Java 9 Module System

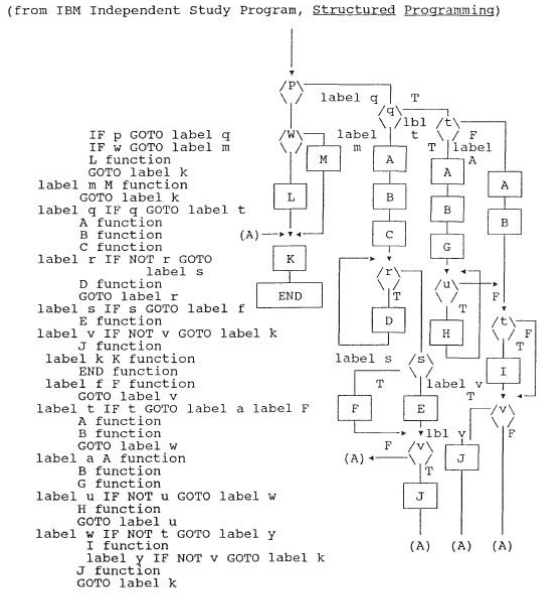
School of Engineering

- Complex Software and Programming Language
- History of Modules
- Module Concepts and Tools
- Modularization of the JDK

Problem of Complexity and Programming Language

Early/Modern Programming Languages

- No Encapsulation
- No language features to control visibility
- No structural language features
- Dynamic Typing
- Jumps
- -> Unmaintainable (spaghetti) code




```
google.load('visualization', '1.0', {'packages':['corechart']});
google.setOnLoadCallback(drawChart);
function drawChart() {
    var options = {
        chart: {
            title: 'Occupancy stats',
        },
        bars: 'horizontal' // Required for Material Bar Charts.
    };
<?php   if($isPercentage == 1) { ?>
    var data = google.visualization.arrayToDataTable([
        ['Centre', 'Mean', 'Median', 'Mode', 'Average Occupancy'],
<?php   if(count($averages) > 0 ) { ?>
<?php       foreach($averages as $branchId => $average) {

            echo "['" . $allBranches[$branchId]['BUSINESS_BRANCH_NAME'] . "(ID " . $branchId
        }?>
    ]);
<?php   } ?>

    options.hAxis = {
        format: 'percent',
    };

<?php   } else { ?>
    var data = google.visualization.arrayToDataTable([
        ['Centre', 'Mean', 'Median', 'Mode', 'Average Appointment Slots', 'Average Occupancy'],
<?php   if(count($averages) > 0 ) { ?>
<?php       foreach($averages as $branchId => $average) {
            echo "['" . $allBranches[$branchId]['BUSINESS_BRANCH_NAME'] . "(ID " . $branchId . ")
                }?>
    ]);
<?php   } ?>
<?php   } ?>

    var chart = new google.visualization.BarChart(document.getElementById('barchart_material'));
```

Mix JavaScript with PHP

© Google

```
var UserGist = React.createClass({
  getInitialState: function() {
    return {
      username: '',
      lastGistUrl: ''
    };
  },

  componentDidMount: function() {
    $.get(this.props.source, function(result) {
      var lastGist = result[0];
      this.setState({
        username: lastGist.owner.login,
        lastGistUrl: lastGist.html_url
      });
    }.bind(this));
  },

  render: function() {
    return (
      <div>
        {this.state.username}'s last gist is
        <a href={this.state.lastGistUrl}>here</a>.
      </div>
    );
  }
});
```

Mix HTML tags (presentation) with JavaScript for cross cutting concerns and business logic in an inverted control flow

© Facebook

OOP has solved the Problem

OOP allows

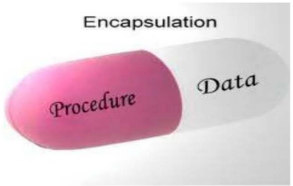
- Encapsulation of
  - data
  - functionality
- Information hiding
- Separation of concerns

- Inheritance for reuse
  - Extension of functionality
  - Adaptation of functionality by overriding

- All problems solved!

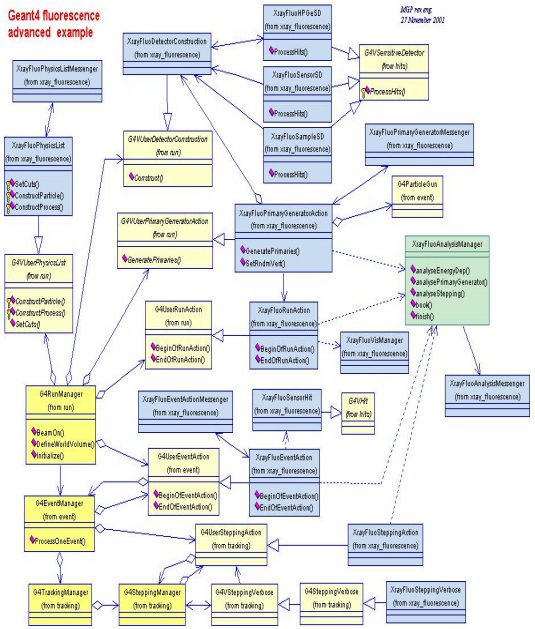
Data abstraction or information hiding refers to providing only essential information to the outside world and hiding their background details.

Data Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data and that keep both safe from outside interference and misuse.



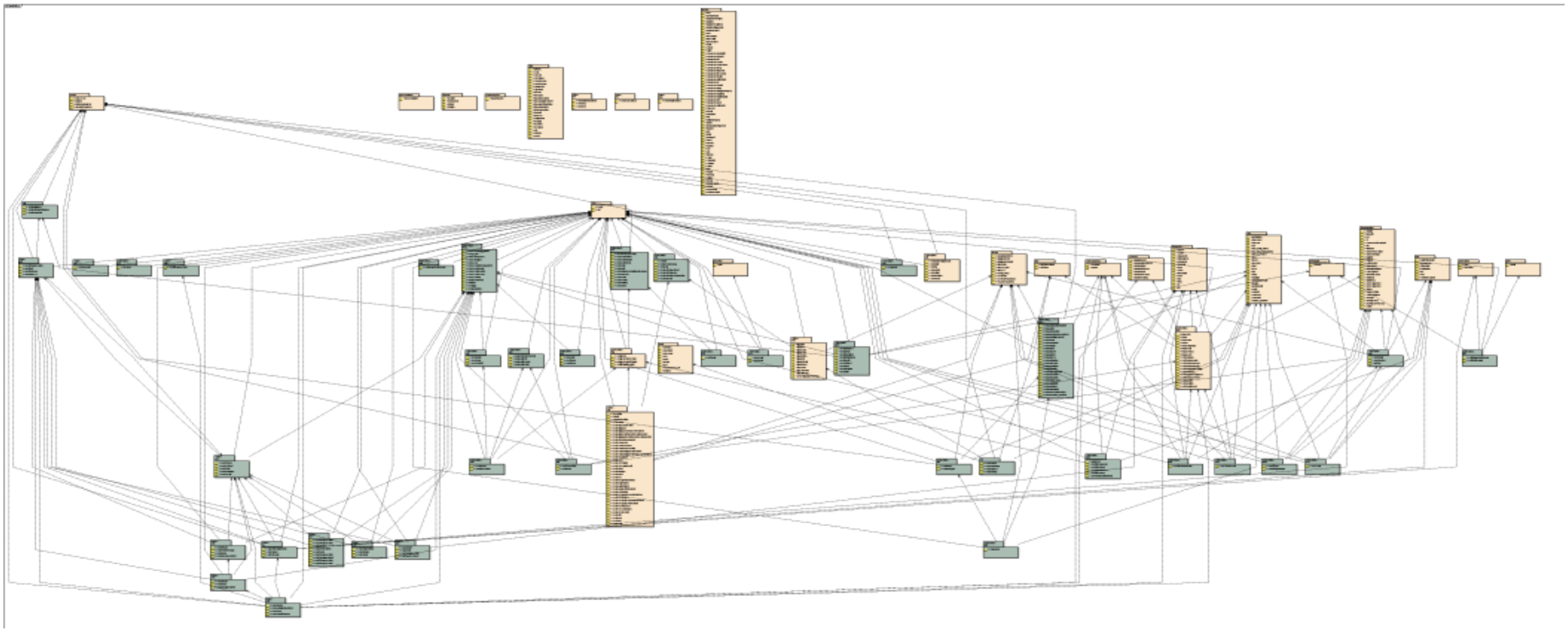
Limits of OOP

- Object are interconnected
- Coupling between objects:
  - reference
  - composition
  - inheritance
    - *class -> extension*
    - *interface -> implementation*
- Complexity grows
  - n^2 number classes
  - linear with time (versions)
- e.g.Eclipse
  - 20'000 classes; 1.5 Mio interdependencies
- Plug-in architecture
  - run time extensibility



