Guava

a quick and dirty overview

of just some of the highlights

2009-09-15

What libraries we talkin bout?

Google's Java *Core* Libraries include these (and more), all under **com.google.common**:

- **base**
  - the "corest of the core"
- **primitives**
  - working with int, long, double, byte, . . .
- **collect**
  - collection types, implementations, algorithms
- **io**
  - streams, buffers, files, . . .
- **net**
  - URIs, domain names, IP addrs, user agents, . . . (not yet released)
- **util**
  - Higher-level classes that may build on the above

Platforms

You can use these libraries on...

- On JDK 6
- On Android
  - We think. Need volunteers to help us test it out.
- On Google App Engine
  - We think. Need volunteers to help us test it out.
- On GWT -- spotty!
  - Because GWT's JDK library support is spotty
  - Maybe 2/3 the stuff in these libraries is GWT-safe so far
  - Just look for **@GwtCompatible**!
  - Performance? Not necessarily good.

Charsets

This one's easy! Don't do this:

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

Do this:

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

Our **common.base.Charsets** class contains six predefined **java.nio.Charset** constants.

Charsets 2

These predefined charsets are the six that are guaranteed to be supported on **all** JDK platforms.

From now on, references in your code to **`UnsupportedEncodingException`** or the literal string **`"UTF-8"`** are probable code smells!

CharMatcher

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

allAscii, collapse, collapseControlChars, collapseWhitespace, indexOfChars, lastIndexNotOf, numSharedChars, removeChars, removeCrLf, replaceChars, retainAllChars, strip, stripAndCollapse, stripNonDigits, ...

They represent a partial cross product of two notions:

(a) what constitutes a "matching" character?
(b) what to *do* with those "matching" characters?

This approach could not scale, so we created **CharMatcher**.

An instance of this type represents part (a), and the operation you invoke on it represents part (b). So you now get the *full* cross product, with minimal API surface area.

Getting an instance of CharMatcher

- Use a predefined constant (examples)
  - **CharMatcher.WHITESPACE**
  - **CharMatcher.JAVA_DIGIT**
  - **CharMatcher.ASCII**
  - **CharMatcher.ANY**
- Use a factory method (examples)
  - **CharMatcher.is('x')**
  - **CharMatcher.isNot('_')**
  - **CharMatcher.oneOf("aeiou").negate()**
  - **CharMatcher.inRange('a', 'z').or(inRange('A', 'Z'))**
- Or subclass CharMatcher and implement **matches(char c)**

Now check out all that you can do . . .

Using your new CharMatcher

- boolean matchesAllOf(CharSequence)
- boolean matchesAnyOf(CharSequence)
- boolean matchesNoneOf(CharSequence)
- int indexIn(CharSequence, int)
- int lastIndexIn(CharSequence, int)
- int countIn(CharSequence)
- String removeFrom(CharSequence)
- String retainFrom(CharSequence)
- String trimFrom(CharSequence)
- String trimLeadingFrom(CharSequence)
- String trimTrailingFrom(CharSequence)
- String collapseFrom(CharSequence, char)
- String trimAndCollapseFrom(CharSequence, char)
- String replaceFrom(CharSequence, char)

Putting it together

To scrub an id number, you might use

```
String id = CharMatcher.DIGIT.or(CharMatcher.is('-'))
    .retainFrom(input);
```

If inside a loop, move your CharMatcher definition outside the loop, or to a private class constant.

```
private static final CharMatcher ID_CHARS =
    CharMatcher.DIGIT.or(CharMatcher.is('-'));

. . .

String id = ID_CHARS.retainFrom(input);
```

Joiner

From the *Top 3 Most Bizarrely Missing From The JDK Class Libraries* list: joining pieces of text with a separator.

```
String s = Joiner.on(", ").join(numbers);
```

Joiner is configurable:

```
StringBuilder sb = ...;
Joiner.on("|").skipNulls().appendTo(sb, attrs);
```

It can even handle maps:

```
static final MapJoiner MAP_JOINER = Joiner.on("; ")
    .useForNull("NODATA")
    .withKeyValueSeparator(":");
```

Joiner (2)

Like **CharEscaper** and **CharMatcher**, **Joiner** separates the configuration phase from the actual processing of data.

