# Google Guava Collections

… and a little bit more

# Google Guava … only a small part of it.

## User Guide

The Guava project contains several of Google's core libraries that we rely on in our Java-based projects: collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. Each of these tools really do get used every day by Googlers, in production services.

But trawling through Javadoc isn't always the most effective way to learn how to make best use of a library. Here, we try to provide readable and pleasant explanations of some of the most popular and most powerful features of Guava.

*This wiki is a work in progress, and parts of it may still be under construction.*

- Basic utilities: Make using the Java language more pleasant.
  - Using and avoiding null: `null` can be ambiguous, can cause confusing errors, and is sometimes just plain unpleasant. Many Guava utilities reject and fail fast on nulls, rather than accepting them blindly.
  - Preconditions: Test preconditions for your methods more easily.
  - Common object methods: Simplify implementing `Object` methods, like `hashCode()` and `toString()`.
  - Ordering: Guava's powerful "fluent `Comparator`" class.
  - Throwables: Simplify propagating and examining exceptions and errors.
- Collections: Guava's extensions to the JDK collections ecosystem. These are some of the most mature and popular parts of Guava.
  - Immutable collections, for defensive programming, constant collections, and improved efficiency.
  - New collection types, for use cases that the JDK collections don't address as well as they could: multisets, multimaps, tables, bidirectional maps, and more.
  - Powerful collection utilities, for common operations not provided in `java.util.Collections`.
  - Extension utilities: writing a `Collection` decorator? Implementing `Iterator`? We can make that easier.
- Caches: Local caching, done right, and supporting a wide variety of expiration behaviors.
- Functional idioms: Used sparingly, Guava's functional idioms can significantly simplify code.
- Concurrency: Powerful, simple abstractions to make it easier to write correct concurrent code.
  - ListenableFuture: Futures, with callbacks when they are finished.
  - Service: Things that start up and shut down, taking care of the difficult state logic for you.
- Strings: A few extremely useful string utilities: splitting, joining, padding, and more.
- Primitives: operations on primitive types, like `int` and `char`, not provided by the JDK, including unsigned variants for some types.
- Ranges: Guava's powerful API for dealing with ranges on `Comparable` types, both continuous and discrete.
- I/O: Simplified I/O operations, especially on whole I/O streams and files, for Java 5 and 6.
- Hashing: Tools for more sophisticated hashes than what's provided by `Object.hashCode()`, including Bloom filters.
- EventBus: Publish-subscribe-style communication between components without requiring the components to explicitly register with one another.
- Math: Optimized, thoroughly tested math utilities not provided by the JDK.
- Reflection: Guava utilities for Java's reflective capabilities.
- Tips: Getting your application working the way you want it to with Guava.
  - Philosophy: what Guava is and isn't, and our goals.
  - Using Guava in your build, with build systems including Maven, Gradle, and more.
  - Using ProGuard to avoid bundling parts of Guava you don't use with your JAR.
  - Apache Commons equivalents, helping you translate code from using Apache Commons Collections.
  - Compatibility, details between Guava versions.
  - Idea Graveyard, feature requests that have been conclusively rejected.
  - Friends, open-source projects we like and admire.

**NOTE:** To discuss the contents of this wiki, please just use the guava-discuss mailing list.

https://code.google.com/p/guava-libraries/wiki/GuavaExplained

## Optional

… to express that result is optional.

… to avoid null and null checks.

# Optional

```
assertFalse(findUser(12).isPresent());
assertEquals("foo",emailForUser(12).or("foo"));

loadExistingUser(12); -> IllegalStateException

static Optional<User> findUser(int userId) {
  return Optional.absent();
}

static Optional<String> emailForUser(int userId) {
  return findUser(userId).transform(new Function<User, String>() {
      @Override
      public String apply(User user) {
        return user.name;
      }
  });
}

static User loadExistingUser(int userId) {
  return findUser(userId).get();
}
```

# Extensions - noneOrOne

```
Optional<String> result = Expectations.noneOrOne(Lists.<String>newArrayList());
assertFalse(result.isPresent());
result = Expectations.noneOrOne(Lists.newArrayList("foo"));
assertTrue(result.isPresent());

Expectations.noneOrOne(Lists.newArrayList("foo","bar")); -> IllegalArgumentException
```

## Preconditions

.. fail as soon as possible.