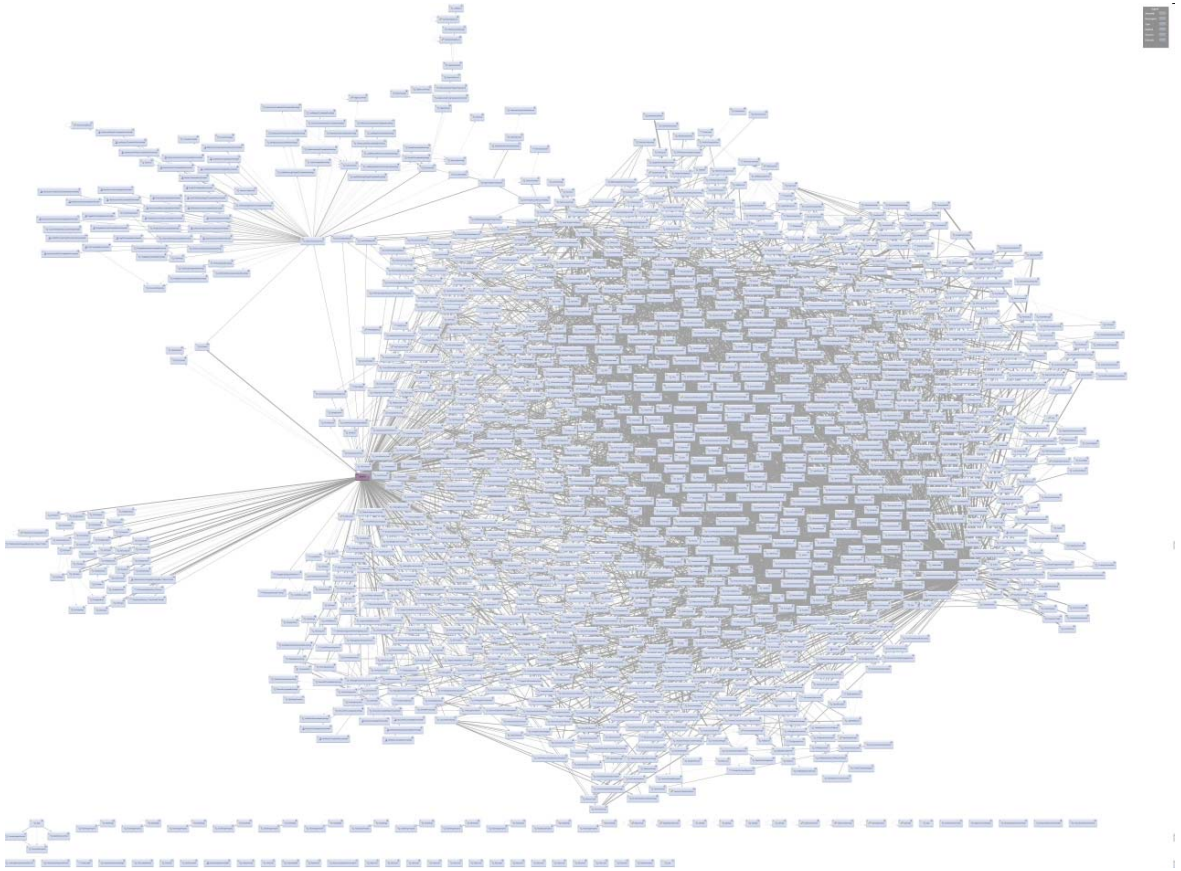
… limits of OOP

- In reality quickly several 100 classes in dozens of packages
- Only package dependency shown here



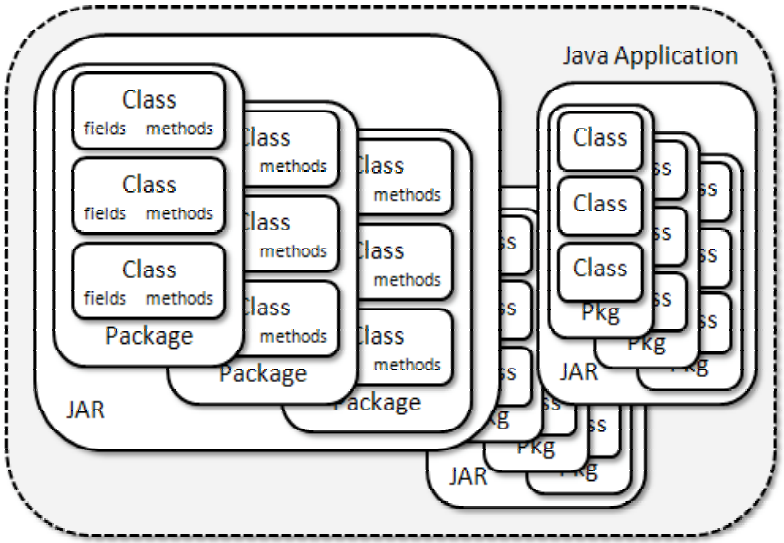
Failed Due to Complexity

- Large projects prone to failer due complexity
- 2008 Amarta
  - Swiss Life
  - canceled
  - ca. 500-800 Mio
- 2012 Insieme
  - Steuerverwaltung
  - canceled
  - 150 Mio SFR
- 2012 FIS
  - Schweizer Armee
  - up to 700 Mio SFR
- 2013 Mistra
  - costs: 100 Mio
  - continued
- 2015 ISS
  - Interception System Schweiz
  - 18 + 13 Mio
  - Leaks to Mossad detected
  - continued

