Protocol Buffer Basics: Java

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

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

This isn't a comprehensive guide to using protocol buffers in Java. For more detailed reference information, see the Protocol Buffer Language Guide, the Java API Reference, the Java Generated Code Guide, and the Encoding Reference.

Why Use Protocol Buffers?

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

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

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

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

Where to Find the Example Code

The example code is included in the source code package, under the "examples" directory. Download it here.

Defining Your Protocol Format

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

```
package tutorial;

option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

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

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

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

Next, you have your message definitions. A message is just an aggregate containing a set of typed fields. Many standard simple data types are available as field types, including `bool`, `int32`, `float`, `double`, and `string`. You can also add further structure to your messages by using other message types as field types – in the above example the `Person` message contains `PhoneNumber` messages, while the `AddressBook` message contains `Person` messages. You can even define message

types nested inside other messages – as you can see, the `PhoneNumber` type is defined inside `Person`. You can also define `enum` types if you want one of your fields to have one of a predefined list of values – here you want to specify that a phone number can be one of `MOBILE`, `HOME`, or `WORK`.

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

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

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

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

You'll find a complete guide to writing `.proto` files – including all the possible field types – in the Protocol Buffer Language Guide. Don't go looking for facilities similar to class inheritance, though – protocol buffers don't do that.

Compiling Your Protocol Buffers

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

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

   ```
   protoc -I=$SRC_DIR --java_out=$DST_DIR $SRC_DIR/addressbook.proto
   ```

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

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

The Protocol Buffer API

Let's look at some of the generated code and see what classes and methods the compiler has created for you. If you look in `AddressBookProtos.java`, you can see that it defines a class called `AddressBookProtos`, nested within which is a class for each message you specified in `addressbook.proto`. Each class has its own `Builder` class that you use to create instances of that class. You can find out more about builders in the Builders vs. Messages section below.

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

```
// required string name = 1;
public boolean hasName();
public String getName();

// required int32 id = 2;
public boolean hasId();
public int getId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();

// repeated .tutorial.Person.PhoneNumber phone = 4;
public List<PhoneNumber> getPhoneList();
public int getPhoneCount();
public PhoneNumber getPhone(int index);
```

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

```
// required string name = 1;
public boolean hasName();
public java.lang.String getName();
public Builder setName(String value);
public Builder clearName();

// required int32 id = 2;
public boolean hasId();
public int getId();
public Builder setId(int value);
public Builder clearId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();
public Builder setEmail(String value);
public Builder clearEmail();

// repeated .tutorial.Person.PhoneNumber phone = 4;
public List<PhoneNumber> getPhoneList();
public int getPhoneCount();
public PhoneNumber getPhone(int index);
```

```
public Builder setPhone(int index, PhoneNumber value);
public Builder addPhone(PhoneNumber value);
public Builder addAllPhone(Iterable<PhoneNumber> value);
public Builder clearPhone();
```

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

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

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

For more information on exactly what members the protocol compiler generates for any particular field definition, see the Java generated code reference.

Enums and Nested Classes

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

```
public static enum PhoneType {
  MOBILE(0, 0),
  HOME(1, 1),
  WORK(2, 2),
  ;
  ...
}
```

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

Builders vs. Messages

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

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

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

```
Person john =
  Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
```

```
    .addPhone(
      Person.PhoneNumber.newBuilder()
        .setNumber("555-4321")
        .setType(Person.PhoneType.HOME))
    .build();
```

Standard Message Methods

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

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

These methods implement the `Message` and `Message.Builder` interfaces shared by all Java messages and builders. For more information, see the complete API documentation for `Message`.

Parsing and Serialization

Finally, each protocol buffer class has methods for writing and reading messages of your chosen type using the protocol buffer binary format. These include:

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

These are just a couple of the options provided for parsing and serialization. Again, see the `Message` API reference for a complete list.

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

Writing A Message

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

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

compiler are highlighted.