# Preconditions - Example

```
static <T> T elementOf(Index index,
List<T> list) {
    return list.get(index.value());
}
interface Index {
    int value();
}
```

```
class IndexWrapper implements Index {
  private final int value;

  public IndexWrapper(int value) {
    this.value = value;
  }

  @Override
  public int value() {
    return value;
  }
}
```

```
class IndexChecked implements Index {
  private final int value;

  public IndexChecked(int value) {
    Preconditions.checkArgument(value >= 0, "value < 0");
    this.value = value;
  }
  public int value() {
    return value;
  }
}
```

# Preconditions - Example

```
IndexWrapper index = new IndexWrapper(-1);
elementOf(index, Lists.newArrayList("foo"));
```

```
java.lang.ArrayIndexOutOfBoundsException: -1
  at java.util.ArrayList.elementData(ArrayList.java:400)
  at java.util.ArrayList.get(ArrayList.java:413)
  at de.flapdoodle.guava.talk.jan2014.preconditions.TestPreconditions.elementOf(TestPreconditio  java:36)
  at de.flapdoodle.guava.talk.jan2014.preconditions.TestPreconditions.withoutPreconditions(Test  conditions.java:16)
```

```
IndexChecked index = new IndexChecked(-1);
elementOf(index, Lists.newArrayList("foo"));
```

```
java.lang.IllegalArgumentException: value < 0
  at com.google.common.base.Preconditions.checkArgument(Preconditions.java:93)
  at de.flapdoodle.guava.talk.jan2014.preconditions.TestPreconditions$IndexChecked.<init>(TestPreconditions.java:63)
  at de.flapdoodle.guava.talk.jan2014.preconditions.TestPreconditions.withPreconditions(TestPreconditions.java:26)
```

# Preconditions

- fails on construction not on usage
- stacktrace show cause not effect
- mix well with immutables
- checked only once vs. many times
- …

# Immutable[List,Set,Map,..]

- why?
- Immutable[List] vs Collections.unmodifiable [List]
- List<X> vs ImmutableList<X> as returnType.
- bigger Picture (return value, constructor args)
- Builder

# Immutable[List,Set,Map,..] - why?

```
static List<String> badBeginsWith(List<String> list, char c) {
  final Iterator<String> iterator = list.iterator();
  while (iterator.hasNext()) {
    String val=iterator.next();
    if (val.charAt(0)!=c) {
      iterator.remove();
    }
  }
  return list;
}
```

```
List<String> source=Lists.newArrayList("foo","bar","boo");

List<String> result=badBeginsWith(source, 'b');
assertEquals("[bar, boo]",result.toString());
assertEquals("NOT expected","[bar, boo]",source.toString());

source=ImmutableList.of("foo","bar","boo");
badBeginsWith(source, 'b'); -> UnsupportedOperationException
```

# Immutable vs. Unmodifyable

```
List<String> source=Lists.newArrayList("a","b","c");

StringList readOnlyWrapper=new StringList(Collections.unmodifiableList(source));
StringList readOnlyCopy=new StringList(ImmutableList.copyOf(source));

assertEquals("a",readOnlyWrapper.getFirst());
assertEquals("a",readOnlyCopy.getFirst());

source.remove(0);
assertEquals("not expected", "b",readOnlyWrapper.getFirst());
assertEquals("a",readOnlyCopy.getFirst());

static class StringList {
  private final List<String> data;
  public StringList(List<String> data) {
    this.data = data;
  }
  public String getFirst() {
    return data.get(0);
  }
}
```

# Immutable[List,Set,Map,..]

You should copy all incoming collections if you keep them.

You may return an unmodifyable wrapper, but you could simply return your immutable copy.

# Immutable[X] - as return type

```
List<String> list;
list=classicWay();
list=alternativeWay();

ImmutableList<String> immutable;
// immutable=classicWay(); --> compile error
immutable=alternativeWay();

// list.remove(0); --> looks ok
// immutable.remove(0); --> marked as deprecated, name suggest: dont use it

static List<String> classicWay() {
  return Lists.newArrayList("foo","bar");
}

static ImmutableList<String> alternativeWay() {
  return ImmutableList.of("foo","bar");
}
```

# Immutable[X] - bigger Picture

```
Path path = new Path(Lists.newArrayList("foo","bar"));
Path copy = new Path(path.parts()); // share same list instance

static class Path {
  final ImmutableList<String> parts;
  public Path(ImmutableList<String> parts) {
    this.parts = parts; // use without copy instance
  }

  public Path(Collection<String> parts) {
    this(ImmutableList.copyOf(parts)); // make a copy
  }

  public ImmutableList<String> parts() {
    return parts;
  }
}
```

# Immutable[X] - Builder

```
ImmutableList<String> list = ImmutableList.<String>builder()
                                          .add("foo")
                                          .add("bla","bar","boo")
                                          .addAll(Lists.newArrayList("ying","yang"))
                                          .build();

assertEquals("[foo, bla, bar, boo, ying, yang]", list.toString());
```

# Immutable[List,Set,Map,..]

… there is a lot more.

