Модульность в Java 9

Минимальные знания для комфортного прохождения курса

Основы ООП.

Основы языковых конструкций Java 8 и выше.

Базовые принципы построения приложений на языке Java.

Введение

Одним из крупнейших изменений, представленных в Java 9, является модульность (проект *Jigsaw*).

Реальные проекты на Java (и не только) состоят из сотен или тысяч классов, сгруппированных в пакеты. Пакеты сгруппированы в архивы Java – jar-файлы.

Помимо основного кода проекта, написанного командой, любое серьёзное приложение использует сторонние драйвера, библиотеки, фреймворки и т.д. Так, Java имеет активное сообщество разработчиков программного обеспечения с открытым исходным кодом (OSS) и все дополнения поставляются в формате jar.

Некоторые проекты с открытым исходным кодом зависят от функциональности в других проектах с открытым исходным кодом. Например, чтобы использовать Spring (фреймворк) необходимо убедиться, что есть совместимые версии всех соответствующих jar-файлов, доступных во время выполнения, для JUnit (библиотека для тестирования). Эта сложная цепочка зависимостей и минимальных версий, часто называемая как “Jar Hell” (ад jar), является отличным способом описания неправильной версии загружаемого класса или даже ClassNotFoundException во время выполнения.

Система ***Java Platform Module System*** (***JPMS***) была введена в Java 9 для группировки кода на более высоком уровне и пытается решить проблемы, с которыми сталкивалась Java с самого начала.

Основная цель модуля – предоставить группы связанных пакетов так, чтобы был доступен специфичный набор функций.

Система модулей платформы Java включает в себя следующие нововведения [1]:

1. Формат для файлов модуля jar (объявление модуля).
2. Разбиение JDK на модули.
3. Дополнительные параметры командной строки.

1 Основные цели и преимущества модулей

1. Надежная конфигурация [2].

Модульность предоставляет механизмы для явного объявления зависимостей между модулями таким образом, который распознается как во время компиляции, так и во время выполнения. Система может пройти через эти зависимости, чтобы определить подмножество всех модулей, необходимых для функционирования и поддержки приложения.

1. Надежная инкапсуляция (сильная инкапсуляция) [2].

Пакеты в модуле доступны для других модулей, только если модуль явно их экспортирует. Даже в этом случае другой модуль не может использовать эти пакеты, если в нем явно не указано, что ему требуется доступ к тому модулю. Это повышает безопасность платформы, поскольку потенциальным злоумышленникам доступно меньше классов. А учитывая модульность, появляется возможность разработки более чистых и логичных конструкций.

Классы в пакетах, которые не экспортируются, могут использоваться только внутри модуля, который их содержит. Такие пакеты называются скрытыми или инкапсулированными.

1. Масштабируемая платформа Java [2].

Ранее платформа Java представляла собой монолит, состоящий из огромного количества пакетов, что усложняло разработку, поддержку и развитие. Поскольку и API-интерфейсы платформы за последние годы существенно увеличились, приложение стали включать большое количество неиспользуемых классов (соответственно, и распространяемое приложение больше, чем нужно). Это может быть проблемой для небольших устройств, таких как мобильные телефоны, Raspberry Pi и т.д.

Теперь платформа объединена в 95 модулей (это число может меняться по мере развития). Появилась возможность создавать пользовательские среды выполнения, состоящие только из тех модулей, которые необходимы для приложений или целевых устройств. Например, если устройство не поддерживает GUI, можно создать среду выполнения, не включающую данные модули GUI.

Таким образом, готовые приложения, распространяемые через модульную платформу Java, обладают меньшим размером.

1. Большая целостность платформы [2].

До Java 9 в платформе можно было использовать много классов, которые не предназначались для использования классами приложений. Благодаря сильной инкапсуляции эти внутренние API-интерфейсы действительно инкапсулированы и скрыты от приложений, использующих платформу. Это может затруднить перенос устаревшего кода на модульную систему, если код зависит от внутренних API.

1. Обнаружение отсутствующих модулей при запуске [3].

