FREE eBook

LEARNING guava

Free unaffiliated eBook created from
Stack Overflow contributors.

#guava

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: guava

It is an unofficial and free guava ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official guava.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with guava

Remarks

This section provides an overview of what guava is, and why a developer might want to use it.

It should also mention any large subjects within guava, and link out to the related topics. Since the Documentation for guava is new, you may need to create initial versions of those related topics.

Versions

Version	Release Date
r01%	2009-09-15
r02	2010-01-04
r03	2010-04-09
r04	2010-04-27
r05	2010-05-28
r06	2010-07-07
7.0	2010-09-22
8.0	2011-01-27
9.0	2011-04-07
10.0	2011-09-28
11.0	2011-12-18
12.0	2012-04-30
13.0	2012-08-03
14.0	2013-02-25
15.0	2013-09-06
16.0	2014-01-17
17.0	2014-04-22
18.0	2014-08-25

Version	Release Date
19.0	2015-12-09

% did not include Google Collections, which existed separately at that time

Note: Releases 1.0 through 11.0 require JDK 1.5 or newer. Releases 12.0 through 20.0 require JDK 1.6 or newer. Release 21.0 is expected to require JDK 1.8 or newer.

Examples

Setup

Dependency on Guava can be added in your Java project by using any build system.

Maven:

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>19.0</version>
</dependency>
```

Gradle:

```
dependencies {
  compile 'com.google.guava:guava:19.0'
}
```

Ivy

```
<dependency org="com.google.guava" name="guava" rev="19.0" />
```

Buildr

```
compile.with 'com.google.guava:guava:jar:19.0'
```

Manual Dependency

You can also just manually download JARs from Guava's release page for the classes, sources and javadocs.

Note that JDK 1.6 or newer is required for Guava 12.0 through 20.0. See Version list for more info. Guava users who target Java 5 should use the Guava JDK5 backport. This includes users who target Android Froyo and earlier.

Read Getting started with guava online: https://riptutorial.com/guava/topic/4391/getting-started-with-guava

Chapter 2: I/O

Examples

Handling existing InputStreams and OutputStreams

Reading the content of an `InputStream` as a `byte` array:

```
// Reading from a file
try (InputStream in = new FileInputStream("in.dat")) {
  byte[] content = ByteStreams.toByteArray(in);
  // do something with content
}
```

Copying an `InputStream` to an `OutputStream`:

```
// Copying the content from a file in.dat to out.dat.
try (InputStream in = new FileInputStream("in.dat");
     OutputStream out = new FileOutputStream("out.dat")) {
  ByteStreams.copy(in, out);
}
```

Note: to copy files directly, it's better to use `Files.copy(sourceFile, destinationFile)`.

Reading an entire predefined `byte` array from an `InputStream`:

```
try (InputStream in = new FileInputStream("in.dat")) {
  byte[] bytes = new byte[16];
  ByteStreams.readFully(in, bytes);
  // bytes is totally filled with 16 bytes from the InputStream.
} catch (EOFException ex) {
  // there was less than 16 bytes in the InputStream.
}
```

Skipping `n` bytes from the `InputStream`:

```
try (InputStream in = new FileInputStream("in.dat")) {
  ByteStreams.skipFully(in, 20);
  // the next byte read will be the 21st.
  int data = in.read();
} catch (EOFException e) {
  // There was less than 20 bytes in the InputStream.
}
```

Creating an `OutputStream` that discards everything that is written to it:

```
try (InputStream in = new FileInputStream("in.dat");
     OutputStream out = ByteStreams.nullOutputStream()) {
  ByteStreams.copy(in, out);
  // The whole content of in is read into... nothing.
}
```

Handling existing Readers and Writers

Reading the content of a `Reader` as a `String`:

```
// Reading from a file
try (Reader reader = new FileReader("in.txt")) {
  String content = CharStreams.toString(reader);
  // do something with content
}
```

Reading the content of a `Reader` as a list of line contents:

```
try (Reader reader = new FileReader("in.txt")) {
  List<String> lines = CharStreams.readLines(reader);
  for (String line: lines) {
    // Do something with line
  }
}
```

Copying a `Reader` to a `Writer`:

```
try (Reader reader = new FileReader("in.txt");
     Writer writer = new FileWriter("out.txt")) {
  CharStreams.copy(reader, writer);
}
```

Note: to copy files directly, it's better to use Files.copy(sourceFile, destinationFile).

Skipping `n` bytes from the `Reader`:

```
try (Reader reader = new FileReader("in.txt")) {
  CharStreams.skipFully(reader, 20);
  // The next char read will be the 21st.
} catch (EOFException e) {
  // There was less than 20 chars in the Reader.
}
```

Creating a `Writer` that discards everything that is written to it:

```
try (Reader reader = new FileReader("in.txt");
     Writer writer = CharStreams.nullWriter()) {
  CharStreams.copy(reader, writer);
  // The whole content of reader is read into... nothing.
}
```

Sources and sinks

Sources and sinks are objects that know how to open streams.

	Bytes	Chars
Reading	`ByteSource`	`CharSource`

	Bytes	Chars
Writing	`ByteSink`	`CharSink`

Creating sources and sinks

Note: for all examples, consider `UTF_8` as if the following import is set:

```
import static java.nio.charset.StandardCharsets.UTF_8;
```

Reading from a file

```
ByteSource dataSource = Files.asByteSource(new File("input.dat"));
CharSource textSource = Files.asCharSource(new File("input.txt"), UTF_8);
```

Writing to a file

```
ByteSink dataSink = Files.asByteSink(new File("output.dat"));
CharSink textSink = Files.asCharSink(new File("output.txt"), UTF_8);
```

Reading from a URL

```
ByteSource dataSource = Resources.asByteSource(url);
CharSource textSource = Resources.asCharSource(url, UTF_8);
```

Reading from in memory data

```
ByteSource dataSource = ByteSource.wrap(new byte[] {1, 2, 3});
CharSource textSource = CharSource.wrap("abc");
```

Converting from bytes to chars

```
ByteSource originalSource = ...
CharSource textSource = originalSource.asCharSource(UTF_8);
```

Converting from chars to bytes

(From Guava 20 onwards)

```
CharSource originalSource = ...
ByteSource dataSource = originalSource.asByteSource(UTF_8);
```

Using sources and sinks

Common operations

Opening a stream

```
InputStream inputStream = byteSource.openStream();
OutputStream outputStream = byteSink.openStream();
Reader reader = charSource.openStream();
Writer writer = charSink.openStream();
```

Opening a buffered stream

```
InputStream bufferedInputStream = byteSource.openBufferedStream();
OutputStream bufferedOutputStream = byteSink.openBufferedStream();
BufferedReader bufferedReader = charSource.openBufferedStream();
Writer bufferedWriter = charSink.openBufferedStream();
```

Source operations

Reading from a source:

```
ByteSource source = ...
byte[] bytes = source.read();

CharSource source = ...
String text = source.read();
```

Reading lines from a source:

```
CharSource source = ...
ImmutableList<String> lines = source.readLines();
```

Reading the first line from a source:

```
CharSource source = ...
String firstLine = source.readFirstLine();
```

Copying from a source to a sink:

```
ByteSource source = ...
ByteSink sink = ...
source.copyTo(sink);

CharSource source = ...
CharSink sink = ...
source.copyTo(sink);
```

Typical usage

```
CharSource source = ...
try (Reader reader = source.openStream()) {
  // use the reader
}
```

Read I/O online: https://riptutorial.com/guava/topic/6929/i-o

Chapter 3: Strings

Examples

Checking a string for unwanted characters

As a developer, you frequently find yourself dealing with strings that are not created by your own code.

These will often be supplied by third party libraries, external systems, or even end users. Validating strings of unclear provenance is considered to be one of the hallmarks of defensive programming, and in most cases you will want to reject string input that does not meet your expectations.

A fairly common case is where you would only want to allow alphanumeric characters in an input string, so we'll use that as an example. In plain Java, the following two methods both serve the same purpose:

```
public static boolean isAlphanumeric(String s) {
    for (char c : s.toCharArray()) {
        if (!Character.isLetterOrDigit(c)) {
            return false;
        }
    }

    return true;
}
```

```
public static boolean isAlphanumeric(String s) {
    return s.matches("^[0-9a-zA-Z]*$");
}
```

The first version converts the string to a character array, and then uses the `Character` class' static `isLetterOrDigit` method to determine whether the characters contained in the array are alphanumeric or not. This approach is predictable and readable, albeit a little bit verbose.

The second version uses a regular expression to achieve the same purpose. It is more concise, but can be somewhat enigmatic to developers with limited or no knowledge of regular expressions.

Guava introduces the `CharMatcher` class to deal with these types of situations. Our alphanumeric test, using Guava, would look as follows:

```
import static com.google.common.base.CharMatcher.javaLetterOrDigit;

/* ... */

public static boolean isAlphanumeric(String s) {
    return javaLetterOrDigit().matchesAllOf(s);
}
```

The method body contains only one line, but there's actually a lot going on here, so let's break things down a little bit further.

If you take a look at the API of Guava's `CharMatcher` class, you'll notice that it implements the `Predicate<Character>` interface. If you would create a class that implements `Predicate<Character>` yourself, it could look something like this:

```
import com.google.common.base.Predicate;

public class AlphanumericPredicate implements Predicate<Character> {
    @Override
    public boolean apply(Character c) {
        return Character.isLetterOrDigit(c);
    }
}
```

In Guava, as in a number of other programming languages and libraries that cater to a functional style of programming, a predicate is a construct that evaluates a given input to either true or false. In Guava's `Predicate<T>` interface, this is made evident by the presence of the sole `boolean apply(T t)` method. The `CharMatcher` class is built on this concept, and will evaluate a character or sequence of characters to check whether or not they match the criteria laid out by the used `CharMatcher` instance.

Guava currently provides the following predefined character matchers:

Matcher	Description
`any()`	Matches any character.
`none()`	Matches no characters.
`javaDigit()`	Matches digits, according to the Java definition.
`javaUpperCase()`	Matches any upper case character, according to Java's definition.
`javaLowerCase()`	Matches any lower case character, according to Java's definition.
`javaLetter()`	Matches any letter, according to Java's definition.
`javaLetterOrDigit()`	Matches any letter or digit, according to Java's definition.
`javaIsoControl()`	Matches any ISO control character, according to Java's definition.
`ascii()`	Matches any character in the ASCII character set.
`invisible()`	Matches characters that are not visible, according to the Unicode standard.
`digit()`	Matches any digit, according to the Unicode specification.
`whitespace()`	Matches any whitespace character, according to the Unicode

Matcher	Description
	specification.
`breakingWhitespace()`	Matches any breaking whitespace character, according to the unicode specification.
`singleWidth()`	Matches any single-width character.

If you have read through the above table, you've undoubtedly noticed the amount of definition and specification involved in determining which characters belong to a certain category. Guava's approach, so far, has been to provide `CharMatcher` wrappers for a number of the character categories defined by Java, and you can consult the API of Java's `Character` class to get more information about these categories. On the other hand, Guava attempts to supply a number of `CharMatcher` instances that are in line with the current Unicode specification. For the nitty-gritty details, consult the `CharMatcher` API documentation.

Getting back to our example of checking a string for unwanted characters, the following `CharMatcher` methods provide the capabilities you need to check whether a given string's character usage meets your requirements:

- `boolean matchesNoneOf(CharSequence sequence)`
  Returns true if none of the characters in the argument string match the `CharMatcher` instance.

- `boolean matchesAnyOf(CharSequence sequence)`
  Returns true if at least one character in the argument string matches the `CharMatcher` instance.

- `boolean matchesAllOf(CharSequence sequence)`
  Returns true if all of the characters in the argument string match the CharMatcher instance.

Finding and counting characters in a string

To help you find and count characters in a string, `CharMatcher` provides the following methods:

- `int indexIn(CharSequence sequence)`
  Returns the index of the first character that matches the `CharMatcher` instance. Returns -1 if no character matches.

- `int indexIn(CharSequence sequence, int start)`
  Returns the index of the first character after the specified start position that matches the `CharMatcher` instance. Returns -1 if no character matches.

- `int lastIndexIn(CharSequence sequence)`
  Returns the index of the last character that matches the `CharMatcher` instance. Returns -1 if no character matches.

- `int countIn(CharSequence sequence)`
  Returns the number of characters that match the `CharMatcher` instance.

Using these methods, here's a simple console application called `NonAsciiFinder` that takes a string as an input argument. First, it prints out the total number of non-ASCII characters contained in the string. Subsequently, it prints out the Unicode representation of each non-ASCII character it encounters. Here's the code:

