(9) Java Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Name Clashes – Part I

- The idea of oo programming leads of course to a lot of types.
- Additionally, our own code also uses other libraries, often not only from the JDK, but 3rd party libraries.
- And our library will be used by yet other code.
- Sooner or later, we or others will get into trouble, because names of types can be reused independently.
  - Assume following situation, in which we defined our own class *System* and use the library commons-lang3-3.1.jar.
  - We use *ArrayUtils.isEmpty()* from the package *org.apache.commons.lang3* of library commons-lang3-3.1.jar:

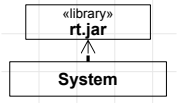
```
// <System.java>
import org.apache.commons.lang3.*;

public class System {
        public static void main(String[] args) {
            if (ArrayUtils.isEmpty(args)) {
                System.out.println("You must pass some arguments!"); // Invalid! cannot find symbol 'out'
            }
        }
}
```

  - However, this code won't compile! – What is going on here?

Name Clashes – Part II

- The problem is, that the compiler assumes, that our class *System* offers the field *out*!
  - Sure, this is incorrect, rather *java.lang.System* offers the field *out*.

- What we have here is a name clash: we have two types with the same name, while the compiler chooses the wrong one.
  - The UML offers a way to represent the scenario in a class diagram with the «library» stereotype:



- Actually, we can easily select the correct *System* type for the compiler, by fully qualifying its name with the package name:

```
// <System.java>

public class System {
        public static void main(String[] args) {
            if (0 == args.length) {
                java.lang.System.out.println("You must pass some arguments!");
            }
        }
}
```

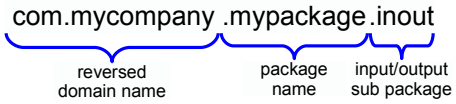
Good to know
Java imports the package *java.lang* into each java-file automatically. However, here we have to fully qualify *java.lang.System* to allow the compiler to tell it from the class *System* in the default package.

- So, Java's types can have the same name, as long as they reside in different packages!

- Esp. Groovy imports more packages than only *java.lang*, e.g. *java.util*, because the types in *java.util* are part of the Groovy language itself.

Packages and Package Names – Part I

- The best way to deal with potential name clashes is to introduce own packages for own projects/libraries consequently.
  - This allows maximal control and flexibility for other developers and us as library provider!
- So, we already unleashed the fact, that packages are used to group types and "shield" them from each other.
- Sure, in the beginning we have to think about naming our package.
  - Of course there are also conventions in Java, how to name packages.
- The usual naming is to use the reversed domain name as root, then the project name and finally other aspects.



  - The names are written in all lower case letters.
  - Esp. if the domain name has invalid letters for Java symbols, it should be replaced with '_'.
  - If there are name clashes within a company, often substructures can agreed upon, e.g. the region as part in the name.
  - If segments of the package names are Java keywords or start with a digit, they should be prefixed with '_'.
  - packages are not allowed to begin with java or javax, those prefixes are reserved for the JDK.

Packages and Package Names – Part II

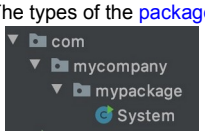
- Let's use the package name *com.mycompany.mypackage* for our discussion.

```
// <System.java>
package com.mycompany.mypackage;

import java.util.*;

public class System {
	public static void main(String[] args) {
		if (0 == args.length) {
			System.out.println("You must pass some arguments!");
		} else {
			java.lang.System.out.printf("You entered: %s%nPress enter", Arrays.toString(args));
			new Scanner(java.lang.System.in).hasNextLine();
		}
	}
}
```

- As can be seen, the package statement sets the package name. This statement must be the first statement in a java-file!
- However, the code above does not yet compile, the compile time error reads:
  - "Package name 'com.mycompany.mypackage' does not correspond to the file path"
- The answer is, that we have to adapt the structure of our source code files to make the compiler happy!

Packages and Package Names – Part III

- Actually, we have to change the Java project's file structure to match the package name.
  - The types of the package *com.mycompany.mypackage* must reside in the directory com/mycompany/mypackage.



