Google Guava

Source code available on GitHub:
https://github.com/tdziurko/Guava-Lessons

Author: Tomasz Dziurko

Blog: http://tomaszdziurko.pl

Twitter: https://twitter.com/#!/TomaszDziurko

import com.google.common.base.Functions (1)

- *Function* can be used to execute defined action (function) on many objects
- It is used to change collections into collections of different type
- Main usage with *transform()* method from classes *Colllections2, Lists, Iterables, Iterators*, etc.
- *Functions* class provides a few pre-defined functions and allows to compose them into more complicated ones

import com.google.common.base.Functions (2)

- Simple example – transforming collection of countries into collection of capital cities

```
@Test
public void shouldPrintCountryWithCapitalCityUpperCase() throws Exception {

    // given
    Function<Country, String> capitalCityFunction = new Function<~>() {
        public String apply(@Nullable Country country) {
            if (country == null) {
                return "";
            }
            return country.getCapitalCity();
        }
    };

    // when
    Collection<String> capitalCities = Collections2.transform(Country.getCountries(), capitalCityFunction);

    // then
    assertThat(capitalCities).contains("Warsaw", "Madrid");
}
```

import com.google.common.base.Functions (3)

- *Functions.compose()* – composition of two or more functions to use as a one

```
@Test
public void shouldComposeTwoFunctions() throws Exception {
    Function<Country, String> upperCaseFunction = new Function<Country, String>() {
        public String apply(@Nullable Country country) {
            if (country == null) {
                return "";
            }
            return country.getName().toUpperCase() + ": " + country.getCapitalCity().toUpperCase();
        }
    };

    Function<String, String> reverseFunction = new Function<String, String>() {
        public String apply(String string) {
            if(string == null) {
                return null;
            }
            return new StringBuilder(string).reverse().toString();
        }
    };
    Function<Country, String> composedFunction = Functions.compose(reverseFunction, upperCaseFunction);

    // when
    Collection<String> reversedCapitalCities = Collections2.transform(Country.getCountries(), composedFunction);

    // then
    assertThat(reversedCapitalCities).contains("WASRAW", "DIRDAM");
}
```

import com.google.common.base.Functions (4)

- *Functions.forMap()* – pre-defined function loading values from map for provided list of keys

```
@Test
public void shouldUseForMapFunction() throws Exception {

    // given
    Map<String, String> map = Maps.newHashMap();
    map.put(Country.POLAND.getName(), Country.POLAND.getCapitalCity());
    map.put(Country.BELGIUM.getName(), Country.BELGIUM.getCapitalCity());
    map.put(Country.SPAIN.getName(), Country.SPAIN.getCapitalCity());
    map.put(Country.ENGLAND.getName(), Country.ENGLAND.getCapitalCity());

    // when
    Function<String, String> capitalCityFromCountryName = Functions.forMap(map);

    List<String> countries = Lists.newArrayList();
    countries.add(Country.POLAND.getName());
    countries.add(Country.BELGIUM.getName());

    // then
    Collection<String> capitalCities = Collections2.transform(countries, capitalCityFromCountryName);

    assertThat(capitalCities).containsOnly(Country.POLAND.getCapitalCity(), Country.BELGIUM.getCapitalCity());
}
```

import com.google.common.base.Functions (5)

- *Functions.forMap()* – when there is no value for a given key, exception is thrown. This behaviour can be changed to return defined default value

```
@Test
public void shouldUseForMapFunctionWithDefaultValue() throws Exception {

    // given
    Map<String, String> map = Maps.newHashMap();
    map.put(Country.POLAND.getName(), Country.POLAND.getCapitalCity());

    // we omit this one intentionally
    // map.put(Country.BELGIUM.getName(), Country.BELGIUM.getCapitalCity());
    map.put(Country.SPAIN.getName(), Country.SPAIN.getCapitalCity());
    map.put(Country.ENGLAND.getName(), Country.ENGLAND.getCapitalCity());

    // when
    Function<String, String> capitalCityFromCountryName = Functions.forMap(map, "Unknown");

    List<String> countries = Lists.newArrayList();
    countries.add(Country.POLAND.getName());
    countries.add(Country.BELGIUM.getName());

    // then
    Collection<String> capitalCities = Collections2.transform(countries, capitalCityFromCountryName);

    assertThat(capitalCities).containsOnly(Country.POLAND.getCapitalCity(), "Unknown");
}
```

import com.google.common.base.Predicates (1)

- *Predicate* checks if given condition is met for passed object.
- It is mainly used to filter collections using *filter()* method from *Iterables, Iterators, Collections2,* etc.
- I can be also used to check if all elements from collections are satisfying defined rule - method *Iterables.all()*
- *Predicates* class provides a few utility methods to work use and allows to compose many Predicates in one

import com.google.common.base.Predicates (2)

- Simple predicate checking if country has a capital city defined

```
@Test
public void shouldUseCustomPredicate() throws Exception {

    // given
    Predicate<Country> capitalCityProvidedPredicate = new Predicate<Country>() {

        @Override
        public boolean apply(@Nullable Country country) {
            return !Strings.isNullOrEmpty(country.getCapitalCity());
        }
    };

    // when
    boolean allCountriesSpecifyCapitalCity = Iterables.all(
            Lists.newArrayList(Country.POLAND, Country.BELGIUM, Country.FINLAND_WITHOUT_CAPITAL_CITY),
                capitalCityProvidedPredicate);

    // then
    assertFalse(allCountriesSpecifyCapitalCity);
}
```

import com.google.common.base.Predicates (3)

- Predicate objects can be composed using *Predicates.and()*, *Predicates.or()* and *Predicates.not()*

```
@Test
public void shouldComposeTwoPredicates() throws Exception {

    // given
    Predicate<Country> fromEuropePredicate = new Predicate<Country>() {

        @Override
        public boolean apply(@Nullable Country country) {
            return Continent.EUROPE.equals(country.getContinent());
        }
    };

    Predicate<Country> populationPredicate = new Predicate<Country>() {

        @Override
        public boolean apply(@Nullable Country country) {
            return country.getPopulation() < 20;
        }
    };

    Predicate<Country> composedPredicate = Predicates.and(fromEuropePredicate, populationPredicate);

    // when
    Iterable<Country> filteredCountries = Iterables.filter(Country.getCountries(), composedPredicate);

    // then
    assertThat(filteredCountries).containsOnly(Country.BELGIUM);
}
```

import com.google.common.base.Predicates (4)

- *Predicates.containsPattern()* – used to create condition using regular expressions

```
@Test
public void shouldCheckPattern() throws Exception {

    // given
    Predicate<CharSequence> twoDigitsPredicate = Predicates.containsPattern("\\d\\d");

    // then
    assertThat(twoDigitsPredicate.apply("Hello world 40")).isTrue();
}
```

import com.google.common.base.CharMatcher (1)

