# Sources location

Doclet sources currently located here: <https://github.com/v-anpunko/docfx-doclet>

# Build prerequisites

OpenJDK 11+

Maven 3.6.0+

# Build

Use next command to build: *maven clean install*

As a result of build in addition to usual jar with doclet, with help of the *maven-assembly-plugin* we make fat jar, which contains all dependent libraries, so result jars names will be something like next:

* *docfx-doclet-1.0-SNAPSHOT.jar*
* *docfx-doclet-1.0-SNAPSHOT-jar-with-dependencies.jar*

Both jars have next *groupId* and *artifactId* values:

<**groupId**>com.microsoft</**groupId**>  
<**artifactId**>docfx-doclet<**artifactId**>

The *maven-compiler-plugin* used for adjustment of compiler configuration to show more details of warnings.

The *jacoco-maven-plugin* used to build test coverage report. Check it at: *target/site/jacoco/index.html*

# Doclet usage

Before description of ways of doclet usage we need to describe its params.

## Doclet command line params

Doclet support next command line params:

**-outputpath** Path where generated yml files should be placed. Mandatory param

For example: *./test-out*

**-excludepackages** List of ecluded packages with ‘:’ delimeter. Regex supported. Not mandatory param

For example: *com\.msdn\..\*:com\.ms\.news\..\**

**-excludeclasses** List of excluded classes with ‘:’ delimeter. Regex supported. Not mandatory param

For example: *.\*SomeClass:com\.ms\..\*AnyClass*

Aforementioned params supported by doclet. But to configure *javadoc* tool we need to note about others useful params of *javadoc* tool

## Useful javadoc command line params

**-encoding**  Encoding of source files

(For example when we faced with issue with commons-lang3-3.8.1-sources it was useful to use UTF-8 encoding to resolve it)

**-docletpath** Path to jar with doclet

**-doclet** Name of doclet class

**-cp** Specify where to find user class files (jars with dependencies)

Need to note that delimeter here could be platform-dependent as I remember

**-sourcepath** Location of jar with sources

**-subpackages** Specify subpackages to recursively load

So, using *javadoc* tool with our doclet as param we could generate docfx files.

We could use doclet in different ways – with help of maven plugin or in stanalone mode.

## Usage with help of maven-javadoc-plugin

When we have existing java project where maven used as a build tool, we could add *maven-javadoc-plugin* to root *pom.xml*:

<**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-javadoc-plugin</**artifactId**>  
 <**version**>3.0.1</**version**>  
 <**configuration**>  
 <**doclet**>com.microsoft.doclet.DocFxDoclet</**doclet**>  
 <**docletArtifact**>  
 <**groupId**>com.microsoft</**groupId**>  
 <**artifactId**>docfx-doclet<**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**docletArtifact**>  
 <**useStandardDocletOptions**>false</**useStandardDocletOptions**>  
 <**additionalOptions**>-outputpath ./generated-files</**additionalOptions**>

<!-- Add additional options here when needed -->  
 </**configuration**>  
</**plugin**>

And run docs generation by command: *mvn javadoc:javadoc*

## Standalone usage (preferable for me as more generic)

When we have jar with sources we need to unpack it, and download its compile time dependencies too. After that run javadoc tool with command-line params next way:

javadoc \  
-encoding <encoding of source files> \  
-docletpath <path to fat doclet jar (with its dependencies)> \  
-doclet com.microsoft.doclet.DocFxDoclet \  
-classpath <list of jars with dependencies> \  
-sourcepath <path to sources jar> \  
-outputpath <path to output folder> \

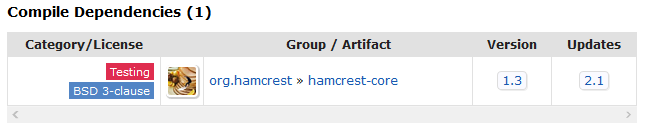
-excludepackages <list of excluded packages> \

-excludeclasses <list of excluded classes> \

-subpackages <subpackages to recursively load>

For example, for *JUnit-4.12* library on this page of Maven Central we see that it has one compile time dependency on *hamcrest-core-1.3* library:

<https://mvnrepository.com/artifact/junit/junit/4.12>



So, need to download *hamcrest-core-1.3.jar* and put it on classpath during run of Javadoc command.