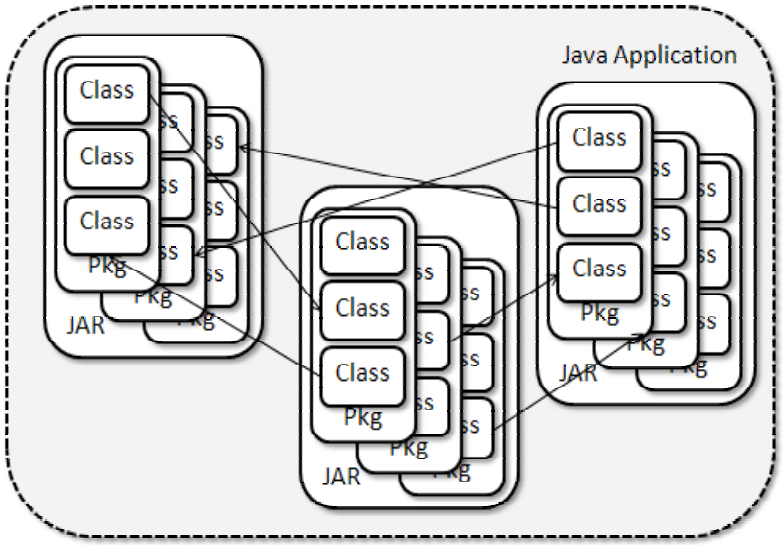


Java is Modular

- Packages as containers for classes
- Jar Files to package classes

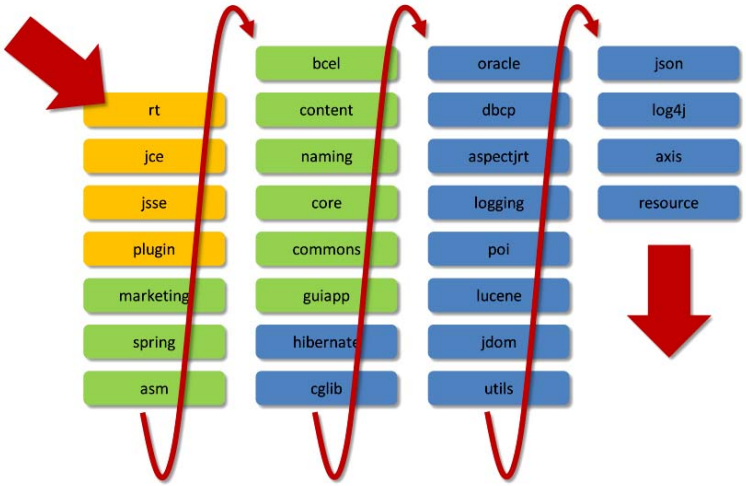


- Wrong -> Problem
  - public classes are globally accessible



JAR Hell

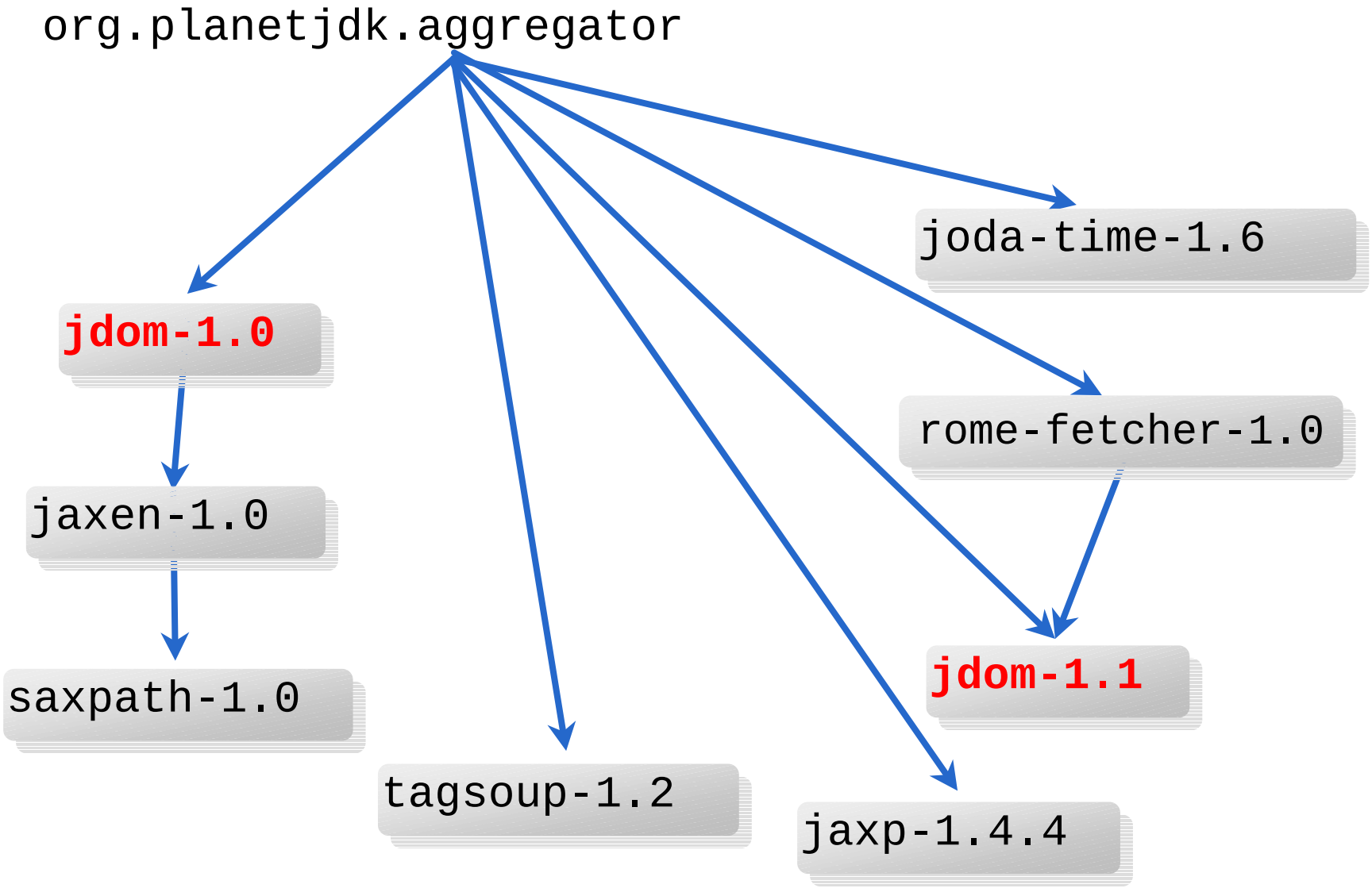
- All used Jars



- have to be listed in the classpath

```
[search path for class files: C:\Program Files\Java\jdk1.7.0_80\jre\lib\resources.jar,C:\Program Files\Java\jdk1.7.0_80\
jre\lib\rt.jar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\sunrsasign.jar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\jsse.j
ar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\jce.jar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\charsets.jar,C:\Program F
iles\Java\jdk1.7.0_80\jre\lib\jfr.jar,C:\Program Files\Java\jdk1.7.0_80\jre\classes,C:\Program Files\Java\jdk1.7.0_80\jr
e\lib\ext\access-bridge-64.jar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\dnsns.jar,C:\Program Files\Java\jdk1.7.0_80
\jre\lib\ext\jaccess.jar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\localedata.jar,C:\Program Files\Java\jdk1.7.0_80\
jre\lib\ext\sunec.jar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\sunjce_provider.jar,C:\Program Files\Java\jdk1.7.0_8
0\jre\lib\ext\sunmscapi.jar,C:\Program Files\Java\jdk1.7.0_80\jre\lib\ext\zipfs.jar,.,Person.jar,archive\PersonIF.jar]
```

… JAR Hell: Version Conflict



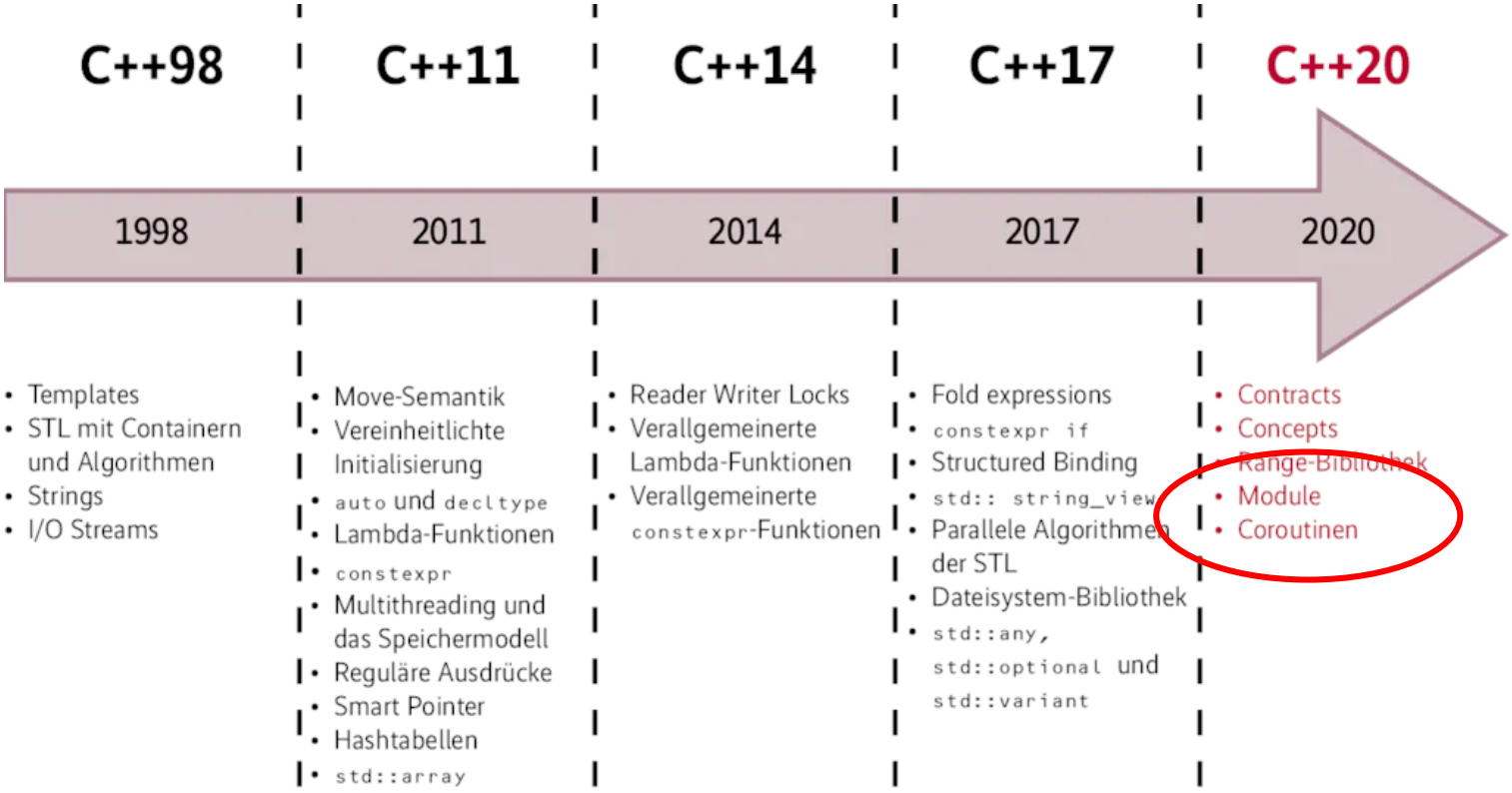
What is missing in Java

School of Engineering

- Coarse grained units of (full) encapsulation
  - Packages are not enough
- Clearly specified dependencies
  - what is **used** in other modules -> imports
  - what is **offered to** other modules -> exports
- Clearly specified visibility
  - Full encapsulation
- Versioning of modules
  - had to be taken out of Java 9 -> build system: maven, gradle
- Independent lifecycle of modules
  - only working solution so far: COM/ActiveX

