

# Java 8 Streams Cheat Sheet

For more awesome cheat sheets  
visit [rebellabs.org](http://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
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	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.

# Java Collections Cheat Sheet

For more awesome cheat sheets  
visit [rebellabs.org](https://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	✗	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗
HashBiMap (Guava)	Maps.synchronizedBiMap (new HashBiMap())	✗	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗
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



# Git Cheat Sheet

For more awesome cheat sheets  
visit [rebellabs.org](https://rebellabs.org)!



## Create a Repository

From scratch -- Create a new local repository

```
$ git init [project name]
```

Download from an existing repository

```
$ git clone my_url
```

## Observe your Repository

List new or modified files not yet committed

```
$ git status
```

Show the changes to files not yet staged

```
$ git diff
```

Show the changes to staged files

```
$ git diff --cached
```

Show all staged and unstaged file changes

```
$ git diff HEAD
```

Show the changes between two commit ids

```
$ git diff commit1 commit2
```

List the change dates and authors for a file

```
$ git blame [file]
```

Show the file changes for a commit id and/or file

```
$ git show [commit]:[file]
```

Show full change history

```
$ git log
```

Show change history for file/directory including diffs

```
$ git log -p [file/directory]
```

## Working with Branches

List all local branches

```
$ git branch
```

List all branches, local and remote

```
$ git branch -av
```

Switch to a branch, my\_branch, and update working directory

```
$ git checkout my_branch
```

Create a new branch called new\_branch

```
$ git branch new_branch
```

Delete the branch called my\_branch

```
$ git branch -d my_branch
```

Merge branch\_a into branch\_b

```
$ git checkout branch_b
```

```
$ git merge branch_a
```

Tag the current commit

```
$ git tag my_tag
```

## Make a change

Stages the file, ready for commit

```
$ git add [file]
```

Stage all changed files, ready for commit

```
$ git add .
```

Commit all staged files to versioned history

```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history

```
$ git commit -am "commit message"
```

Unstages file, keeping the file changes

```
$ git reset [file]
```

Revert everything to the last commit

```
$ git reset --hard
```

## Synchronize

Get the latest changes from origin (no merge)

```
$ git fetch
```

Fetch the latest changes from origin and merge

```
$ git pull
```

Fetch the latest changes from origin and rebase

```
$ git pull --rebase
```

Push local changes to the origin

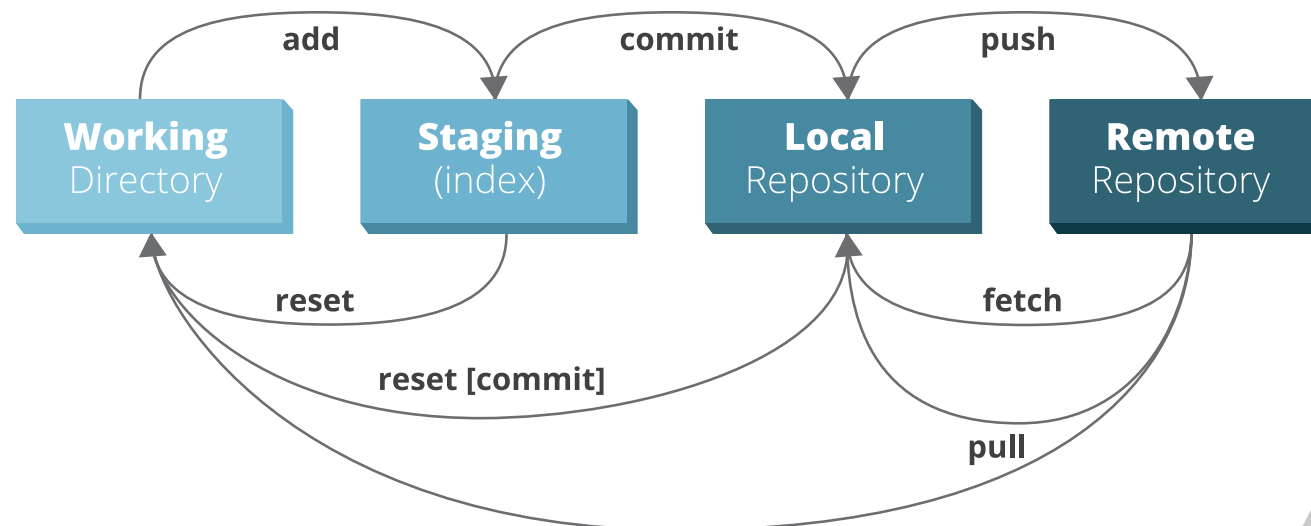
```
$ git push
```

## Finally!

When in doubt, use git help