Warning
That the package structure must adhere to the file structure of the file system bears a risk: the packages structure must not lead to exceedingly long file paths!

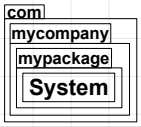
```
// <System.java>
package com.mycompany.mypackage;

import java.util.*;

public class System {
        public static void main(String[] args) {
            if (0 == args.length) {
                System.out.println("You must pass some arguments!");
            } else {
                java.lang.System.out.printf("You entered: %s%nPress enter", Arrays.toString(args));
                new Scanner(java.lang.System.in).hasNextLine();
            }
        }
}
```

- After that change we can compile *System*, or more exactly c*om.mycompany.mypackage.System*:

Terminal

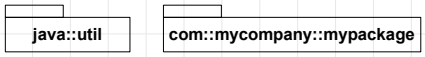
```
NicosMBP:src nico$ javac com/mycompany/mypackage/System.java
NicosMBP:src nico$ java com/mycompany/mypackage/System
You must pass some arguments!
NicosMBP:src
```

- The corresponding class diagram shows the naming structure of the packages nicely:



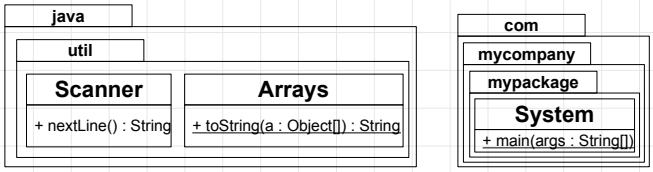
Packages – UML Notations – Part I

- A package can be represented as folder symbol, that shows the package name or fully qualified path within the box.

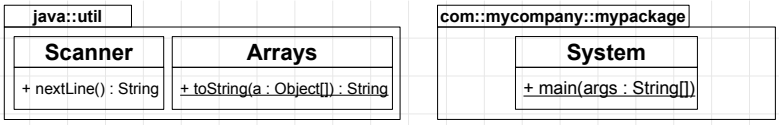


  - Also connectors (i.e. lines with or without arrows) can be drawn between package folders.

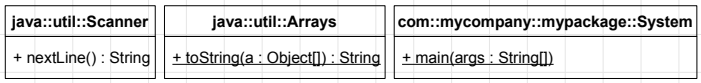
- We also draw packages and sub packages as cascades symbols:
  - If a package symbol contains other symbols (classes or sub packages), the UML standard suggests to write package names into the label of the package folder symbol.



- Alternatively, the cascade can be condensed showing the package path in the label of an "aggregating package":



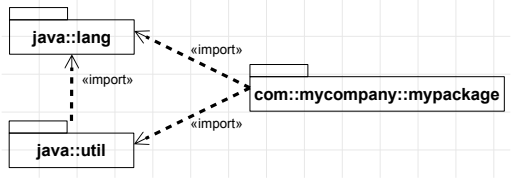
- And we can just use fully qualified class names:



Packages – UML Notations – Part II

```
// <System.java>
package com.mycompany.mypackage;

import java.util.*;

public class System {
        public static void main(String[] args) {
            if (0 == args.length) {
                System.out.println("You must pass some arguments!");
            } else {
                java.lang.System.out.printf("You entered: %s%nPress enter", Arrays.toString(args));
                new Scanner(java.lang.System.in).hasNextLine();
            }
        }
}
```

- This code can also be abstracted into a package view, i.e. the very 20 miles perspective showing package dependencies:



- Mind, that we explicitly mention the package *java.lang* here, although it is automatically imported into all java-files

Visibility of Classes among Packages

- Although package's and sub package's names have names suggesting a hierarchical access allowance, this not the case.
  - Consider following case, although *ClassB* is in a sub package of *ClassA*, *ClassB* and *ClassA* don't see each other:

```
// <ClassA.java>
package com.mycompany.mypackage;

public class ClassA {
        public void test() {
                ClassB classB = new ClassB();
        }
}
```

```
// <ClassB.java>
package com.mycompany.mypackage.utils;

public class ClassB {
        public void test() {
                ClassA classA = new ClassA();
        }
}
```