Начиная с Java 9, все приложения должны быть упакованы как модули Java. Модуль приложения явно указывает, какие другие модули (модули API Java или модули сторонних производителей) он использует. Поэтому JVM может проверить весь граф зависимостей приложения, при запуске JVM. Если какие-либо обязательные модули не найдены, JVM сообщает об отсутствующем модуле и завершает работу.

До Java 9 отсутствующие классы (например, из отсутствующего JAR-файла) не обнаруживались до тех пор, пока приложение фактически не попыталось использовать отсутствующий класс. Это могло произойти в любой момент времени выполнения.

Уведомление об отсутствующих модулях во время запуска приложения является большим преимуществом для разработчиков.

1. Повышение производительности [2].

JVM использует различные методы оптимизации для повышения производительности приложений. Например, JSR 376 указывает, что эти методы стали более эффективными, так как заранее известно, что требуемые типы находятся только в определенных модулях.

1. Низкая связность модулей [4].

Цель состоит в том, чтобы максимально уменьшить межмодульные связи, и это достигается путем указания интерфейсов для связывания. Интерфейсы скрывают реализацию модуля, позволяя им быть независимыми и взаимозаменяемыми без нарушения функционирования модулей и всего приложения.

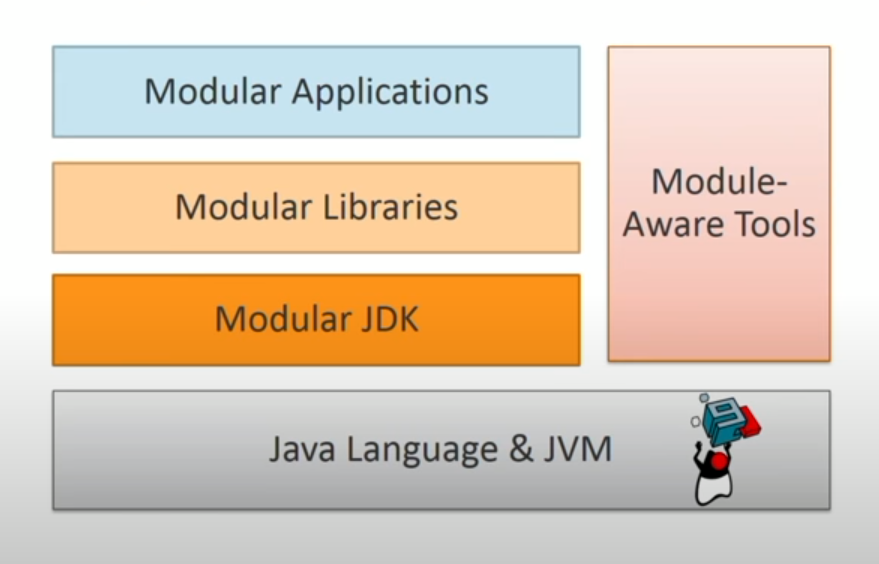
2 Основы модульной системы

2.1 Структура модульной среды

Модуль представляет набор пакетов, который имеет смысл объединить и предназначенный для повторного использования. Модуль может быть, как полным Java-приложением, так и API-платформой Java или сторонним API.