# Transformations

- convert type container (dto -> dto)
- change data to new type (String to Int,...)
- combine data
- ...

https://github.com/flapdoodle-oss/de.flapdoodle.guava

# Transformations - the classic way

```
List<Integer> numbers = Lists.newArrayList(1, 2, 3, 4);
List<String> result = numbersAsDaysClassic(numbers);
assertEquals("[1 Tag, 2 Tage, 3 Tage, 4 Tage]", result.toString());

private static List<String> numbersAsDaysClassic(List<Integer> numbers) {
  List<String> ret=new ArrayList<String>();
  for (Integer number : numbers) {
    ret.add(numberAsDay(number));
  }
  return ret;
}

private static String numberAsDay(int input) {
 if (input == 1) return input + " Tag";
 return input + " Tage";
}
```

# Transformations

- cases to test:
  - empty list
  - list with one element
    - 1
    - >1
  - list with more elements
    - first 1, other >1
    - first >1, other 1
- or test numberAsDay and one test for numbersAsDays

# Transformations - alternative

```
private static List<String> numbersAsDays(List<Integer> numbers) {
  return Lists.transform(numbers, new Function<Integer, String>() {
    @Override
    public String apply(Integer input) {
      return numberAsDay(input);
    }
  });
}
```

- “Don’t know what transform does.”
- anon type

# Transformations - alternative

```
private static List<String> numbersAsDaysRefactored(List<Integer> numbers) {
  return Lists.transform(numbers, new NumberAsDay());
}
private static class NumberAsDay implements Function<Integer, String> {
  @Override
  public String apply(Integer input) {
    return numberAsDay(input);
  }
}
```

- “Don’t know what transform does.”
  - then find out… -> for each entry in list apply function and put result into new list
- transformation function now with more meaningful name

# Transformations

but there is more...

# Extensions

map, foldLeft, etc.

# Extensions - map

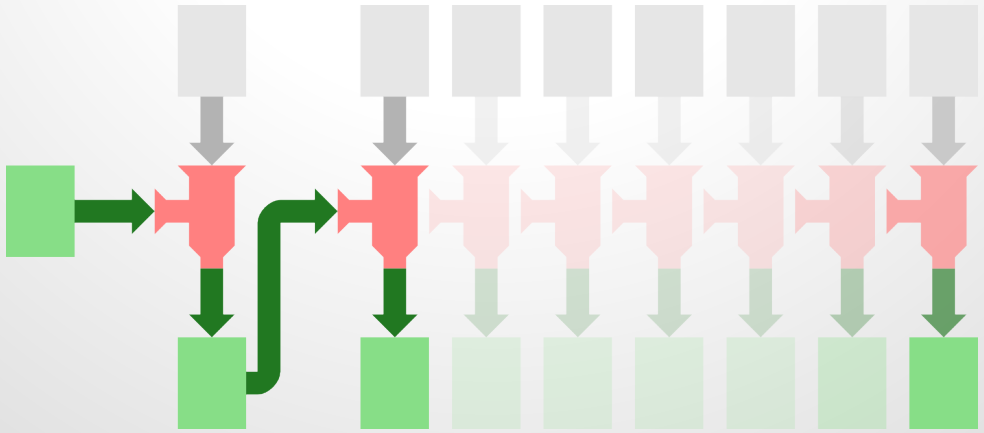
```
List<Order> orders = Lists.newArrayList(new Order(1), new Order(2), new Order(4));

Function<Order, Integer> keytransformation = new Function<Order, Integer>() {
  @Override
  public Integer apply(Order order) {
    return order.getId();
  }
};

Map<Integer, Order> orderMap = Transformations.map(orders, keytransformation);

Order order = orderMap.get(2);
assertNotNull(order);
```