- To make the visible to each other, we have to import each package respectively:

```
// <ClassA.java>
package com.mycompany.mypackage;

import com.mycompany.mypackage.utils.*;

public class ClassA {
        public void test() {
                ClassB classB = new ClassB();
        }
}
```

```
// <ClassB.java>
package com.mycompany.mypackage.utils;

import com.mycompany.mypackage.*;

public class ClassB {
        public void test() {
                ClassA classA = new ClassA();
        }
}
```

- Similar issue: classes in the default package cannot be imported into a "named" package!
- Take away: also sub packages of packages are completely different logical packages for Java!

Java Archives (JARs) as Libraries

- packages are just logical groupings of types in Java, but not yet "real" libraries.
  - In programming, a library is usually a file, more formally, a physical artifact.
  - A library can be versioned, given to other developers and tested in isolation, we say it is a deployment unit.

- As a matter of fact, we have already used other physical artifacts in Java, namely class-files.

- The handling of multiple class-files is cumbersome, therefor we can aggregate class-files into a Java archive (JAR).
  - JARs can be used to define libraries of class-files and those libraries can then be easier deployed than class-files.

- For our further discussion, assume, we have following two classes:

```
// <System.java>
package com.mycompany.mypackage;
// pass
```

```
// <Algorithms.java>
package com.mycompany.mypackage.utils;
// pass
```

  - We assume the class *com.mycompany.mypackage.System* as we have already discussed it.
  - The new package *com.mycompany.mypackage.utils* contains the class *Algorithms*, with some helper methods.
  - Our task is to put these classes into a common library, i.e. a JAR.

Creating JARs

- A JAR is just a ZIP-file containing the class-files to be aggregated into a library:

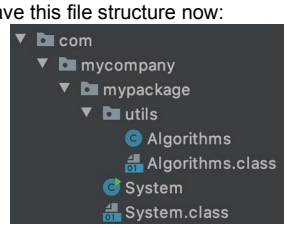
```
// <System.java>
package com.mycompany.mypackage;
// pass
```

```
// <Algorithms.java>
package com.mycompany.mypackage.utils;
// pass
```

compilation → <System.class> <Algorithms.class> → archiving →

```
com/
    mycompany/
        mypackage/
            <System.class>
            utils/
                <Algorithms.class>
<MyLibrary.jar>
```

- Practically, class-files are packed into a JAR with the jar-tool on the command line, which is part of the JDK:
  - Assume, we compiled the java-files and have this file structure now:



Good to know
Originally, JARs were introduced to support Java applets. Java applets are "small" Java applications, that are deployed via the web and executed in a web browser. The idea was, that downloading a single compact JAR via the web is cheaper than downloading multiple class-files via the web.

  - We use the command **jar cf MyLibrary.jar [class-files]** to add class-files into the jar:

```
Terminal
NicosMBP:src nico$ jar cf MyLibrary.jar com/mycompany/mypackage/System.class com/mycompany/mypackage/utils/Algorithms.class
NicosMBP:src nico$ ls
MyLibrary.jar com
NicosMBP:src nico$
```

13

- The jar-tool uses command line switches like the program "tar" (Tape Archiver), but it creates a ZIP-file!
  - So, instead of using the jar-tool, we could also just zip the class-files and package folders of a Java library to a zip-file.
  - However, jars are usually the better target-file-type, i.e. using the jar-extension instead of the zip-extension.

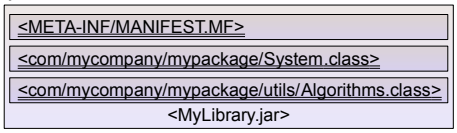
Inspecting and extracting JARs

- We can also have a look into the jar-file, with the jar tool's **t**-option, which shows the **t**able of contents.
  - We issue the command **jar tvf MyLibrary.jar** on the console:
  - (The option **v** prints verbose information (size and creation date) of each contained file to the console.)