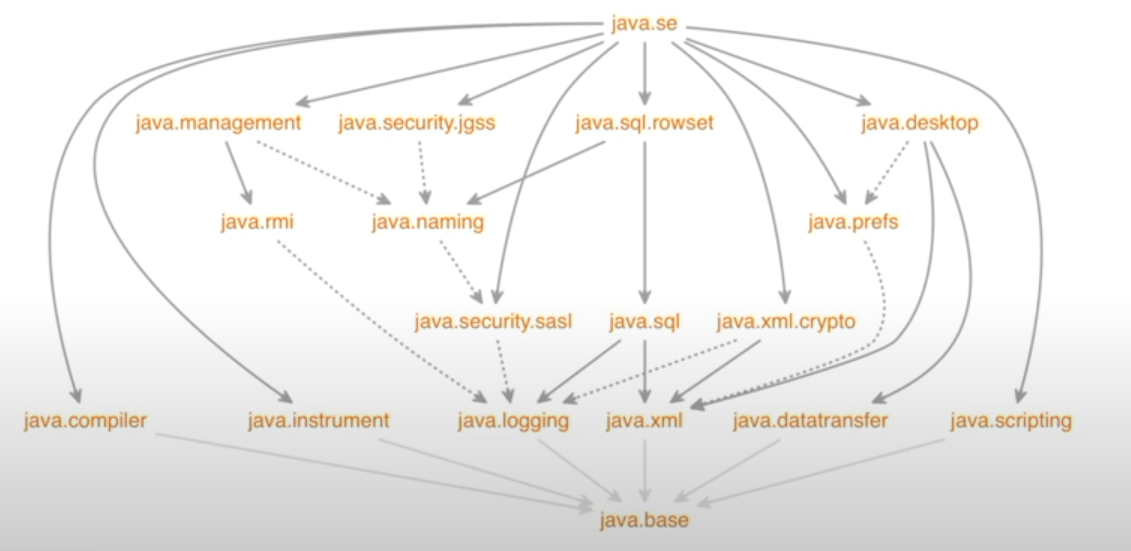
Суть модулей заключается в выражении «*the unit of reuse is the unit of release*».

Структура среды модульного приложения представлена на рисунке [5].



На самом нижнем уровне находится JVM, которая выполняет все низкоуровневые операции, она обеспечивает должный уровень абстракции между скомпилированной программой и базовой аппаратной платформой, и операционной системой.

Следующим слоем представлен JDK, разбитый на модули. Основные модули JDK и их связи представлены на рисунке [5]. Таким образом, основные классы сгруппированы в модуле *java.base*. Все остальные модули, в том числе и модули приложений, зависят от базового.

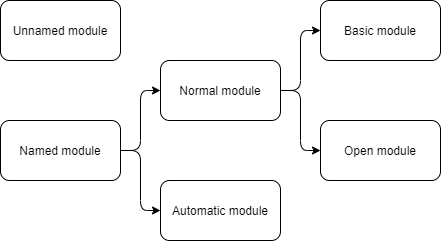


Далее располагаются сторонние библиотеки (modular libraries) также в формате модулей и на самом верхнем уровне – те модули, которые обеспечивают бизнес-логику приложения (modular applications).

Для работы с модулями отдельно представлены инструменты (module-aware tools), обеспечивающие такие процессы как сборка модулей, оптимизация, поиск зависимостей и другие.

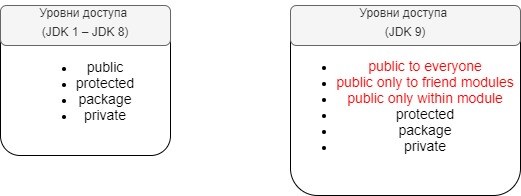
В связи с разделением выделилось несколько типов модулей [6]:

1. Системные (system) – модули платформы, например, JSE, JDK.
2. Модули приложений (application) – именованные модули, необходимые для построения приложения.
3. Автоматические (automatic) – неофициальные модули, которые подключаются как jar, с полностью доступными пакетами.
4. Безымянные (unnamed) – модули, загружаемые по classpath (но не по modulepath). Добавляются автоматически, предназначены для обратной совместимости.



2.2 Уровни доступа

Так как модули предназначены для усиления инкапсуляции, увеличилась и гибкость доступа [7].

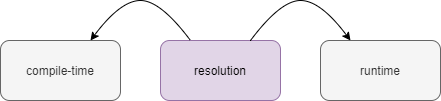


Публичный уровень доступа разделился. Самый открытым является модификатор, разрешающий доступ для всех (public to everyone). Другим вариантом является предоставление доступа только для некоторых (friend), явно указанных, модулей. Также доступ может быть предоставлен всем классам в пределах одного модуля. Таким может быть, например, класс утилит для внутреннего форматирования логов.