- It is a class similar to *Predicate*, but working with *chars*
- It allows to check if a sequence of characters satisfies given condition
- We can also use it to filter chars
- Additionaly it provides methods to modify char sequences: *removeFrom(), replaceFrom(), trimFrom(), collapseFrom(), retainFrom()*

import com.google.common.base.CharMatcher (2)

- A few simple matchers checking if different conditions are met

```
@Test
public void shouldNotMatchChar() throws Exception {
    assertThat(CharMatcher.noneOf("xZ").matchesAnyOf("anything")).isTrue();
}

@Test
public void shouldMatchAny() throws Exception {
    assertThat(CharMatcher.ANY.matchesAllOf("anything")).isTrue();
}

@Test
public void shouldMatchBreakingWhitespace() throws Exception {
    assertThat(CharMatcher.BREAKING_WHITESPACE.matchesAllOf("\r\n\r\n")).isTrue();
}

@Test
public void shouldMatchDigits() throws Exception {
    assertThat(CharMatcher.DIGIT.matchesAllOf("1231212")).isTrue();
}

@Test
public void shouldMatchDigitsWithWhitespace() throws Exception {
    assertThat(CharMatcher.DIGIT.matchesAnyOf("1231 aa212")).isTrue();
}
```

import com.google.common.base.CharMatcher (3)

- *CharMatcher.retainFrom()* – allows to keep only those chars that are satisfying defined matching rule
- Matching rules can be joined with *or()* or *and()* methods

```
@Test
public void shouldRetainOnlyDigits() throws Exception {
    assertThat(CharMatcher.DIGIT.retainFrom("Hello 1234 567")).isEqualTo("1234567");
}

@Test
public void shouldRetainDigitsOrWhiteSpaces() throws Exception {
    assertThat(CharMatcher.DIGIT.or(CharMatcher.WHITESPACE).retainFrom("Hello 1234 567")).isEqualTo(" 1234 567");
}

@Test
public void shouldRetainNothingAsConstrainsAreExcluding() throws Exception {
    assertThat(CharMatcher.DIGIT.and(CharMatcher.JAVA_LETTER).retainFrom("Hello 1234 567")).isEqualTo("");
}

@Test
public void shouldRetainLettersAndDigits() throws Exception {
    assertThat(CharMatcher.DIGIT.or(CharMatcher.JAVA_LETTER).retainFrom("Hello 1234 567")).isEqualTo("Hello1234567");
}
```

import com.google.common.base.CharMatcher (4)

- *CharMatcher.collapseFrom()* – replaces group of chars satysfying matching rule with defined sequence of characters
- *CharMatcher.replaceFrom()* - replaces every char satysfying matching rule with defined sequence of characters

```
@Test
public void shouldCollapseAllDigitsByX() throws Exception {
    assertThat(CharMatcher.DIGIT.collapseFrom("Hello 1234 567", 'x')).isEqualTo("Hello x x");
}

 @Test
public void shouldReplaceAllDigitsByX() throws Exception {
    assertThat(CharMatcher.DIGIT.replaceFrom("Hello 1234 567", 'x')).isEqualTo("Hello xxxx xxx");
}
```

import com.google.common.base.CharMatcher (5)

- Another interesting methods: *countIn(),* indexIn() finding position of first match and *lastIndexIn()* finding position of last match

```
@Test
public void shouldCountDigits() throws Exception {
    assertThat(CharMatcher.DIGIT.countIn("Hello 1234 567")).isEqualTo(7);
}

@Test
public void shouldReturnFirstIndexOfFirstWhitespace() throws Exception {
    assertThat(CharMatcher.WHITESPACE.indexIn("Hello 1234 567")).isEqualTo(5);
}

@Test
public void shouldReturnLastIndexOfFirstWhitespace() throws Exception {
    assertThat(CharMatcher.WHITESPACE.lastIndexIn("Hello 1234 567")).isEqualTo(10);
}
```

import com.google.common.base.CharMatcher (6)

- *inRange()* allows to define a matching rule with a range of characters
- *negate()* creates negated matching rule

```
@Test
public void shouldRemoveDigitsBetween3and6() throws Exception {
    assertThat(CharMatcher.inRange('3', '6').removeFrom("Hello 1234 567")).isEqualTo("Hello 12 7");
}

@Test
public void shouldRemoveAllExceptDigitsBetween3and6() throws Exception {
    assertThat(CharMatcher.inRange('3', '6').negate().removeFrom("Hello 1234 567")).isEqualTo("3456");
}
```

import com.google.common.base.CharMatcher (7)

- Methods *trimFrom(), trimLeadingFrom(), trimTrailingFrom() or trimAndCollapseFrom*() find and trim sequences located at the start or end of passed element that match the rule

```
@Test
public void shouldRemoveStartingAndEndingDollarsAndKeepOtherUnchanged() throws Exception {
    assertThat(CharMatcher.is('$').trimFrom("$$$ This is a $ sign $$$")).isEqualTo(" This is a $ sign ");
}

@Test
public void shouldRemoveOnlyStartingDollarsAndKeepOtherUnchanged() throws Exception {
    assertThat(CharMatcher.is('$').trimLeadingFrom("$$$ This is a $ sign $$$")).isEqualTo(" This is a $ sign $$$");
}

@Test
public void shouldRemoveStartingEndEndingDollarsAndReplaceOthersWithX() throws Exception {
    assertThat(CharMatcher.is('$').trimAndCollapseFrom("$$$This is a $ sign$$$", 'X')).isEqualTo("This is a X sign");
}
```

import com.google.common.base.Joiner(1)

- Class used to convert collections into single String object containing elements separated with a defined delimeter
- We can declare to omit nulls or replace them with a default value
- It also works with *Maps*

import com.google.common.base.Joiner (2)

- Simple example with *List*

```
public static List<String> languages = Arrays.asList("Java", "Haskell", "Scala", "Brainfuck");

@Test
public void shouldJoinWithCommas() throws Exception {
    assertThat(Joiner.on(",").join(languages)).isEqualTo("Java,Haskell,Scala,Brainfuck");
}
```

import com.google.common.base.Joiner (3)

- If there is a null in the collection and we don't define how to treat it, we will get *NullPointerException*
- Nulls can be omitted with *skipNulls()* or replaced with a default value using *useForNull()*

```
public static List<String> countriesWithNullValue = Arrays
        .asList("Poland", "Brasil", "Ukraine", null, "England", "Croatia");

@Test
public void shouldJoinWithCommasAndOmitNulls() throws Exception {
    assertThat(Joiner.on(",").skipNulls().join(countriesWithNullValue))
            .isEqualTo("Poland,Brasil,Ukraine,England,Croatia");
}

@Test
public void shouldJoinWithCommasAndReplaceNullsWithWordNothing() throws Exception {
    assertThat(Joiner.on(",").useForNull("NONE").join(countriesWithNullValue))
            .isEqualTo("Poland,Brasil,Ukraine,NONE,England,Croatia");
}
```

import com.google.common.base.Joiner (4)