Terminal

```
NicosMBP:src nico$ jar tvf MyLibrary.jar
     0 Wed Dec 25 23:20:46 CET 2019 META-INF/
    65 Wed Dec 25 23:20:46 CET 2019 META-INF/MANIFEST.MF
   870 Tue Dec 24 21:46:54 CET 2019 com/mycompany/mypackage/System.class
   224 Wed Dec 25 10:41:42 CET 2019 com/mycompany/mypackage/utils/Algorithms.class
NicosMBP:src nico$
```

- Obviously, MyLibrary.jar contains not only class-files, but also the directory META-INF, which contains the file MANIFEST.MF.
  - We'll discuss those files in short, they basically contain common information about the JAR. Actually MyLibrary.jar looks like so:



- We can also extract the files contained in a jar-file using the jar-tool's **x**-option.
  - We issue the command **jar xvf MyLibrary.jar** on the console:
  - As can be seen, we did a little bit more than only the **x**-option:
    - (1) We create a new directory *extracted* and cd'd into it.
    - (2) We issue **jar xvf ../MyLibrary.jar** to extract the jar's content from the parent directory into the current directory extracted.
    - (3) The option **v** yields verbose output of the operation to STDOUT.

Terminal

```
NicosMBP:extracted nico$ jar xvf ../MyLibrary.jar
  created: META-INF/
 expanded: META-INF/MANIFEST.MF
 expanded: com/mycompany/mypackage/System.class
 expanded: com/mycompany/mypackage/utils/Algorithms.class
NicosMBP:extracted nico$
```

Adding Resources to JARs

- We can also add other files into a JAR:
  - E.g. the source code of the compiled classes besides the class-files.
    - Open source projects often create a separate JAR with sources.
  - Other resource-files, e.g. properties-files, which contain data, e.g. to configure libraries or javadoc HTML pages:

```
# <app.properties>
t=42
```

- Property-files are simple text files, that contain lines of key=value pairs.
  - Each key=value pair represents an individual property.
  - The values of the properties can then be used to configure executables/programs/libraries.

- To do this, we use the jar-tool with **jar uf MyLibrary.jar app.properties** to add the properties-file into the JAR.

```
Terminal
NicosMBP:src nico$ jar uf MyLibrary.jar app.properties
NicosMBP:src nico$
```


- In principle, we could add other jar-files into a JAR, but Java provides no means to load other JARs from within a JAR. To do something like this, a special class loader must be used or programmed.

Reading Property-Resource-Files of a JAR

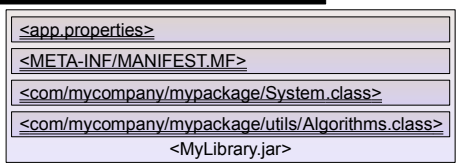
- Next, we want to change the code in *System.main()* to read app.properties from within the JAR:

```
// <System.java>
public class System {
        public static void main(String[] args) {
            String textFromResource = new Scanner(System.class.getClassLoader().getResourceAsStream("app.properties")).nextLine();
            java.lang.System.out.printf("Hello from the JAR: %s%n", textFromResource);
        }
}
```

  - Compile the changed java-file to a class-file and add it to MyLibrary.jar

```
Terminal
NicosMBP:src nico$ javac com/mycompany/mypackage/System.java
NicosMBP:src nico$ jar uf MyLibrary.jar com/mycompany/mypackage/System.class
NicosMBP:src nico$
```

```
# <app.properties>
t=42
```



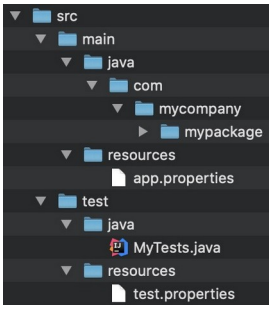
- Executing *System* in MyLibrary.jar prints the first line of app.properties to the console:

```
Terminal
NicosMBP:src nico$ java -cp MyLibrary.jar com/mycompany/mypackage/System
Hello from the JAR: t=42
NicosMBP:src nico$
```

  - The -cp-switch (cp = class path) of the java-interpreter allows to execute a class with *main()*-method in a JAR.