```
import com.google.common.base.CharMatcher;

public class NonAsciiFinder {
    private static final CharMatcher NON_ASCII = CharMatcher.ascii().negate();

    public static void main(String[] args) {
        String input = args[0];
        int nonAsciiCount = NON_ASCII.countIn(input);

        echo("Non-ASCII characters found: %d", nonAsciiCount);

        if (nonAsciiCount > 0) {
            int position = -1;
            char character = 0;

            while (position != NON_ASCII.lastIndexIn(input)) {
                position = NON_ASCII.indexIn(input, position + 1);
                character = input.charAt(position);

                echo("%s => \\u%04x", character, (int) character);
            }
        }
    }

    private static void echo(String s, Object... args) {
        System.out.println(String.format(s, args));
    }
}
```

Note in the above example how you can simply invert a `CharMatcher` by calling its `negate` method. Similarly the `CharMatcher` below matches all double-width characters and is created by negating the predefined `CharMatcher` for single-width characters.

```
final static CharMatcher DOUBLE_WIDTH = CharMatcher.singleWidth().negate();
```

Running the `NonAsciiFinder` application produces the following output:

```
$> java NonAsciiFinder "Maître Corbeau, sur un arbre perché"
Non-ASCII characters found: 2
î => \u00ee
é => \u00e9
```

```
$> java NonAsciiFinder "□□や□□び□む□の□"
NonASCII characters found: 11
□ => \u53e4
□ => \u6c60
や => \u3084
□ => \u86d9
□ => \u98db
び => \u3073
□ => \u8fbc
```

```
む => \u3080
 => \u6c34
の => \u306e
 => \u97f3
```

Removing unwanted characters from a string

The example Checking a string for unwanted characters, describes how to test and reject strings that don't meet certain criteria. Obviously, rejecting input outright is not always possible, and sometimes you just have to make do with what you receive. In these cases, a cautious developer will attempt to sanitize the provided strings to remove any characters that might trip up further processing.

To remove, trim, and replace unwanted characters, the weapon of choice will again be Guava's `CharMatcher` class.

Removing characters

The two `CharMatcher` methods of interest in this section are:

- `String retainFrom(CharSequence sequence)`
  Returns a string containing all the characters that matched the `CharMatcher` instance.
- `String removeFrom(CharSequence sequence)`
  Returns a string containing all the characters that did not match the `CharMatcher` instance.

As an example, we'll use `CharMatcher.digit()`, a predefined `CharMatcher` instance that, unsurprisingly, only matches digits.

