

Советы и секреты

Краткие решения и подсказки

- [Collections](#)
- [Maven](#)
- [Regex](#)
- [Streams](#)

Collections

Java Collections Cheat Sheet

For more awesome cheat sheets
visit rebellabs.org!



Notable Java collections libraries

Fastutil

<http://fastutil.di.unimi.it/>

Fast & compact type-specific collections for Java
Great default choice for collections of primitive types, like int or long. Also handles big collections with more than 2^{31} elements well.

Guava

<https://github.com/google/guava>

Google Core Libraries for Java 6+

Perhaps the default collection library for Java projects. Contains a magnitude of convenient methods for creating collection, like fluent builders, as well as advanced collection types.

Eclipse Collections

<https://www.eclipse.org/collections/>

Features you want with the collections you need
Previously known as gs-collections, this library includes almost any collection you might need: primitive type collections, multimaps, bidirectional maps and so on.

JCTools

<https://github.com/JCTools/JCTools>

Java Concurrency Tools for the JVM.

If you work on high throughput concurrent applications and need a way to increase your performance, check out JCTools.

What can your collection do for you?

Collection class	Thread-safe alternative	Your data				Operations on your collections						
		Individual elements	Key-value pairs	Duplicate element support	Primitive support	Order of iteration			Performant 'contains' check	Random access		
						FIFO	Sorted	LIFO		By key	By value	By index
HashMap	ConcurrentHashMap	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
HashMap (Guava)	Maps.synchronizedBIMap (new HashMap())	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
ArrayListMultimap (Guava)	Maps.synchronizedMultiMap (new ArrayListMultimap())	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗
LinkedHashMap	Collections.synchronizedMap (new LinkedHashMap())	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗
TreeMap	ConcurrentSkipListMap	✗	✓	✗	✗	✗	✓	✗	✓*	✓*	✗	✗
Int2IntMap (Fastutil)		✗	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓
ArrayList	CopyOnWriteArrayList	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✓
HashSet	Collections.newSetFromMap (new ConcurrentHashMap<>())	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗
IntArrayList (Fastutil)		✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✓
PriorityQueue	PriorityBlockingQueue	✓	✗	✓	✗	✗	✓**	✗	✗	✗	✗	✗
ArrayDeque	ArrayBlockingQueue	✓	✗	✓	✗	✓**	✗	✓**	✗	✗	✗	✗

* $O(\log(n))$ complexity, while all others are $O(1)$ - constant time

** when using Queue interface methods: `offer()` / `poll()`

How fast are your collections?

Collection class	Random access by index / key	Search / Contains	Insert
ArrayList	$O(1)$	$O(n)$	$O(n)$
HashSet	$O(1)$	$O(1)$	$O(1)$
HashMap	$O(1)$	$O(1)$	$O(1)$
TreeMap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Remember, not all operations are equally fast. Here's a reminder of how to treat the Big-O complexity notation:

$O(1)$ - constant time, really fast, doesn't depend on the size of your collection

$O(\log(n))$ - pretty fast, your collection size has to be extreme to notice a performance impact

$O(n)$ - linear to your collection size: the larger your collection is, the slower your operations will be

BROUGHT TO YOU BY
JRebel

Maven

maven cheat sheet

For more awesome cheat sheets
visit rebellabs.org!



Getting started with Maven

Create Java project

```
mvn archetype:generate
-DgroupId=org.yourcompany.project
-DartifactId=application
```

Create web project

```
mvn archetype:generate
-DgroupId=org.yourcompany.project
-DartifactId=application
-DarchetypeArtifactId=maven-archetype-webapp
```

Create archetype from existing project

```
mvn archetype:create-from-project
```

Main phases

clean — delete target directory
validate — validate, if the project is correct
compile — compile source code, classes stored in target/classes
test — run tests
package — take the compiled code and package it in its distributable format, e.g. JAR, WAR
verify — run any checks to verify the package is valid and meets quality criteria
install — install the package into the local repository
deploy — copies the final package to the remote repository

Useful command line options

-DskipTests=true compiles the tests, but skips running them
-Dmaven.test.skip=true skips compiling the tests and does not run them
-T - number of threads:
 -T 4 is a decent default
 -T 2C - 2 threads per CPU
-rf, --resume-from resume build from the specified project
-pl, --projects makes Maven build only specified modules and not the whole project
-am, --also-make makes Maven figure out what modules our target depends on and build them too
-o, --offline work offline
-X, --debug enable debug output
-P, --activate-profiles comma-delimited list of profiles to activate
-U, --update-snapshots forces a check for updated dependencies on remote repositories
-ff, --fail-fast stop at first failure

Essential plugins

Help plugin — used to get relative information about a project or the system.

mvn help:describe describes the attributes of a plugin
mvn help:effective-pom displays the effective POM as an XML for the current build, with the active profiles factored in.

Dependency plugin — provides the capability to manipulate artifacts.

mvn dependency:analyze analyzes the dependencies of this project

mvn dependency:tree prints a tree of dependencies

Compiler plugin — compiles your java code.