Splitters!

Google™

Splitter

Breaks strings into substrings

- by recognizing a separator (delimiter), which could be
  - a single character: **Splitter.on('\n')**
  - a literal string: **Splitter.on(", ")**
  - a regex: **Splitter.onPattern(",\\s*")**
  - any **CharMatcher** (hey, remember that?)
- or using a fixed substring length
  - **Splitter.fixedLength(8)**

```
Iterable<String> pieces =
    Splitter.on(',').split("trivial,example")
```

returns an iterable containing **"trivial"** and **"example"** in order.

Google™

But the JDK does have splitting!

JDK has this:

```
String[] pieces = "foo.bar".split("\\.");
```

It's fine to use this if you want exactly what it does:

- regular expression
- result as an array
- its way of handling empty pieces

Mini-puzzler: `",a,,b,".split(",")` returns...

(a) **"", "a", "", "b", ""**
(b) **null, "a", null, "b", null**
(c) **"a", null, "b"**
(d) **"a", "b"**
(e) None of the above

But the JDK does have splitting! (2)

Answer: (e) None of the above.

```
",a,,b,".split(",")
```

returns

```
"", "a", "", "b"
```

Only trailing empties are skipped! (Who knows the workaround to prevent the skipping? It's a fun one...)

In any case, our **Splitter** is simply more flexible, as we're about to see . . .

Splitter: more interesting examples

The default behavior is simplistic:

```
Splitter.on(',').split(" foo, ,bar,  quux,")
 --> [" foo", " ", "bar", "  quux", ""]
```

If you want extra features, ask for them!

```
Splitter.on(',')
    .trimResults()
    .omitEmptyStrings()
    .split(" foo, ,bar,  quux,")
 --> ["foo", "bar", "quux"]
```

Order of config methods doesn't matter -- during splitting, trimming happens before checking for empties.

Dude, where's my *brevity?*

"Brevity is the soul of wit."

But is wit/cleverness the right goal to have for the code you write?

Good code is not witty. It's prosaic, possibly dull. But it's easily understandable to its readers.

Google™

common.primitives

common.primitives is a new package that helps you work with the primitive types: **int**, **long**, **double**, **float**, **char**, **byte**, **short**, and **boolean.**

If you need help doing a primitive task:

1. check the wrapper class (e.g. **java.lang.Integer**)
2. check **java.util.Arrays**
3. check **com.google.common.primitives**
4. it might not exist!

common.primitives (2)

common.primitives contains the classes **Booleans**, **Bytes**, **Chars**, **Doubles**, **Floats**, **Ints**, **Longs** and **Shorts**. Each has the exact same structure (but has only the subset of operations that make sense for its type).

Many of the **byte**-related methods have alternate versions in the classes **SignedBytes** and **UnsignedBytes**. (Bytes are peculiar; always signed, but very often you wish to *treat* them as unsigned.)

(There are a few more general utilities in a class called **Primitives**.)

common.primitives: The Table

Method	Longs	Ints	Shorts	Chars	Doubles	Bytes	S.Bytes	U.Bytes	Booleans
hashCode	X	X	X	X	X	X			X
compare	X	X	X	X	X		X	X	X
checkedCast		X	X	X			X	X	
saturatedCast		X	X	X			X	X	
contains	X	X	X	X	X	X			
indexOf	X	X	X	X	X	X			X
lastIndexOf	X	X	X	X	X	X			X
min	X	X	X	X	X		X	X	
max	X	X	X	X	X		X	X	
concat	X	X	X	X	X	X			X
join	X	X	X	X	X		X	X	X
toArray	X	X	X	X	X	X			X
asList	X	X	X	X	X	X			X
toByteArray	X	X	X	X					
fromByteArray	X	X	X	X					

common.primitives: asList

The most complex method in here is **asList**:

```
int[] nums = ...;
List<Integer> list = Ints.asList(nums);
```

You are viewing the array directly as a List. All methods that don't structurally modify the list are supported.

We've done it as efficiently as we can, but be careful: you can easily get killed by boxing/unboxing.

