = "Some string";
  }
}
```

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

```
// foo.proto
import "google/protobuf/descriptor.proto";
package foo;
extend google.protobuf.MessageOptions {
  optional string my_option = 51234;
}
```

```
// bar.proto
import "foo.proto";
package bar;
message MyMessage {
  option (foo.my_option) = "Hello world!";
}
```

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

```
message FooOptions {
  optional int32 opt1 = 1;
  optional string opt2 = 2;
}

extend google.protobuf.FieldOptions {
  optional FooOptions foo_options = 1234;
}

// usage:
message Bar {
  optional int32 a = 1 [(foo_options).opt1 = 123, (foo_options).opt2 = "baz"];
  // alternative aggregate syntax (uses TextFormat):
  optional int32 b = 2 [(foo_options) = { opt1: 123 opt2: "baz" }];
}
```

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

Generating Your Classes

To generate the Java, Python, or C++ code you need to work with the message types defined in a `.proto` file, you need to run the protocol buffer compiler `protoc` on the `.proto`. If you haven't installed the compiler, download the package and follow the instructions in the README.

The Protocol Compiler is invoked as follows:

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --python_out=DST_DIR path/to/file.pro
to
```

- `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.
  - `--python_out` generates Python code in `DST_DIR`. See the Python 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.