Guava

Google's Core Libraries for Java

Kevin Bourrillion, Google Inc.

as presented at Netflix 2010-04-26

Overview

Guava: Google's core Java libraries for Java 5+.

This presentation: broad overview, partial highlight reel, and lots of questions?

Presenter (me):

- At Google >5 years
- Lead engineer, Java core libraries >3 years

Overview

Guava: Google's core Java libraries for Java 5+.

This presentation: broad overview, partial highlight reel, and lots of questions?

Presenter (me):

- At Google >5 years
- Lead engineer, Java core libraries >3 years
- Devoted Netflix subscriber >9 years!
    - your company changed my life
        - *I OWE YOU GUYS*

Overview (of library)

http://guava-libraries.googlecode.com

Apache 2 license (very permissive).
Frequent releases ("r03" a few weeks ago, "r04" this week).

Under com.google.common:
base, collect, io, net*, primitives, util.concurrent

Er, what about the "Google Collections Library?"
(most of collect, some of base)

We want you to use Guava!

"I could just write that myself." But...

- These things are *much* easier to mess up than it seems
- With a library, other people will make your code faster for you
- When you use a popular library, your code is in the mainstream
- When you find an improvement to your private library, how many people did you help?

Well argued in *Effective Java 2e*, Item 47.

1. com.google.common.base

"The corest of the core."

"java.langy" stuff.

The Objects class

```
public class Person {
  final String name, nickname;
  final Movie favMovie;

  @Override public boolean equals(Object object) {
    if (object instanceof Person) {
      Person that = (Person) object;
      return Objects.equal(this.name, that.name)
          && Objects.equal(this.nickname, that.nickname)
          && Objects.equal(this.favMovie, that.favMovie);
    }
    return false;
  }

  @Override public int hashCode() {
    return Objects.hashCode(name, nickname, favMovie);
  }
```

Objects example cont.

```
public class Person {
  final String name, nickname;
  final Movie favMovie;
  // ...

  @Override public String toString() {
    return Objects.toStringHelper(this)
        .add("name", name)
        .add("nickname", nickname)
        .add("favMovie", favMovie)
        .toString();
  }

  public String preferredName() {
    return Objects.firstNonNull(nickname, name);
  }
}
```

Preconditions

Our class **com.google.common.base.Preconditions** supports defensive coding. You can choose either

```
if (state != State.PLAYABLE) {
  throw new IllegalStateException(
      "Can't play movie; state is " + state);
}
```

. . . or . . .

```
Preconditions.checkState(state == State.PLAYABLE,
    "Can't play movie; state is %s", state);
```

(*what's the difference? none!*)

Preconditions (2)

Or compare . . .

```
public void setRating(StarRating rating) {
  if (rating == null) {
    throw new NullPointerException();
  }
  this.rating = rating;
}
```

. . . with (using static import) . . .

```
public void setRating(StarRating rating) {
  this.rating = checkNotNull(rating);
}
```

CharMatcher

We once had a **StringUtil** class. It grew large:

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

These represent a partial cross product of two notions:

(a) what's 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).

Getting a CharMatcher

- Use a predefined constant (examples)
  - **CharMatcher.WHITESPACE** (tracks Unicode defn.)
  - **CharMatcher.JAVA_DIGIT**
  - **CharMatcher.ASCII**
  - **CharMatcher.ANY**

Getting a CharMatcher

- Use a predefined constant (examples)
  - **CharMatcher.WHITESPACE** (tracks Unicode defn.)
  - **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'))**

Getting a CharMatcher

- Use a predefined constant (examples)
  - **CharMatcher.WHITESPACE** (tracks Unicode defn.)
  - **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'))**
- Subclass CharMatcher, 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)**

(Sure, there's overlap between this and regex.)

Putting it together

To scrub an id number, you might use

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

Putting it together

To scrub an id number, you might use

```
String seriesId =
    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 SERIES_ID_CHARS =
    CharMatcher.DIGIT.or(CharMatcher.is('-'));

 . . .

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

Joiner

Bizarrely Missing From The JDK Class Libraries:
joining pieces of text with a separator.

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

Joiner is configurable:

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

Joiner

Bizarrely Missing From The JDK Class Libraries:
joining pieces of text with a separator.

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

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(":");
```

Splitters!

Splitter

Breaks strings into substrings

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

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

returns "trivial" and "example" in order.

But the JDK does have splitting!

JDK has this:

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

It's convenient to use this... *if* you want exactly what it does:

- regular expression
- result as an array
- its way of handling empty pieces
    - *which is very strange*

Our **Splitter** is very flexible (next slide...)

