RProtoBuf: An R API for Protocol Buffers

Dirk Eddelbuettel\(^a\), Romain François\(^b\), and Murray Stokely\(^c\)

\(^a\)http://dirk.eddelbuettel.com; \(^b\)https://roiman.rbind.io/; \(^c\)https://stokely.org

This version was compiled on February 7, 2020

**Protocol Buffers** is a software project by Google that is used extensively internally and also released under an Open Source license. It provides a way of encoding structured data in an efficient yet extensible format. Google formally supports APIs for C++, Java and Python. This vignette describes version the **RProtoBuf** package which brings support for Protocol Buffer messages to R.

## Contents

1 Protocol Buffers  
2 Basic use: Protocol Buffers and R  
2.1 Importing proto files dynamically  
2.2 Creating a message  
2.3 Access and modify fields of a message  
2.4 Display messages  
2.5 Serializing messages  
2.6 Parsing messages  
2.7 Classes, Methods and Pseudo Methods  
2.8 Messages  
2.8.1 Retrieve fields  
2.8.2 Modify fields  
2.8.3 Message\$has method  
2.8.4 Message\$clone method  
2.8.5 Message\$isInitialized method  
2.8.6 Message\$serialize method  
2.8.7 Message\$clear method  
2.8.8 Message\$size method  
2.8.9 Message\$bytesize method  
2.8.10 Message\$swap method  
2.8.11 Message\$set method  
2.8.12 Message\$fetch method  
2.8.13 Message\$setExtension method  
2.8.14 Message\$getExtension method  
2.8.15 Message\$add method  
2.8.16 Message\$str method  
2.8.17 Message\$as.character method  
2.8.18 Message\$toString method  
2.8.19 Message\$as.list method  
2.8.20 Message\$update method  
2.8.21 Message\$descriptor method  
2.8.22 Message\$fileDescriptor method  
2.9 Message descriptors  
2.9.1 Extracting descriptors  
2.9.2 The new method  
2.9.3 The read method  
2.9.4 The readASCII method  
2.9.5 The toString method  
2.9.6 The as.character method  
2.9.7 The as.list method  
2.9.8 The asMessage method
2.10 Field descriptors

2.10.1 The as.character method
2.10.2 The toString method
2.10.3 The asMessage method
2.10.4 The name method
2.10.5 The fileDescriptor method
2.10.6 The containing_type method
2.10.7 The is_extension method
2.10.8 The number method
2.10.9 The type method
2.10.10 The cpp_type method
2.10.11 The label method
2.10.12 The is_repeated method
2.10.13 The is_required method
2.10.14 The is_optional method
2.10.15 The has_default_value method
2.10.16 The default_value method
2.10.17 The message_type method
2.10.18 The enum_type method

2.11 Enum descriptors

2.11.1 Extracting descriptors
2.11.2 The as.list method
2.11.3 The as.character method
2.11.4 The toString method
2.11.5 The asMessage method
2.11.6 The name method
2.11.7 The fileDescriptor method
2.11.8 The containing_type method
2.11.9 The length method
2.11.10 The has method
2.11.11 The value_count method
2.11.12 The value method

2.12 Enum value descriptors

2.12.1 The number method
2.12.2 The name method
2.12.3 The enum_type method
2.12.4 The as.character method
2.12.5 The toString method
2.12.6 The asMessage method

2.13 File descriptors

2.13.1 The as.character method
2.13.2 The toString method
2.13.3 The asMessage method
2.13.4 The as.list method
2.13.5 The name method
2.13.6 The package method

2.14 Service descriptors
Protocol Buffers are a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more.

Protocol Buffers offer key features such as an efficient data interchange format that is both language- and operating system-agnostic yet uses a lightweight and highly performant encoding, object serialization and de-serialization as well as data and configuration management. Protocol Buffers are also forward compatible: updates to the proto files do not break programs built against the previous specification.

While benchmarks are not available, Google states on the project page that in comparison to XML, Protocol Buffers are at the same time simpler, between three to ten times smaller, between twenty and one hundred times faster, as well as less ambiguous and easier to program.

The Protocol Buffers code is released under an open-source (BSD) license. The Protocol Buffer project (http://code.google.com/p/protobuf/) contains a C++ library and a set of runtime libraries and compilers for C++, Java and Python.

With these languages, the workflow follows standard practice of so-called Interface Description Languages (IDL) (c.f. Wikipedia on IDL). This consists of compiling a Protocol Buffer description file (ending in .proto) into language specific classes that can be used to create, read, write and manipulate Protocol Buffer messages. In other words, given the ‘proto’ description file, code is automatically generated for the chosen target language(s). The project page contains a tutorial for each of these officially supported languages: http://code.google.com/apis/protocolbuffers/docs/tutorials.html

Besides the officially supported C++, Java and Python implementations, several projects have been created to support Protocol Buffers for many languages. The list of known languages to support protocol buffers is compiled as part of the project page: http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns

The Protocol Buffer project page contains a comprehensive description of the language: http://code.google.com/apis/protocolbuffers/docs/proto.html

2. Basic use: Protocol Buffers and R

This section describes how to use the R API to create and manipulate protocol buffer messages in R, and how to read and write the binary payload of the messages to files and arbitrary binary R connections.

2.1. Importing proto files dynamically. In contrast to the other languages (Java, C++, Python) that are officially supported by Google, the implementation used by the RProtoBuf package does not rely on the protoc compiler (with the exception of the two functions discussed in the previous section). This means that no initial step of statically compiling the proto file into C++ code that is then accessed by R code is necessary. Instead, proto files are parsed and processed at runtime by the protobuf C++ library—which is much more appropriate for a dynamic language.

The readProtoFiles function allows importing proto files in several ways.
Using the `file` argument, one can specify one or several file paths that ought to be proto files.

```r
pdir <- system.file("proto", package = "RProtoBuf")
pfile <- file.path(pdir, "addressbook.proto")
readProtoFiles(pfile)
```

With the `dir` argument, which is ignored if the `file` is supplied, all files matching the `.proto` extension will be imported.

```r
dir(pdir, pattern = "\.proto$", full.names = TRUE)
readProtoFiles(dir = pdir)
```

Finally, with the `package` argument (ignored if `file` or `dir` is supplied), the function will import all `.proto` files that are located in the `proto` sub-directory of the given package. A typical use for this argument is in the `.onLoad` function of a package.

```r
readProtoFiles( package = "RProtoBuf" )
```

Once the proto files are imported, all message descriptors are available in the R search path in the `RProtoBuf:DescriptorPool` special environment. The underlying mechanism used here is described in more detail in section 4.2.

```r
ls("RProtoBuf:DescriptorPool")
```

### 2.2. Creating a message

The objects contained in the special environment are descriptors for their associated message types. Descriptors will be discussed in detail in another part of this document, but for the purpose of this section, descriptors are just used with the `new` function to create messages.

```r
p <- new(tutorial.Person, name = "Romain", id = 1)
```

### 2.3. Access and modify fields of a message

Once the message is created, its fields can be queried and modified using the dollar operator of R, making protocol buffer messages seem like lists.

```r
p$name
```

```r
# [1] "Romain"
```

```r
p$id
```

```r
# [1] 1
```
However, as opposed to R lists, no partial matching is performed and the name must be given entirely. The [[ operator can also be used to query and set fields of a message, supplying either their name or their tag number:

```r
p["name"] <- "Romain Francois"
p["email"]
```

# [1] "francoisromain@free.fr"

Protocol buffers include a 64-bit integer type, but R lacks native 64-bit integer support. A workaround is available and described in Section 4.3 for working with large integer values.