```
import com.example.tutorial.AddressBookProtos.AddressBook;
import com.example.tutorial.AddressBookProtos.Person;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintStream;

class AddPerson {
  // This function fills in a Person message based on user input.
  static Person PromptForAddress(BufferedReader stdin,
                                 PrintStream stdout) throws IOException {
    Person.Builder person = Person.newBuilder();

    stdout.print("Enter person ID: ");
    person.setId(Integer.valueOf(stdin.readLine()));

    stdout.print("Enter name: ");
    person.setName(stdin.readLine());

    stdout.print("Enter email address (blank for none): ");
    String email = stdin.readLine();
    if (email.length() > 0) {
      person.setEmail(email);
    }

    while (true) {
      stdout.print("Enter a phone number (or leave blank to finish): ");
      String number = stdin.readLine();
      if (number.length() == 0) {
        break;
      }

      Person.PhoneNumber.Builder phoneNumber =
        Person.PhoneNumber.newBuilder().setNumber(number);

      stdout.print("Is this a mobile, home, or work phone? ");
      String type = stdin.readLine();
      if (type.equals("mobile")) {
        phoneNumber.setType(Person.PhoneType.MOBILE);
      } else if (type.equals("home")) {
        phoneNumber.setType(Person.PhoneType.HOME);
      } else if (type.equals("work")) {
        phoneNumber.setType(Person.PhoneType.WORK);
      } else {
        stdout.println("Unknown phone type.  Using default.");
      }

      person.addPhone(phoneNumber);
    }

    return person.build();
  }
```

```
  // Main function:  Reads the entire address book from a file,
  //   adds one person based on user input, then writes it back out to the same
  //   file.
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage:  AddPerson ADDRESS_BOOK_FILE");
      System.exit(-1);
    }

    AddressBook.Builder addressBook = AddressBook.newBuilder();

    // Read the existing address book.
    try {
      addressBook.mergeFrom(new FileInputStream(args[0]));
    } catch (FileNotFoundException e) {
      System.out.println(args[0] + ": File not found.  Creating a new file.");
    }

    // Add an address.
    addressBook.addPerson(
      PromptForAddress(new BufferedReader(new InputStreamReader(System.in)),
                       System.out));

    // Write the new address book back to disk.
    FileOutputStream output = new FileOutputStream(args[0]);
    addressBook.build().writeTo(output);
    output.close();
  }
}
```

Reading A Message

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

```
import com.example.tutorial.AddressBookProtos.AddressBook;
import com.example.tutorial.AddressBookProtos.Person;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;

class ListPeople {
  // Iterates though all people in the AddressBook and prints info about them.
  static void Print(AddressBook addressBook) {
    for (Person person: addressBook.getPersonList()) {
      System.out.println("Person ID: " + person.getId());
      System.out.println("  Name: " + person.getName());
      if (person.hasEmail()) {
        System.out.println("  E-mail address: " + person.getEmail());
      }

      for (Person.PhoneNumber phoneNumber : person.getPhoneList()) {
        switch (phoneNumber.getType()) {
          case MOBILE:
            System.out.print("  Mobile phone #: ");
            break;
```

```
          case HOME:
            System.out.print("  Home phone #: ");
            break;
          case WORK:
            System.out.print("  Work phone #: ");
            break;
        }
        System.out.println(phoneNumber.getNumber());
      }
    }
  }

  // Main function:  Reads the entire address book from a file and prints all
  //   the information inside.
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage:  ListPeople ADDRESS_BOOK_FILE");
      System.exit(-1);
    }

    // Read the existing address book.
    AddressBook addressBook =
      AddressBook.parseFrom(new FileInputStream(args[0]));

    Print(addressBook);
  }
}
```

Extending a Protocol Buffer

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

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

(There are some exceptions to these rules, but they are rarely used.)

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

Advanced Usage

Protocol buffers have uses that go beyond simple accessors and serialization. Be sure to explore the Java API reference to see what else you can do with them.

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

Reflection is provided as part of the `Message` and `Message.Builder` interfaces.

Last updated 四月 *2, 2012.*