- *Joiner* also works with *Maps*, we can define how to separate key-value pairs and what chars should be used as a element between a key and a value

```
public static Map<Integer, String> numbersWords = new HashMap<~>();

static {
    numbersWords.put(1, "one");
    numbersWords.put(2, "two");
    numbersWords.put(3, null);
    numbersWords.put(4, "four");
}

@Test
public void shouldJoinMap() throws Exception {
    assertThat(Joiner.on(" | ").withKeyValueSeparator(" -> ")
            .useForNull("Unknown").join(numbersWords))
            .isEqualTo("1 -> one | 2 -> two | 3 -> Unknown | 4 -> four");
}
```

import com.google.common.base.Objects(1)

- Utility class with helper method to implement *equals()*, *hashcode()* and *toString() methods*
- Additionally it contains useful method *firstNonNull()*

import com.google.common.base.Objects(2)

- *Objects.equals()* compares two objects that can be nulls
- *Objects.hashcode()* takes unlimited number of parameters

```
public class UserProfile {

    private String name;
    private String nickname;
    private Integer age;


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof UserProfile)) return false;

        UserProfile objectsLesson = (UserProfile) o;
        return Objects.equal(this.name, objectsLesson.name) &&
                Objects.equal(this.age, objectsLesson.age) &&
                Objects.equal(this.nickname, objectsLesson.nickname);
    }

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

import com.google.common.base.Objects(3)

- *Objects.toStringHelper()* makes it easier to write *toString()*

```
public String toString() {
    return Objects.toStringHelper(this).add("name", name)
            .add("nickname", nickname)
            .addValue(age).toString();
}
```

```
@Test
public void shouldShowHowToStringMethodWorks() throws Exception {

    assertThat(objectsLesson.toString())
            .isEqualTo("UserProfile{name=name, nickname=nickname, 20}");

}
```

import com.google.common.base.Objects(4)

- *Objects.firstNonNull()* returns first argument which is not null

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

import com.google.common.base.Preconditions(1)

- This class allows to check correctness of parameters passed to our method and throw an appropriate exception when necessary

import com.google.common.base.Preconditions(2)

- We can check if state of the object/parameter is correct and throw *IllegalStateException* otherwise

```
public void getSomeSuntan(Weather weather) {
    Preconditions.checkState(weather.equals(Weather.SHINY), "Weather is not the best for a sunbath");
}
```

```
@Test(expectedExceptions = IllegalStateException.class, expectedExceptionsMessageRegExp =
        "Weather is not the best for a sunbath")
public void shouldThrowIllegalState() throws Exception {

   preconditionsLesson.getSomeSuntan(PreconditionsLesson.Weather.CLOUDY);
}
```

import com.google.common.base.Preconditions(3)

- We can check if passed argument is not null or if it satisfies defined condition

```
public void displayFootballTeamMembers(List<String> teamMembers) {
    Preconditions.checkNotNull(teamMembers, "Team can not be null");
    Preconditions.checkArgument(teamMembers.size() == 11, "Full team should consist of 11 players");
}
```

```
@Test(expectedExceptions = NullPointerException.class, expectedExceptionsMessageRegExp =
        "Team can not be null")
public void shouldNotAcceptNullFootballTeam() throws Exception {

    // when
    preconditionsLesson.displayFootballTeamMembers(null);
}

@Test(expectedExceptions = IllegalArgumentException.class, expectedExceptionsMessageRegExp =
        "Full team should consist of 11 players")
public void shouldNotAcceptNotFullFootballTeam() throws Exception {

    // when
    preconditionsLesson.displayFootballTeamMembers(Arrays.asList("Casillas", "Pepe", "Ramos", "Marcelo"));
}
```

import com.google.common.base.Splitter(1)

- Class working in the opposite direction than *Joiner*
- It allows to split String into collection of elements
- Delimeter can be defined as a sequence of chars, reg exp or *CharMatcher* object

import com.google.common.base.Splitter(2)

- Simple example splitting *String* on semicolons

```
@Test
public void shouldSplitOnSemicolons() throws Exception {
    // when
    Iterable<String> iterable = Splitter.on(";").split("Java;Scala;Haskell;Brainfuck;Kotlin");
    List<String> splittedList = convertToList(iterable.iterator());

    // then
    assertThat(splittedList.size()).isEqualTo(5);
    assertThat(splittedList.get(3)).isEqualTo("Brainfuck");
}
```

import com.google.common.base.Splitter(3)

- Split using regular expression

```
@Test
public void shouldSplitOnRegExp() throws Exception {
    // when
    Iterable<String> iterable = Splitter.onPattern("\\d+").split("Java3Scala4Haskell0Brainfuck5Kotlin");
    List<String> splittedList = convertToList(iterable.iterator());

    // then
    assertThat(splittedList.size()).isEqualTo(5);
    assertThat(splittedList.get(2)).isEqualTo("Haskell");
}
```

import com.google.common.base.Splitter(4)

- Split using *CharMatcher*

```
@Test
public void shouldSplitUsingCharMatcher() throws Exception {

    // when
    Iterable<String> iterable = Splitter
            .on(CharMatcher.inRange('3', '5')).split("Java3Scala4Haskell0Brainfuck5Kotlin");
    List<String> splittedList = convertToList(iterable.iterator());

    // then
    assertThat(splittedList.size()).isEqualTo(4);
    assertThat(splittedList.get(2)).isEqualTo("Haskell0Brainfuck");

}
```

import com.google.common.base.Splitter(5)

- Define Splitter to omit empty elements with *omitEmptyStrings()* or trim white spaces from extracted elements using *trimResults()*

```
@Test
public void shouldSplitAndOmitEmptyElementsAndWhitespaces() throws Exception {
    // when
    Iterable<String> iterable = Splitter.on(";").omitEmptyStrings()
            .trimResults().split("Java;;  ;Scala;;;Haskell;Brainfuck;Kotlin");
    List<String> splittedList = convertToList(iterable.iterator());

    // then
    assertThat(splittedList.size()).isEqualTo(5);
    assertThat(splittedList.get(1)).isEqualTo("Scala");

}
```

import com.google.common.base.Splitter(6)

- We can split given String into elements with defined length using *fixedLength()*

```
@Test
public void shouldSplitForEqualLength() throws Exception {

    // when
    Iterable<String> iterable = Splitter.fixedLength(5).split("HorseHouseGroupDemosScrum");
     List<String> splittedList = convertToList(iterable.iterator());

    // then
    assertThat(splittedList.size()).isEqualTo(5);
    assertThat(splittedList.get(4)).isEqualTo("Scrum");
}
```

import com.google.common.base.Stopwatch(1)

- Class replacing traditional way of calculating method execution time using *System.nanoTime()*
- It provides methods that automatically calculate time between *start()* and *stop()* execution
- Can be easily mocked with our custom passing time provider
- Returns counted time using different units

import com.google.common.base.Stopwatch(2)

- Simple time counting with mocked *Ticker*, for real usage we can use default *Ticker* using system clock