### 2.4 Display messages

Protocol buffer messages and descriptors implement `show` methods that provide basic information about the message:

```r
p
```

# message of type 'tutorial.Person' with 3 fields set

For additional information, such as for debugging purposes, the `as.character` method provides a more complete ASCII representation of the contents of a message.

```r
cat(as.character(p))
```

# name: "Romain Francois"
# id: 3
# email: "francoisromain@free.fr"

### 2.5 Serializing messages

However, the main focus of protocol buffer messages is efficiency. Therefore, messages are transported as a sequence of bytes. The `serialize` method is implemented for protocol buffer messages to serialize a message into the sequence of bytes (raw vector in R speech) that represents the message.

```r
serialize( p, NULL )
```

# [1] 0a 0f 52 6f 6d 61 69 6e 20 46 72 61 6e 63 6f 69 73 10 03 1a 16 66 72 61 6e 63 6f 69 73 72 6f 6d 61 69 6e 40 66 72 65 65 2e 66 72

The same method can also be used to serialize messages to files:

```r
tf1 <- tempfile()
tf1
```

# [1] "/tmp/RtmpRBfpcs/file31743d74b662"

```r
serialize( p, tf1 )
readBin(tf1, raw(0), 500)
```
2.6. Parsing messages. The RProtoBuf package defines the `read` function to read messages from files, raw vector (the message payload) and arbitrary binary connections.

```r
args(read)
```

```r
# function (descriptor, input)  
# NULL
```

The binary representation of the message (often called the payload) does not contain information that can be used to dynamically infer the message type, so we have to provide this information to the `read` function in the form of a descriptor:

```r
message <- read(tutorial.Person, tf1)  
cat(as.character(message))
```

```r
# name: "Romain Francois"  
# id: 3  
# email: "francoisromain@free.fr"
```

The input argument of `read` can also be a binary readable R connection, such as a binary file connection:

```r
con <- file(tf2, open = "rb")  
message <- read(tutorial.Person, con)  
close(con)  
cat(as.character(message))
```

```r
# name: "Romain Francois"  
# id: 3  
# email: "francoisromain@free.fr"
```

Finally, the payload of the message can be used:
# reading the raw vector payload of the message
payload <- readBin(tf1, raw(0), 5000)
message <- read(tutorial.Person, payload)

read can also be used as a pseudo method of the descriptor object:

# reading from a file
message <- tutorial.Person$read(tf1)
# reading from a binary connection
con <- file(tf2, open = "rb")
message <- tutorial.Person$read(con)
close(con)
# read from the payload
message <- tutorial.Person$read(payload)

2.7. Classes, Methods and Pseudo Methods. The RProtoBuf package uses the S4 system to store information about descriptors and messages, but the information stored in the R object is very minimal and mainly consists of an external pointer to a C++ variable that is managed by the proto C++ library.

str(p)

Using the S4 system allows the RProtoBuf package to dispatch methods that are not generic in the S3 sense, such as new and serialize.

The RProtoBuf package combines the R typical dispatch of the form method( object, arguments) and the more traditional object oriented notation object$method(arguments).

2.8. Messages. Messages are represented in R using the Message S4 class. The class contains the slots pointer and type as described on the Table~1.

<table>
<thead>
<tr>
<th>slot</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>external pointer to the Message object of the C++ proto library. Documentation for the Message class is available from the protocol buffer project page: <a href="http://code.google.com/apis/protocolbuffers/docs/reference/cpp/google.protobuf.message.html#Message">http://code.google.com/apis/protocolbuffers/docs/reference/cpp/google.protobuf.message.html#Message</a></td>
</tr>
<tr>
<td>type</td>
<td>fully qualified path of the message. For example a Person message has its type slot set to tutorial.Person</td>
</tr>
</tbody>
</table>

Table 1. Description of slots for the Message S4 class

Although the RProtoBuf package uses the S4 system, the @ operator is very rarely used. Fields of the message are retrieved or modified using the $ or [[ operators as seen on the previous section, and pseudo-methods can also be called using the $ operator. Table~2 describes the methods defined for the Message class:

2.8.1. Retrieve fields. The $ and [[ operators allow extraction of a field data.

message <- new(tutorial.Person, 
    name = "foo", email = "foo@bar.com", id = 2,
    phone = list(new(tutorial.Person.PhoneNumber, 
        number = "+33(0)...", type = "HOME"),
        new(tutorial.Person.PhoneNumber, 
            number = "+33(0)###", type = "MOBILE")
    )
)
message$name
Table 2. Description of methods for the Message S4 class

<table>
<thead>
<tr>
<th>method</th>
<th>section</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>has</td>
<td>2.8.3</td>
<td>Indicates if a message has a given field.</td>
</tr>
<tr>
<td>clone</td>
<td>2.8.4</td>
<td>Creates a clone of the message.</td>
</tr>
<tr>
<td>isInitialized</td>
<td>2.8.5</td>
<td>Indicates if a message has all its required fields set</td>
</tr>
<tr>
<td>serialize</td>
<td>2.8.6</td>
<td>serialize a message to a file or a binary connection or retrieve the message</td>
</tr>
<tr>
<td></td>
<td></td>
<td>payload as a raw vector</td>
</tr>
<tr>
<td>clear</td>
<td>2.8.7</td>
<td>Clear one or several fields of a message, or the entire message</td>
</tr>
<tr>
<td>size</td>
<td>2.8.8</td>
<td>The number of elements in a message field</td>
</tr>
<tr>
<td>bytesize</td>
<td>2.8.9</td>
<td>The number of bytes the message would take once serialized</td>
</tr>
<tr>
<td>swap</td>
<td>2.8.10</td>
<td>swap elements of a repeated field of a message</td>
</tr>
<tr>
<td>set</td>
<td>2.8.11</td>
<td>set elements of a repeated field</td>
</tr>
<tr>
<td>fetch</td>
<td>2.8.12</td>
<td>fetch elements of a repeated field</td>
</tr>
<tr>
<td>setExtension</td>
<td>2.8.13</td>
<td>set an extension of a message</td>
</tr>
<tr>
<td>getExtension</td>
<td>2.8.14</td>
<td>get the value of an extension of a message</td>
</tr>
<tr>
<td>add</td>
<td>2.8.15</td>
<td>add elements to a repeated field</td>
</tr>
<tr>
<td>str</td>
<td>2.8.16</td>
<td>the R structure of the message</td>
</tr>
<tr>
<td>as.character</td>
<td>2.8.17</td>
<td>character representation of a message</td>
</tr>
<tr>
<td>toString</td>
<td>2.8.18</td>
<td>character representation of a message (same as as.character)</td>
</tr>
<tr>
<td>as.list</td>
<td>2.8.19</td>
<td>converts message to a named R list</td>
</tr>
<tr>
<td>descriptor</td>
<td>2.8.20</td>
<td>get the descriptor of the message type of this message</td>
</tr>
<tr>
<td>fileDescriptor</td>
<td>2.8.21</td>
<td>get the file descriptor of this message's descriptor</td>
</tr>
</tbody>
</table>

# [1] "foo"

message$email

# [1] "foo@bar.com"

message[["phone"]]

# [[1]]
# message of type 'tutorial.Person.PhoneNumber' with 2 fields set
# # [[2]]
# message of type 'tutorial.Person.PhoneNumber' with 2 fields set

# using the tag number
message[[2]] # id

# [1] 2

Neither $ nor [[ support partial matching of names. The $ is also used to call methods on the message, and the [[ operator can use the tag number of the field. Table~3 details correspondence between the field type and the type of data that is retrieved by $ and [[.

2.8.2. Modify fields. The $<- and [[<- operators are implemented for Message objects to set the value of a field. The R data is coerced to match the type of the message field.

message <- new(tutorial.Person, name = "foo", id = 2)
messagemessage@email <- "foo@bar.com"
messagemessage[["id"]]<-42
<table>
<thead>
<tr>
<th>field type</th>
<th>R type (non repeated)</th>
<th>R type (repeated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>double vector</td>
<td>double vector</td>
</tr>
<tr>
<td>float</td>
<td>double vector</td>
<td>double vector</td>
</tr>
<tr>
<td>uint32</td>
<td>double vector</td>
<td>double vector</td>
</tr>
<tr>
<td>fixed32</td>
<td>double vector</td>
<td>double vector</td>
</tr>
<tr>
<td>int32</td>
<td>integer vector</td>
<td>integer vector</td>
</tr>
<tr>
<td>sint32</td>
<td>integer vector</td>
<td>integer vector</td>
</tr>
<tr>
<td>sfixed32</td>
<td>integer vector</td>
<td>integer vector</td>
</tr>
<tr>
<td>int64</td>
<td>integer or character vector</td>
<td>integer or character vector</td>
</tr>
<tr>
<td>uint64</td>
<td>integer or character vector</td>
<td>integer or character vector</td>
</tr>
<tr>
<td>sint64</td>
<td>integer or character vector</td>
<td>integer or character vector</td>
</tr>
<tr>
<td>fixed64</td>
<td>integer or character vector</td>
<td>integer or character vector</td>
</tr>
<tr>
<td>sfixed64</td>
<td>integer or character vector</td>
<td>integer or character vector</td>
</tr>
<tr>
<td>bool</td>
<td>logical vector</td>
<td>logical vector</td>
</tr>
<tr>
<td>string</td>
<td>character vector</td>
<td>character vector</td>
</tr>
<tr>
<td>bytes</td>
<td>character vector</td>
<td>character vector</td>
</tr>
<tr>
<td>enum</td>
<td>integer vector</td>
<td>integer vector</td>
</tr>
<tr>
<td>message</td>
<td>S4 object of class Message</td>
<td>list of S4 objects of class Message</td>
</tr>
</tbody>
</table>

Table 3. Correspondence between field type and R type retrieved by the extractors. 1. R lacks native 64-bit integers, so the `RProtoBuf.int64AsString` option is available to return large integers as characters to avoid losing precision. This option is described in Section 4.3. R also lacks an unsigned integer type.

```r
data.frame <- data.frame(message[[1]] <- "foobar",
cat(message$as.character()))
# name: "foobar"
# id: 42
# email: "foo@bar.com"
```

Table 4 describes the R types that are allowed in the right hand side depending on the target type of the field.