Такие уровни могут вводить в заблуждение в начале, однако после понимания позволяют реализовывать гибкие и защищенные инфраструктуры.

2.3 Разрешение зависимостей

Разрешение зависимостей (module resolution) представляет собой процесс, который проверяет правильность пути к модулю, а также анализирует и оптимизирует все зависимости, существующие в модульной системе. Это происходит как во время компиляции, так и во время выполнения [4].

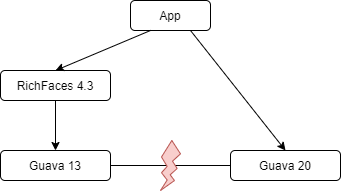


Целью процесса разрешения модулей является получение минимально необходимого набора разрешенных модулей для запуска приложения.

Как уже было сказано, модульная система во время компиляции сообщит об отсутствии необходимых для приложения модулей. Кроме того, во время компиляции во время разрешения зависимостей модульная система проверяет структуру на циклические зависимости. И в случае нахождения, также выдаст ошибку компиляции.

Другую проблему, которую способна обнаружить система при разрешении зависимостей – неявные конфликты версий модулей.

Например, две требуемые библиотеки зависят от разных несовместимых версий третьей библиотеки. Если в пути к классу присутствуют обе версии, поведение будет непредсказуемым. С другой стороны, если одна из версий отсутствует, программа, скорее всего, не будет работать правильно, потому что обе версии необходимы и по предположению несовместимы, что означает, что они не могут заменять друг друга [4].



Описанные версионные проблемы также способны обнаружить некоторые сборщики приложений (например, Maven) во время компиляции либо во время выполнения.

3 Объявление модуля

3.1 Дескриптор модуля

Структура модуля не отличается от структуры классического приложения на Java. За исключением его объявления, которое происходит в корневой директории проекта в файле *module-info.java*.

Минимальное объявление модуля (пустой дескриптор) очень простое [8]:

**module** <module.name> {

}

Оно включает ключевое слово ***module***, название модуля и фигурные скобки; все директивы опциональны.

Компиляция объявления модуля создает дескриптор модуля – файл с именем *module-info.class* в корневой папке модуля.

В целом структура файла описания модуля следующая [9]:

1. Имя модуля – наименование, удовлетворяющее JNC (Java Name Convention).
2. Зависимости – список модулей, от которых зависит данный.
3. Видимые (public) пакеты – пакеты, которые будут доступны извне.
4. Предоставляемые сервисы.
5. Используемые сервисы.
6. Разрешение рефлексии – необходимо явно дать разрешение другим классам использовать рефлексию для доступа к зарытым членам пакета.

Описание эффективного модуля задача не тривиальная, поэтому будут рассмотрены только основные части.

Так, базовыми директивами являются [2, 4, 8, 9]:

1. requires;
2. exports;
3. uses;
4. provides;
5. opens;

3.2 Requires и Exports

Директива ***requires*** указывает, что данный модуль зависит от другого. Каждый модуль должен явно указать свои зависимости. Когда для модуля A требуется модуль B, говорят, что модуль A читает модуль B, а модуль B читается модулем A.

Для указания зависимости от другого модуля, используется следующая запись:

**requires** <module.name>;

Данная зависимость является compile-time & runtime, другими словами отсутствии данного модуля на любом этапе выдаст ошибку.

Существует также статическая директива ***requires***, указывающая, что модуль требуется во время компиляции, но является необязательным во время выполнения.

**requires static** <module.name>;

Чтобы указать зависимость от другого модуля и убедиться, что модули, читающие текущий модуль, также прочитают эту зависимость (подразумеваемая читаемость, “ implied readability”, транзитивная зависимость) – используют следующую форму:

**requires transitive** <module.name>;

Системные модули обеспечивают транзитивную зависимость во всех случаях. Кроме того, хотя стандартный модуль Java SE может зависеть от нестандартных модулей, он не разрешает им подразумеваемую читаемость. Это гарантирует, что код, зависящий только от стандартных модулей Java SE, будет переносимым между реализациями Java SE.