```
$ git command --help
```

Or visit <https://training.github.com/> for official GitHub training.



# maven cheat sheet

For more awesome cheat sheets  
visit [rebellabs.org](https://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:

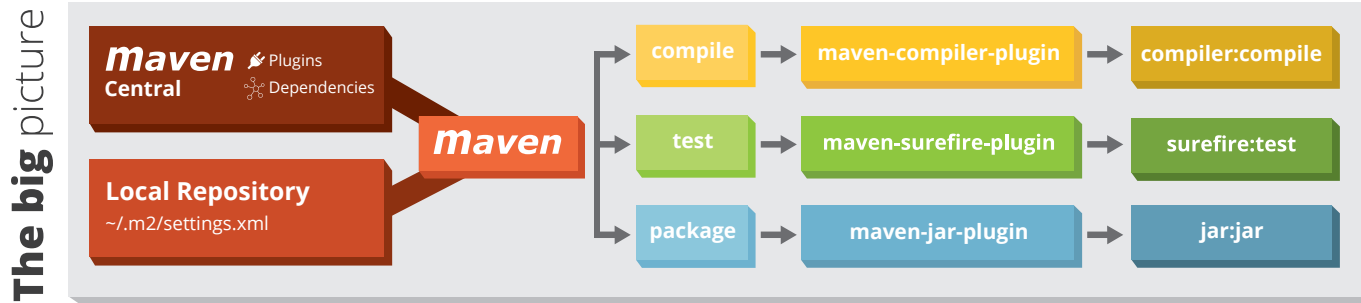
```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.6.1</version>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```

**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,



# SQL cheat sheet

For more awesome cheat sheets  
visit [rebellabs.org](http://rebellabs.org)!



## Basic Queries

- filter your columns  
**SELECT** col1, col2, col3, ... **FROM** table1
- filter the rows  
**WHERE** col4 = 1 **AND** col5 = 2
- aggregate the data  
**GROUP** by ...
- limit aggregated data  
**HAVING** count(\*) > 1
- order of the results  
**ORDER BY** col2

Useful keywords for **SELECTS**:

- DISTINCT** - return unique results
- BETWEEN** a **AND** b - limit the range, the values can be numbers, text, or dates
- LIKE** - pattern search within the column text
- IN** (a, b, c) - check if the value is contained among given.

## Data Modification

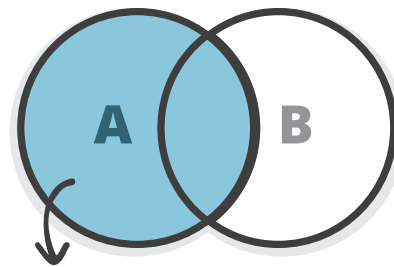
- update specific data with the **WHERE** clause  
**UPDATE** table1 **SET** col1 = 1 **WHERE** col2 = 2
- insert values manually  
**INSERT INTO** table1 (**ID**, **FIRST\_NAME**, **LAST\_NAME**)  
**VALUES** (1, 'Rebel', 'Labs');
- or by using the results of a query  
**INSERT INTO** table1 (**ID**, **FIRST\_NAME**, **LAST\_NAME**)  
**SELECT** id, last\_name, first\_name **FROM** table2