What was missing in C++, is new in C++20

- Module
- Coroutines



What was missing in Java Script -> ECMA 6

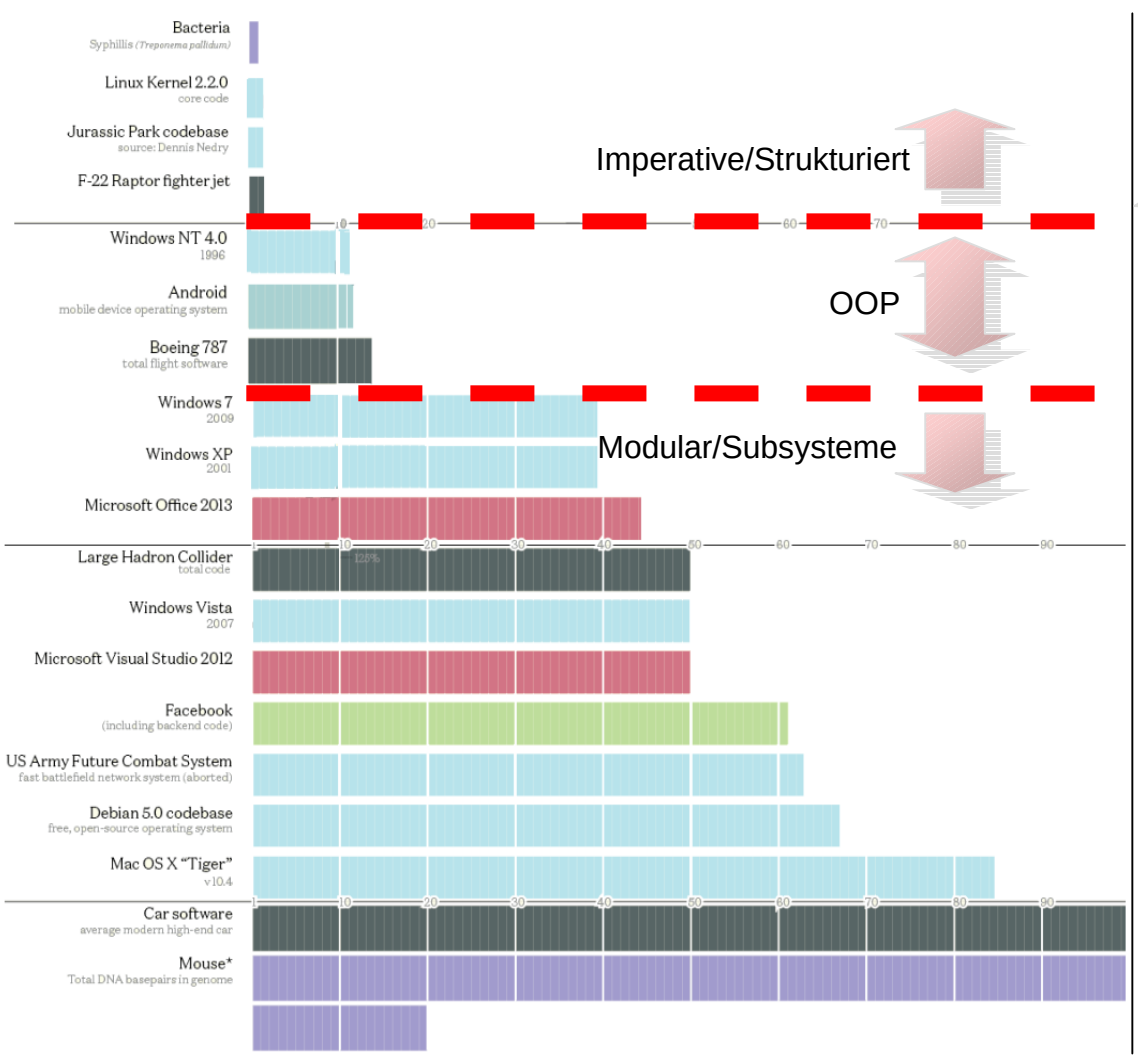
School of Engineering

- Proposed Standard to ECMA Script 6
- Pros
  - Synchronous and asynchronous loading supported.
  - Syntactically simple.
  - Support for static analysis tools.
  - Integrated in the language (eventually supported everywhere, no need for libraries).
  - Circular dependencies supported.
- Cons
  - Still not supported everywhere.
- System.js as Temporary Solution?
  - https://auth0.com/blog/javascript-module-systems-showdown/?utm_source=sitepoint&utm_medium=gp&utm_campaign=webassembly_overdue

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

Komplexität: Million Lines of Code (Vergleich)



Ab einer bestimmten Grösse reichen (Struktur) Mittel nicht mehr

Zum Vergleich Mensch: Gene 3.3 Milliarden "Lines of Code"

Project History

History Java Module System

School of Engineering

- 2000 OSGi Java-based module system
  - Developped by IBM
  - Uses separations of classes loaded by different class loaders
    - *standard for eclipse, WAS, wildfly*
    - *rather complex*
    - *conflicts with low level mechanisms, reflection*
- 2005 first trials JSR 277 / JSR 294
  - canceled during draft status face
- 2008 Project Jigsaw started

Modularity Specifications

School of Engineering

- JSR (Java Specification Request - former JCP)
  - JSR 376: Java Platform Module System
- JEP (Java Enhancement Proposal introduced by Oracle)
  - 200: The Modular JDK: Define a modular structure for the JDK
  - 201: Modular Source Code: Reorganize the JDK source code into modules, enhance the build system to compile modules, and enforce module boundaries at build time220: Modular Run-Time Images
  - 260: Encapsulate Most Internal APIs
  - 261: Module System: Implement the Java Platform Module System
  - 282:The Java Linker: Create a tool that can assemble and optimize a set of modules and their dependencies into a custom run-time image
- related to JEP
  - 238 Multi-Release JAR Files
  - 253 JavaFX Modularization
  - 275: Modular Java Application Packaging

Java 9 Modules

A Module in Java

- A module is a grouping of code
  - For Java this is a collection of **packages**
- A module can contain also
  - Native Code
  - Resources
  - Configuration data

```
module com.azul.zoop {
   exports com.azul.zoop.alpha;
   exports com.azul.zoop.beta;
}
```

module-info.java

Physical Structure of a Module

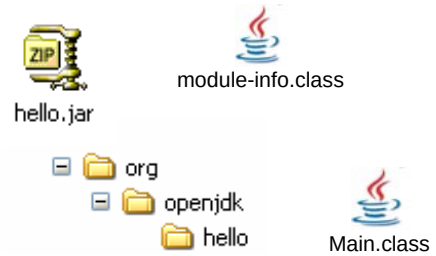
- Module-info is placed into the root
- Java files are placed according package structure
- Compiled output is packaged into a jar file
  - with according directory structure
- the module (jar file) is placed in a central directory
  - this *directory* is references later in the module path
- btw. all modules of the jdk are in the jdk/jmods directory as jmods
  - jmod is an extended module format ("modules on steroids")

Single Module and Multi Module Jars

- A single or multiple modules may be placed into jar
- Single Module Jar
  - module-info is placed into root
  - classes are placed into subdirectories

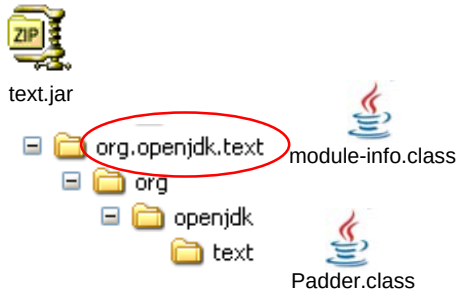
always imported automatically

```
module org.openjdk.hello {
   requires java.base; // not needed
   exports org.openjdk.hello;
}
```



- Multi Module Jar
  - a directory named equals to **module name** is created
    - *e.g. org.openjdk.text*
  - module-info is placed into that directory
  - classes are placed into subdirectories starting with above module directory

```
module org.openjdk.text {
   requires java.base; // not needed
   exports org.openjdk.text;
}
```

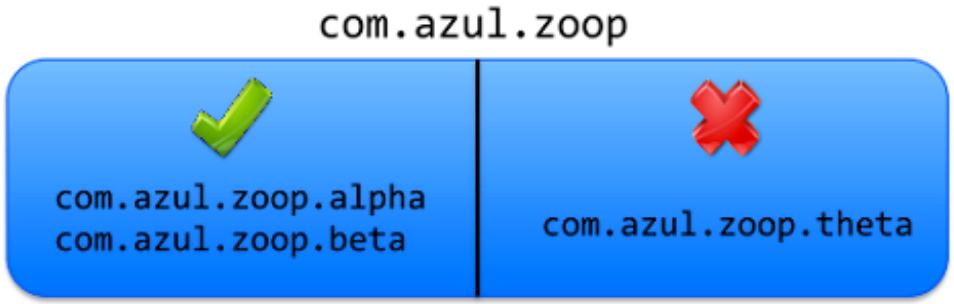


Module Visibility

School of Engineering

- All packages that should be visible from outside have to be exported
- Classes in all other packages are not visible from outside
  - independent of class visibility attributes (public)