For bona fide primitive collections, use the third-party library *fastutil* or *trove4j*.

common.primitives: to/fromByteArray

```
Ints.fromByteArray({0x01, 0x02, 0x03, 0x04})
 --> 0x01020304
Ints.toByteArray(0x01020304)
 --> {0x01, 0x02, 0x03, 0x04}
```

Sometimes useful, but if you need little-endian, or have multiple conversions to do at once, use **java.nio.ByteBuffer**!

```
// to array                // from array
ByteBuffer.allocate(8)  ByteBuffer b = ByteBuffer.wrap(a);
   .putInt(id)          id = b.readLong();
   .putInt(value)       value = b.readInt();
   .array();
```

Or for a dynamically-sized array, use **common.io.ByteStreams.newDataOutput()**.

Google™

common.io

If what you need pertains to streams, buffers, files and the like, look to our package **com.google.common.io**.

Key interfaces:

```
public interface InputSupplier<T> {
  T getInput() throws IOException;
}
public interface OutputSupplier<T> {
  T getOutput() throws IOException;
}
```

Typically: **InputSupplier<InputStream>**, **OutputSupplier<Writer>**, etc. This lets our utilities be useful for various kinds of I/O, such as GFS.

Google™

common.io: Streams

Our terms:

- **byte** stream
  - means "**InputStream** or **OutputStream**"
- **char** stream
  - means "**Reader** or **Writer**."

Utilities for these things are in the classes **ByteStreams** and **CharStreams** (which have largely parallel structure).

common.io: ByteStreams

- **byte[] toByteArray(InputStream)**
- **byte[] toByteArray(InputSupplier)**
- **void readFully(InputStream, byte[])**
- **void write(byte[], OutputSupplier)**
- **long copy(InputStream, OutputStream)**
- **long copy(InputSupplier, OutputSupplier)**
- **long length(InputSupplier)**
- **boolean equal(InputSupplier, InputSupplier)**
- **InputSupplier slice(InputSupplier, long, long)**
- **InputSupplier join(InputSupplier...)**

CharStreams is similar, but deals in **Reader**, **Writer**, **String** and **CharSequence** (often requiring you to specify a **Charset**).

common.io: Files

The **Files** class builds on top of **ByteStreams** and **CharStreams** and has a few other tricks.

- **byte[] toByteArray(File)**
- **String toString(File, Charset)**
- **void write(byte[], File)**
- **void write(CharSequence, File, Charset)**
- **long copy(File, File)**
- **long copy(InputSupplier, File)**
- **long copy(File, OutputSupplier)**
- **long copy(File, Charset, Appendable)**
- **long move(File, File)**
- **boolean equal(File, File)**
- **List<String> readLines(File, Charset)**

common.io: Files.toByteArray

Pre-Java 7, this stuff is an incredible pain without a library!

```
public static byte[] toByteArray(File file) throws IOException {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  boolean threw = true;
  InputStream in = new FileInputStream(file);
  try {
    byte[] buf = new byte[BUF_SIZE];
    long total = 0;
    while (true) {
      int r = in.read(buf);
      if (r == -1) break;
      out.write(buf, 0, r);
    }
    threw = false;
  } finally {
    try {
      in.close();
    } catch (IOException e) {
      if (threw) {
        warn("IOException thrown while closing", e);
      } else { throw e; }
    }
  }
  return out.toByteArray();
}
```

common.io: LineProcessor example

Count all the lines of a UTF-8 encoded file:

```
int count = Files.readLines(
    new File("/path/to/file"),
    Charsets.UTF_8, // remember that? :)
    new LineProcessor<Integer>() {
      int count = 0;
      public boolean processLine(String line) {
        count++;
      }
      public Integer getResult() { return count; }
    });
```

Callback-based APIs allow us to deal with all the mess for you!

Thanks

I (kevinb) made some of this stuff, but had a lot of help, from people including

jlevy cpovirk chrisn mbostock killianey jessewilson fry benyu hhchan konigsberg nnk eh harryh georgevdd crazybob cquinn cnwan elharo ziff rocketman matevossian dbeaumont tobe sclark mgp . . . **and 231 others**

(that is only by raw change count, a very coarse and flawed measure.)

Google™