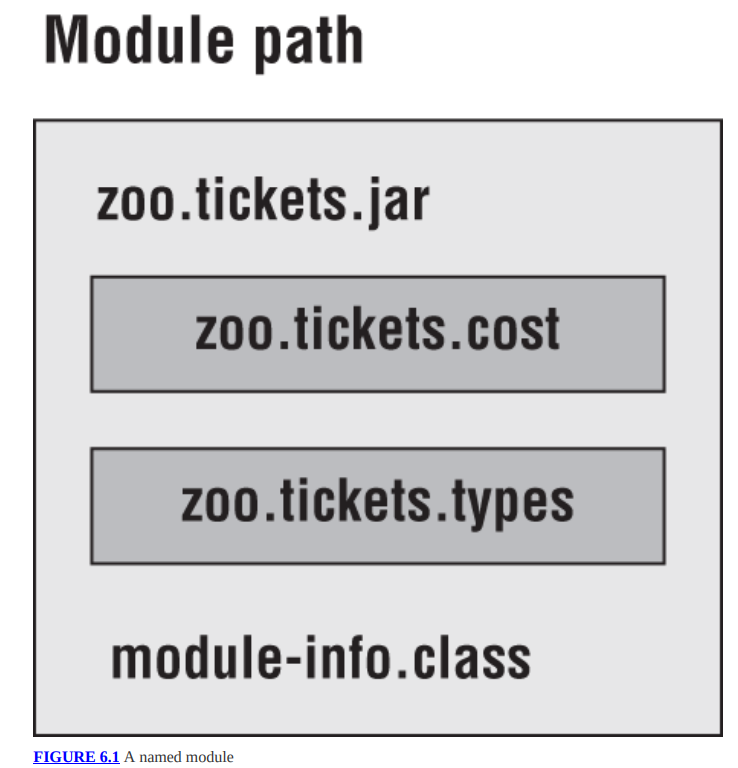
**Глава 6. Модульные приложения**

Существует 3 типа модулей, которые изучены выше – именованые модули, а так же присутствуют неименованные, автоматические моудли.

Именованый модуль – это тот модуль, который содержит module-info.java файл

Примера структуры именнованного модуля:

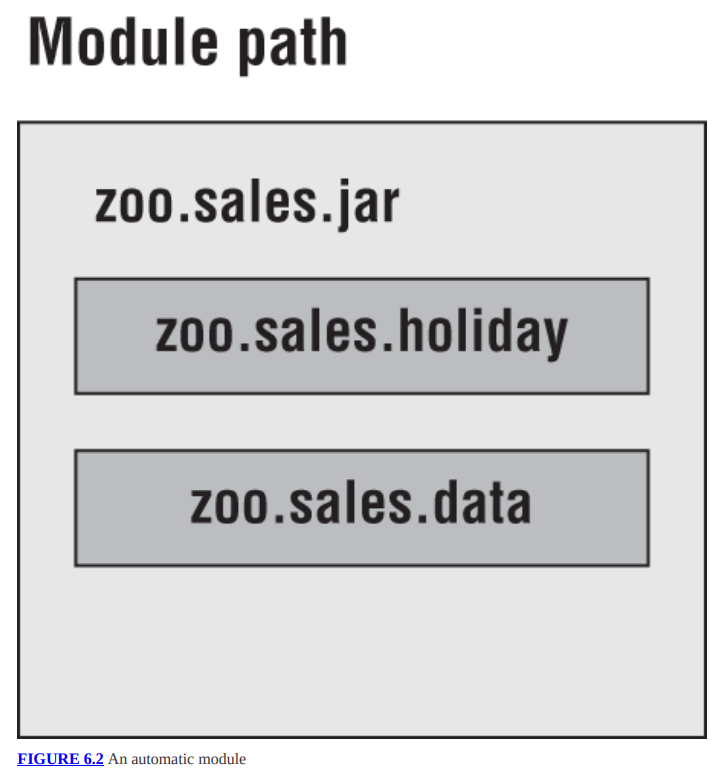


**Автоматические модули**

Автоматический модуль появляется в пути модуля но не содержит module-info файл. Это просто обычный JAR файл, который расположен в module path и принимает обработку как модуль.