Директива модуля ***export*** указывает один из пакетов модуля, чьи открытые (public) типы (и их вложенные public и protected типы) должны быть доступны для кода во всех других модулях.

Записывается в следующей форме:

**exports** <com.package.name>;

Для предоставления доступа к пакету только определенным (friend) модулям используется конструкция, называемая квалифицированным экспортом:

**exports** <com.package.name> to <module.name.A>,

<module.name.C>;

Каждая директива ***exports*** / ***requires*** для соответствующего модуля / пакета должна располагаться на новой строке.

3.3 Uses и Provides

Директива ***uses*** определяет сервис, используемый данным модулем, что делает модуль потребителем сервиса.

Под сервисом (службой) понимается объект класса, который реализует интерфейс или расширяет абстрактный класс, указанный в директиве uses.

**uses** <interface.name>;

Здесь следует отметить, что между директивой ***require*** и директивой ***users*** есть разница.

Может потребоваться модуль, предоставляющий сервис, нужный приложению, но этот сервис реализует интерфейс из одной из своих транзитивных зависимостей.

Вместо того, чтобы заставлять модуль требовать все транзитивные зависимости «на всякий случай», следует использовать директиву ***uses*** для добавления необходимого интерфейса в путь к модулю.

Директива ***provides*** указывает реализацию сервиса, предоставляемую данным модулем, что делает модуль поставщиком сервиса.

Первая часть директивы указывает интерфейс или абстрактный класса. Во второй части предоставляется имя класса реализации, которое либо реализует интерфейс, либо расширяет абстрактный класс.

**provides** <interface.name> **with** <impl.class.name>;

Каждая директива ***uses*** и ***provides*** для соответствующего интерфейса API должна располагаться на новой строке.

3.4 Opens

Ранее было упомянуто, что инкапсуляция была движущей силой при разработке модульной системы. До Java 9 было возможно использовать рефлексию, чтобы исследовать каждый тип и элемент в пакете, даже private. Ничто не было действительно инкапсулировано, что открывало все виды проблем для разработчиков библиотек.

Поскольку Java 9 обеспечивает сильную инкапсуляцию, теперь необходимо явно предоставлять разрешение другим модулям на рефлексию классов.

Существует несколько уровней доступа для модуля.

1. Разрешение полной рефлексии модуля (через ключевое слово):

**open module** <module.name> {}

1. Разрешение рефлексии специфических классов (директива):

**opens** <com.package.name>;

1. Разрешение рефлексии специфических классов только некоторым (friend) модулям (директива):

**opens** <com.package.name> to <module.name.A>, <module.name.B>;

Во всех случаях доступ предоставляется только во время выполнения.

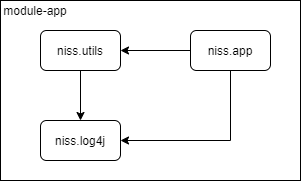
4 Пример модульного приложения

Рассмотри простой пример – пусть требуется некоторое приложение для нахождения значения функций с поддержкой логирования.

Данный пример предназначен, чтобы показать объявление модулей, их применение и сборку встроенными средствами Java 9 и выше.

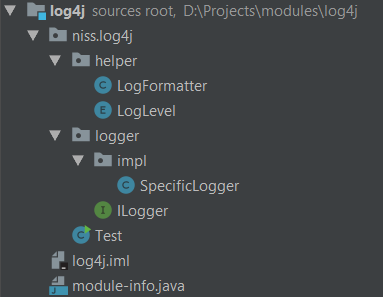
Пример НЕ стоит рассматривать как лучшее архитектурное решение.

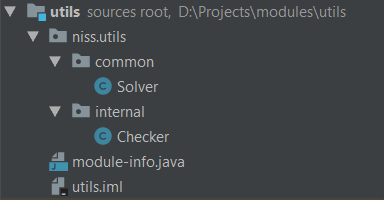
Можно выделить 3 модуля.