Gradle's Project Structure with Resources

- With the introduction of resources, we can further precise Gradle's Standard Directory Layout:



  - (1) The effective program code should be stored in <projectDirectory>/src/**main**/**<languageName>**.
  - (2) All the resources for the effective program should reside in <projectDirectory>/src/**main**/**resources**.
  - (3) The test code should reside in <projectDirectory>/src/**test**/**<languageName>**.
  - (4) All the resources for the tests should reside in <projectDirectory>/src/**test**/**resources**.

Multiple Classes with main()-Methods

- If one or many compiled classes in a JAR have *main()* methods, we can execute these classes directly from the JAR!

- E.g. with these executable classes in MyAlgorithms.jar:

```
// <System.java>
public class System {
        public static void main(String[] args) {
            // ...
            java.lang.System.out.printf("Hello from the JAR: %s%n", textFromResource);
        }
}
```

```
// <Algorithms.java>
public class Algorithms {
        public static void main(String[] args) {
            System.out.println("Hello from the Algos!");
        }
}
```

  - … we can use the java command with the -cp switch addressing the JAR to execute specific classes in a JAR:

Terminal

```
NicosMBP:src nico$ java -cp MyLibrary.jar com/mycompany/mypackage/System
Hello from the JAR: t=42
NicosMBP:src nico$ java -cp MyLibrary.jar com/mycompany/mypackage/utils/Algorithms
Hello from the Algos!
NicosMBP:src nico$
```

- The -cp switch, the class path switch, is used to specify the directories and archives, in which classes are searched.
  - We will discuss this in more depth soon.

Manifest-Files

- We mentioned that Java supports special resources to be put into a jar-file, which enrich JARs with meta-information.
- JAR meta-information is stored in the directory META-INF, which is packed into the JAR at the top level.
- META-INF can contain different files storing information for different purposes, the most important file is MANIFEST.MF.
  - The jar-tool creates a default-manifest, which looks like this:

<MANIFEST.MF>

```
Manifest-Version: 1.0
Created-By: 1.8.0 (Oracle Corporation)
```

- We can add many additional meta-information into our own MANIFEST.MF, here a selection:

<MANIFEST.MF>

```
Implementation-Title: MyLibrary (Test JAR)
Implementation-Version: 3.14
Implementation-Vendor: MyCompany
```

  - We can just save this file as META-INF/MANIFEST.MF
  - Then we issue **jar uvmf META-INF/MANIFEST.MF MyLibrary.jar** on the command line to add the manifest to MyLibrary.jar.
    - The m-option tells the jar-tool, that we specify our own manifest-file as first file parameter.

The Main-Class of a JAR

- An interesting setting in the manifest is the specification of a main class.

- When we specify a class in a JAR to be the main class, we can directly execute this class from a jar.
  - The manifest's Main-Class property accepts this specification:

<MANIFEST.MF>

```
Manifest-Version: 1.0
Main-Class: com.mycompany.mypackage.System
Implementation-Title: MyLibrary (Test JAR)
Implementation-Version: 3.14
Implementation-Vendor: MyCompany
```

- After the MyLibrary.jar was build with this new manifest, we can execute it directly with java's -jar-option:

Terminal

```
NicosMBP:src nico$ java -jar MyLibrary.jar
Hello from the JAR: t=42
NicosMBP:src nico$
```

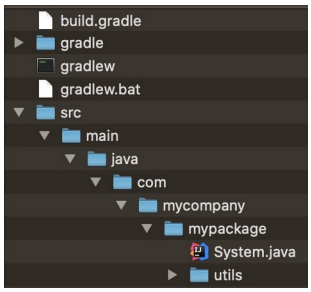
  - A JAR with a Main-Class is sometimes called executable JAR.

- The Main-Class specification for a JAR is an alternative to using the java command with the -cp-option.
  - The difference is, that we can only have one Main-Class, but with the -cp-option we can address each class with a *main()*-method.