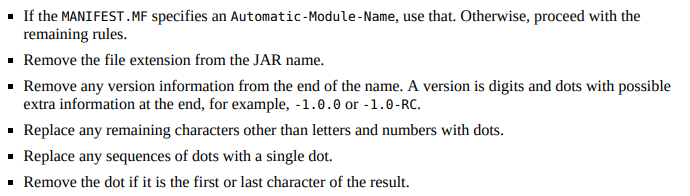
Как вариант запомнить это, Жаба автоматически определяет имя модуля.

Пример автоматического модуля с двумя пакетами:

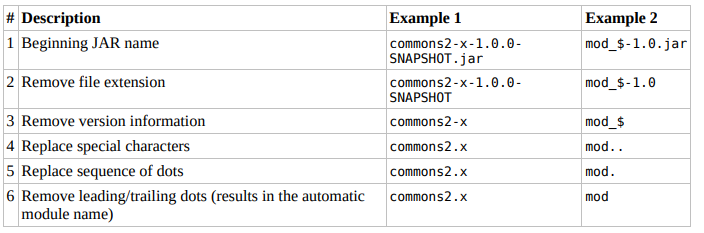


В имени модуля не разрешен дефисы. При автоматическом проставлении наименовании модуля джава сама заменяет их на точки. И далее соседние точки или в самом начале или конце удаляются.

Правила при автоматическом именовании модуля:



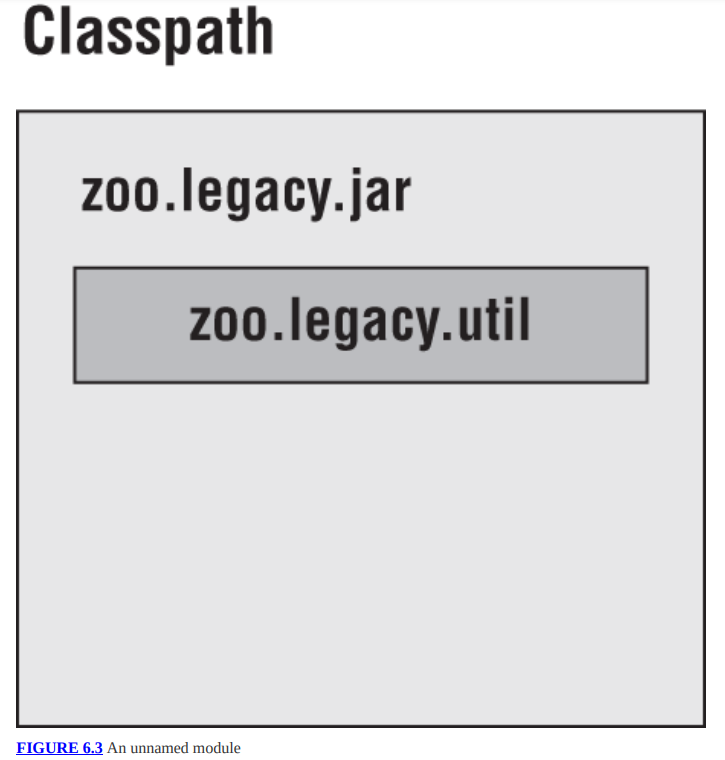
Пример автоматического именования:



**Неименованные модули:**

Неименованные модули появляются в classpath. Как автоматический модуль это обычный JAR. Неименованный модуль обрабатывается как старый код и как второстортный для модулей

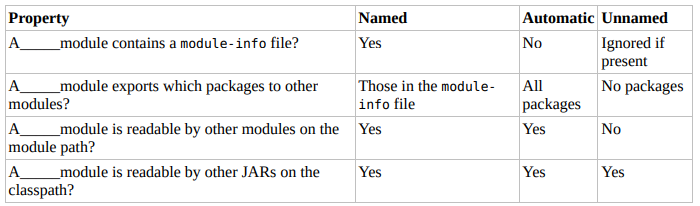
Пример структуры неименованых модулей:



Обычно неименованный модуль не содержит файл module-info, но даже если содержит, то он игнорируется, т.к. находится в classpath.

Неименованные модули не экспортируют какие либо пакеты именованным или автоматическим модулям. Неименованные моудли читают из любых JAR в classpath или module path. Можно думать, что неименоыванные модули выступают как код, который работает так же как и до введения модулей.