For example for JUnit sources we used next .bat script on Win OS:

**set SOURCES\_FOLDER**=**"./unpacked-sources"  
set GENERATED\_FOLDER**=**"./generated-yml-files"  
  
echo "Remove folders"**rmdir /S /Q %SOURCES\_FOLDER%  
rmdir /S /Q %GENERATED\_FOLDER%  
  
**echo "Create directory to extract sources"**mkdir %SOURCES\_FOLDER%  
  
**echo "Extract sources from jar file"**pushd %SOURCES\_FOLDER%  
jar xf ../libs/junit-4.12-sources.jar  
popd  
  
**echo "Generate yml files"**javadoc ^  
-encoding UTF-8 ^  
-docletpath ../target/docfx-doclet-1.0-SNAPSHOT-jar-with-dependencies.jar ^  
-doclet com.microsoft.doclet.DocFxDoclet ^  
-cp ./libs/hamcrest-core-1.3.jar ^  
-sourcepath %SOURCES\_FOLDER% ^  
-outputpath %GENERATED\_FOLDER% ^  
-subpackages org:junit

Here both *junit-4.12-sources.jar* and *hamcrest-core-1.3.jar* were at *lib* folder.

In result generated files placed to *generated-yml-files* folder.

In similar way script for another OS could be created.

# Serve DocFx documentation

* Install *Chocolately* using [its instructions](https://chocolatey.org/docs/installation)
* Install *docfx* through *Chocolatey*: *choco install docfx -y*
* Generate set of yml files by java sources using one of aforementioned ways
* Start DocFx web-server: *docfx docfx.json --serve*
* Connect to <http://localhost:8080> to see how it looks like

# Appendix I. Doclet development

For doclet development it could be useful to use *DocletRunner* class. It makes javadoc call (as described in the part of standalone doclet usage) and takes parameters from external text file. So for that:

* Create Run/Debug configuration in IDE with main class *com.microsoft.doclet.DocletRunner*
* Add *src\test\resources\test-doclet-params.txt* as program arguments of configuration

Now we could run/debug doclet versus source code of classes located at *com.microsoft.samples*   
package as mentioned in *test-doclet-params.txt* config file. Or you could adjust text file content to run doclet versus your own sources.

Run of *DocletRunner* versus test sources from *com.microsoft.samples* package used as integration test when generated content compared to expected files (see *DocletRunnerTest* for details).

In addition we added *./sandbox* folder with required scripts and libs as playground to generate yml-files and serve DocFx documentation for JUnit sources.

# Appendix II. Description of how doclet works

There are two basic versions of doclet API exist in java world, and in Java 9 it was significantly reworked to make it consistent.

We used latest doclet API.

Class *DocFxDoclet* which implements *Doclet* interface is entry point during call of doclet. Its method *getSupportedOptions()* describes options supported by doclet.

After doclet call instance of *DocletEnvironment* class passed to run() method of *DocFxDoclet*. So instance of *DocletEnvironment* is container with info about enclosing elements from which essential information could be extracted. We make it by next steps:

* Retrieve information about packages except packages which we should skip according to appropriate doclet param.

After what

* For each package retrieve its classes and inner classes recursively except classes that we should skip according to appropriate doclet param. In process of work information about packages/classes aggregated into instances of metadata classes: instance per package/class. For that purpose we create set of ‘model’ classes which represent how data will be looks like in yml files after serializarion. By using some annotations on model classes and their fields we make generated ymls look like according to our expectations.

To avoid calculation of metadata fields content multiple times, we realized cache using several lookups in which items identified by uid.

* In background short info about all items (both packages and classes) aggregated into separate instance of metadata class, so it’s responsible for ‘table of content’.

After when previous steps finished we are ready to serialize metadata classes to yml files but need to make additional postprocessing steps, using information of all generated staff. So we need to resolve names collision and populate uid values of xref tags.

* Resolve names collision when some package contains subpackage and class with same names, except casing, for example *org.junit.runner* package and *org.junit.Runner* class. To prevent creation of yml files with overlapping names we add suffixes (package) and (class) to their names appropriately.
* Populate uid attribute of xref tags. We made it in two steps: first build two layer lookups for that (local – lookup per item and global lookup – which contains all lookups info) and after that - perform population using lookups priority: try to resolve uid value using local lookup, and in case of data absence – resolve by global lookup. The reason of do it this way is multiple cases of how @link/@linkplain items could be written in Javadoc comment.

During uids population using Remark lib we perform conversion of summary fields from html to markdown with addition of ‘java’ prefix when needed to highlight <code> and @code blocks.

And finally

* Serialize all instances of metadata classes to yml files with help of Jackson. Create required output folders when needed during save of files to disk.