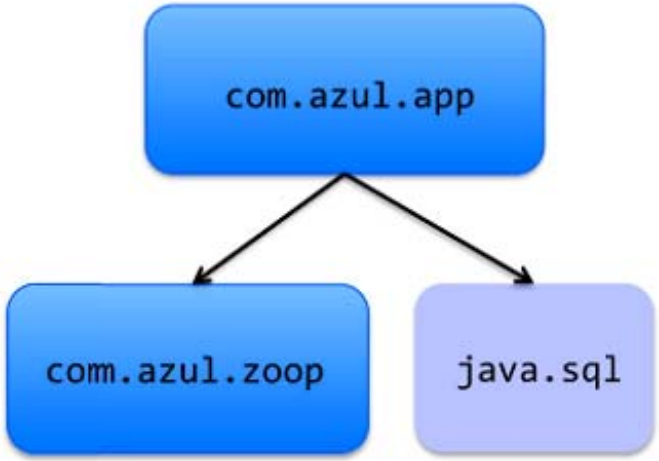
```
module com.azul.zoop {
   exports com.azul.zoop.alpha;
   exports com.azul.zoop.beta;
}
```



Module Dependencies

- In the Module Info all dependent Modules are declared

```
module com.azul.app {
    requires com.azul.zoop;
    requires java.sql;
}
```

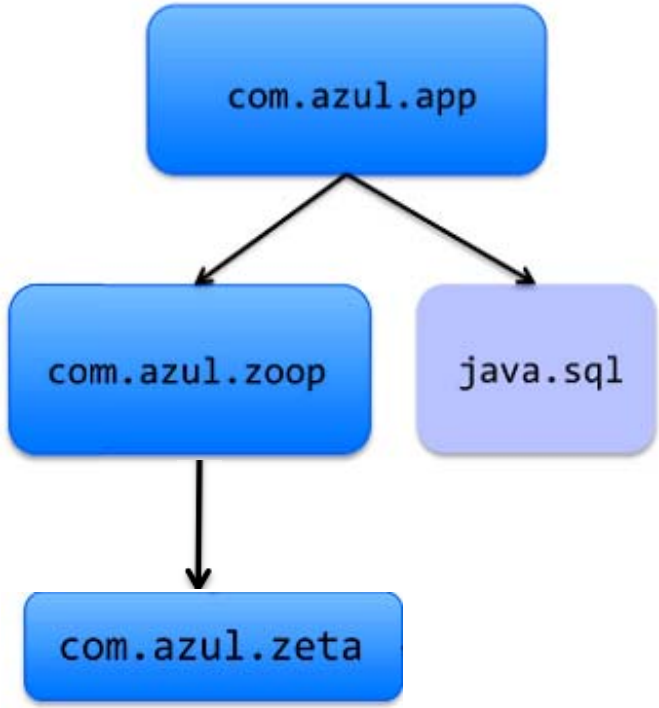


Transient Module Dependencies

- Module is dependent of another
  - e.g. type in method parameter

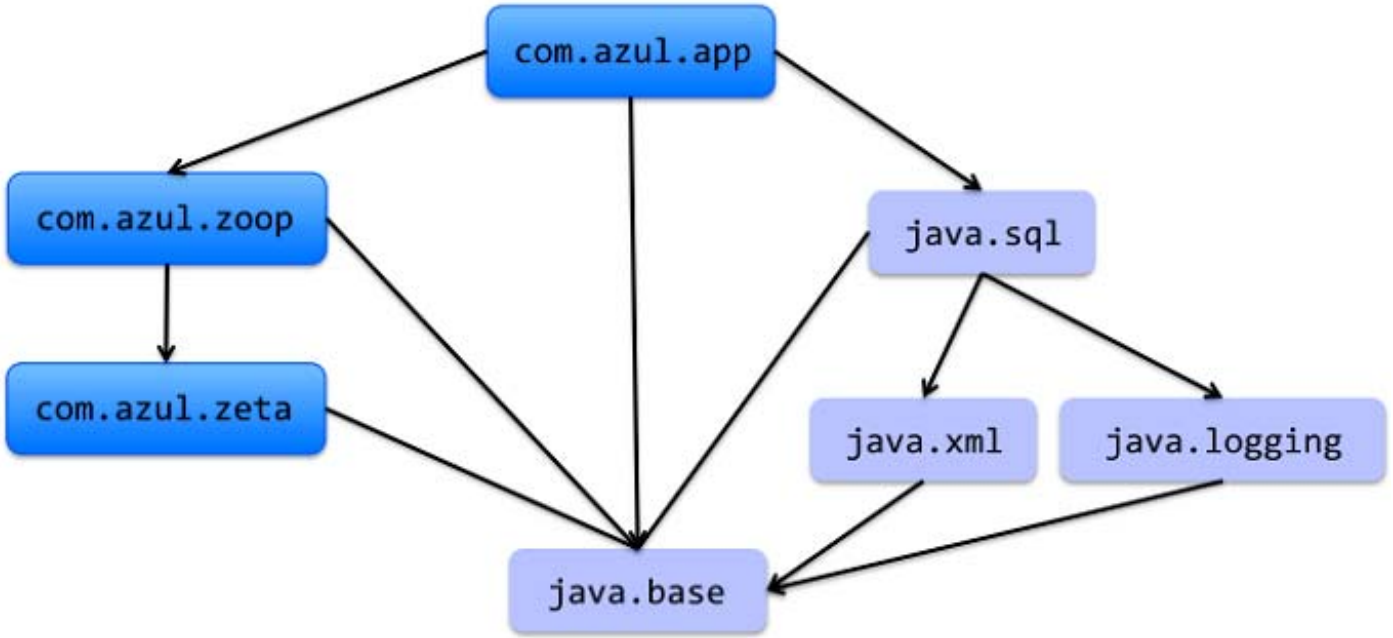
```
module com.azul.app {
    requires com.azul.zoop;
    requires java.sql;
}
```

```
module com.azul.zoop {
    exports com.azul.zoop.alpha;
    exports com.azul.zoop.beta;
    requires transient com.azul.zeta;
}
```



Convex Hull

- Due to the module declarations Java may determine the "convex hull" of all modules required
- i.e. all modules needed to build and run the application



Automatic &
Unnamed Modules

Automatic Modules

- Problem: a lot of libraries are plain jars not modules
- a) module-info can be added to the jar
- b) a plain JAR on the *module path* becomes an Automatic Module
  - Module name derived from JAR name (without versioning info)
  - exports everything
  - reads all other module
  - using module has simply import package

```
module automatic-demonstrator {
    requires commons.lang;
}
```

Versioning Info is stripped

commons.lang3-3.4.jar

Unnamed Modules

School of Engineering

- JARs (modular or not) and classes on the classpath will be contained in the Unnamed Module set.

	`--module-path`	`--classpath`
Modular JAR	application module	unnamed module
Non-Modular JAR	automatic module	unnamed module

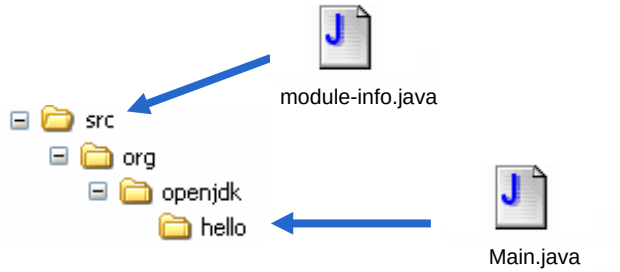
- Similar to automatic modules
  - it exports all packages and reads all other modules.
  - but it does not have a name.
  - for that reason, it cannot be required and read by named application modules.
  - the unnamed module in turn can access all other modules.
- Module Readability

Module type	Origin	Exports packages	Can read modules
(Named) Platform Modules	provided by JDK	explicitly	
(Named) Application Modules	any JAR *containing* `module-info.class` on the *module path*	explicitly	Platform Application Automatic
Automatic Modules	any JAR *without* `module-info.class` on the *module path*	all	Platform Application Automatic Unnamed
Unnamed Module	all JARs and classes on the *classpath*	all	Platform Automatic Application

Tools to Build Modules

Make a Standalone Module Hello

- To define a Module a module-info.java has to be placed in root
  - Java sources are in a src-directory
  - are compiled to a classes-directory and packaged to lib-directory




```
module org.openjdk.hello {
   requires java.base; // not needed
   exports org.openjdk.hello;
}
```

lookup leftpad and npm :-)

```
package org.openjdk.hello;

public class Main {
    private static String leftPad (String s, int w) {
      StringBuilder sb = new StringBuilder();
      int n = w - s.length();
      for (int i =0; i < n; i++) {
          sb.append(" ");
      }
      sb.append(s);
      return sb.toString();
    }

    public static void main(String ... args) {
      String adj = args.length > 0 ? (" "+ args[0]) : "";
      System.out.println(leftPad("Hello,"+ adj + "world!",90));
    }
}
```