Set language level with the following configuration:

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.6.1</version>
<configuration>
  <source>1.8</source>
  <target>1.8</target>
</configuration>
</plugin>
```

Version plugin — used when you want to manage the versions of artifacts in a project's POM.

Wrapper plugin — an easy way to ensure a user of your Maven build has everything that is necessary.

Spring Boot plugin — compiles your Spring Boot app, build an executable fat jar.

Exec — amazing general purpose plugin,

The big picture



BROUGHT TO YOU BY
JRebel

Regex

Regex cheat sheet

For more awesome cheat sheets
visit rebellabs.org!



Character classes

[abc] matches **a** or **b**, or **c**.
[^abc] negation, matches everything except **a**, **b**, or **c**.
[a-c] range, matches **a** or **b**, or **c**.
[a-c[f-h]] union, matches **a**, **b**, **c**, **f**, **g**, **h**.
[a-c&&[b-c]] intersection, matches **b** or **c**.
[a-c&&[^b-c]] subtraction, matches **a**.

Predefined character classes

. Any character.
\d A digit: **[0-9]**
\D A non-digit: **[^0-9]**
\s A whitespace character: **[\t\n\r\b\f\z]**
\S A non-whitespace character: **[^\s]**
\w A word character: **[a-zA-Z_0-9]**
\W A non-word character: **[^\w]**

Boundary matches

^ The beginning of a line.
\$ The end of a line.
\b A word boundary.
\B A non-word boundary.
\A The beginning of the input.
\G The end of the previous match.
\Z The end of the input but for the final terminator, if any.
\z The end of the input.

Pattern flags

Pattern.CASE_INSENSITIVE - enables case-insensitive matching.
Pattern.COMMENTS - whitespace and comments starting with **#** are ignored until the end of a line.
Pattern.MULTILINE - one expression can match multiple lines.
Pattern.UNIX_LINES - only the **\n** line terminator is recognized in the behavior of **.**, **\A**, and **\$**.

Useful Java classes & methods

PATTERN

A pattern is a compiler representation of a regular expression.

Pattern.compile(String regex)

Compiles the given regular expression into a pattern.

Pattern.compile(String regex, int flags)

Compiles the given regular expression into a pattern with the given flags.

boolean matches(String regex)

Tells whether or not this string matches the given regular expression.

String[] split(CharSequence input)

Splits the given input sequence around matches of this pattern.

String.quote(String s)

Returns a literal pattern String for the specified String.

Predicate<String> asPredicate()

Creates a predicate which can be used to match a string.

MATCHER

An engine that performs match operations on a character sequence by interpreting a Pattern.

boolean matches()

Attempts to match the entire region against the pattern.

boolean find()

Attempts to find the next subsequence of the input sequence that matches the pattern.

int start()

Returns the start index of the previous match.

int end()

Returns the offset after the last character matched.

Quantifiers

Greedy	Reluctant	Possessive	Description
X?	X??	X?+	X, once or not at all.
X*	X*?	X*+	X, zero or more times.
X+	X+?	X++	X, one or more times.
X{n}	X{n}?	X{n}+	X, exactly n times.
X{n,}	X{n,}?	X{n,}+	X, at least n times.
X{n,m}	X{n,m}?	X{n,m}+	X, at least n but not more than m times.

Greedy - matches the longest matching group.

Reluctant - matches the shortest group.

Possessive - longest match or bust (no backoff).

Groups & backreferences

A group is a captured subsequence of characters which may be used later in the expression with a backreference.

(...) - defines a group.

\N - refers to a matched group.

(\d\d) - a group of two digits.

(\d\d)/\1 - two digits repeated twice.

\1 - refers to the matched group.

Logical operations

XY X then Y.

X|Y X or Y.



Streams

Java 8 Streams Cheat Sheet

For more awesome cheat sheets
visit rebellabs.org!



Definitions

- ✓ A stream **is** a pipeline of functions that can be evaluated.
- ✓ Streams **can** transform data.
- ✗ A stream **is not** a data structure.
- ✗ Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
<i>map</i>	✓	✗	✓
<i>filter</i>	✗	✓	✓
<i>distinct</i>	✗	✓	✓
<i>sorted</i>	✓	✓	✗
<i>peek</i>	✓	✓	✓

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
    .map(book -> book.getAuthor())
    .filter(author -> author.getAge() >= 50)
    .distinct()
    .limit(15)
    .map(Author::getSurname)
    .map(String::toUpperCase)
    .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()
    .map(Book::getAuthor)
    .filter(a -> a.getGender() == Gender.FEMALE)
    .map(Author::getAge)
    .filter(age -> age < 25)
    .reduce(0, Integer::sum);
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
<i>reduce</i>	concrete type	to cumulate elements
<i>collect</i>	list, map or set	to group elements
<i>forEach</i>	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(
    groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()
    .map(member -> member.getFollowers())
    .flatMap(followers -> followers.stream())
    .collect(toList());
```

Pitfalls

- ✗ Don't update shared mutable variables i.e.

```
List<Book> myList = new ArrayList<>();
library.stream().forEach(e -> myList.add(e));
```
- ✗ Avoid blocking operations when using parallel streams.

BROUGHT TO YOU BY
JRebel