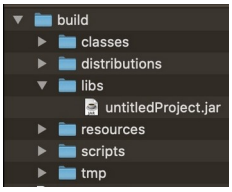
Creating JARs with Gradle – Part I

- By default, Gradle creates a JAR from a Java project, which carries the name of the build.gradle-file's directory.
  - Let's assume *System* and *Algorithms* incl. their package structure are represented in a Gradle project in the directory untitledProject:

```
// <build.gradle>
apply plugin: 'java'
```



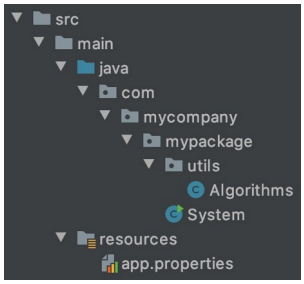
- With Gradle's default configuration, and running *gradlew build*, we get untitledProject.jar in directory build/lib:



Creating JARs with Gradle – Part II

- We have to do some changes to our Gradle configuration to match what we have done before with the jar-tool.
  - Besides the classes *System* and *Algorithms* we also want to add app.properties into the JAR.
  - We should rename the JAR to MyLibrary.jar.

- Resources such as app.properties have to be put into the directory src/main/resources to be processed by Gradle:



<app.properties>
<META-INF/MANIFEST.MF>
<com/mycompany/mypackage/System.class>
<com/mycompany/mypackage/utils/Algorithms.class>
<MyLibrary.jar>

  - By default Gradle puts the file structure under resources at the root of the internal JAR-structure, exactly as we need it!

- As can be seen, Gradle also creates META-INF/MANIFEST.MF for us, but only with default content.

Creating JARs with Gradle – Part III

- With some changes in build.gradle we rename the jar-file to be created and customize the manifest-file:

```
// <build.gradle>
apply plugin: 'java'

archivesBaseName = 'MyLibrary'

jar {
        manifest {
            attributes (
                    'Main-Class' : 'com.mycompany.mypackage.System',
                    'Manifest-Version' : 1.0,
                    'Implementation-Title' : 'MyLibrary (Test JAR)',
                    'Implementation-Version' : 3.14,
                    'Implementation-Vendor' : 'MyCompany'
            )
        }
}
```

  - Setting the property *archivesBaseName* specifies the name of the resulting JAR.
  - The cascaded blocks *jar* and *manifest* contain a call to the Gradle-function *attributes()*.
  - *attributes()* accepts a list of key:value pairs, corresponding to the ones we have seen for manifest-files.

- This build.gradle-file will create MyLibrary.jar, which has the manifest we have had before.

Creating JARs with Gradle – Part IV

- In the beginning of the lecture, we presented code using *ArrayUtils.isEmpty()* of the package *org.apache.commons.lang3*.
  - This package is provided by the library commons-lang3-3.1.jar, so it's not part of the JDK.
    - (Its package name is neither prefixed with java nor javax.)

```
// <System.java>
import org.apache.commons.lang3.*;

public class System {
	public static void main(String[] args) {
		if (ArrayUtils.isEmpty(args)) {
			java.lang.System.out.println("You must pass some arguments!");
		}
	}
}
```

- Gradle provides means to load dependent libraries/jars from a Maven repository:

```
// <build.gradle>
apply plugin: 'java'

archivesBaseName = 'MyLibrary'

repositories {
	mavenCentral()
}
dependencies {
	compile 'org.apache.commons:commons-lang3:3.1'
}
task copyDependencies(type: Copy) {
	from configurations.default
	into "$buildDir/libs"
}
→
```

```
←
jar {
	manifest {
		attributes(
			'Main-Class' : 'com.mycompany.mypackage.System',
			'Class-Path': configurations.runtime.files.collect {it.name}.join(' ')
		)
	}
}
```

- Let's discuss the required changes in build.gradle.