## Views

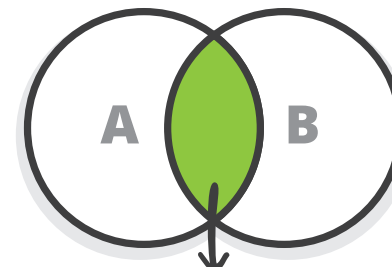
A **VIEW** is a virtual table, which is a result of a query.  
They can be used to create virtual tables of complex queries.

```
CREATE VIEW view1 AS  
SELECT col1, col2  
FROM table1  
WHERE ...
```

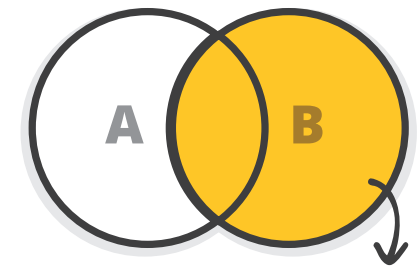
## The Joy of JOINS



**LEFT OUTER JOIN** - all rows from table A,  
even if they do not exist in table B



**INNER JOIN** - fetch the results that  
exist in both tables



**RIGHT OUTER JOIN** - all rows from table B,  
even if they do not exist in table A

## Updates on JOINed Queries

You can use **JOINS** in your **UPDATES**

```
UPDATE t1 SET a = 1  
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.t1_id  
WHERE t1.col1 = 0 AND t2.col2 IS NULL;
```

NB! Use database specific syntax, it might be faster!

## Semi JOINS

You can use subqueries instead of **JOINS**:

```
SELECT col1, col2 FROM table1 WHERE id IN  
(SELECT t1_id FROM table2 WHERE date >  
CURRENT_TIMESTAMP)
```

## Indexes

If you query by a column, index it!

```
CREATE INDEX index1 ON table1 (col1)
```

Don't forget:

Avoid overlapping indexes

Avoid indexing on too many columns

Indexes can speed up **DELETE** and **UPDATE** operations

## Useful Utility Functions

- convert strings to dates:  
**TO\_DATE** (Oracle, PostgreSQL), **STR\_TO\_DATE** (MySQL)
- return the first non-NULL argument:  
**COALESCE** (col1, col2, "default value")
- return current time:  
**CURRENT\_TIMESTAMP**
- compute set operations on two result sets  
**SELECT** col1, col2 **FROM** table1  
**UNION / EXCEPT / INTERSECT**  
**SELECT** col3, col4 **FROM** table2;

*Union* - returns data from both queries

*Except* - rows from the first query that are not present  
in the second query

*Intersect* - rows that are returned from both queries

## Reporting

Use aggregation functions

**COUNT** - return the number of rows

**SUM** - cumulate the values

**AVG** - return the average for the group

**MIN / MAX** - smallest / largest value

# Regex cheat sheet

For more awesome cheat sheets  
visit [rebellabs.org](https://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\x0B\f\r]`
- `\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 `.`, `^`, 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
<code>X?</code>	<code>X??</code>	<code>X?+</code>	<i>X, once or not at all.</i>
<code>X*</code>	<code>X*?</code>	<code>X*+</code>	<i>X, zero or more times.</i>
<code>X+</code>	<code>X+?</code>	<code>X++</code>	<i>X, one or more times.</i>
<code>X{n}</code>	<code>X{n}?</code>	<code>X{n}+</code>	<i>X, exactly n times.</i>
<code>X{n,}</code>	<code>X{n,}?</code>	<code>X{n,}+</code>	<i>X, at least n times.</i>
<code>X{n,m}</code>	<code>X{n,m}?</code>	<code>X{n,m}+</code>	<i>X, at least n but not more than m times.</i>

**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`.