<table>
<thead>
<tr>
<th>internal type</th>
<th>allowed R types</th>
</tr>
</thead>
<tbody>
<tr>
<td>double, float</td>
<td>integer, raw, double, logical</td>
</tr>
<tr>
<td>int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64</td>
<td>integer, raw, double, logical, character</td>
</tr>
<tr>
<td>bool</td>
<td>integer, raw, double, logical</td>
</tr>
<tr>
<td>bytes, string</td>
<td>character</td>
</tr>
<tr>
<td>enum</td>
<td>integer, double, raw, character</td>
</tr>
<tr>
<td>message, group</td>
<td>S4, of class Message of the appropriate message type, or a list of S4 objects of class Message of the appropriate message type.</td>
</tr>
</tbody>
</table>

Table 4. Allowed R types depending on internal field types.

### 2.8.3. Message$has method.

The has method indicates if a field of a message is set. For repeated fields, the field is considered set if there is at least on object in the array. For non-repeated fields, the field is considered set if it has been initialized.

The has method is a thin wrapper around the HasField and FieldSize methods of the `google::protobuf::Reflection` C++ class.

```r
message <- new(tutorial.Person, name = "foo")
message$has("name")
```

# [1] TRUE
message$has("id")

# [1] FALSE

message$has("phone")

# [1] FALSE

### 2.8.4. Message$clone method.
The `clone` function creates a new message that is a clone of the message. This function is a wrapper around the methods `New` and `CopyFrom` of the `google::protobuf::Message` C++ class.

```r
m1 <- new(tutorial.Person, name = "foo")
m2 <- m1$clone()
m2$email <- "foo@bar.com"
cat(as.character(m1))
```

# name: "foo"

```r
cat(as.character(m2))
```

# name: "foo"
# email: "foo@bar.com"

### 2.8.5. Message$isInitialized method.
The `isInitialized` method quickly checks if all required fields have values set. This is a thin wrapper around the `IsInitialized` method of the `google::protobuf::Message` C++ class.

```r
message <- new(tutorial.Person, name = "foo")
message$isInitialized()
```

# [1] FALSE

```r
message$id <- 2
message$isInitialized()
```

# [1] TRUE

### 2.8.6. Message$serialize method.
The `serialize` method can be used to serialize the message as a sequence of bytes into a file or a binary connection.

```r
message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)
tf1 <- tempfile()
tf1
```

# [1] "/tmp/RtmpRBfpcs/file317419c2144f"
message$serialize(tf1)

tf2 <- tempfile()
tf2

# [1] "/tmp/RtmpRBfpcs/file31746f1b8c63"

con <- file(tf2, open = "wb")
message$serialize(con)
close(con)

The (temporary) files tf1 and tf2 both contain the message payload as a sequence of bytes. The `readBin` function can be used to read the files as a raw vector in R:

```r
readBin(tf1, raw(0), 500)
# [1] 0a 03 66 6f 6f 10 02 1a 0b 66 6f 6f 40 62 61 72 2e 63 6f 6d
```

```r
readBin(tf2, raw(0), 500)
# [1] 0a 03 66 6f 6f 10 02 1a 0b 66 6f 6f 40 62 61 72 2e 63 6f 6d
```

The `serialize` method can also be used to directly retrieve the payload of the message as a raw vector:

```r
message$serialize(NULL)
# [1] 0a 03 66 6f 6f 10 02 1a 0b 66 6f 6f 40 62 61 72 2e 63 6f 6d
```

### 2.8.7. Message$clear method

The `clear` method can be used to clear all fields of a message when used with no argument, or a given field.

```r
message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)
cat(as.character(message))

# name: "foo"
# id: 2
# email: "foo@bar.com"
```

```r
message$clear()
cat(as.character(message))
```

```r
message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)
message$clear("id")
cat(as.character(message))

# name: "foo"
# email: "foo@bar.com"
```

The `clear` method is a thin wrapper around the `Clear` method of the `google::protobuf::Message` C++ class.

### 2.8.8. Message$size method

The `size` method is used to query the number of objects in a repeated field of a message:
message <- new(tutorial.Person, name = "foo",
    phone = list(new(tutorial.Person.PhoneNumber,
        number = "+33(0)...", type = "HOME"),
        new(tutorial.Person.PhoneNumber,
            number = "+33(0)###", type = "MOBILE"))
)

message$size("phone")

# [1] 2

size( message, "phone")

# [1] 2

The size method is a thin wrapper around the FieldSize method of the google::protobuf::Reflection C++ class.

2.8.9. Message$bytesize method. The bytesize method retrieves the number of bytes the message would take once serialized. This is a thin wrapper around the ByteSize method of the google::protobuf::Message C++ class.

message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)
message$bytesize()

# [1] 20

bytesize(message)

# [1] 20

length(message$serialize(NULL))

# [1] 20

2.8.10. Message$swap method. The swap method can be used to swap elements of a repeated field.

message <- new(tutorial.Person, name = "foo",
    phone = list(new(tutorial.Person.PhoneNumber,
        number = "+33(0)...", type = "HOME" ),
        new(tutorial.Person.PhoneNumber,
            number = "+33(0)###", type = "MOBILE" )))

message$swap("phone", 1, 2)
cat(as.character(message$phone[[1]]))

# number: "+33(0)###"
# type: MOBILE
```r
# number: "+33(0)...
# type: HOME

swap(message, "phone", 1, 2)

# number: "+33(0)...
# type: HOME

cat(as.character(message$phone[[2]]))

# number: "+33(0)###
# type: MOBILE

cat(as.character(message$phone[[2]]))
```

### 2.8.11. `Message$set` method

The `set` method can be used to set values of a repeated field.

```r
message <- new(tutorial.Person, name = "foo",
                phone = list(new(tutorial.Person.PhoneNumber,
                                 number = "+33(0)...", type = "HOME"),
                                 new(tutorial.Person.PhoneNumber,
                                     number = "+33(0)###", type = "MOBILE")))

number <- new(tutorial.Person.PhoneNumber, number = "+33(0)---", type = "WORK")
message$set("phone", 1, number)

# name: "foo"
# phone {
#   number: "+33(0)---"
#   type: WORK
# }
# phone {
#   number: "+33(0)###"
#   type: MOBILE
# }

cat(as.character(message))
```

### 2.8.12. `Message$fetch` method

The `fetch` method can be used to get values of a repeated field.

```r
message <- new(tutorial.Person, name = "foo",
                phone = list(new(tutorial.Person.PhoneNumber,
                                 number = "+33(0)...", type = "HOME"),
                                 new(tutorial.Person.PhoneNumber,
                                     number = "+33(0)###", type = "MOBILE" )))

message$fetch("phone", 1)

# [[1]]
# message of type 'tutorial.Person.PhoneNumber' with 2 fields set
```

### 2.8.13. `Message$setExtension` method

The `setExtension` method can be used to set an extension field of the Message.
if (!exists("protobuf_unittest.TestAllTypes", "RProtoBuf:DescriptorPool")) {
    unittest.proto.file <- system.file("tinytest", "data", "unittest.proto", package="RProtoBuf")
    readProtoFiles(file=unittest.proto.file)
}