```
@Test
public void shouldCalculateIterationsTime() throws Exception {

    // given
    Ticker ticker = mock(Ticker.class);
    when(ticker.read()).thenReturn(0L, 2000000000L);
    Stopwatch stopwatch = new Stopwatch(ticker);

    // when
    stopwatch.start();
    // some method is called here
    stopwatch.stop();

    // then
    assertThat(stopwatch.elapsedMillis()).isEqualTo(2000L);
}
```

import com.google.common.base.Stopwatch(3)

- Returning time using different units

```
@Test
public void shouldPrintIterationsTime() throws Exception {

    // given
    Ticker ticker = mock(Ticker.class);
    when(ticker.read()).thenReturn(0L, 2*60*60*1000000000L); // 2 hours
    Stopwatch stopwatch = new Stopwatch(ticker);

    // when
    stopwatch.start();
    // some method is called here
    stopwatch.stop();

    // then
    assertThat(stopwatch.toString()).isEqualTo("7200 s");
    assertThat(stopwatch.elapsedTime(TimeUnit.MINUTES)).isEqualTo(120);
    assertThat(stopwatch.elapsedTime(TimeUnit.HOURS)).isEqualTo(2);
}
```

import com.google.common.base.Strings(1)

- Helper class to work with *String* objects
- Can replace null with empty *String* or empty *String* with null
- It provides methods to pad elements to left or right

import com.google.common.base.Strings(2)

- Check if *String* is null or empty with *isNullOrEmpty()*
- Empty/null *Strings* convertion to null/empty ones

```
@Test
public void shouldReturnTrueOnNullString() throws Exception {
  assertThat(Strings.isNullOrEmpty(null)).isTrue();
}

@Test
public void shouldConvertNullToEmpty() throws Exception {
    assertThat(Strings.nullToEmpty(null)).isEqualTo("");
}

@Test
public void shouldConvertEmptyToNull() throws Exception {
    assertThat(Strings.emptyToNull("")).isNull();
}
```

import com.google.common.base.Strings(3)

- String padding to left or right using defined character as a ”filler”

```
@Test
public void shouldPadEnd() throws Exception {

    assertThat(Strings.padEnd("Nothing special", 20, '*')).isEqualTo("Nothing special*****");
}

@Test
public void shouldPadStart() throws Exception {
    assertThat(Strings.padStart("Nothing special", 20, ' ')).isEqualTo("     Nothing special");
}
```

import com.google.common.base.Strings(4)

- Create new *String* as a n-time repetition

```
@Test
public void shouldRepeatGivenString() throws Exception {
    assertThat(Strings.repeat("Hello ", 3)).isEqualTo("Hello Hello Hello ");
}
```

import com.google.common.base.Throwables(1)

- Helper class to work with *exceptions*
- It allows to extract root exception from the chain
- Converts stack trace to *String*
- Extracts list of exceptions from exceptions chain

import com.google.common.base.Throwables(2)

- Extract the root cause from exceptions chain

```
@Test
public void shouldExtractInnermostException() throws Exception {

    try {
        try {
            try {
                throw new RuntimeException("Innermost exception");
            }
            catch(Exception e) {
                throw new SQLException("Middle tier exception", e);
            }
        }
        catch(Exception e) {
            throw new IllegalStateException("Last exception", e);
        }
    }
    catch(Exception e) {
        assertThat(Throwables.getRootCause(e).getMessage()).isEqualTo("Innermost exception");
    }
}
```

import com.google.common.base.Throwables(3)

- List of exceptions from chain

```
@Test
public void shouldListAllExceptionsChain() throws Exception {

    try {
        try {
            try {
                throw new RuntimeException("Innermost exception");
            }
            catch(Exception e) {
                throw new SQLException("Middle tier exception", e);
            }
        }
        catch(Exception e) {
            throw new IllegalStateException("Last exception", e);
        }
    }
    catch(Exception e) {
        List<Throwable> exceptionsChain = Throwables.getCausalChain(e);
        assertThat(exceptionsChain).onProperty("message")
                .containsExactly("Last exception", "Middle tier exception", "Innermost exception");
    }
}
```

import com.google.common.base.Throwables(4)

- Convert stack trace to *String*

```
@Test
public void shouldGetStackTrace() throws Exception {

    try {
        try {
            try {
                throw new RuntimeException("Innermost exception");
            }
            catch(Exception e) {
                throw new SQLException("Middle tier exception", e);
            }
        }
        catch(Exception e) {
            throw new IllegalStateException("Last exception", e);
        }
    }
    catch(Exception e) {
        assertThat(Throwables.getStackTraceAsString(e))
                .contains("Caused by: java.lang.RuntimeException: Innermost exception");
    }
}
```

import com.google.common.collect.Collections2(1)

- Class to filter and transform Collections using functions and predicates

import com.google.common.collect.Collections2(2)

- *transform()* – transforming collection using function

```
@Test
public void shouldTransformCollection() throws Exception {

    // given
    ArrayList<Country> countries = Lists.newArrayList(Country.POLAND, Country.BELGIUM, Country.ENGLAND);

    // when
    Collection<String> capitalCities = Collections2.transform(countries,
            new Function<Country, String>() {

                @Override
                public String apply(@Nullable Country country) {
                    return country.getCapitalCity();
                }
            });

    // then
    assertThat(capitalCities).containsOnly("Warsaw", "Brussels", "London");
}
```

import com.google.common.collect.Collections2(3)

- filter() – filtering collections

```
@Test
public void shouldFilterCountriesOnlyFromAfrica() throws Exception {

    // given
    ArrayList<Country> countries = Lists.newArrayList(Country.POLAND, Country.BELGIUM, Country.SOUTH_AFRICA);

    // when
    Collection<Country> countriesFromAfrica = Collections2.filter(countries, new Predicate<Country>() {

        @Override
        public boolean apply(@Nullable Country country) {
            return Continent.AFRICA.equals(country.getContinent());
        }
    });

    // then
    assertThat(countriesFromAfrica).containsOnly(Country.SOUTH_AFRICA);
}
```

import com.google.common.collect.Collections2(4)

- Important! What we get is a ”view” of passed collection!