Creating JARs with Gradle – Part V

We have to define a task, which copies the dependent, i.e. downloaded libraries/jars into the output directory of MyLibrary.jar.

```
// <build.gradle>
apply plugin: 'java'

archivesBaseName = 'MyLibrary'

repositories {
        mavenCentral()
}
dependencies {
        compile 'org.apache.commons:commons-lang3:3.1'
}
task copyDependencies(type: Copy) {
        from configurations.default
        into "$buildDir/libs"
}
jar {
        manifest {
            attributes(
                    'Main-Class' : 'com.mycompany.mypackage.System',
                    'Class-Path': configurations.runtime.files.collect {it.name}.join(' ')
            )
        }
}
```

We have to set a class path, which refers to the dependent libraries/jars, so that MyLibrary.jar's Main-Class can be called.

- When we create MyLibrary.jar, we have to call the new task *copyDependencies*, to copy dependent libraries to the output.

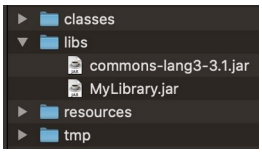
Terminal

```
NicosMBP:src nico$ ./gradlew build copyDependencies
NicosMBP:src nico$
```

Creating JARs with Gradle – Part VI

```
Terminal
NicosMBP:src nico$ ./gradlew build copyDependencies
NicosMBP:src nico$
```

- After this command was issued, we get following output in $buildDir/libs:



- The manifest-file created by Gradle has the Class-Path key, which points to the dependent commons-lang3-3.1.jar:

<MANIFEST.MF>

```
Manifest-Version: 1.0
Main-Class: com.mycompany.mypackage.System
Class-Path: commons-lang3-3.1.jar
```

- Finally, we can run MyLibrary.jar in a directory, that contains all dependent JARs:

```
Terminal
NicosMBP:src nico$ java -jar MyLibrary.jar
You must pass some arguments!
NicosMBP:src nico$
```


- If an OS allows to assign standard-applications to the jar-extension, we could assign the Java interpreter and start jars via a double-click on its icon as Java programs. Additionally, jars must be declared with a Main-Class in its manifest.

The Class Path – Part I

- What is a class path?
- Put simple, the class path specifies directories and individual jars, that are searched for classes, when a Java program is started.
- Besides the class path, some essential classes must be loaded, when a Java program starts, namely the JDK:
  - (1) So called bootstrap classes, which make up the JDK in directory jre/lib, esp. rt.jar.
  - (2) So called extension classes, which contribute types extending the Java runtime in directory jre/lib/ext.
  - (3) So called user classes, which make up our own classes and 3rd party classes, like those in commons-lang3-3.1.jar.
    - (3.1) User classes are searched in the current directory.
    - (3.2) User classes are searched in the directories/libraries/jars specified in
      - (3.2.1) the environment variable CLASSPATH (if any),
      - (3.2.2) the Class-Path specified in MANIFEST.MF (if any) and
      - (3.2.3) the Java interpreter's -cp switch (this is the preferred method right now).
- The take away: By default, the class path only points to the current directory!
  - This is the reason, we have to specify the Class-Path in the manifest-file to refer the dependent jar for MyLibrary.jar.

The Class Path – Part II

- Let's try specifying the class path via the environment variable CLASSPATH, therefor we remove it from the manifest-file:

```
// <build.gradle>
apply plugin: 'java'
archivesBaseName = 'MyLibrary'
repositories {
        mavenCentral()
}
dependencies {
        compile 'org.apache.commons:commons-lang3:3.1'
}
task copyDependencies(type: Copy) {
    from configurations.default
    into "$buildDir/libs"
}
jar {
        manifest {
            attributes(
                'Main-Class' : 'com.mycompany.mypackage.System'
                //, 'Class-Path': configurations.runtime.files.collect {it.name}.join(' ')
            )
        }
}
```

- Relying on CLASSPATH, we mustn't use -jar, instead we have to set MyLibrary.jar and commons-lang3-3.1.jar there:

Terminal

```
NicosMBP:libs nico$ export CLASSPATH=MyLibrary.jar:commons-lang3-3.1.jar
NicosMBP:libs nico$ java com.mycompany.mypackage.System
You must pass some arguments!
NicosMBP:libs nico$
```

  - If we use -jar is used to run an executable instead, CLASSPATH is ignored.