Commands to Build the Module Hello

School of Engineering

- Compile all sources in a classes directory

```
% compile
javac -d classes src/org/openjdk/hello/Main.java src/module-info.java
```

- jar class-files to a lib hello.jar

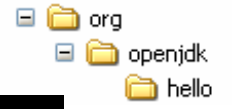
```
% jar it
jar --create --file lib/hello.jar -C classes .
```

module desc

hello.jar


module-info.class

jar file




Main.class

- run it

```
% run it
java --module-path lib -m org.openjdk.hello/org.openjdk.hello.Main
```

-modulepath can be abbreviated to -mp

Module to run

main class to run

- or - jar class-files and define main class (run becomes simpler)

```
% jar it
jar --create --file lib/hello.jar --main-class org.openjdk.hello.Main  -C classes
```

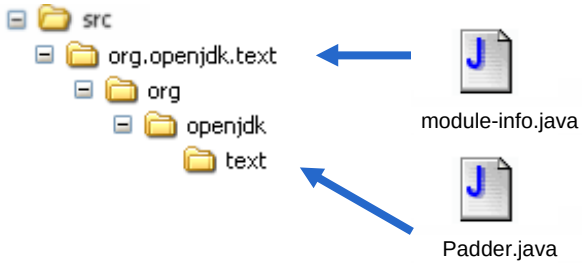
- run it

```
% run it
java --module-path lib -m org.openjdk.hello
```

Make a Module Text with class Padder

Subdirectory with same name as module

- A directory with name org.openjdk.text has to be created
- A module-info.java has to be placed in that directory




```
module org.openjdk.text {
   requires java.base; // not needed
   exports org.openjdk.text;
}
```

```
package org.openjdk.text;

public class Padder {
   public static String leftPad (String s, int w) {
      StringBuilder sb = new StringBuilder();
      int n = w - s.length();
      for (int i =0; i < n; i++) {
          sb.append(" ");
      }
      sb.append(s);
      return sb.toString();
   }

}
```

Commands to build the Module Text

- Clean classes directory

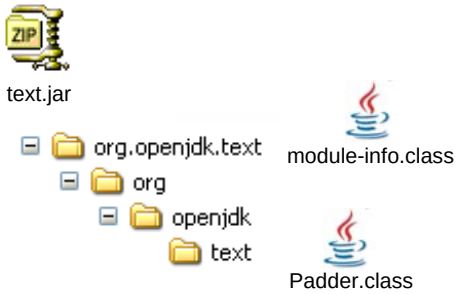
```
rm -rf classes
```

- Compile sources into the classes directory

```
% compile
javac -d classes src/org.openjdk.text/org/openjdk/text/Padder.java
                 src/org.openjdk.text/Module-info.java
```

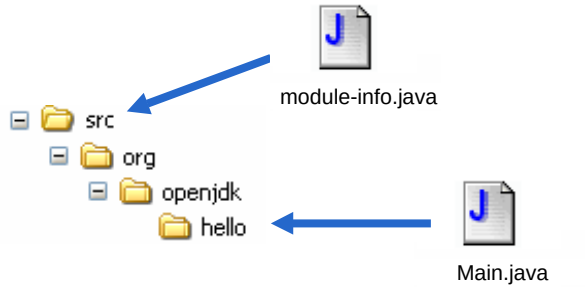
- jar class-files to a lib text.jar

```
% jar it
jar --create --file lib/text.jar -C classes .
```



Make a Module Hello that imports Text

- To define a Module a module-info.java has to be placed in root




```
module org.openjdk.hello {
   requires java.base; // not needed
   requires org.openjdk.text;
   exports org.openjdk.hello;
}
```

```
package org.openjdk.hello;

import org.openjdk.text.Padder;

class Main {
    public static void main(String ... args) {
      String adj = args.length > 0 ? (" "+ args[0]) : "";
      System.out.println(Padder.leftPad("Hello,"+ adj + "world!",70));
    }
}
```

Commands to build and run the Module Hello

School of Engineering

- Clean classes directory (for clean build)

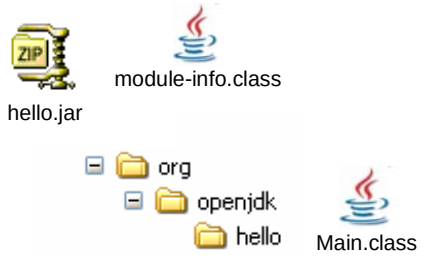
```
rm -rf classes
```

- Compile all sources into the classes directory
- The text.jar (module dependency) is in the lib directory

```
% compile
javac -d classes --module-path lib src\org\openjdk\hello\Main.java src/module-info.java
```

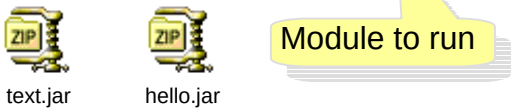
- jar class-files to a lib hello.jar

```
% jar it
jar --create --file lib/hello.jar -C classes .
```



- To run it the module-path is required (text.jar and hello.jar)

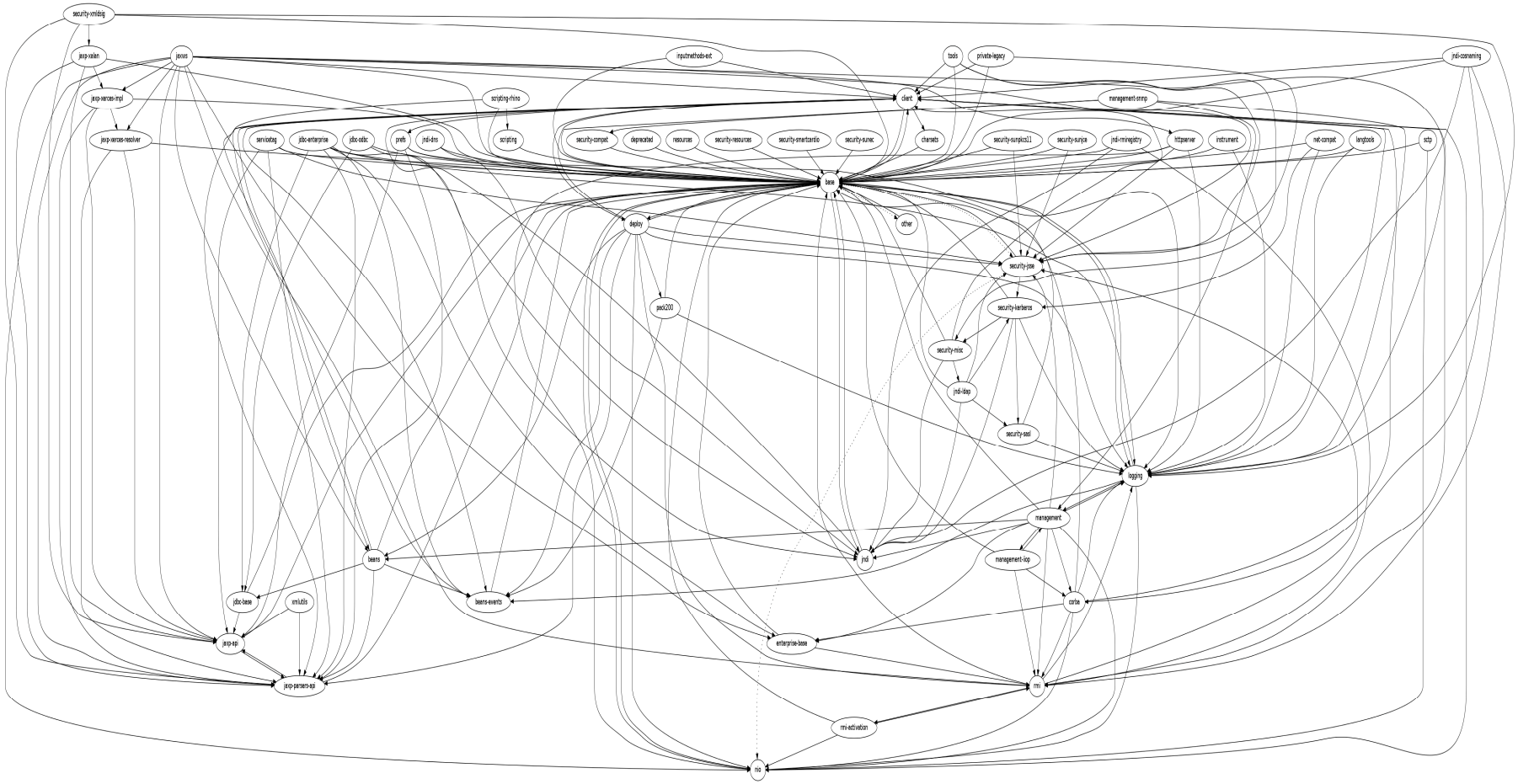
```
% run it
java --module-path lib -m org.openjdk.hello/org.openjdk.hello.Main
```