```
String rock = "1, 2, 3 o'clock, 4 o'clock rock!";

CharMatcher.digit().retainFrom(rock); // "1234"
CharMatcher.digit().removeFrom(rock); // ", , o'clock, o'clock rock!"
CharMatcher.digit().negate().removeFrom(rock); // "1234"
```

The last line in this example illustrates that `removeFrom` is actually the inverse operation of `retainFrom`. Invoking `retainFrom` on a `CharMatcher` has the same effect as invoking `removeFrom` on a negated version of that `CharMatcher`.

Trimming leading and trailing characters

Removing leading and trailing characters is a very common operation, most frequently used to trim whitespace from strings. Guava's `CharMatcher` offers these trimming methods:

- `String trimLeadingFrom(CharSequence sequence)`
  Removes all leading characters that match the `CharMatcher` instance.

- `String trimTrailingFrom(CharSequence sequence)`
  Removes all trailing characters that match the `CharMatcher` instance.

- `String trimFrom(CharSequence sequence)`
  Removes all leading and trailing characters that match the `CharMatcher` instance.

When used with `CharMatcher.whitespace()`, these methods will effectively take care of all your whitespace trimming needs:

```
CharMatcher.whitespace().trimFrom("   Too much space   "); // returns "Too much space"
```

Replacing characters

Often, applications will replace characters that are not allowed in a certain situation with a placeholder character. To replace characters in a string, `CharMatcher`'s API provides the following methods:

- `String replaceFrom(CharSequence sequence, char replacement)`
  Replaces all occurrences of characters that match the `CharMatcher` instance with the provided replacement character.

- `String replaceFrom(CharSequence sequence, CharSequence replacement)` Replaces all occurrences of characters that match the `CharMatcher` instance with the provided replacement character sequence (string).

- `String collapseFrom(CharSequence sequence, char replacement)`
  Replaces groups of consecutive characters that match the `CharMatcher` instance with a single instance of the provided replacement character.

- `String trimAndCollapseFrom(CharSequence sequence, char replacement)`
  Behaves the same as `collapseFrom`, but matching groups at the start and the end are removed rather than replaced.

Let's look at an example that demonstrates how the behavior of these methods differs. Say that we're creating an application that lets the user specify output filenames. To sanitize the input provided by the user, we create a `CharMatcher` instance that is a combination of the predefined whitespace `CharMatcher` and a custom `CharMatcher` that specifies a set of characters that we would rather avoid in our filenames.

```
CharMatcher illegal = CharMatcher.whitespace().or(CharMatcher.anyOf("<>:|?*\"/\\"));
```

Now, if we invoke the discussed replacement methods as follows on a filename that is in dire need of cleanup:

```
String filename = "<A::12> first draft???";

System.out.println(illegal.replaceFrom(filename, '_'));
```

```
System.out.println(illegal.collapseFrom(filename, '_'));
System.out.println(illegal.trimAndCollapseFrom(filename, '_'));
```

We'll see the output below in our console.

```
_A__12___first_draft___
_A_12_first_draft_
A_12_first_draft
```

Splitting a string into a list

To split strings, Guava introduces the `Splitter` class.

Why not use Java's splitting capabilities?

As a rule, Guava does not duplicate functionality that is readily available in Java. Why then do we need an additional `Splitter` class? Do the split methods in Java's `String` class not provide us with all the string splitting mechanics we'll ever need?

The easiest way to answer that question is with a couple of examples. First off, we'll deal with the following gunslinging duo:

```
String gunslingers = "Wyatt Earp+Doc Holliday";
```

To try and split up the legendary lawman and his dentist friend, we might try the following:

```
String[] result = gunslingers.split("+"); // wrong
```

At runtime, however, we are confronted with the following exception:

```
Exception in thread "main" java.util.regex.PatternSyntaxException:
Dangling meta character '+' near index 0
```

After an involuntary facepalm, we're quick to remember that `String`'s split method takes a regular expression as an argument, and that the `+` character is used as a quantifier in regular expressions. The solution is then to escape the `+` character, or enclose it in a character class.

```
String[] result = gunslingers.split("\\+");
String[] result = gunslingers.split("[+]");
```

Having successfully resolved that issue, we move on to the three musketeers.

```
String musketeers = ",Porthos , Athos ,Aramis,";
```

The comma has no special meaning in regular expressions, so let's count the musketeers by applying the `String.split()` method and getting the length of the resulting array.