## Test setting a singular extensions.
test <- new(protobuf_unittest.TestAllExtensions)
test$setExtension(protobuf_unittest.optional_int32_extension, as.integer(1))

2.8.14. **Message$getExtension method.** The getExtension method can be used to get values of an extension.

```r

test$getExtension(protobuf_unittest.optional_int32_extension)
```

# [1] 1

2.8.15. **Message$add method.** The add method can be used to add values to a repeated field.

```r
message <- new(tutorial.Person, name = "foo")
phone <- new(tutorial.Person.PhoneNumber, number = "+33(0)...", type = "HOME")
message$add("phone", phone)
cat(message$toString())
```

# name: "foo"
# phone {
#     number: "+33(0)...
#     type: HOME
# }

2.8.16. **Message$str method.** The str method gives the R structure of the message. This is rarely useful.

```r
message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)
message$str()
```

# Formal class 'Message' [package "RProtoBuf"] with 2 slots
# ..@ pointer:<externalptr>
# ..@ type : chr "tutorial.Person"

```r
str(message)
```

# Formal class 'Message' [package "RProtoBuf"] with 2 slots
# ..@ pointer:<externalptr>
# ..@ type : chr "tutorial.Person"

2.8.17. **Message$as.character method.** The as.character method gives the debug string of the message.

```r
message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)
cat(message$as.character())
```
# name: "foo"  # id: 2  # email: "foo@bar.com"

cat(as.character(message))

# name: "foo"  # id: 2  # email: "foo@bar.com"

cat(toString(message))

# name: "foo"  # id: 2  # email: "foo@bar.com"

cat(toString(message))

2.8.18. **Message$toString method.** toString currently is an alias to the as.character function.

message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)  
cat(message$toString())

# name: "foo"  # id: 2  # email: "foo@bar.com"

cat(toString(message))

# name: "foo"  # id: 2  # email: "foo@bar.com"

2.8.19. **Message$as.list method.** The as.list method converts the message to a named R list.

message <- new(tutorial.Person, name = "foo", email = "foo@bar.com", id = 2)  
as.list(message)

# $name  # [1] "foo"  #  # $id  # [1] 2  #  # $email  # [1] "foo@bar.com"  #  # $phone  # list()

The names of the list are the names of the declared fields of the message type, and the content is the same as can be extracted with the $ operator described in section 2.8.1.

2.8.20. **Message$update method.** The update method can be used to update several fields of a message at once.

message <- new(tutorial.Person)  
update(message, name = "foo", id = 2, email = "foo@bar.com")
2.8.21. `Message$descriptor` method. The `descriptor` method retrieves the descriptor of a message. See section 2.9 for more information about message type descriptors.

```r
message <- new(tutorial.Person)
message$descriptor()
```

# descriptor for type 'tutorial.Person'

```r
descriptor(message)
```

2.8.22. `Message$fileDescriptor` method. The `fileDescriptor` method retrieves the file descriptor of the descriptor associated with a message. See section 2.13 for more information about file descriptors.

```r
message <- new(tutorial.Person)
message$fileDescriptor()
```

# file descriptor for package tutorial (addressbook.proto)

```r
descriptor(message)
```

2.9. Message descriptors. Message descriptors are represented in R with the `Descriptor` S4 class. The class contains the slots `pointer` and `type`:

<table>
<thead>
<tr>
<th>slot</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>fully qualified path of the message type.</td>
</tr>
</tbody>
</table>

Similarly to messages, the `$` operator can be used to extract information from the descriptor, or invoke pseudo-methods. Table 6 describes the methods defined for the `Descriptor` class:

2.9.1. Extracting descriptors. The `$` operator, when used on a descriptor object retrieves descriptors that are contained in the descriptor.

This can be a field descriptor (see section 2.10), an enum descriptor (see section 2.11) or a descriptor for a nested type.
<table>
<thead>
<tr>
<th>Method</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>2.9.2</td>
<td>Creates a prototype of a message described by this descriptor.</td>
</tr>
<tr>
<td>read</td>
<td>2.9.3</td>
<td>Reads a message from a file or binary connection.</td>
</tr>
<tr>
<td>readASCII</td>
<td>2.9.4</td>
<td>Read a message in ASCII format from a file or text connection.</td>
</tr>
<tr>
<td>name</td>
<td>2.9.10</td>
<td>Retrieve the name of the message type associated with this descriptor.</td>
</tr>
<tr>
<td>as.character</td>
<td>2.9.6</td>
<td>character representation of a descriptor</td>
</tr>
<tr>
<td>toString</td>
<td>2.9.5</td>
<td>character representation of a descriptor (same as as.character)</td>
</tr>
<tr>
<td>as.list</td>
<td>2.9.7</td>
<td>return a named list of the field, enum, and nested descriptors included in this descriptor.</td>
</tr>
<tr>
<td>asMessage</td>
<td>2.9.8</td>
<td>return DescriptorProto message.</td>
</tr>
<tr>
<td>fileDescriptor</td>
<td>2.9.9</td>
<td>Retrieve the file descriptor of this descriptor.</td>
</tr>
<tr>
<td>containing_type</td>
<td>2.9.11</td>
<td>Retrieve the descriptor describing the message type containing this descriptor.</td>
</tr>
<tr>
<td>field_count</td>
<td>2.9.12</td>
<td>Return the number of fields in this descriptor.</td>
</tr>
<tr>
<td>field</td>
<td>2.9.13</td>
<td>Return the descriptor for the specified field in this descriptor.</td>
</tr>
<tr>
<td>nested_type_count</td>
<td>2.9.14</td>
<td>The number of nested types in this descriptor.</td>
</tr>
<tr>
<td>nested_type</td>
<td>2.9.15</td>
<td>Return the descriptor for the specified nested type in this descriptor.</td>
</tr>
<tr>
<td>enum_type_count</td>
<td>2.9.16</td>
<td>The number of enum types in this descriptor.</td>
</tr>
<tr>
<td>enum_type</td>
<td>2.9.17</td>
<td>Return the descriptor for the specified enum type in this descriptor.</td>
</tr>
</tbody>
</table>

Table 6. Description of methods for the Descriptor S4 class

```
# field descriptor
tutorial.Person$`email`

# descriptor for field 'email' of type 'tutorial.Person'

# enum descriptor
tutorial.Person$`PhoneType`

# descriptor for enum 'PhoneType' with 3 values

# nested type descriptor
tutorial.Person$`PhoneNumber`

# descriptor for type 'tutorial.Person.PhoneNumber'

# same as
tutorial.Person.PhoneNumber

# descriptor for type 'tutorial.Person.PhoneNumber'

2.9.2. The new method. The new method creates a prototype of a message described by the descriptor.

tutorial.Person$`new()`

# message of type 'tutorial.Person' with 0 fields set

new(tutorial.Person)```
Passing additional arguments to the method allows directly setting the fields of the message at construction time.

```r
tutorial.Person$new(email = "foo@bar.com")
```

```r
# message of type 'tutorial.Person' with 1 field set

# same as

update(tutorial.Person$new(), email = "foo@bar.com")
```

### 2.9.3. The read method.
The read method is used to read a message from a file or a binary connection.

```r
# start by serializing a message
message <- new(tutorial.Person.PhoneNumber,
               type = "HOME", number = "+33(0)....")
tf <- tempfile()
serialize(message, tf)

# now read back the message
m <- tutorial.Person.PhoneNumber$read(tf)
cat(as.character(m))
```

```r
# number: "+33(0)...."
# type: HOME

m <- read(tutorial.Person.PhoneNumber, tf)
cat(as.character(m))
```

### 2.9.4. The readASCII method.
The readASCII method is used to read a message from a text file or a character vector.