```
@Test
public void shouldShowThatResultIsOnlyAView() throws Exception {

    // given
    ArrayList<Country> countries = Lists.newArrayList(Country.POLAND, Country.BELGIUM, Country.ENGLAND);

    // when
    Collection<String> capitalCities = Collections2.transform(countries,
            new Function<Country, String>() {

                @Override
                public String apply(@Nullable Country country) {
                    return country.getCapitalCity();
                }
            });

    // then
    assertThat(capitalCities).containsOnly("Warsaw", "Brussels", "London");
    assertThat(capitalCities).excludes("Pretoria");

    countries.add(Country.SOUTH_AFRICA);

    assertThat(capitalCities).contains("Pretoria");
}
```

import com.google.common.collect.BiMap(1)

- *BiMap* is a *Map* with unique both keys and values

import com.google.common.collect.BiMap(2)

- Adding new pair with existing key will cause an exception

```
@Test(expectedExceptions = IllegalArgumentException.class,
        expectedExceptionsMessageRegExp = "value already present: one")
public void shouldNotAllowToPutExistingValue() throws Exception {

    BiMap<Integer, String> bimap = HashBiMap.create();

    // when
    bimap.put(1, "one");
    bimap.put(2, "two");
    bimap.put(10, "ten");
    bimap.put(10, "one");

    fail("Should throw IllegalArgumentException");
}
```

import com.google.common.collect.BiMap(3)

- If we really want to add a key-value pair with value that is already in BiMap we can use *forcePut()*. Key that was previously linked with added value will be re-linked to null.

```
@Test
public void shouldAllowToPutExistingValueWithForcePut() throws Exception {

    BiMap<Integer, String> bimap = HashBiMap.create();

    // when
    bimap.put(1, "one");
    bimap.put(2, "two");
    bimap.put(10, "ten");
    bimap.forcePut(10, "one");

    assertThat(bimap.get(10)).isEqualTo("one");
    assertThat(bimap.get(1)).isNull();

}
```

import com.google.common.collect.BiMap(4)

- *inverse()* – because *BiMap* is bidirectional we can easily inverse it and use values as keys

```
@Test
public void shouldInverseBiMap() throws Exception {

    BiMap<Integer, String> bimap = HashBiMap.create();

    // when
    bimap.put(1, "one");
    bimap.put(2, "two");
    bimap.put(10, "ten");

    BiMap<String, Integer> inversedBiMap = bimap.inverse();

    // then
    assertThat(inversedBiMap.get("one")).isEqualTo(1);
}
```

import com.google.common.collect.Constraints(1)

- *Constraint* class defines a condition which must be met to add a new element to collection
- *Constraints* is a helper class used to create collections with added constraint

import com.google.common.collect.Constraints(2)

- Simple example using *Constraint* to create collection not accepting null values

```
@Test(expectedExceptions = NullPointerException.class)
public void shouldThrowExceptionOnNullAdd() throws Exception {

    // given
    List<Integer> numbers = Constraints
            .constrainedList(Lists.newArrayList(1, 2, 3), Constraints.<~>notNull());

    // when
    numbers.add(null);

    // then
    fail("Should throw a NullPointerException");
}
```

import com.google.common.collect.Constraints(3)

- It is easy to create custom Constraint, only one method have to be implemented – *checkElement()*

```
@Test(expectedExceptions = IllegalArgumentException.class)
public void shouldThrowExceptionOnInvalidAdd() throws Exception {

    // given
    List<Integer> userAgesList = Constraints.constrainedList(Lists.newArrayList(1, 2, 3), new Constraint<Integer>() {

        @Override
        public Integer checkElement(Integer age) {
            Preconditions.checkNotNull(age);
            Preconditions.checkArgument(age.intValue() > 0);
            return age;
        }
    });

    // when
    userAgesList.add(-2);

    // then
    fail("Should throw a IllegalArgumentException");
}
```

import com.google.common.collect.ForwardingCollection(1)

- Class forwarding method calls to its internal collection
- Allows to override methods to e.g. automate some operations on collection
- There are similar classes for *List*, *Set*, *Multiset*, *Queue* and *Iterator*

import com.google.common.collect.ForwardingCollection(2)

- Only method we have to implement is *delegate()*. It should point to the internal collection on which all methods will be called.

```
Collection<Integer> numbersList = new ForwardingCollection<Integer>() {
    private List<Integer> list = Lists.newArrayList();

    @Override
    protected List<Integer> delegate() {
        return list;
    }
};
```

import com.google.common.collect.ForwardingCollection(3)

- Overriding all methods from *Collection* interface is optional
- With override we can customize behaviour of our *ForwardingCollection* instance e.g. add opposite number for each added element

```
@Override
public boolean add(Integer element) {
    if(element == null) {
        return super.add(element);
    }
    if(element.intValue() == 0) {
        return super.add(element);
    }
    else {
        return super.add(element) && super.add(-element);
    }
}

@Override
public boolean addAll(Collection<? extends Integer> integers) {
    if(integers == null) {
        return add(null);
    }
    boolean result = true;
    for (Integer element : integers) {
        result = result && add(element);
    }
    return result;
}
```

import com.google.common.collect.ForwardingCollection(4)

- Example from previous page in action

```
@Test
public void shouldAddOppositeNumber() throws Exception {

    // given
    Collection<Integer> numbersList = new ForwardingCollection<Integer>() {
        private List<Integer> list = Lists.newArrayList();

        @Override
        protected List<Integer> delegate() {...}

        @Override
        public boolean add(Integer element) {...}

        @Override
        public boolean addAll(Collection<? extends Integer> integers) {...}
    };

    numbersList.add(10);
    numbersList.add(12);
    numbersList.add(0);
    numbersList.add(null);

    assertThat(numbersList).containsOnly(10, 12, -10, -12, 0, null);

}
```

import com.google.common.collect.ImmutableMap(1)

- Class creating read-only *Maps*
- It has methods to create and initialize *Maps* with a few key ↔ value pairs
- There are similar classes for *List*, *Set*, *Multiset etc.*

import com.google.common.collect.ImmutableMap(2)

- Using *Builder* to create new *Map*

```
@Test
public void shouldUseMapBuilder() throws Exception {

    // when
    ImmutableMap<String, Integer> numbersMap = new ImmutableMap.Builder<String, Integer>()
            .put("one", 1)
            .put("two", 2)
            .put("three", 3)
            .put("four", 4)
            .build();

    // then
    assertThat(numbersMap.get("two")).isEqualTo(2);
}
```

import com.google.common.collect.ImmutableMap(3)

- Creating and initializing Map using *of()* method with up to five key ↔ value pairs

```
@Test
public void shouldUseQuickMapCreator() throws Exception {

    // when
    ImmutableMap<String, Integer> numbersMap = ImmutableMap.of("one", 1,
            "two", 2, "three", 3, "four", 4, "five", 5);


    assertThat(numbersMap.get("two")).isEqualTo(2);
}
```

import com.google.common.collect.Iterables(1)

- Helper class to work with classes implementing *Iterable*
- It contains methods *transform()* oraz *filter()* mentioned earlier
- There is a ”clone” class: *Iterators* with almost the same methods as *Iterables* but working with *Iterators*

import com.google.common.collect.Iterables(2)

- *all()* method allows to check if defined condition is satisfied by all elements from *Collection*

```
@Test
public void shouldCheckLengthOfAllElements() throws Exception {

    // given
    Predicate<String> lengthPredicate = new Predicate<String> () {
        @Override
        public boolean apply(@Nullable String input) {
            if(input == null) {
                return false;
            }
            return input.length() > 3;
        }
    };

