Functional Utilities

Caveats

Java 8 includes the `java.util.function` and `java.util.stream` packages, which supercede Guava's functional programming classes for projects at that language level.

While Guava's functional utilities are usable on Java versions prior to Java 8, functional programming without Java 8 requires awkward and verbose use of anonymous classes.

Excessive use of Guava's functional programming idioms can lead to verbose, confusing, unreadable, and inefficient code. These are by far the most easily (and most commonly) abused parts of Guava, and when you go to preposterous lengths to make your code "a one-liner," the Guava team weeps.

Compare the following code:

```
Function<String, Integer> lengthFunction = new Function<String, Integer>() {
  public Integer apply(String string) {
    return string.length();
  }
};
Predicate<String> allCaps = new Predicate<String>() {
  public boolean apply(String string) {
    return CharMatcher.javaUpperCase().matchesAllOf(string);
  }
};
Multiset<Integer> lengths = HashMultiset.create(
  Iterables.transform(Iterables.filter(strings, allCaps), lengthFunction));
```

or the `FluentIterable` version

```
Multiset<Integer> lengths = HashMultiset.create(
  FluentIterable.from(strings)
    .filter(new Predicate<String>() {
       public boolean apply(String string) {
         return CharMatcher.javaUpperCase().matchesAllOf(string);
       }
     })
    .transform(new Function<String, Integer>() {
       public Integer apply(String string) {
         return string.length();
       }
     }));
```

with:

```
Multiset<Integer> lengths = HashMultiset.create();
```

```
for (String string : strings) {
  if (CharMatcher.javaUpperCase().matchesAllOf(string)) {
    lengths.add(string.length());
  }
}
```

Even using static imports, even if the Function and the Predicate declarations are moved to a different file, the first implementation is less concise, less readable, and less efficient.

Imperative code should be your *default*, your *first choice* as of Java 7. You should not use functional idioms unless you are *absolutely* sure of one of the following:

- Use of functional idioms will result in *net* savings of lines of code for your entire project. In the example above, the "functional" version used 11 lines, the imperative version 6. Moving the definition of a function to another file, or a constant, does not help.
- For efficiency, you need a lazily computed view of the transformed collection and cannot settle for an explicitly computed collection. Additionally, you have read and reread Effective Java, item 55, and besides following those instructions, you have actually done benchmarking to prove that this version is faster, and can cite numbers to prove it.

Please be sure, when using Guava's functional utilities, that the traditional imperative way of doing things isn't more readable. Try writing it out. Was that so bad? Was that more readable than the preposterously awkward functional approach you were about to try?

Functions and Predicates

This article discusses only those Guava features dealing directly with `Function` and `Predicate`. Some other utilities are associated with the "functional style," such as concatenation and other methods which return views in constant time. Try looking in the collection utilities article.

Guava provides two basic "functional" interfaces:

- `Function<A, B>`, which has the single method `B apply(A input)`. Instances of `Function` are generally expected to be referentially transparent – no side effects – and to be consistent with equals, that is, `a.equals(b)` implies that `function.apply(a).equals(function.apply(b))`.
- `Predicate<T>`, which has the single method `boolean apply(T input)`. Instances of `Predicate` are generally expected to be side-effect-free and consistent with equals.

Special predicates

Characters get their own specialized version of `Predicate`, `CharMatcher`, which is typically more efficient and more useful for those needs. `CharMatcher`

already implements `Predicate<Character>`, and can be used correspondingly, while conversion from a `Predicate` to a `CharMatcher` can be done using `CharMatcher.forPredicate`. Consult the `CharMatcher` article for details.

Additionally, for comparable types and comparison-based predicates, most needs can be fulfilled using the `Range` type, which implements an immutable interval. The `Range` type implements `Predicate`, testing containment in the range. For example, `Range.atMost(2)` is a perfectly valid `Predicate<Integer>`. More details on using ranges can be found in the [`Range` article].

Manipulating Functions and Predicates

Simple `Function` construction and manipulation methods are provided in `Functions`, including

- `forMap(Map<A, B>)`
- `compose(Function<B, C>, Function<A, B>)`
- `constant(T)`
- `identity()`
- `toStringFunction()`

Consult the Javadoc for details.

There are considerably more construction and manipulation methods available in `Predicates`, but a sample includes:

- `instanceOf(Class)`
- `assignableFrom(Class)`
- `contains(Pattern)`
- `in(Collection)`
- `isNull()`
- `alwaysFalse()`, `alwaysTrue()`
- `equalTo(Object)`
- `compose(Predicate, Function)`
- `and(Predicate...)`, `or(Predicate...)`, `not(Predicate)`

Consult the Javadoc for details.

Using

Guava provides many tools to manipulate collections using functions and predicates. These can typically be found in the collection utility classes `Iterables`, `Lists`, `Sets`, `Maps`, `Multimaps`, and the like.

Predicates

The most basic use of predicates is to filter collections. All Guava filter methods return *views*.

Collection type	Filter methods
`Iterable`	`Iterables.filter(Iterable, Predicate)FluentIterable.filter(Predicate)`