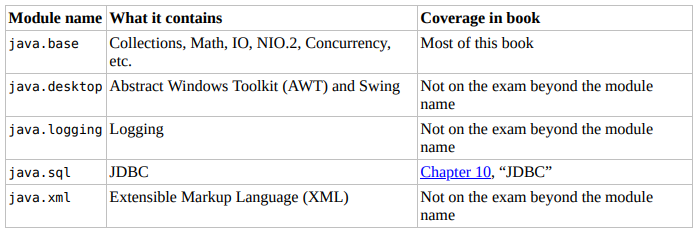
Свойства весь типов модулей для сравнения:



**Анализ зависимостей JDK**

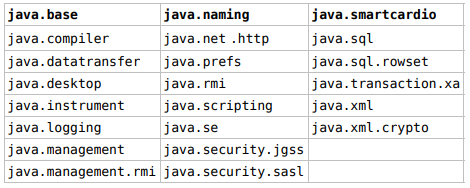
Самый важный модуль в jdk это java.base – это общий модуль, который содержит в себе все пакеты и классы необходимые для экзамена. Он автоматически экспортируется во все созданные модули и его явный экспорт не обязателен.

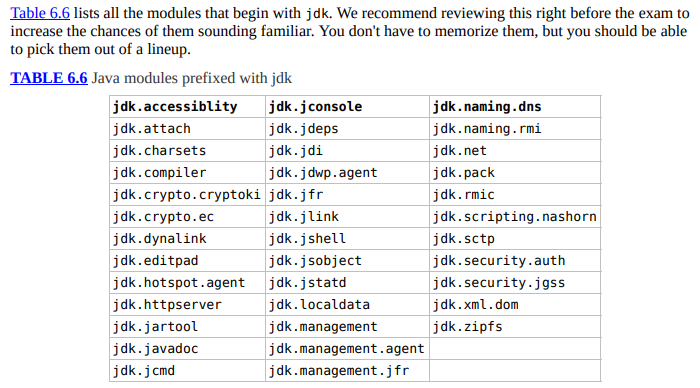
Пример общих модулей:



Так же нужно знать, что есть модули в jdk начинающиеся на java. и jdk.

Примеры данных модулей:





**Using jdeps**

Jdeps – команнда, позволяющая вывести информацию о зависимостях в модуле. Данная команда точно скажет где были использованы зависимости, а не просто опишет их.

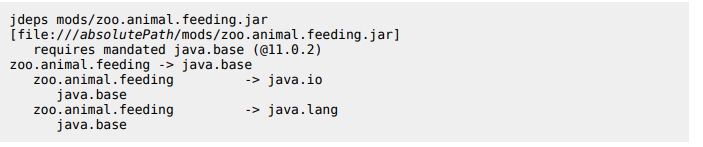
Для того, чтобы получить описание зависимостей использутеся:



Вывод команды представляет собой вывод зависимостей модулей от других:

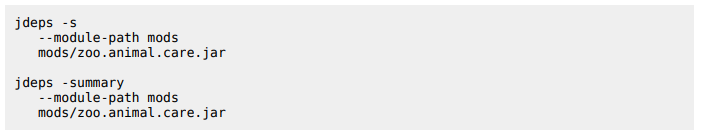


Если же убрать флаг –summary, то получим более длинный вывод:



Здесь более подробная информация и показывает как называется модуль, его путь, его требуемые зависимости от других модулей и показывает зависимости на пакетном уровне, т.е. какие пакеты используют другие пакеты и из каких модулей.

Но у нас можно использовать следующие команды:



При помощи флага –jdk-internals можно получить список модулей, которые вызывают internal API и далее приводит рекомендации для их замены:

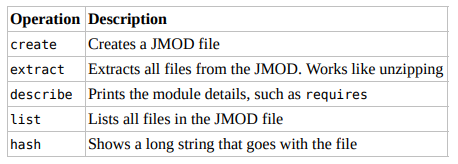


-jdkinternals === --jdk-internals

**Jmod команда**

Данная утилита позволяет работать только с такими же jmod файлами и лучше вообще избегать их использования в реальном мире.

Операции jmod:



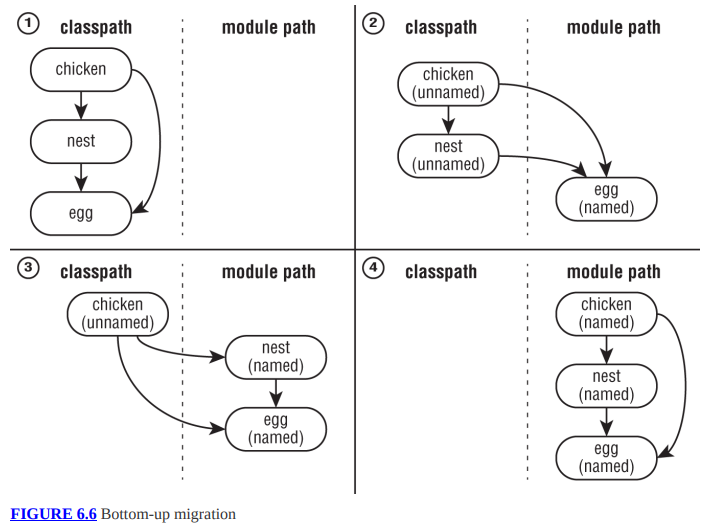
**Миграция на модульную систему.**

Всего существует несколько способов для миграции.