    // then
    assertThat(Iterables.all(Lists.newArrayList("Java", "Scala", "Haskell", "Groovy", "Java", "Lisp"),
            lengthPredicate)).isTrue();
}
```

import com.google.common.collect.Iterables(3)

- *any()* method allows to check if defined condition is satisfied by at least one element from *Collection*

```
@Test
public void shouldCheckIfAtLeastOneElementIsEmptyOrNull() throws Exception {

    // given
    Predicate<String> emptyOrNullPredicate = new Predicate<~> () {

        @Override
        public boolean apply(@Nullable String input) {
            return Strings.isNullOrEmpty(input);
        }
    };

    // then
    assertThat(Iterables.any(Lists.newArrayList("Java", "Scala",
            "Haskell", "Groovy", "Java", "Lisp"), emptyOrNullPredicate)).isFalse();
}
```

import com.google.common.collect.Iterables(4)

- *cycle()* returns *Iterable* which *Iterator* cycles indefinitely over its elements

```
@Test
public void shouldCycleOverIterable() throws Exception {

    Iterable<String> cycleIterables = Iterables.cycle("Right", "Left");

    // then
    Iterator<String> iterator = cycleIterables.iterator();

    for(int i = 0; i < 100; i++) {
        iterator.next();
    }

    assertThat(iterator.next()).is(new Condition<String>() {
        @Override
        public boolean matches(String value) {
            return "Left".equals(value) || "Right".equals(value);
        }
    });
}
```

import com.google.common.collect.Iterables(5)

- *filter()* method takes not only predicates but also classes

```
@Test
public void shouldFilterOnlyLongs() throws Exception {

    // given
    List<Number> numbersList = Lists.newArrayList();

    numbersList.add(1L);
    numbersList.add(2);
    numbersList.add(3L);
    numbersList.add(4);

    // when
    Iterable<Long> filteredList = Iterables.filter(numbersList, Long.class);

    // then
    assertThat(filteredList).hasSize(2).contains(1L, 3L);
}
```

import com.google.common.collect.Iterables(6)

- *frequency()* counts number of elemens in the *Collection*

```
@Test
public void shouldCountElementsInIterable() throws Exception {

    // given
    List<Integer> numbersList = Lists.newArrayList(1, 2, 3, 0, -2, 2, 430, 2);

    int frequency = Iterables.frequency(numbersList, 2);


    // then
    assertThat(frequency).isEqualTo(3);
}
```

import com.google.common.collect.Iterables(7)

- *getFirst()*, *getLast()* – an easy way to get first and last element from collection

```
@Test
public void shouldGetFirstAndLast() throws Exception {

    // given
    List<Integer> numbersList = Lists.newArrayList(1, 2, 3, 0, -12, 22, 430, -1024);

    // when
    Integer first = Iterables.getFirst(numbersList, null);
    Integer last = Iterables.getLast(numbersList);

    // then
    assertThat(first).isEqualTo(1);
    assertThat(last).isEqualTo(-1024);
}
```

import com.google.common.collect.Iterables(8)

- *partition()* splits collection into sub-collections with defined length

```
@Test
public void shouldPartition() throws Exception {

    // given
    List<Integer> numbersList = Lists.newArrayList(1, 2, 3, 0, -12, 22, 430, -1024);

    // when
    Iterable<List<Integer>> partitionedLists = Iterables.partition(numbersList, 5);

    // then
    assertThat(Iterables.size(partitionedLists)).isEqualTo(2);
    Iterator<List<Integer>> iterator = partitionedLists.iterator();
    assertThat(iterator.next().size()).isEqualTo(5);
    assertThat(iterator.next().size()).isEqualTo(3);

}
```

import com.google.common.collect.Iterables(9)

- *toArray()* is a nicer way to convert *Collection* to array

```
@Test
public void shouldConvertToArray() throws Exception {

    // given
    List<Integer> numbersList = Lists.newArrayList(1, 2, 3, 0, -12, 22, 430, -1024);

    // when
    Number[] numbers = Iterables.toArray(numbersList, Number.class);
    Number[] numbersWithTraditionalWay = numbersList.toArray(new Number[] {});


    // then
    assertThat(numbers).contains(1, 2, 3, 0, -12, 22, 430, -1024);
}
```

import com.google.common.collect.Iterables(10)

- *removeIf()* removes from *Collection* only elements that satisfy defined predicate

```
@Test
public void shouldRemoveNegativeNumbers() throws Exception {

    // given
    List<Integer> numbersList = Lists.newArrayList(1, 2, 3, 0, -12, 22, 430, -1024);

    // when
    Iterables.removeIf(numbersList, new Predicate<Integer>() {
        @Override
        public boolean apply(@Nullable Integer input) {
            return input < 0;
        }
    });

    // then
    assertThat(numbersList).excludes(-12, -1024);
}
```

import com.google.common.collect.Multiset(1)

- Class allowing to insert the same element multiple times
- Returns number of occurences of an element
- Allows to set this number without calling *add()* N times

import com.google.common.collect.Multiset(2)

- Simple example

```
@Test
public void shouldAddElementTwoTimes() throws Exception {

    // given
    Multiset<String> multiset = HashMultiset.create();

    // when
    multiset.add("nothing");
    multiset.add("nothing");

    // then
    assertThat(multiset.count("nothing")).isEqualTo(2);
    assertThat(multiset.count("something")).isEqualTo(0);
}
```

import com.google.common.collect.Multiset(3)

- *Multiset* provides method to add/remove element multiple times
- And method *setCount()* to set counter to a particular value

```
@Test
public void shouldUserCustomAddRemoveAndSetCount() throws Exception {

    // given
    Multiset<String> multiset = HashMultiset.create();

    // when
    multiset.add("ball");
    multiset.add("ball", 10);

    // then
    assertThat(multiset.count("ball")).isEqualTo(11);


    // when
    multiset.remove("ball", 5);

    // then
    assertThat(multiset.count("ball")).isEqualTo(6);


    // when
    multiset.setCount("ball", 2);

    // then
    assertThat(multiset.count("ball")).isEqualTo(2);
}
```

import com.google.common.collect.Multiset(4)

- *retainAll()* allows to keep only defined keys in the collection

```
@Test
public void shouldRetainOnlySelectedKeys() throws Exception {

    // given
    Multiset<String> multiset = HashMultiset.create();

    multiset.add("ball");
    multiset.add("ball");
    multiset.add("cow");
    multiset.setCount("twelve", 12);

    // when
    multiset.retainAll(Arrays.asList("ball", "horse"));

    assertThat(multiset.count("ball")).isEqualTo(2);
    assertThat(multiset.count("twelve")).isEqualTo(0);
}
```

import com.google.common.collect.Multimap(1)

- Class to replace objects similar to *Map<String, List<String>>*
- Developer is no longer forced to check if lists exists for a given key before adding something

import com.google.common.collect.Multimap(2)

- Simple example