```
System.out.println(musketeers.split(",").length);
```

Which yields the following result in the console:

```
4
```

Four? Given the fact that the string contains a leading and a trailing comma, a result of five would have been within the realm of normal expectations, but four? As it turns out, the behavior of Java's `split` method is to preserve leading, but to discard trailing empty strings, so the actual contents of the array are `["", "Porthos ", " Athos ", "Aramis"]`.

Since we don't need any empty strings, leading nor trailing, let's filter them out with a loop:

```
for (String musketeer : musketeers.split(",")) {
    if (!musketeer.isEmpty()) {
        System.out.println(musketeer);
    }
}
```

This gives us the following output:

```
Porthos
 Athos
Aramis
```

As you can see in the output above, the extra spaces before and after the comma separators have been preserved in the output. To get around that, we can trim off the unneeded spaces, which will finally yield the desired output:

```
for (String musketeer : musketeers.split(",")) {
    if(!musketeer.isEmpty()) {
        System.out.println(musketeer.trim());
    }
}
```

(Alternatively, we could also adapt the regular expression to include whitespace surrounding the comma separators. However, keep in mind that leading spaces before the first entry or trailing spaces after the last entry would still be preserved.)

After reading through the examples above, we can't help but conclude that splitting strings with Java is mildly annoying at best.

Splitting strings with Guava

The best way to demonstrate how Guava turns splitting strings into a relatively painfree experience, is to treat the same two strings again, but this time using Guava's `Splitter` class.

```
List<String> gunslingers = Splitter.on('+')
```

```
        .splitToList("Wyatt Earp+Doc Holliday");
```

```
List<String> musketeers = Splitter.on(",")
        .omitEmptyStrings()
        .trimResults()
        .splitToList(",Porthos , Athos ,Aramis,");
```

As you can see in the code above, `Splitter` exposes a fluent API, and lets you create instances through a series of static factory methods:

- `static Splitter on(char separator)`
  Lets you specify the separator as a character.

- `static Splitter on(String separator)`
  Lets you specify the separator as a string.

- `static Splitter on(CharMatcher separatorMatcher)`
  Lets you specify the separator as a Guava `CharMatcher`.

- `static Splitter on(Pattern separatorPattern)`
  Lets you specify the separator as a Java regular expression `Pattern`.

- `static Splitter onPattern(String separatorPattern)`
  Lets you specify the separator as a regular expression string.

In addition to these separator-based factory methods, there's also a `static Splitter fixedLength(int length)` method to create `Splitter` instances that split strings into chunks of the specified length.

After the `Splitter` instance is created, a number of modifiers can be applied:

- `Splitter omitEmptyStrings()`
  Instructs the `Splitter` to exclude empty strings from the results.

- `Splitter trimResults()`
  Instructs the `Splitter` to trim results using the predefined whitespace `CharMatcher`.

- `Splitter trimResults(CharMatcher trimmer)`
  Instructs the `Splitter` to trim results using the specified `CharMatcher`.

After creating (and optionally modifying) a `Splitter`, it can be invoked on a character sequence by invoking its `split` method, which will return an object of type `Iterable<String>`, or its `splitToList` method, which will return an (immutable) object of type `List<String>`.

You might wonder in which cases it would be beneficial to use the `split` method (which returns an `Iterable`) instead of the `splitToList` method (which returns the more commonly used `List` type). The short answer to that is: you probably want to use the `split` method only for processing very large strings. The slightly longer answer is that because the `split` method returns an `Iterable`, the split operations can be lazily evaluated (at iteration time), thus removing the need to keep the entire result of the split operation in memory.

Read Strings online: https://riptutorial.com/guava/topic/4576/strings

Credits

S. No	Chapters	Contributors
1	Getting started with guava	Community, Daniel Käfer, jayantS, Omar Hrynkiewicz
2	I/O	Olivier Grégoire
3	Strings	jbduncan, Robby Cornelissen, Xaerxess