# String utilities

## Joiner

Joining together a sequence of strings with a separator can be unnecessarily tricky – but it shouldn't be. If your sequence contains nulls, it can be even harder. The fluent style of `Joiner` makes it simple.

```
Joiner joiner = Joiner.on("; ").skipNulls();
return joiner.join("Harry", null, "Ron", "Hermione");
```

returns the string "Harry; Ron; Hermione". Alternately, instead of using `skipNulls`, you may specify a string to use instead of null with `useForNull(String)`.

You may also use `Joiner` on objects, which will be converted using their `toString()` and then joined.

```
Joiner.on(",").join(Arrays.asList(1, 5, 7)); // returns "1,5,7"
```

**Warning:** joiner instances are always immutable. The joiner configuration methods will always return a new `Joiner`, which you must use to get the desired semantics. This makes any `Joiner` thread safe, and usable as a `static final` constant.

## Splitter

The built in Java utilities for splitting strings have some quirky behaviors. For example, `String.split` silently discards trailing separators, and `StringTokenizer` respects exactly five whitespace characters and nothing else.

Quiz: What does `",a„b,".split(",")` return?

1. `"", "a", "", "b", ""`
2. `null, "a", null, "b", null`
3. `"a", null, "b"`
4. `"a", "b"`
5. None of the above

The correct answer is none of the above: `"", "a", "", "b"`. Only trailing empty strings are skipped. What is this I don't even.

`Splitter` allows complete control over all this confusing behavior using a reassuringly straightforward fluent pattern.

```
Splitter.on(',')
    .trimResults()
    .omitEmptyStrings()
    .split("foo,bar,,   qux");
```

returns an `Iterable<String>` containing "foo", "bar", "qux". A `Splitter` may be set to split on any `Pattern`, `char`, `String`, or `CharMatcher`.

**Base Factories**

| Method | Description | Example |
|---|---|---|
| `Splitter.on(char)` | Split on occurrences of a specific, individual character. | `Splitter.on(';')` |
| `Splitter.on(CharMatcher)` | Split on occurrences of any character in some category. | `Splitter.on(CharMatcher.BREA` |
| `Splitter.on(String)` | Split on a literal `String`. | `Splitter.on(", ")` |
| `Splitter.on(Pattern)` `Splitter.onPattern(String)` | Split on a regular expression. | `Splitter.onPattern("\r?\n")` |
| `Splitter.fixedLength(int)` | Splits strings into substrings of the specified fixed length. The last piece can be smaller than `length`, but will never be empty. | `Splitter.fixedLength(3)` |

**Modifiers**

| Method | Description | Example |
|---|---|---|
| `omitEmptyStrings()` | Automatically omits empty strings from the result. | `Splitter.on(',').omitEmptyStr` re- turns `"a", "c", "d"` |
| `trimResults()` | Trims whitespace from the results; equivalent to `trimResults(CharMatcher.WHITESPACE)`. | `Splitter.on(',').trimResults(` `b, c, d")` re- turns `"a", "b", "c", "d"` |
| `trimResults(CharMatcher)` | Trims characters matching the specified `CharMatcher` from results. | `Splitter.on(',').trimResults(` `,_b_ ,c__")` re- turns `"a ", "b_ ", "c".` |

| Method | Description | Example |
|---|---|---|
| `limit(int)` | Stops splitting after the specified number of strings have been returned. | `Splitter.on(',').limit(3).spli` returns `"a", "b", "c,d"` |

If you wish to get a `List`, use `splitToList()` instead of `split()`.

**Warning:** splitter instances are always immutable. The splitter configuration methods will always return a new `Splitter`, which you must use to get the desired semantics. This makes any `Splitter` thread safe, and usable as a `static final` constant.

**Map Splitters** You can also use a splitter to deserialize a map by specifying a second delimiter using `withKeyValueSeparator()`. The resulting `MapSplitter` will split the input into entries using the splitter's delimiter, and then split those entries into keys and values using the given key-value separator, returning a `Map<String, String>`.

## CharMatcher

In olden times, our `StringUtil` class grew unchecked, and had many methods like these:

- `allAscii`
- `collapse`
- `collapseControlChars`
- `collapseWhitespace`
- `lastIndexNotOf`
- `numSharedChars`
- `removeChars`
- `removeCrLf`
- `retainAllChars`
- `strip`
- `stripAndCollapse`
- `stripNonDigits`

They represent a partial cross product of two notions:

1. what constitutes a "matching" character?
2. what to do with those "matching" characters?

To simplify this morass, we developed `CharMatcher`.

Intuitively, you can think of a `CharMatcher` as representing a particular class of characters, like digits or whitespace. Practically speaking, a `CharMatcher` is just a boolean predicate on characters – indeed, `CharMatcher` implements [`Predicate<Character>`] – but because it is so common to refer to "all whitespace characters" or "all lowercase letters," Guava provides this specialized syntax and API for characters.