```r
# start by generating the ASCII representation of a message
text <- as.character(new(tutorial.Person, id=1, name="Murray"))
text
```

```r
# [1] "name: "Murray"

# Then read the ascii representation in as a new message object.
msg <- tutorial.Person$readASCII(text)
```

### 2.9.5. The toString method.
`toString` currently is an alias to the `as.character` function.

### 2.9.6. The as.character method.
`as.character` prints the text representation of the descriptor as it would be specified in the `.proto` file.
```r
desc <- tutorial.Person
cat(desc$toString())

# message Person {
#   message PhoneNumber {
#     required string number = 1;
#     optional .tutorial.Person.PhoneType type = 2 [default = HOME];
#   }
#   enum PhoneType {
#     MOBILE = 0;
#     HOME = 1;
#     WORK = 2;
#   }
#   required string name = 1;
#   required int32 id = 2;
#   optional string email = 3;
#   repeated .tutorial.Person.PhoneNumber phone = 4;
#   extensions 100 to 199;
# }

cat(toString(desc))

# message Person {
#   message PhoneNumber {
#     required string number = 1;
#     optional .tutorial.Person.PhoneType type = 2 [default = HOME];
#   }
#   enum PhoneType {
#     MOBILE = 0;
#     HOME = 1;
#     WORK = 2;
#   }
#   required string name = 1;
#   required int32 id = 2;
#   optional string email = 3;
#   repeated .tutorial.Person.PhoneNumber phone = 4;
#   extensions 100 to 199;
# }

cat(as.character(tutorial.Person))

# message Person {
#   message PhoneNumber {
#     required string number = 1;
#     optional .tutorial.Person.PhoneType type = 2 [default = HOME];
#   }
#   enum PhoneType {
#     MOBILE = 0;
#     HOME = 1;
#     WORK = 2;
#   }
#   required string name = 1;
#   required int32 id = 2;
```
# optional string email = 3;
# repeated .tutorial.Person.PhoneNumber phone = 4;
# extensions 100 to 199;
# }

2.9.7. The as.list method. The as.list method returns a named list of the field, enum, and nested descriptors included in this descriptor.

```r
tutorial.Person$as.list()
```

```r
# $name
descriptor for field 'name' of type 'tutorial.Person'
#
# $id
descriptor for field 'id' of type 'tutorial.Person'
#
# $email
descriptor for field 'email' of type 'tutorial.Person'
#
# $phone
descriptor for field 'phone' of type 'tutorial.Person'
#
# $PhoneNumber
descriptor for type 'tutorial.Person.PhoneNumber'
#
# $PhoneType
descriptor for enum 'PhoneType' with 3 values
```


```r
tutorial.Person$asMessage()
```

```r
# message of type 'google.protobuf.DescriptorProto' with 5 fields set
```

2.9.9. The fileDescriptor method. The fileDescriptor method retrieves the file descriptor of the descriptor. See section~2.13 for more information about file descriptors.

```r
desc <- tutorial.Person
desc$fileDescriptor()
```

```r
# file descriptor for package tutorial (addressbook.proto)
```

```r
fileDescriptor(desc)
```

```r
# file descriptor for package tutorial (addressbook.proto)
```

2.9.10. The name method. The name method can be used to retrieve the name of the message type associated with the descriptor.
# simple name
tutorial.Person$name()

# [1] "Person"

# name including scope
tutorial.Person$name(full = TRUE)

# [1] "tutorial.Person"

2.9.11. The `containing_type` method. The `containing_type` method retrieves the descriptor describing the message type containing this descriptor.

tutorial.Person$containing_type()

# NULL

tutorial.Person$PhoneNumber$containing_type()

# descriptor for type 'tutorial.Person'

2.9.12. The `field_count` method. The `field_count` method retrieves the number of fields in this descriptor.

tutorial.Person$field_count()

# [1] 4

2.9.13. The `field` method. The `field` method returns the descriptor for the specified field in this descriptor.

tutorial.Person$field(1)

# descriptor for field 'name' of type 'tutorial.Person'

2.9.14. The `nested_type_count` method. The `nested_type_count` method returns the number of nested types in this descriptor.

tutorial.Person$nested_type_count()

# [1] 1

2.9.15. The `nested_type` method. The `nested_type` method returns the descriptor for the specified nested type in this descriptor.

tutorial.Person$nested_type(1)

# descriptor for type 'tutorial.Person.PhoneNumber'

2.9.16. The `enum_type_count` method. The `enum_type_count` method returns the number of enum types in this descriptor.
2.9.17. The `enum_type` method. The `enum_type` method returns the descriptor for the specified enum type in this descriptor.

```r
tutorial.Person$enum_type()# [1] 1
```

2.10. Field descriptors. The class `FieldDescriptor` represents field descriptor in R. This is a wrapper S4 class around the `google::protobuf::FieldDescriptor` C++ class. Table 8 describes the methods defined for the `FieldDescriptor` class.

<table>
<thead>
<tr>
<th>slot</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>External pointer to the <code>FieldDescriptor</code> C++ variable</td>
</tr>
<tr>
<td>name</td>
<td>simple name of the field</td>
</tr>
<tr>
<td>full_name</td>
<td>fully qualified name of the field</td>
</tr>
<tr>
<td>type</td>
<td>name of the message type where the field is declared</td>
</tr>
</tbody>
</table>

Table 7. Description of slots for the `FieldDescriptor` S4 class

<table>
<thead>
<tr>
<th>method</th>
<th>section</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as.character</code></td>
<td>2.10.1</td>
<td>character representation of a descriptor</td>
</tr>
<tr>
<td><code>toString</code></td>
<td>2.10.2</td>
<td>character representation of a descriptor (same as <code>as.character</code>)</td>
</tr>
<tr>
<td><code>asMessage</code></td>
<td>2.10.3</td>
<td>return <code>FieldDescriptorProto</code> message.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>2.10.4</td>
<td>Return the name of the field descriptor.</td>
</tr>
<tr>
<td><code>fileDescriptor</code></td>
<td>2.10.5</td>
<td>Return the fileDescriptor where this field is defined.</td>
</tr>
<tr>
<td><code>containing_type</code></td>
<td>2.10.6</td>
<td>Return the containing descriptor of this field.</td>
</tr>
<tr>
<td><code>is_extension</code></td>
<td>2.10.7</td>
<td>Return TRUE if this field is an extension.</td>
</tr>
<tr>
<td><code>number</code></td>
<td>2.10.8</td>
<td>Gets the declared tag number of the field.</td>
</tr>
<tr>
<td><code>type</code></td>
<td>2.10.9</td>
<td>Gets the type of the field.</td>
</tr>
<tr>
<td><code>cpp_type</code></td>
<td>2.10.10</td>
<td>Gets the C++ type of the field.</td>
</tr>
<tr>
<td><code>label</code></td>
<td>2.10.11</td>
<td>Gets the label of a field (optional, required, or repeated).</td>
</tr>
<tr>
<td><code>is_repeated</code></td>
<td>2.10.12</td>
<td>Return TRUE if this field is repeated.</td>
</tr>
<tr>
<td><code>is_required</code></td>
<td>2.10.13</td>
<td>Return TRUE if this field is required.</td>
</tr>
<tr>
<td><code>is_optional</code></td>
<td>2.10.14</td>
<td>Return TRUE if this field is optional.</td>
</tr>
<tr>
<td><code>has_default_value</code></td>
<td>2.10.15</td>
<td>Return TRUE if this field has a default value.</td>
</tr>
<tr>
<td><code>default_value</code></td>
<td>2.10.16</td>
<td>Return the default value.</td>
</tr>
<tr>
<td><code>message_type</code></td>
<td>2.10.17</td>
<td>Return the message type if this is a message type field.</td>
</tr>
<tr>
<td><code>enum_type</code></td>
<td>2.10.18</td>
<td>Return the enum type if this is an enum type field.</td>
</tr>
</tbody>
</table>

Table 8. Description of methods for the `FieldDescriptor` S4 class

2.10.1. The `as.character` method. The `as.character` method gives the debug string of the field descriptor.

```r
cat(as.character(tutorial.Person$PhoneNumber))
```

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

2.10.2. The `toString` method. `toString` is an alias of `as.character`. 
# message PhoneNumber {
#   required string number = 1;
#   optional .tutorial.Person.PhoneType type = 2 [default = HOME];
# }

## 2.10.3. The asMessage method

The `asMessage` method returns a message of type `google.protobuf.FieldDescriptorProto` of the `FieldDescriptor`.

```r
tutorial.Person$id$asMessage()
```

# message of type 'google.protobuf.FieldDescriptorProto' with 4 fields set

```r
cat(as.character(tutorial.Person$id$asMessage()))
```

# name: "id"
# number: 2
# label: LABEL_REQUIRED
# type: TYPE_INT32

## 2.10.4. The name method

The `name` method can be used to retrieve the name of the field descriptor.