```
@Test
public void shouldTestHowMultimapWorks() throws Exception {

    // given
    Multimap<String, String> multimap = ArrayListMultimap.create();

    // when
    multimap.put("Poland", "Warsaw");
    multimap.put("Poland", "Cracow");
    multimap.put("Poland", "Plock");
    multimap.put("Poland", "Gdansk");

    multimap.put("Germany", "Berlin");
    multimap.put("Germany", "Bremen");
    multimap.put("Germany", "Dortmund");
    multimap.put("Germany", "Koln");

    multimap.put("Portugal", "Lisbone");
    multimap.put("Portugal", "Porto");
    multimap.put("Portugal", "Leira");
    multimap.put("Portugal", "Funchal");
    multimap.put("Portugal", "Funchal");


    // then
    assertThat(multimap.get("Poland").size()).isEqualTo(4);
    assertThat(multimap.get("Portugal").size()).isEqualTo(5); // duplicate values are fine
    assertThat(multimap.get("Poland")).contains("Warsaw", "Plock");
    assertThat(multimap.keys().size()).isEqualTo(13); // keys can have duplicates

}
```

import com.google.common.collect.ObjectArrays(1)

- Utility class to operate on arrays of any type
- It allows to concatenate arrays and add single element before first or after last position

import com.google.common.collect.ObjectArrays(2)

- *concat()* – concatenate two arrays

```
String[] array1 = new String[] {"one", "two", "three"};
String[] array2 = new String[] {"four", "five"};

@Test
public void shouldContactTwoArrays() throws Exception {

    // when
    String[] newArray = ObjectArrays.concat(array1, array2, String.class);

    // then
    assertThat(newArray.length).isEqualTo(5);
}
```

import com.google.common.collect.ObjectArrays(3)

- *concat()* – add element to start or end of the array

```
String[] array1 = new String[] {"one", "two", "three"};
String[] array2 = new String[] {"four", "five"};

@Test
public void shouldAppendElement() throws Exception {

    // when
    String[] newArray = ObjectArrays.concat(array2, "six");

    // then
    assertThat(newArray.length).isEqualTo(3);
    assertThat(newArray[2]).isEqualTo("six");
}

@Test
public void shouldPrependElement() throws Exception {

    // when
    String[] newArray = ObjectArrays.concat("zero", array1);

    // then
    assertThat(newArray.length).isEqualTo(4);
    assertThat(newArray[0]).isEqualTo("zero");
}
```

import com.google.common.collect.Ranges(1)

- *Ranges* and *Range* classes are used to define ranges and then checking if a given object is contained within defined range
- They work similarly to *Predicate*
- We can define open (2,10), closed <2,10> and mixed <2, 10) ranges

import com.google.common.collect.Ranges(2)

- *contains()* – simple check if element is within a range
- Nice way to replace two *if-s*

```
@Test
public void shouldCheckThatElementIsInRange() throws Exception {

    // given
    Range<Integer> range = Ranges.closed(2, 20);
    Range<Integer> rangeWithRightOpen = Ranges.closedOpen(2, 20);

    // then
    assertThat(range.contains(20)).isTrue();
    assertThat(rangeWithRightOpen.contains(20)).isFalse();
}
```

import com.google.common.collect.Ranges(3)

- *encloses()*, *intersection()*, *span()* – various operations on ranges

```
@Test
public void shouldCheckThatRangeIsEnclosedInAnotherOne() throws Exception {

    // given
    Range<Long> range = Ranges.openClosed(10L, 20L);
    Range<Long> smallerRange = Ranges.closed(10L, 15L);

    // then
    assertThat(range.encloses(smallerRange)).isFalse();
}
```

import com.google.common.collect.Ranges(4)

- *containsAll()* – checks if all elements from collection are contained within a defined range

```
@Test
public void shouldCheckThatAllElementAreInRange() throws Exception {

    // given
    Range<Integer> range = Ranges.closed(2, 20);

    // then
    assertThat(range.containsAll(Lists.newArrayList(3, 4, 5, 6, 7, 8, 9, 10))).isTrue();
}
```

import com.google.common.collect.Ranges(5)

- *asSet()* – we can generate collection of all elements that are contained by defined range

```
@Test
public void shouldGenerateSetOfElementsInRange() throws Exception {

    // given
    Range<Integer> range = Ranges.open(2, 20);

    // when
    Set<Integer> integersInRange = range.asSet(DiscreteDomains.integers());

    // then
    assertThat(integersInRange).contains(3);
    assertThat(integersInRange).contains(19);
    assertThat(integersInRange).excludes(2, 20);
}
```

import com.google.common.collect.Ranges(6)

- *encloseAll()* – generates range that encloses passed list of elements

```
@Test
public void shouldCreateRangeForGivenNumbers() throws Exception {

    // given
    ArrayList<Integer> numbers = Lists.newArrayList(4, 3, 10, 30, 20);

    // when
    Range<Integer> range = Ranges.encloseAll(numbers);

    // then
    assertThat(range.lowerEndpoint()).isEqualTo(3);
    assertThat(range.upperEndpoint()).isEqualTo(30);
}
```

import com.google.common.primitives.Ints(1)

- Util class to work with *int*s and array of *int*s
- primitives.* package contains similar classes for *boolean, float, double*, etc.

import com.google.common.primitives.Ints(2)

- *contains()* oraz *indexOf()* – quick checking for occurence of a given element and finding its position

```
final int[] array2 = new int[] {0, 14, 99};

final int[] array = new int[] {5, 2, 4, -12, 100, 450, 22, 7};

@Test
public void shouldFindGivenNumberInArray() throws Exception {

    assertThat(Ints.contains(array, 22)).isTrue();
}

@Test
public void shouldFindIndexOfGivenNumber() throws Exception {

    assertThat(Ints.indexOf(array, 5)).isEqualTo(0);
}
```

import com.google.common.primitives.Ints(3)

- *concat()* and *join()* – arrays concatenation and conversion to *String* using defined delimeter (similarly to *Joiner* class)

```
final int[] array2 = new int[] {0, 14, 99};

final int[] array = new int[] {5, 2, 4, -12, 100, 450, 22, 7};

@Test
public void shouldConcatArrays() throws Exception {
    assertThat(Ints.concat(array, array2).length).isEqualTo(array.length + array2.length);
}

@Test
public void shouldJoinArrayUsingSeparator() throws Exception {
    assertThat(Ints.join(":", array2)).isEqualTo("0:14:99");
}
```

import com.google.common.primitives.Ints(4)

- *min(), max()* – finding minimum and maximum in the array

```
final int[] array = new int[] {5, 2, 4, -12, 100, 450, 22, 7};

@Test
public void shouldFindMaxAndMinInArray() throws Exception {

   assertThat(Ints.min(array)).isEqualTo(-12);
   assertThat(Ints.max(array)).isEqualTo(450);