But the utility of a `CharMatcher` is in the *operations* it lets you perform on occurrences of the specified class of characters: trimming, collapsing, removing, retaining, and much more. An object of type `CharMatcher` represents notion 1: what constitutes a matching character? It then provides many operations answering notion 2: what to do with those matching characters? The result is that API complexity increases linearly for quadratically increasing flexibility and power. Yay!

```
String noControl = CharMatcher.javaIsoControl().removeFrom(string); // remove control charac
String theDigits = CharMatcher.digit().retainFrom(string); // only the digits
String spaced = CharMatcher.whitespace().trimAndCollapseFrom(string, ' ');
  // trim whitespace at ends, and replace/collapse whitespace into single spaces
String noDigits = CharMatcher.javaDigit().replaceFrom(string, "*"); // star out all digits
String lowerAndDigit = CharMatcher.javaDigit().or(CharMatcher.javaLowerCase()).retainFrom(st
  // eliminate all characters that aren't digits or lowercase
```

**Note:** `CharMatcher` deals only with `char` values; it does not understand supplementary Unicode code points in the range 0x10000 to 0x10FFFF. Such logical characters are encoded into a `String` using surrogate pairs, and a `CharMatcher` treats these just as two separate characters.

### Obtaining CharMatchers

Many needs can be satisfied by the provided `CharMatcher` factory methods:

- `any()`
- `none()`
- `whitespace()`
- `breakingWhitespace()`
- `invisible()`
- `digit()`
- `javaLetter()`
- `javaDigit()`
- `javaLetterOrDigit()`
- `javaIsoControl()`
- `javaLowerCase()`
- `javaUpperCase()`
- `ascii()`
- `singleWidth()`

Other common ways to obtain a `CharMatcher` include:

| Method | Description |
|---|---|
| `anyOf(CharSequence)` | Specify all the characters you wish matched. For example, `CharMatcher.anyOf("aeiou")` matches lowercase English vowels. |
| `is(char)` | Specify exactly one character to match. |
| `inRange(char, char)` | Specify a range of characters to match, e.g. `CharMatcher.inRange('a', 'z')`. |

Additionally, `CharMatcher` has `negate()`, `and(CharMatcher)`, and `or(CharMatcher)`. These provide simple boolean operations on `CharMatcher`.

### Using CharMatchers

`CharMatcher` provides a wide variety of methods to operate on occurrences of the specified characters in any `CharSequence`. There are more methods provided than we can list here, but some of the most commonly used are:

| Method | Description |
|---|---|
| `collapseFrom(CharSequence, char)` | Replace each group of consecutive matched characters with the specified character. For example, `WHITESPACE.collapseFrom(string, ' ')` collapses whitespaces down to a single space. |

| Method | Description |
|---|---|
| `matchesAllOf(CharSequence)` | Test if this matcher matches all characters in the sequence. For example, `ASCII.matchesAllOf(string)` tests if all characters in the string are ASCII. |
| `removeFrom(CharSequence)` | Removes matching characters from the sequence. |
| `retainFrom(CharSequence)` | Removes all non-matching characters from the sequence. |
| `trimFrom(CharSequence)` | Removes leading and trailing matching characters. |
| `replaceFrom(CharSequence, CharSequence)` | Replace matching characters with a given sequence. |

(Note: all of these methods return a `String`, except for `matchesAllOf`, which returns a `boolean`.)

## Charsets

Don't do this:

```
try {
  bytes = string.getBytes("UTF-8");
} catch (UnsupportedEncodingException e) {
  // how can this possibly happen?
  throw new AssertionError(e);
}
```

Do this instead:

```
bytes = string.getBytes(Charsets.UTF_8);
```

`Charsets` provides constant references to the six standard `Charset` implementations guaranteed to be supported by all Java platform implementations. Use them instead of referring to charsets by their names.

TODO: an explanation of charsets and when to use them

(Note: If you're using JDK7, you should use the constants in `StandardCharsets`

## CaseFormat

`CaseFormat` is a handy little class for converting between ASCII case conventions — like, for example, naming conventions for programming languages. Supported formats include:

| Format | Example |
|---|---|
| `LOWER_CAMEL` | `lowerCamel` |
| `LOWER_HYPHEN` | `lower-hyphen` |
| `LOWER_UNDERSCORE` | `lower_underscore` |
| `UPPER_CAMEL` | `UpperCamel` |
| `UPPER_UNDERSCORE` | `UPPER_UNDERSCORE` |

Using it is relatively straightforward:

```
CaseFormat.UPPER_UNDERSCORE.to(CaseFormat.LOWER_CAMEL, "CONSTANT_NAME")); // returns "const
```

We find this especially useful, for example, when writing programs that generate other programs.

## Strings

A limited number of general-purpose `String` utilities reside in the `Strings` class.