```r
# simple name.
name(tutorial.Person$id)
```

# [1] "id"

```r
# name including scope.
name(tutorial.Person$id, full=TRUE)
```

# [1] "tutorial.Person.id"

## 2.10.5. The fileDescriptor method

The `fileDescriptor` method can be used to retrieve the file descriptor of the field descriptor.

```r
fileDescriptor(tutorial.Person$id)
```

# file descriptor for package tutorial (addressbook.proto)

```r
tutorial.Person$id$fileDescriptor()
```

# file descriptor for package tutorial (addressbook.proto)

## 2.10.6. The containing_type method

The `containing_type` method can be used to retrieve the descriptor for the message type that contains this descriptor.
2.10.7. The is_extension method. The is_extension method returns TRUE if this field is an extension.

```r
is_extension( tutorial.Person$id )
```

# [1] FALSE

2.10.8. The number method. The number method returns the declared tag number of this field.

```r
number( tutorial.Person$id )
```

# [1] 2

2.10.9. The type method. The type method can be used to retrieve the type of the field descriptor.

```r
type( tutorial.Person$id )
```

# [1] 5

2.10.10. The cpp_type method. The cpp_type method can be used to retrieve the C++ type of the field descriptor.

```r
cpp_type( tutorial.Person$id )
```
2.10.11. **The `label` method.** Gets the label of a field (optional, required, or repeated). The `label` method returns the label of a field (optional, required, or repeated). By default it returns a number value, but the optional `as.string` argument can be provided to return a human readable string representation.

```r
label(tutorial.Person$id)
```

# [1] 2

```r
label(tutorial.Person$id, TRUE)
```

# [1] "LABEL_REQUIRED"

```r
tutorial.Person$id$label(TRUE)
```

# [1] "LABEL_REQUIRED"

2.10.12. **The `is_repeated` method.** The `is_repeated` method returns TRUE if this field is repeated.

```r
is_repeated(tutorial.Person$id)
```

# [1] FALSE

```r
tutorial.Person$id$is_repeated()
```

# [1] FALSE

2.10.13. **The `is_required` method.** The `is_required` method returns TRUE if this field is required.

```r
is_required(tutorial.Person$id)
```

# [1] TRUE

```r
tutorial.Person$id$is_required()
```

# [1] TRUE

2.10.14. **The `is_optional` method.** The `is_optional` method returns TRUE if this field is optional.
2.10.15. The `has_default_value` method. The `has_default_value` method returns `TRUE` if this field has a default value.

```r
has_default_value(tutorial.Person$PhoneNumber$type)
```

# [1] TRUE

```r
has_default_value(tutorial.Person$PhoneNumber$number)
```

# [1] FALSE

2.10.16. The `default_value` method. The `default_value` method returns the default value of a field.

```r
default_value(tutorial.Person$PhoneNumber$type)
```

# [1] 1

```r
default_value(tutorial.Person$PhoneNumber$number)
```

# [1] ""

2.10.17. The `message_type` method. The `message_type` method returns the message type if this is a message type field.

```r
message_type(tutorial.Person$phone)
```

# descriptor for type 'tutorial.Person.PhoneNumber'

```r
tutorial.Person$phone$message_type()
```

# descriptor for type 'tutorial.Person.PhoneNumber'

2.10.18. The `enum_type` method. The `enum_type` method returns the enum type if this is an enum type field.

```r
enum_type(tutorial.Person$PhoneNumber$type)
```
# descriptor for enum 'PhoneType' with 3 values

## 2.11. Enum descriptors

The class `EnumDescriptor` is an R wrapper class around the C++ class `google::protobuf::EnumDescriptor`. Table 10 describes the methods defined for the `EnumDescriptor` class.

<table>
<thead>
<tr>
<th>slot</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>External pointer to the <code>EnumDescriptor</code> C++ variable</td>
</tr>
<tr>
<td>name</td>
<td>simple name of the enum</td>
</tr>
<tr>
<td>full_name</td>
<td>fully qualified name of the enum</td>
</tr>
<tr>
<td>type</td>
<td>name of the message type where the enum is declared</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>method</th>
<th>section</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>as.list</td>
<td>2.11.2</td>
<td>return a named integer vector with the values of the enum and their names.</td>
</tr>
<tr>
<td>as.character</td>
<td>2.11.3</td>
<td>character representation of a descriptor</td>
</tr>
<tr>
<td>toString</td>
<td>2.11.4</td>
<td>character representation of a descriptor (same as as.character)</td>
</tr>
<tr>
<td>asMessage</td>
<td>2.11.5</td>
<td>return <code>EnumDescriptorProto</code> message.</td>
</tr>
<tr>
<td>name</td>
<td>2.11.6</td>
<td>Return the name of the enum descriptor.</td>
</tr>
<tr>
<td>fileDescriptor</td>
<td>2.11.7</td>
<td>Return the fileDescriptor where this field is defined.</td>
</tr>
<tr>
<td>containing_type</td>
<td>2.11.8</td>
<td>Return the containing descriptor of this field.</td>
</tr>
<tr>
<td>length</td>
<td>2.11.9</td>
<td>Return the number of constants in this enum.</td>
</tr>
<tr>
<td>has</td>
<td>2.11.10</td>
<td>Return TRUE if this enum contains the specified named constant string.</td>
</tr>
<tr>
<td>value_count</td>
<td>2.11.11</td>
<td>Return the number of constants in this enum (same as length).</td>
</tr>
<tr>
<td>value</td>
<td>2.11.12</td>
<td>Return the <code>EnumValueDescriptor</code> of an enum value of specified index, name, or number.</td>
</tr>
</tbody>
</table>

### 2.11.1. Extracting descriptors

The `$` operator, when used on a `EnumDescriptor` object retrieves `EnumValueDescriptors` that are contained in the descriptor.

```r
tutorial.Person$PhoneType$WORK
```

# [1] 2

```r
name(tutorial.Person$PhoneType$value(number=2))
```

# [1] "WORK"

### 2.11.2. The as.list method

The `as.list` method creates a named R integer vector that captures the values of the enum and their names.

```r
as.list(tutorial.Person$PhoneType)
```

# $MOBILE  # [1] 0  
# # $HOME  # [1] 1  
# # $WORK  # [1] 2
2.11.3. The as.character method. The as.character method gives the debug string of the enum type.

```r
cat(as.character(tutorial.Person$PhoneType ))
```

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

2.11.4. The toString method. The toString method gives the debug string of the enum type.

```r
{ toStringmethod3 } cat(toString(tutorial.Person$PhoneType))
```

2.11.5. The asMessage method. The asMessage method returns a message of type google.protobuf.EnumDescriptorProto of the EnumDescriptor.

```r
tutorial.Person$PhoneType$asMessage()
```

```r
# message of type 'google.protobuf.EnumDescriptorProto' with 2 fields set

```r
cat(as.character(tutorial.Person$PhoneType$asMessage()))
```

```r
# name: "PhoneType"
# value { 
#    name: "MOBILE"
#    number: 0
# } 
# value { 
#    name: "HOME"
#    number: 1
# } 
# value { 
#    name: "WORK"
#    number: 2
# }
```

2.11.6. The name method. The name method can be used to retrieve the name of the enum descriptor.

```r
# simple name.
name( tutorial.Person$PhoneType )
```

```r
# [1] "PhoneType"
```

```r
# name including scope.
name( tutorial.Person$PhoneType, full=TRUE )
```

```r
# [1] "tutorial.Person.PhoneType"
```

2.11.7. The fileDescriptor method. The fileDescriptor method can be used to retrieve the file descriptor of the enum descriptor.

```r
```
2.11.8. The containing_type method. The containing_type method can be used to retrieve the descriptor for the message type that contains this enum descriptor.

```
tutorial.Person$PhoneType$containing_type()
```

# descriptor for type 'tutorial.Person'

2.11.9. The length method. The length method returns the number of constants in this enum.

```
length(tutorial.Person$PhoneType)
```

# [1] 3

```
tutorial.Person$PhoneType$length()
```

# [1] 3

2.11.10. The has method. The has method returns TRUE if this enum contains the specified named constant string.

```
tutorial.Person$PhoneType$has("WORK")
```

# [1] TRUE

```
tutorial.Person$PhoneType$has("nonexistent")
```

# [1] FALSE

2.11.11. The value_count method. The value_count method returns the number of constants in this enum.

```
value_count(tutorial.Person$PhoneType)
```

# [1] 3

```
tutorial.Person$PhoneType$value_count()
```
2.11.12. **The value method.** The value method extracts an EnumValueDescriptor. Exactly one argument of ‘index’, ‘number’, or ‘name’ must be specified to identify which constant is desired.

```r
tutorial.Person$PhoneType$value(1)
```

# enum value descriptor tutorial.Person.MOBILE

```r
tutorial.Person$PhoneType$value(name="HOME")
```

# enum value descriptor tutorial.Person.HOME

```r
tutorial.Person$PhoneType$value(number=1)
```

# enum value descriptor tutorial.Person.HOME

### 2.12. Enum value descriptors.

The class `EnumValueDescriptor` is an R wrapper class around the C++ class `google::protobuf::EnumValueDescriptor`. Table~12 describes the methods defined for the `EnumValueDescriptor` class.