Modularization of the JDK
Platform Modules

Java 8 JDK Package Dependencies

School of Engineering



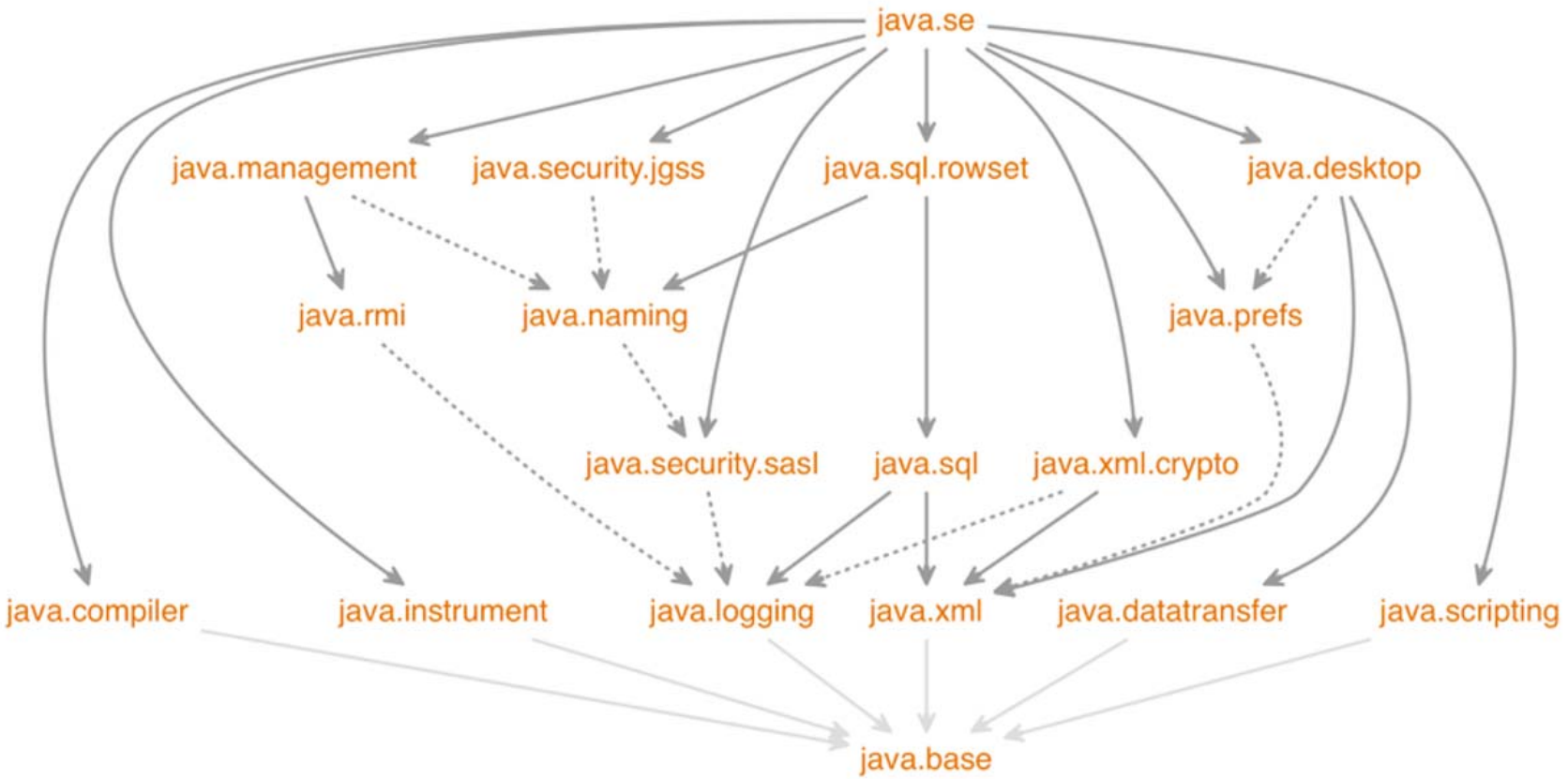
Issues - Solution

- Java SE 8 has 210 packages
- Many are not needed by all applications
  - e.g. CORBA, Swing, AWT, XML
- Cyclic Dependencies
  - everything is dependent on each other
- To big for small devices

- Internal (potentially unsafe packages)
- Solution
  - group packages into larger entities -> Modules
  - no cyclic dependencies
  - allow for custom built (stripped JDK) deployment