The Class Path – Part III

- Let's also try specifying the class path via the command line switch -cp:

```
Terminal
NicosMBP:libs nico$ java -cp MyLibrary.jar:commons-lang3-3.1.jar com.mycompany.mypackage.System
You must pass some arguments!
NicosMBP:libs nico$
```

- Also if relying on -cp, we mustn't use -jar, instead we have to specify MyLibrary.jar and commons-lang3-3.1.jar with -cp.
  - The other way around: if -jar is used to run an executable, -cp is ignored.
- If -jar is used, -cp and CLASSPATH are ignored.
  - The other way around: if we want to use an executable jar (with Main-Class), we have to use Class-Path to specify the class path.
  - The Class-Path property in MANIFEST.MF overrides -cp and CLASSPATH.
- Currently, the Java community recommends using -cp, because esp. setting CLASSPATH can be tricky.
  - In a sense environment variables can be difficult to handle.
- Remark: the jar-files contained in the current path, are not automatically in the class path!
  - Instead the files in the current directory must be explicitly added. This can be done very simply:

```
Terminal
NicosMBP:libs nico$ java -cp "./*" Program
NicosMBP:libs nico$
```


- We have to put -cp's argument into double quotes, because we have to protect it from evaluation through the terminal! The problem is the *, which has a special meaning for resolving files for the java-tool and also has a interfering special meaning for the terminal as console (terminal-globbing).

JAR Signing

- JARs can be signed with a digital signature with the jarsigner-tool.
  - The idea is to sign a JAR with a signature, which can be associated with a specific issuer/vendor.
    - The signature is then added to MANIFEST.MF.
  - This specific issuer/vendor should then be trustworthy.
  - JAR signing was esp. important to guarantee the origin of a JAR containing an applet loaded dynamically by the web browser.
  - Library-code contained in signed JARs has more privileges than unsigned JARs.

- In an earlier example, we saw, that updating a JAR is relatively simple with the jar-tool's u-option.
  - However, there is a problem: it means, that class-files, resources and the manifest-file could be manipulated in a JAR!
  - Alas, the JDK's tool have nothing to offer currently to address this problem.

static imports – Part I

- If we want to use static methods defined in another class, we can simplify the call syntax by using static imports:

```
package com.mycompany.mypackage.utils;
// <Algorithms.java>
import java.util.Scanner;

public class Algorithms {
        public static int number;

        public static int printPrompt(){
                System.out.println("Please enter a number:");
                System.out.println("The number should be greater than ten:");
                Scanner inputScanner = new Scanner(System.in);
                number = inputScanner.nextInt();
                return number;
        }
}
```

```
package com.mycompany.mypackage;
// <System.java>
import java.util.Scanner;
import static com.mycompany.mypackage.utils.Algorithms.*;

public class System {
        public static void main(String[] args) {
                printPrompt();
                java.lang.System.out.println("You entered "+number+"!");
        }
}
```

  - We just use the import keyword together with the static keyword to import all static methods of a class.
  - With this import we can use all static methods of *Algorithms* into *System*, as if they were defined in *System* itself.
  - In effect we can then call the methods of the "statically" imported class just without prefixing class name and dot.

- Similar to importing full packages, using '*' on static imports, imports all accessible members!
  - We can also specify specific static members to be imported, e.g. only *Algorithms.printPrompt()*:

```
// <System.java>
import java.util.Scanner;
import static Algorithms.printPrompt;
// pass
```

static imports – Part II

- An often seen usage of static import is importing all static methods and constants of *Math*, e.g. *Math.sin()* and *Math.PI*:

```
// <Program.java>
import static java.lang.Math.*;

public class Program {
	public static void main(String[] args) {
		System.out.println(sin(PI/2));
		// >1.0
	}
}
```

- We can also statically import all members of an enum:

```
// <Program.java>
import static java.time.Month.*;

public class Program {
	public static void main(String[] args) {
		System.out.println(DECEMBER);
		// >DECEMBER
	}
}
```

- Restrictions:
  - static imports cannot be used for classes of the default package! – It is really meant to shorten the syntax for long package names.
- Personal recommendation: minimize using static imports, it can hide the origin of esp. methods!

Thank you!