<table>
<thead>
<tr>
<th>slot</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>External pointer to the <code>EnumValueDescriptor</code> C++ variable</td>
</tr>
<tr>
<td>name</td>
<td>simple name of the enum value</td>
</tr>
<tr>
<td>full_name</td>
<td>fully qualified name of the enum value</td>
</tr>
</tbody>
</table>

**Table 11. Description of slots for the `EnumValueDescriptor` S4 class**

<table>
<thead>
<tr>
<th>method</th>
<th>section</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>2.12.1</td>
<td>return the number of this <code>EnumValueDescriptor</code>.</td>
</tr>
<tr>
<td>name</td>
<td>2.12.2</td>
<td>Return the name of the enum value descriptor.</td>
</tr>
<tr>
<td>enum_type</td>
<td>2.12.3</td>
<td>return the <code>EnumDescriptor</code> type of this <code>EnumValueDescriptor</code>.</td>
</tr>
<tr>
<td>as.character</td>
<td>2.12.4</td>
<td>character representation of a descriptor.</td>
</tr>
<tr>
<td>asString</td>
<td>2.12.5</td>
<td>character representation of a descriptor (same as as.character).</td>
</tr>
<tr>
<td>asMessage</td>
<td>2.12.6</td>
<td>return <code>EnumValueDescriptorProto</code> message.</td>
</tr>
</tbody>
</table>

**Table 12. Description of methods for the `EnumValueDescriptor` S4 class**

#### 2.12.1. **The number method.**

The `number` method can be used to retrieve the number of the enum value descriptor.

```r
number(tutorial.Person$PhoneType$value(number=2))
```

# [1] 2

#### 2.12.2. **The name method.**

The `name` method can be used to retrieve the name of the enum value descriptor.

```r
# simple name.
name(tutorial.Person$PhoneType$value(number=2))
```
# [1] "WORK"

# name including scope.
name(tutorial.Person$PhoneType$value(number=2), full=TRUE)

# [1] "tutorial.Person.WORK"

## 2.12.3. The enum_type method.
The `enum_type` method can be used to retrieve the `EnumDescriptor` of the enum value descriptor.

`enum_type(tutorial.Person$PhoneType$value(number=2))`

# descriptor for enum 'PhoneType' with 3 values

## 2.12.4. The as.character method.
The `as.character` method gives the debug string of the enum value type.

`cat(as.character(tutorial.Person$PhoneType$value(number=2)))`

# WORK = 2;

## 2.12.5. The toString method.
The `toString` method gives the debug string of the enum value type.

`cat(toString(tutorial.Person$PhoneType$value(number=2)))`

# WORK = 2;

## 2.12.6. The asMessage method.
The `asMessage` method returns a message of type `google.protobuf.EnumValueDescriptorProto` of the `EnumValueDescriptor`.

`tutorial.Person$PhoneType$value(number=2)$asMessage()`

# message of type 'google.protobuf.EnumValueDescriptorProto' with 2 fields set

`cat(as.character(tutorial.Person$PhoneType$value(number=2)$asMessage()))`

# name: "WORK"
# number: 2

### 2.13. File descriptors.
File descriptors describe a whole `.proto` file and are represented in R with the `FileDescriptor S4` class. The class contains the slots `pointer`, `filename`, and `package`:

Similarly to messages, the `$` operator can be used to extract fields from the file descriptor (in this case, types defined in the file), or invoke pseudo-methods. Table~14 describes the methods defined for the `FileDescriptor` class.

```r
f <- tutorial.Person$fileDescriptor()
f
```
Table 13. Description of slots for the FileDescriptor S4 class

<table>
<thead>
<tr>
<th>slot</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer</td>
<td>external pointer to the FileDescriptor object of the C++ proto library. Documentation for the FileDescriptor class is available from the protocol buffer project page: <a href="http://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#FileDescriptor">http://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#FileDescriptor</a></td>
</tr>
<tr>
<td>filename</td>
<td>fully qualified pathname of the .proto file.</td>
</tr>
<tr>
<td>package</td>
<td>package name defined in this .proto file.</td>
</tr>
</tbody>
</table>

Table 14. Description of methods for the FileDescriptor S4 class

<table>
<thead>
<tr>
<th>method</th>
<th>section</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>2.13.5</td>
<td>Return the filename for this FileDescriptorProto.</td>
</tr>
<tr>
<td>package</td>
<td>2.13.6</td>
<td>Return the file-level package name specified in this FileDescriptorProto.</td>
</tr>
<tr>
<td>as.character</td>
<td>2.13.1</td>
<td>character representation of a descriptor.</td>
</tr>
<tr>
<td>toString</td>
<td>2.13.2</td>
<td>character representation of a descriptor (same as as.character).</td>
</tr>
<tr>
<td>asMessage</td>
<td>2.13.3</td>
<td>return FileDescriptorProto message.</td>
</tr>
<tr>
<td>as.list</td>
<td>2.13.4</td>
<td>return named list of descriptors defined in this file descriptor.</td>
</tr>
</tbody>
</table>

2.13.1. The as.character method. The as.character method gives the debug string of the file descriptor.

```
cat(as.character(fileDescriptor(tutorial.Person)))
```

```r
# syntax = "proto2";
#
# package tutorial;
#
# option java_package = "com.example.tutorial";
# option java_outer_classname = "AddressBookProtos";
#
# message Person {
#     message PhoneNumber {
#         required string number = 1;
#         optional .tutorial.Person.PhoneType type = 2 [default = HOME];
#     }
#     enum PhoneType {
#         MOBILE = 0;
#         HOME = 1;
#         WORK = 2;
#     }
#     required string name = 1;
#     required int32 id = 2;
#     optional string email = 3;
#     repeated .tutorial.Person.PhoneNumber phone = 4;
#     extensions 100 to 199;
# }
#
# message AddressBook {
```
2.13.2. The `toString` method. `toString` is an alias of `as.character`.

```r
cat(fileDescriptor(tutorial.Person)$toString())
```

2.13.3. The `asMessage` method. The `asMessage` method returns a protocol buffer message representation of the file descriptor.

```r
asMessage(tutorial.Person$fileDescriptor())
```

```r
# message of type 'google.protobuf.FileDescriptorProto' with 5 fields set
cat(as.character(asMessage(tutorial.Person$fileDescriptor())))
```
# name: "addressbook.proto"
# package: "tutorial"
# message_type {
#   name: "Person"
#   field {
#     name: "name"
#     number: 1
#     label: LABEL_REQUIRED
#     type: TYPE_STRING
#   }
#   field {
#     name: "id"
#     number: 2
#     label: LABEL_REQUIRED
#     type: TYPE_INT32
#   }
#   field {
#     name: "email"
#     number: 3
#     label: LABEL_OPTIONAL
#     type: TYPE_STRING
#   }
#   field {
#     name: "phone"
#     number: 4
#     label: LABEL_REPEATED
#     type: TYPE_MESSAGE
#     type_name: ".tutorial.Person.PhoneNumber"
#   }
#   nested_type {
#     name: "PhoneNumber"
#     field {
#       name: "number"
#       number: 1
#       label: LABEL_REQUIRED
#       type: TYPE_STRING
#     }
#     field {
#       name: "type"
#       number: 2
#       label: LABEL_OPTIONAL
#       type: TYPE_ENUM
#       type_name: ".tutorial.Person.PhoneType"
#       default_value: "HOME"
#     }
#   }
#   enum_type {
#     name: "PhoneType"
#     value {
#       name: "MOBILE"
#       number: 0
#     }
#     value {
#       name: "HOME"
#       number: 1
#     }
#   }
# }
2.13.4. The as.list method. The as.list method creates a named R list that contains the descriptors defined in this file descriptor.

```r
as.list(tutorial.Person$fileDescriptor())
```

```r
# $Person
# descriptor for type 'tutorial.Person'
#
# $AddressBook
# descriptor for type 'tutorial.AddressBook'
#
# $EchoService
```

2.13.5. The name method. The name method can be used to retrieve the file name associated with the file descriptor. The optional boolean argument can be specified if full pathnames are desired.

```r
name(tutorial.Person$fileDescriptor())
```

```r
# [1] "addressbook.proto"
```
# [1] "addressbook.proto"

### 2.13.6. The package method.
The package method can be used to retrieve the package scope associated with this file descriptor.

```r
# [1] "tutorial"
```

Not fully implemented. Needs to be connected to a concrete RPC implementation. The Google Protocol Buffers C++ open-source library does not include an RPC implementation, but this can be connected easily to others.