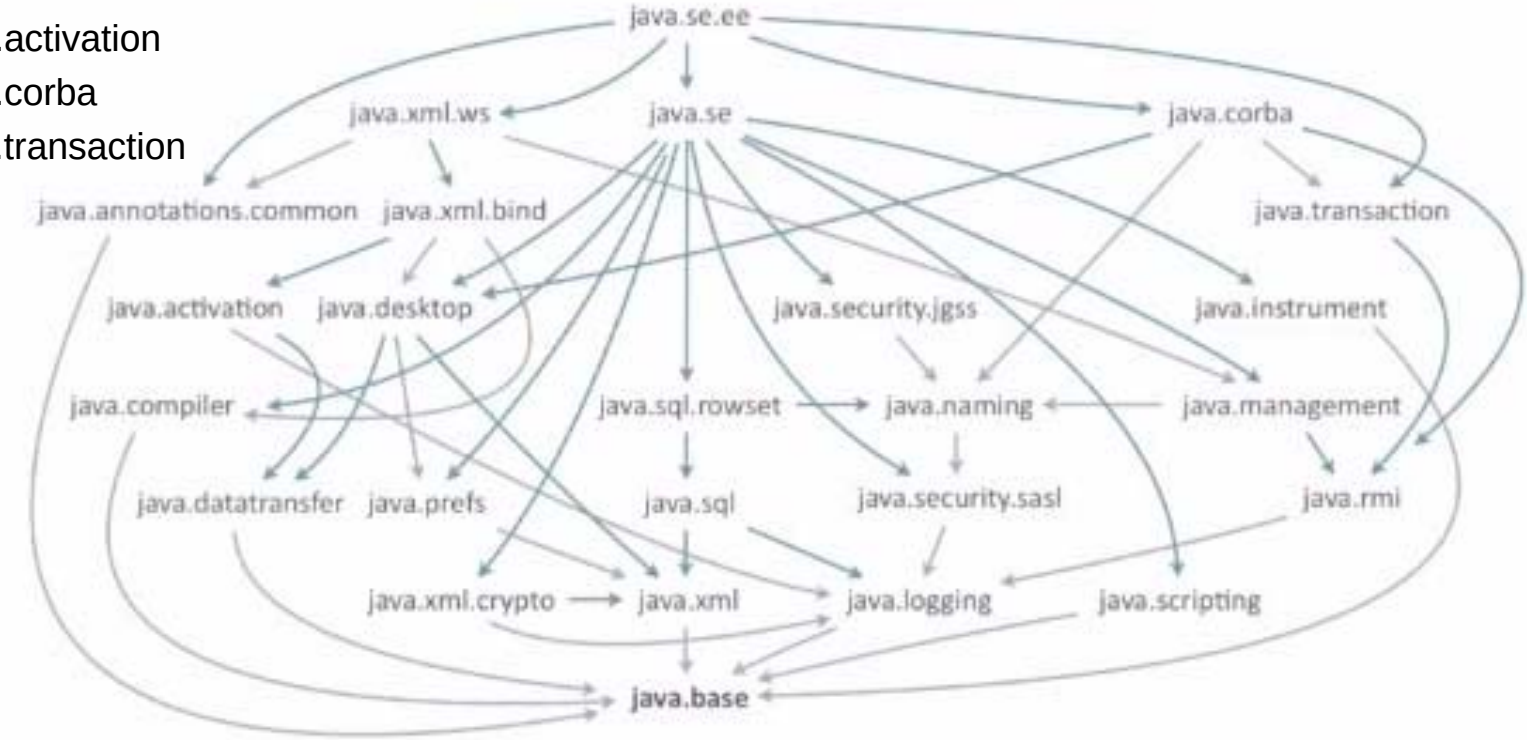
Java 9 SE Module Graph

- java.base is the central base module



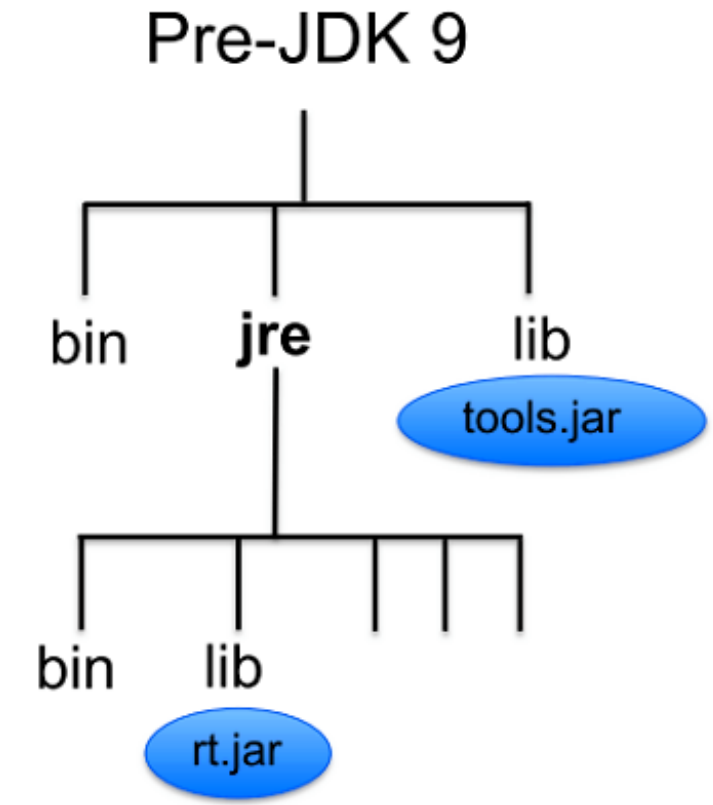
Java 9 EE Modules - Removed in 11

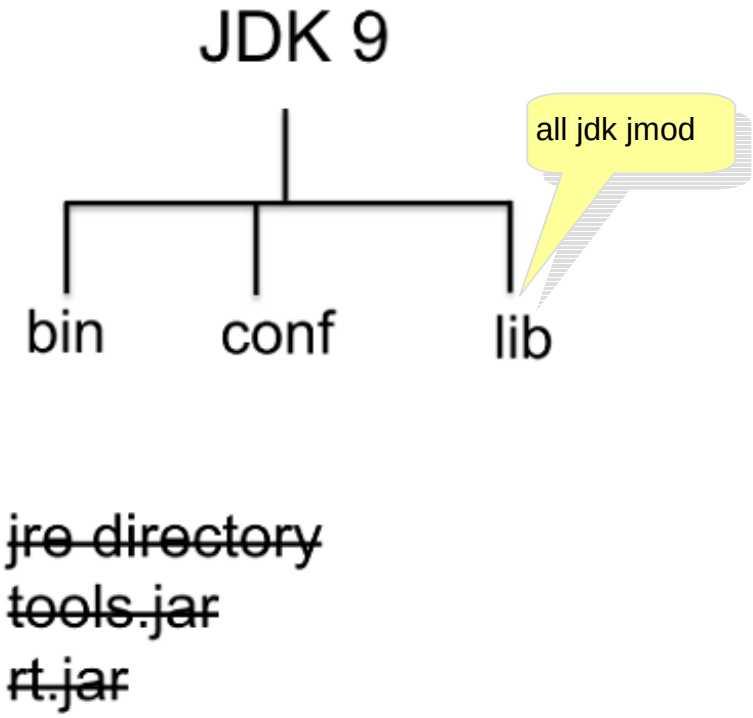
- Modules that are not part of standard edition -> Jakarta EE
  - java.xml.bind
  - java.xml.ws
  - java.xml.ws.annotation
  - java.activation
  - java.corba
  - java.transaction



New JDK Directory Structure

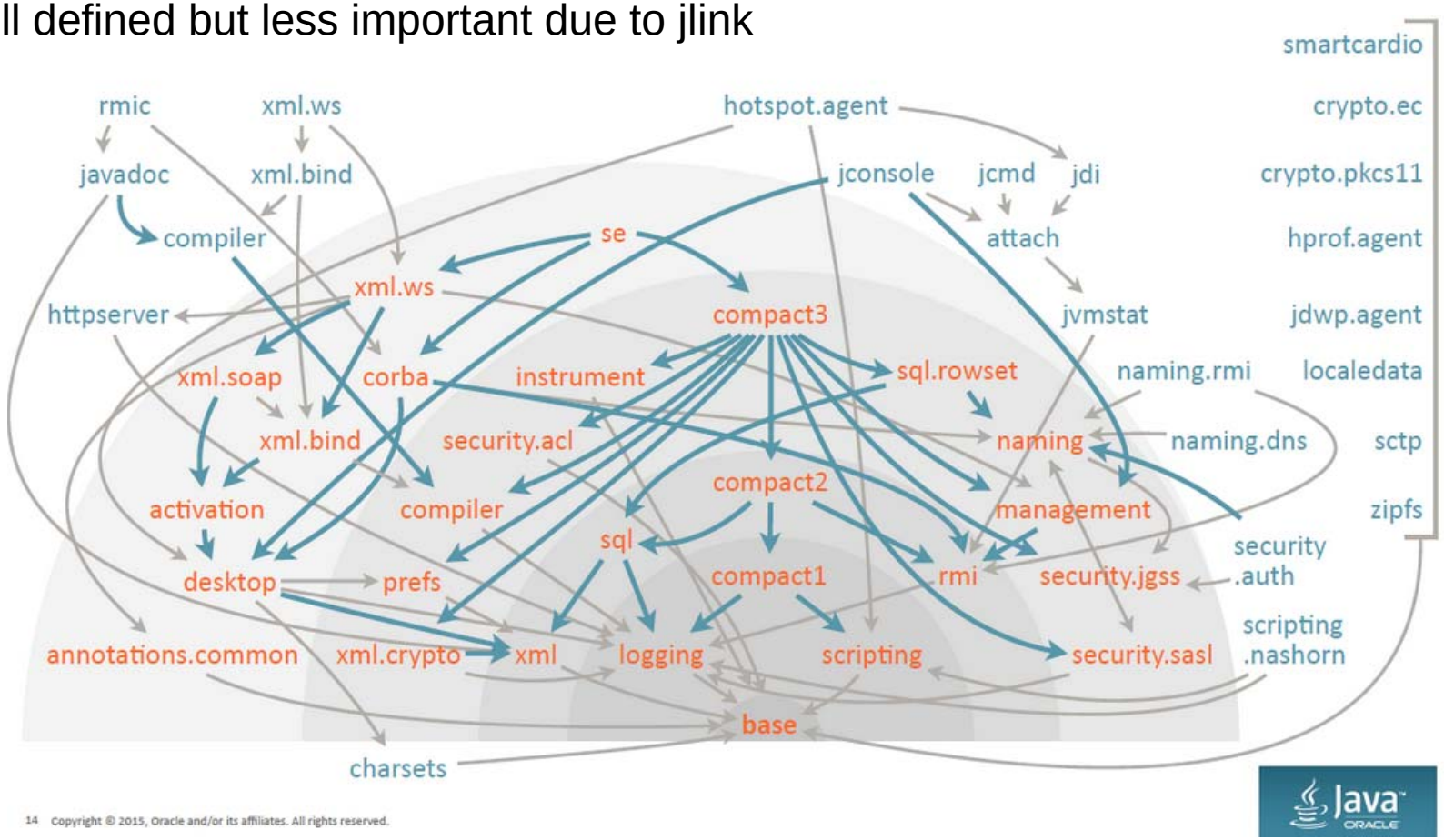
- Java 9 has a cleaned up directory structure





Java 8 Compact Profiles

- In Java 8 Compact Profiles
  - smaller Subsets of the jdk
- Still defined but less important due to jlink



Build Custom Distributions - jlink

- jlink: a tool for building a customized runtime image
  - all dependent modules
  - a stripped down Runtime System *for this platform (ca. 20 mb)*
  - a start script to start the application

- to build it

```
set jmods="C:\Program Files\Java\jdk9\jmods"
jlink --module-path lib;%jmods% --add-modules org.openjdk.hello
     --launcher  start-cmd=org.openjdk.hello --output distdir --compress 2 --strip-debug
```

root of modules

to compress

- to run it

```
distdir\bin\start-cmd
```

Conclusion

School of Engineering

- Java 9 has a module system deeply integrated into the system
- Compilation and configuration of large systems becomes
  - simpler
  - more predictable
- Not addressed is the versioning problem
  - should be handled by build tools: maven, gradle
  - layers if not handled otherwise
    - *https://docs.oracle.com/javase/9/docs/api/java/lang/ModuleLayer.html*
- It will take years until Java 9 modules will be fully adopted
  - Tools (e.g. build tools, IDEs) have to be adapted
  - Java EE has to be modularized
  - OSGi modules have to be integrated/migrated
  - ...

Links

School of Engineering

- Project Jigsaw: Module System Quick-Start Guide
  http://openjdk.java.net/projects/jigsaw/quick-start
- Das neue Modulsystem
  https://www.informatik-aktuell.de/entwicklung/programmiersprachen/java-9-das-neue-modulsystem-jigsaw-tutorial.html
- Modules in One Lesson by Mark Reinhold, Oracle
  https://www.youtube.com/watch?v=C5yX-elG4w0
- Keynote Session by Mark Reinhold, Oracle
  https://www.youtube.com/watch?v=I1s7R85GF1A
- Support for Java 9 in IntelliJ IDEA
  https://www.youtube.com/watch?v=WL48zkLvK3I