Splitter: more examples

The default behavior is simplistic:

```
// yields [" foo", " ", "bar", "  quux", ""]
Splitter.on(',').split(" foo, ,bar,  quux,")
```

If you want extra features, ask for them!

Splitter: more examples

The default behavior is simplistic:

```
  // yields [" foo", " ", "bar", "  quux", ""]
  Splitter.on(',').split(" foo, ,bar,  quux,")
```

If you want extra features, ask for them!

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

Order of config methods doesn't matter.

2. com.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 (wait for it) **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...)

We don't do primitive-based collections; try fastutil, or trove4j, or . . .

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
lexComparator	X	X	X	X	X		X	X	X
toByteArray	X	X	X	X					
fromByteArray	X	X	X	X					

3. com.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 all our utilities be useful for many kinds of I/O.

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 works one level higher than **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: the future?

JDK 7 has a proper abstract filesystem API, and ARM syntax.

You won't need most of our `common.io` anymore then!

4. com.google.common.collect

It would take an entire presentation to tell you about this package... (and it did!)

- Immutable Collections
- Multimaps, Multisets, BiMaps
- Comparator-related utilities
- Forwarding collections, Constrained collections
- Some functional programming support (filter/transform/etc.)

Just search google collections video in your favorite search engine.

One highlight: MapMaker

MapMaker is the jewel of common.collect.

```
ConcurrentMap<User, RecGraph> recommendations =
  new MapMaker()
    .weakKeys()
    .expiration(10, TimeUnit.MINUTES)
    .makeComputingMap(
      new Function<User, RecGraph>() {
        public RecGraph apply(User user) {
          return createExpensiveGraph(user);
        }
      });
```

It merits another entire presentation of its own.

5. com.google.common.util.concurrent

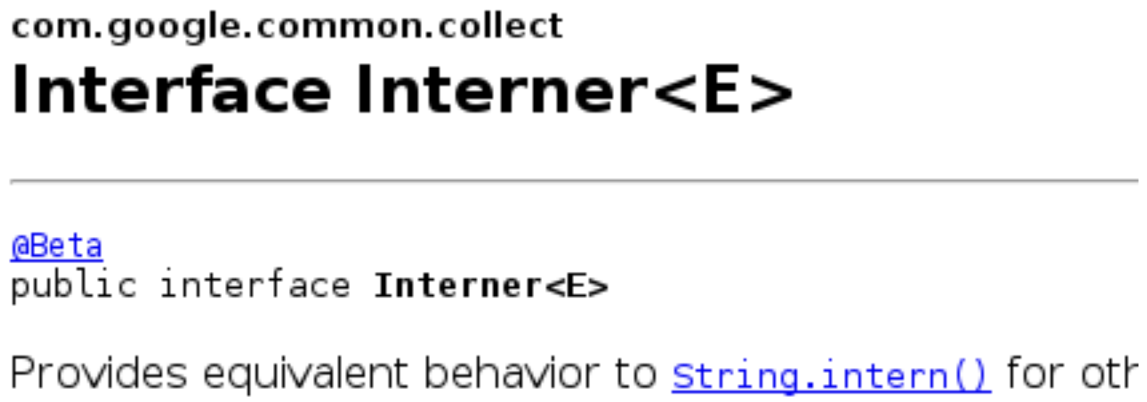
Spend the time to get deeply familiar with java.util.concurrent first!

Then come check out:

Callables, Futures, CheckedFuture, ListenableFuture, UninterruptibleFuture, Service, MoreExecutors, ThreadFactoryBuilder, TimeLimiter, ...

Caveat 1

Libraries marked @Beta are subject to change at any time!



com.google.common.collect

Interface Interner<E>

```
@Beta
public interface Interner<E>
```

Provides equivalent behavior to `String.intern()` for oth

For the rest, we intend to maintain compatibility (modulo deprecation window).

Nothing that was in Google Collections 1.0 is @Beta.

Caveat 2

Serialization compatibility not guaranteed.

Don't assume persisted serialized data can be deserialized in future version of the library.

(Consider not even *using* serialization if you can avoid it!)

What to do now?

- Download it, see online javadocs, etc.
  - http://guava-libraries.googlecode.com
- Watch Collections presentation
  - http://www.youtube.com/watch?v=ZeO_J2OcHYM
  - (or search "google collections video")
- Join discussion list
  - http://groups.google.com/group/guava-discuss
- Ask for help
  - post with "guava" tag to StackOverflow.com

Q & A