Not fully implemented. Needs to be connected to a concrete RPC implementation. The Google Protocol Buffers C++ open-source library does not include an RPC implementation, but this can be connected easily to others. Now that Google gRPC is released, this an obvious possibility. Contributions would be most welcome.

### 3. Utilities

#### 3.1. Coercing objects to messages.
The `asMessage` function uses the standard coercion mechanism of the `as` method, and so can be used as a shorthand:

```r
# coerce a message type descriptor to a message
asMessage(tutorial.Person)
```

# message of type 'google.protobuf.DescriptorProto' with 5 fields set

```r
# coerce an enum descriptor
asMessage(tutorial.Person.PhoneType)
```

# message of type 'google.protobuf.EnumDescriptorProto' with 2 fields set

```r
# coerce a field descriptor
asMessage(tutorial.Person$email)
```

# message of type 'google.protobuf.FieldDescriptorProto' with 4 fields set

```r
# coerce a file descriptor
asMessage(fileDescriptor(tutorial.Person))
```

# message of type 'google.protobuf.FileDescriptorProto' with 5 fields set
3.2. Completion. The RProtoBuf package implements the .DollarNames S3 generic function (defined in the utils package) for all classes.

Completion possibilities include pseudo method names for all classes, plus:

- field names for messages
- field names, enum types, nested types for message type descriptors
- names for enum descriptors
- names for top-level extensions
- message names for file descriptors

In the unlikely event that there is a user-defined field of exactly the same name as one of the pseudo methods, the user-defined field shall take precedence for completion purposes by design, since the method name can always be invoked directly.

3.3. with and within. The S3 generic with function is implemented for class Message, allowing to evaluate an R expression in an environment that allows to retrieve and set fields of a message simply using their names.

```
{r withwithin message <- new(tutorial.Person, email = "foo"
## The com" method with(message, { ##
## set the id field id <- 2

### set the name field from the email field
name <- gsub( "@[\]", " ", email )

sprintf( "%d [%s] : %s", id, email, name )

})
```

The difference between `with` and `within` is the value that is returned. For `with` returns the result of the R expression, for `within` the message is returned. In both cases, the message is modified because RProtoBuf works by reference.

```
## identical

The identical method is implemented to compare two messages.

```{r}
m1 <- new(tutorial.Person, email = "foo@bar.com", id = 2)
m2 <- update(new(tutorial.Person), email = "foo@bar.com", id = 2)
identical(m1, m2)
```

# [1] TRUE

The `==` operator can be used as an alias to `identical`.

```
m1 == m2
```

# [1] TRUE
Alternatively, the `all.equal` function can be used, allowing a tolerance when comparing `float` or `double` values.

### 3.4. `merge`

`merge` can be used to merge two messages of the same type.

```r
m1 <- new(tutorial.Person, name = "foobar")
m2 <- new(tutorial.Person, email = "foo@bar.com")
m3 <- merge(m1, m2)
cat(as.character(m3))
```

```r
# name: "foobar"
# email: "foo@bar.com"
```

### 3.5. `P`

The `P` function is an alternative way to retrieve a message descriptor using its type name. It is not often used because of the lookup mechanism described in section 4.2.

```r
P("tutorial.Person")
```

```r
# descriptor for type 'tutorial.Person'

new(P("tutorial.Person"))
```

```r
# message of type 'tutorial.Person' with 0 fields set

# but we can do this instead
tutorial.Person
```

```r
# descriptor for type 'tutorial.Person'

new(tutorial.Person)
```

```r
# message of type 'tutorial.Person' with 0 fields set

```

### 4. Advanced Features

#### 4.1. Extensions

Extensions allow you to declare a range of field numbers in a message that are available for extension types. This allows others to declare new fields for a given message type possibly in their own .proto files without having to edit the original file. See [https://developers.google.com/protocol-buffers/docs/proto#extensions](https://developers.google.com/protocol-buffers/docs/proto#extensions).

Notice that the last line of the `Person` message schema in `addressbook.proto` is the following line:

```
extensions 100 to 199;
```

This specifies that other users in other .proto files can use tag numbers between 100 and 199 for extension types of this message.
4.2. Descriptor lookup. The RProtoBuf package uses the user defined tables framework that is defined as part of the RObjectTables package available from the OmegaHat project.

The feature allows RProtoBuf to install the special environment RProtoBuf:DescriptorPool in the R search path. The environment is special in that, instead of being associated with a static hash table, it is dynamically queried by R as part of R’s usual variable lookup. In other words, it means that when the R interpreter looks for a binding to a symbol (foo) in its search path, it asks to our package if it knows the binding “foo”, this is then implemented by the RProtoBuf package by calling an internal method of the protobuf C++ library.

4.3. 64-bit integer issues. R does not have native 64-bit integer support. Instead, R treats large integers as doubles which have limited precision. For example, it loses the ability to distinguish some distinct integers:

\[ 2^{53} = (2^{53} + 1) \]

\[ # [1] TRUE \]

Protocol Buffers are frequently used to pass data between different systems, however, and most other systems these days have support for 64-bit integers. To work around this, RProtoBuf allows users to get and set 64-bit integer types by treating them as characters when running on a platform with a 64-bit long long type available.

If we try to set an int64 field in R to double values, we lose precision:

4.4. Deprecated Feature: Protocol Buffer Groups. Groups are a deprecated feature that offered another way to nest information in message definitions. For example, the TestAllTypes message type in unittest.proto includes an OptionalGroup type:

```protobuf
optional group OptionalGroup = 16 {
  optional int32 a = 17;
}
```

And although the feature is deprecated, it can be used with RProtoBuf:

```r
test <- new(probuf_unittest.TestAllTypes)
test$optionalgroup$a <- 3
test$optionalgroup$a
```

# [1] 3

```r
cat(as.character(test))
```

# OptionalGroup {
#   a: 3
# }

Note that groups simply combine a nested message type and a field into a single declaration. The field type is OptionalGroup in this example, and the field name is converted to lower-case ‘optionalgroup’ so as not to conflict with the type name.

Note that groups simply combine a nested message type and a field into a single declaration. The field type is OptionalGroup in this example, and the field name is converted to lower-case ‘optionalgroup’ so as not to conflict with the type name.

5. Other approaches


Jeroen Ooms took a similar approach influenced by Saptarshi in his RProtoBufUtils package. Unlike Saptarshi’s package, RProtoBufUtils depends on RProtoBuf for underlying message operations. This package is available at https://github.com/jeroenooms/RProtoBufUtils.
6. Plans for future releases

Protocol Buffers have a mechanism for remote procedure calls (RPC) that is not yet used by RProtoBuf, but we may one day take advantage of this by writing a Protocol Buffer message R server, and client code as well, probably based on the functionality of the Rserve package. Now that Google gRPC is released, this an obvious possibility. Contributions would be most welcome.

Extensions have been implemented in RProtoBuf and have been extensively used and tested, but they are not currently described in this vignette. Additional examples and documentation are needed for extensions.

7. Acknowledgements

Some of the design of the package is based on the design of the rJava package by Simon Urbanek (dispatch on new, S4 class structures using external pointers, etc). We would like to thank Simon for his indirect involvement on RProtoBuf. The user defined table mechanism, implemented by Duncan Temple Lang for the purpose of the RObjectTables package allowed the dynamic symbol lookup (see section—4.2). Many thanks to Duncan for this amazing feature.