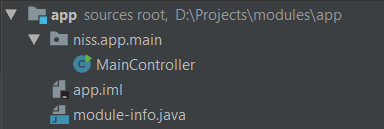


1. Модуль ***app*** как основной, содержащий бизнес-логику приложения.
2. Модуль ***utils***, позволяющий составлять функциональные маппинги операторов для решения уравнений от одной переменной.
3. Модуль ***log4j*** для поддержки элементарного логирования.

Иерархия каждого модуля представлена далее.







Модуль ***log4j*** будет реализовывать консольное цветное логирование и предоставлять этот сервис. Другими словами, он должен предоставить сервис, но сокрыть внутреннее устройство данного процесса.

Это позволяет в дальнейшем заменить его на модуль, предоставляющий, например, запись логов в файлы.

Модуль ***utils*** будет принимать список операторов, преобразовывать его в единый маппинг операторов и с его помощью находить значения полученной функции в некоторых точках. А также фильтровать список значений по какому-то условию, например, возвращать четные или простые числа.

Основному модулю требуются инструменты для решения уравнений. Модули приложения и инструментария должны иметь возможность вести логи своих действий.

Код проекта доступен для скачивания: [].

5 Инструменты командной строки

Сборщики проектов как Maven или Gradle поддерживают работу с модулями. Однако понимание построения модульных приложений остается необходимым. Кроме того, возможно ситуация, когда стандартный плагин сборщика не сможет собрать модули с требуемой структурой.

В Java 9 были добавлены инструменты, обеспечивающие описанные ранее процессы функционирования модулей. Будут рассмотрены основные опции командной строки на примерах созданных ранее модулей.

5.1 Средства компиляции и запуска

Команда для компиляции модуля:

**javac** **-d** <outDir> [\*.java]

где ***–d*** указывает <outDir> – директорию, в которой будут располагаться скомпилированные классы;

*[\*.java]* – перечисление всех java-файлов.

В Unix системах можно использовать команду поиска файлов:

$(**find** dirName -name "\*.java").

P.S. Кто знает “Windows-решение” данной задачи, просьба сообщить.

Так, для компиляции модуля ***log4j*** необходимо выполнить:

**javac** **–d** out\log4j log4j\niss\log4j\helper\\*.java

log4j\niss\log4j\logger\\*.java

log4j\niss\log4j\logger\impl\\*.java

log4j\niss\log4j\\*.java

log4j\module-info.java

Да, нужно искать более изящное решения для Windwos.

Модуль ***utils*** явно зависит от модуля ***log4j*** (указано в описании модуля), поэтому для его сборки необходимо дополнительно указать путь к внешнему модулю. Для этого используется опция ***--module-path*** (или сокращенно ***-p***):

**javac** **–d** out\utils **–p** out\log4j [\*.java]

Указывать в качестве зависимостей обычные директории с классами, как и распространение их в таком виде, не очень удобно. В большинстве случаев опции ***--module-path*** передается jar-файл. Таким же образом можно добавить зависимость от автомодуля (например, mysql-driver-6.0.jar).

Для построения jar-файла используется команда:

**jar** **-cvf** mods\niss.log4j.jar **-C** out\log4j **.**

где флаги ***–cvf*** означают создать jar-файл с именем mods\niss.log4j.jar и вывести информацию о нем;

***-C*** *classes\* ***.*** – в указанной директории (в которой хранятся скомпилированные классы модуля) включить все найденное файлы в jar.