# Extensions - map

```
List<User> users = Lists.newArrayList(new User(1,"klaus"), new User(2,"susi"), new User(4,"klaus"));

Function<User, Integer> keytransformation = new Function<User, Integer>() {
  @Override
  public Integer apply(User user) {
    return user.id();
  }
};

Function<User, String> valuetransformation = new Function<User, String>() {
  @Override
  public String apply(User user) {
    return user.name();
  }
};

Map<Integer, String> userMap = Transformations.map(users, keytransformation, valuetransformation);

String userName = userMap.get(2);
assertEquals("susi",userName);
```

# Extensions - foldLeft

```
interface Foldleft<R, L> {
      L apply(L left, R right);
}


public static <S, D> D foldLeft(Collection<? extends S> collection, Foldleft<? super S, D> foldFunction, D
leftValue) {
  D ret = leftValue;
  for (S value : collection) {
    ret = foldFunction.apply(ret, value);
  }
  return ret;
}
```



# Extensions - foldLeft (sum)

```
List<Integer> numbers = Lists.newArrayList(1,2,3,4,5,6);
int result = Folds.foldLeft(numbers, new Foldleft<Integer, Integer>() {
    @Override
    public Integer apply(Integer left, Integer right) {
      return left+right;
    }
}, 0);
assertEquals(21, result);
```

# Extensions - foldLeft (count)

```
List<Integer> source = Lists.newArrayList(1,2,3,4,5,6);
int result = Folds.foldLeft(source, new Foldleft<Integer, Integer>() {
    @Override
    public Integer apply(Integer left, Integer right) {
      return left+1;
    }
}, 0);
assertEquals(6,result);
```

# Extensions - foldLeft (concat)

```
List<String> source = Lists.newArrayList("fu","man","chu");
String result = Folds.foldLeft(source, new Foldleft<String, String>() {
  @Override
  public String apply(String left, String right) {
    return left+" "+right;
  }
}, "name is");
assertEquals("name is fu man chu",result);
```

# Extensions - map+foldLeft

```
List<String> names = Lists.newArrayList("Achim", "Albert", "Susi", "Sonja", "Bert");
Function<String, Character> keytransformation = new Function<String, Character>() {
  @Override
  public Character apply(String name) {
    return name.charAt(0);
  }
};

Map<Character, ImmutableList<? extends String>> nameMap = Transformations.map(names, keytransformation,
    Folds.asListFold(Transformations.<String> asCollection()));

ImmutableList<? extends String> namesWithA = nameMap.get('A');
assertEquals("[Achim, Albert]", namesWithA.toString());
```

# Extensions - map+foldLeft explained

```
private static <K, V, T, M extends Map<K, V>> M map(MapCreator<K, V, M> mapCreator, Collection<T> collection,
    Function<? super T, K> keyTransformation, Foldleft<? super T, V> valueFold) {
  M map = mapCreator.newInstance();
  for (T value : collection) {
    K key = keyTransformation.apply(value);
    map.put(key, valueFold.apply(map.get(key), value));
  }
  return map;
}
```

# Extensions - map+foldLeft explained

```
static class ImmutableListFold<R> implements CollectingFold<R, ImmutableList<? extends R>> {
  @Override
  public ImmutableList<? extends R> apply(ImmutableList<? extends R> left, Collection<? extends R> right) {
    left = Types.defaultIfNull(left, ImmutableList.<R> of());

    if (right.isEmpty())
      return left;
    if (left.isEmpty())
      return ImmutableList.copyOf(right);
    return ImmutableList.<R> builder().addAll(left).addAll(right).build();
  }
}
```

# Extensions - map+foldLeft 1:1 map

```
List<String> names = Lists.newArrayList("Achim", "Albert", "Susi", "Sonja", "Bert");

Function<String, Character> keytransformation = new Function<String, Character>() {
  @Override
  public Character apply(String name) {
    return name.charAt(0);
  }
};

Map<Character, String> nameMap = Transformations.map(names, keytransformation,new Foldleft<String, String>() {
  @Override
  public String apply(String left, String right) {
    if (left!=null) {
      throw new IllegalArgumentException("key for "+right+" allready mapped to "+left);
    }
    return right;
  }
}); -> throws IllegalArgumentException(“key for Albert allready mapped to Achim”);
```

# Extensions

… and much more.

# Why should we use it?

- separate “packaging” from transformation
  - composition of transformations and packagings
  - separate tests
- reduce code duplication (less for loops and list.add())
- less code if done well …

Q & A