}
```

import com.google.common.eventbus.*(1)

- Classes from *eventbus* package can be used as a simple tool to implement *publisher – subscriber* use case
- Besides standard features these classes have some extras:

  - checking if someone is listening for an event of given type

  - events hierarchy

import com.google.common.eventbus.*(2)

- Listener class needs only one additional element, a method with *@Subscribe* annotation. Argument of this method defines what type of event this class is listening for

```
public class EventListener {

    public int lastMessage = 0;

    @Subscribe
    public void listen(OurTestEvent event) {
        lastMessage = event.getMessage();
    }

    public int getLastMessage() {
        return lastMessage;
    }
}
```

import com.google.common.eventbus.*(3)

- There are no additional restrictions to be an event class. It can be even *String* or *Integer.*

```
public class OurTestEvent {

    private final int message;

    public OurTestEvent(int message) {
        this.message = message;
    }

    public int getMessage() {
        return message;
    }

}
```

import com.google.common.eventbus.*(4)

- Event publishing and receiving – a simple example

```
@Test
public void shouldReceiveEvent() throws Exception {

    // given
    EventBus eventBus = new EventBus("test");
    EventListener listener = new EventListener();

    eventBus.register(listener);

    // when
    eventBus.post(new OurTestEvent(200));

    // then
    assertThat(listener.getLastMessage()).isEqualTo(200);
}
```

import com.google.common.eventbus.*(5)

- One listener can subscribe (listen for) more than one type of event

```
public class MultipleListener {

    public Integer lastInteger;
    public Long lastLong;

    @Subscribe
    public void listenInteger(Integer event) {
        lastInteger = event;
    }

    @Subscribe
    public void listenLong(Long event) {
        lastLong = event;
    }

    public Integer getLastInteger() {
        return lastInteger;
    }

    public Long getLastLong() {
        return lastLong;
    }
}
```

import com.google.common.eventbus.*(6)

- One listener can subscribe (listen for) more than one type of event (continued)

```
@Test
public void shouldReceiveMultipleEvents() throws Exception {

    // given
    EventBus eventBus = new EventBus("test");
    MultipleListener multiListener = new MultipleListener();

    eventBus.register(multiListener);


    // when
    eventBus.post(new Integer(100));
    eventBus.post(new Long(800));

    // then
    assertThat(multiListener.getLastInteger()).isEqualTo(100);
    assertThat(multiListener.getLastLong()).isEqualTo(800L);
}
```

import com.google.common.eventbus.*(7)

- To ensure that every event type has at least one listener we could use pre-defined *DeadEvent* which is fired when *EventBus* receives an event without any listener.
- In such situation we can, for example, log warning message to log file

```
public class DeadEventListener {

    boolean notDelivered = false;

    @Subscribe
    public void listen(DeadEvent event) {
        notDelivered = true;
    }

    public boolean isNotDelivered() {
        return notDelivered;
    }
}
```

import com.google.common.eventbus.*(8)

- Example using *DeadEvent* and *EventBus* without listener for *OurTestEvent*

```
@Test
public void shouldDetectEventWithoutListeners() throws Exception {

    // given
    EventBus eventBus = new EventBus("test");

    DeadEventListener deadEventListener = new DeadEventListener();
    eventBus.register(deadEventListener);

    // when
    eventBus.post(new OurTestEvent(200));

    assertThat(deadEventListener.isNotDelivered()).isTrue();
}
```

import com.google.common.eventbus.*(9)

- Events hierarchy and one listener waiting for concrete events (*Integers*) and second listening for more generic ones (*Numbers*)

```
@Test
public void shouldGetEventsFromSubclass() throws Exception {

    // given
    EventBus eventBus = new EventBus("test");
    IntegerListener integerListener = new IntegerListener();
    NumberListener numberListener = new NumberListener();
    eventBus.register(integerListener);
    eventBus.register(numberListener);

    // when
    eventBus.post(new Integer(100));

    // then
    assertThat(integerListener.getLastMessage()).isEqualTo(100);
    assertThat(numberListener.getLastMessage()).isEqualTo(100);

    //when
    eventBus.post(new Long(200L));

    // then
    // this one should has the old value as it listens only for Integers
    assertThat(integerListener.getLastMessage()).isEqualTo(100);
    assertThat(numberListener.getLastMessage()).isEqualTo(200L);
}
```

import com.google.common.math.*(1)

- Classes from *math* package provides methods to perform some methematical calculations
- Available classes: *IntMath, LongMath, DoubleMath* and *BigIntegerMath*
- They saves developer from being affected by ”silent overflow” when for example adding numbers close to the *Integer.MAX_INT*

import com.google.common.math.*(2)

- Exception is thrown when we reach max allowed value for a given type

```
@Test(expectedExceptions = ArithmeticException.class, expectedExceptionsMessageRegExp = "overflow")
public void shouldThrowExceptionWhenOverflow() throws Exception {

    // given
    int numberOne = Integer.MAX_VALUE - 4;
    int numberTwo = 6;

    // when
    int resultOldWay = numberOne + numberTwo;

    // silent overflow here
    assertThat(resultOldWay).isEqualTo(Integer.MIN_VALUE + 1);

    int result = IntMath.checkedAdd(numberOne, numberTwo);
}
```

import com.google.common.math.*(3)

- *factorial(), gcd(), log2(), log10()* – methods to calculate factorial, greatest common divisor and most popular logarithms

```
@Test
public void shouldCalculateFactorial() throws Exception {
    // when
    int factorial = IntMath.factorial(5);

    // then
    assertThat(factorial).isEqualTo(1 * 2 * 3 * 4 * 5);
}

@Test
public void shouldCalculateGreatestCommonDivisor() throws Exception {
    // when
    int gcd = IntMath.gcd(20, 15);

    // then
    assertThat(gcd).isEqualTo(5);
}

@Test
public void shouldCalculateLogarithms() throws Exception {

    // when
    int log2Result = IntMath.log2(17, RoundingMode.HALF_UP);
    int log10Result = IntMath.log10(100, RoundingMode.UNNECESSARY);

    // then
    assertThat(log2Result).isEqualTo(4);
    assertThat(log10Result).isEqualTo(2);
}
```

import com.google.common.math.*(4)

- Rounding mode can be easily defined when needed for some calculations e.g. *divide()*, *sqrt()* and logarithms

```
@Test
public void shouldDivideWithRoundingMode() throws Exception {

    // when
    int roundedUp = IntMath.divide(10, 4, RoundingMode.HALF_UP);
    int roundedDown = IntMath.divide(10, 4, RoundingMode.HALF_DOWN);

    // then
    assertThat(roundedUp).isEqualTo(3);
    assertThat(roundedDown).isEqualTo(2);
}
```

The End

Thank you for reading!

Source code available on GitHub:
https://github.com/tdziurko/Guava-Lessons

Blog: http://tomaszdziurko.pl