После того, как все модули были собраны в jar-файлы, можно запустить основной модуль:

**java** **–p** <modsDir> **-m** <module.name>**/**<full.class.name>

где ***–p*** (***--module-path***) указывает директорию, в которой хранятся jar-файлы,

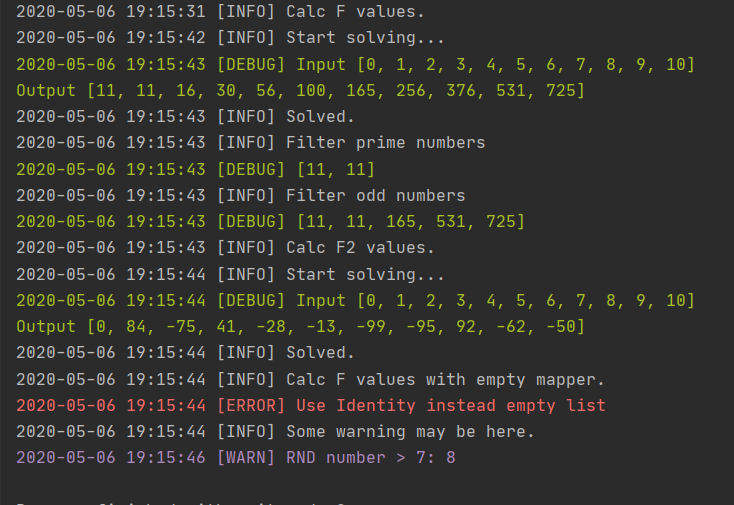
***-m*** (***--module***) указывает полное имя запускаемого класса.

Слеш обязательно прямой, независимо от используемой ОС.

Команда для запуска приложения для вычисления значений функций:

**java** **–p** mods **–m** niss.app**/**niss.app.main.MainController

Пример вывода программы:



5.2 Дескриптивные инструменты

Существуют опции командной строки эквивалентные дескрипторам модуля. При обычных условиях стоит избегать их использование, однако для некоторых вспомогательных модулей они облегчают жизнь разработчику.

***--add-reads*** эквивалент ***requires***

***--add-exports*** эквивалент ***exports***

***--add-opens*** эквивалент ***open***

***--illegal-access={permit|war|deny}*** эквивалент ***open[s]***

5.3 Инструменты для анализа зависимостей

Нередко появляется необходимость узнать больше информации о модуле, чем его название.

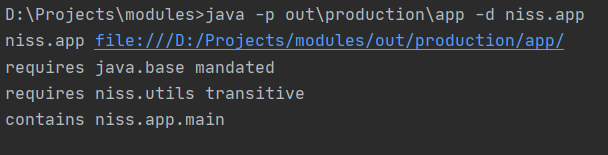
Было добавлено несколько инструментов для анализа описания и анализа модулей. Самый простой – посмотреть информацию, представленную в дескрипторе:

**java** **–p** <modsDir> **--describe-module** <module.name>

где опция ***–d*** (***--describe-module***) указывает, что требуется описание данного модуля.

Для тестового модуля:

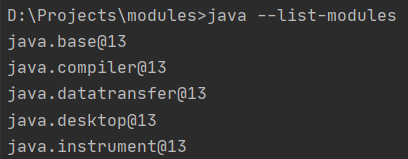
**java** **–p** out\production\app **–d** niss.app



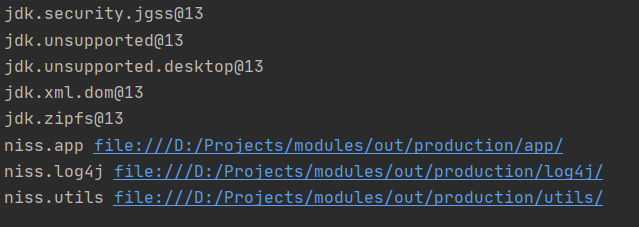
Как уже упоминалось, все модули зависят от *java.base*, который является основой JSE. Для просмотра всех доступных модулей достаточно выполнить:

**java** **--list-modules**

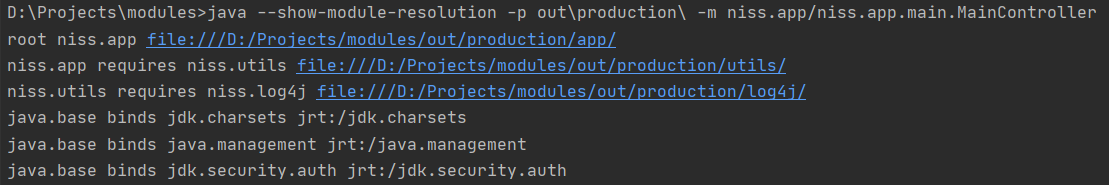
Первые несколько строк для Java 13:



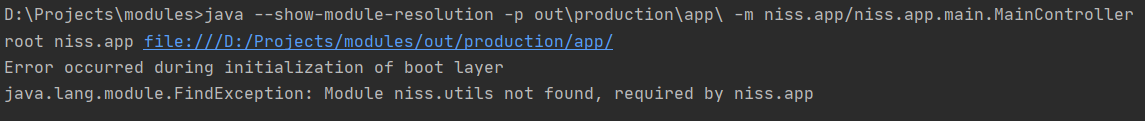
Если добавить путь к другим модулям, они будут выведены после встроенных, например:



Если информации из дескриптора недостаточно, опция ***--show-module-resolution*** выведет подробное дерево зависимостей, начиная от корня проекта, перед непосредственным запуском. Начало вывода для тестового проекта:

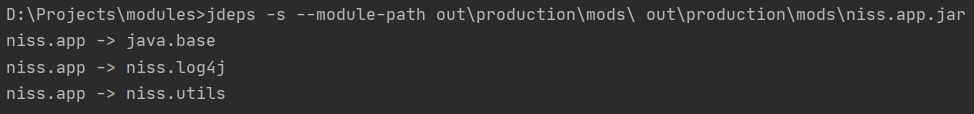


Если какой-либо из требуемых модулей не найден, будет выведена соответствующая ошибка:



Однако, рассматривать подобную информацию перед каждым запуском программы не самый удобный вариант. Для данных целей стоит воспользоваться командой ***jdeps***, предоставляющей информацию о зависимостях внутри модуля в более удобном для чтения виде.

В отличие от простого описания модуля, команда выводит зависимости, которые на самом деле используются, а не просто объявлены.



В качестве опции указана ***–s*** (***–summary***), обеспечивающая краткий вывод. Для полного и подробного вывода следует воспользоваться командой напрямую:



6 Дополнительно

В данном курсе была рассмотрена лишь верхушка айсберга модульной системы. Но она является необходимой основой для всех остальных нововведений в мире модульной Java.

С имеющимися знаниями вы можете создавать собственные модульные приложения, постепенно усложнять их структуру, создавать архитектурные решения.

Для более глубокого понимания следует обратиться к соответствующей литературе (например, [3] и [4]).

Использование иллюстраций и исходного кода без ссылки на автора не допускается.

Использованные источники

1. Java 9 API Specification [Электронный ресурс]. // Oracle Doc. URL: <https://docs.oracle.com/javase/9/docs/api/overview-summary.html> (дата обращения 25.04.2020).
2. Understanding Java 9 Modules [Электронный ресурс]. // Oracle. 2017. URL: <https://www.oracle.com/corporate/features/understanding-java-9-modules.html> (дата обращения 30.04.2020).
3. The Java Module System. 1st Edition. Nicolai Parlog. Manning Publications, 2019, – 440 p.
4. Java 9 Modularity Revealed. Alexandru Jecan. 2017, – 236 p.
5. Modular Development with JDK 9 [Электронный ресурс]. // Oracle. 2017. URL: <https://youtu.be/gtcTftvj0d0> (дата обращения 25.04.2020).
6. A Guide to Java 9 Modularity [Электронный ресурс]. // Baeldung. URL: <https://www.baeldung.com/java-9-modularity> (дата обращения 02.05.2020).
7. JEP 261: Module System [Электронный ресурс]. // OpeJDK Doc. URL: <http://openjdk.java.net/jeps/261> (дата обращения 25.04.2020).
8. Oracle Certified Professional Java SE 11 Programmer I. Study Guide. Jeanne Boyarsky, Scott Selikoff. 2020, – 820 p.
9. Java Modules [Электронный ресурс]. // Jenkov. URL: <http://tutorials.jenkov.com/java/modules.html> (дата обращения 02.05.2020).