Первый способ, называемый **Bottom-Up migration,** т.е. миграция снизу вверх:

1. Взять самый низко уровневый проект который не был еще мигрирован
2. Добавить module-info.java файл в этот проект. Нужно быть уверенным добавить любые exports для того чтобы расширить любые пакеты ипользованные на высшем уровне. Так же добавить requires для модулей, от которых зависит данный проект
3. Поменять новый именованный модуль с classpath на module path.
4. Убедиться что любые проекты который не были еще мигрированы остаются неименованными модулями в classpath
5. Повторить со следующим самым низкоуровневым проектом и так до тех пор пока не закончится миграция.

Пример миграции снизу-вверх:

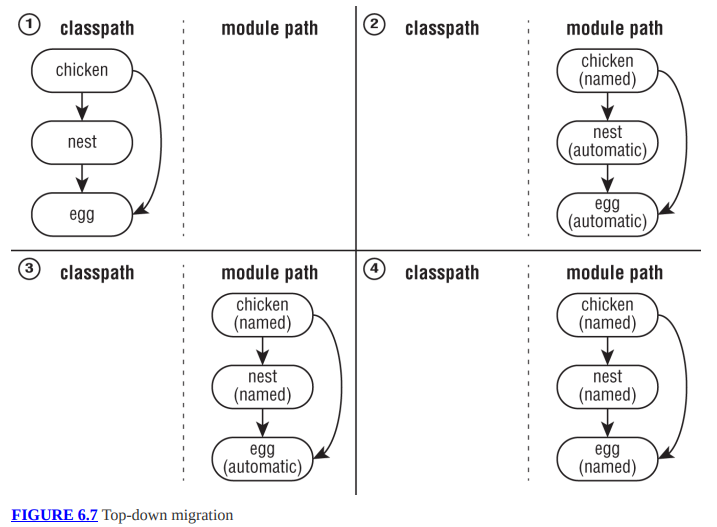


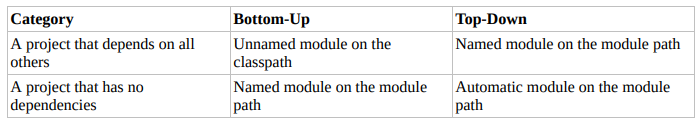
Этот подход работает наилучшим образом, когда у нас есть способность конвертировать любые JAR файлы которые не являются модулями.

Второй подход для миграции: **Top-Down migration**, т.е. сверху-вниз

1. Расположите все проекты в module path
2. Возьмите самый верхнеуровневый проект
3. Добавить module-info.java файл в этот проект для того чтобы конвертировать автоматический модуль в именованный модуль. Далее снова не забыть добавить exports и required директивы. Можно использовать автоматические имена модулей других модулей когда нужно написать requires директиву, т.к. большая часть модулей в проекте еще не имеет имен.
4. Повторить со следующим самым высокоуровневым проектом и так пока миграция не окончится.

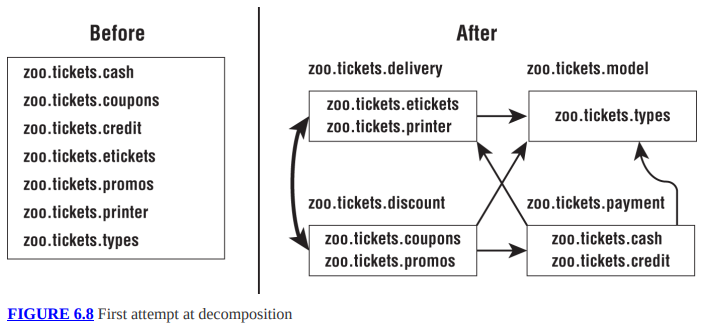
Данный вид миграции наиболее применимый когда нет контроля над каждым jar файлом использующимся в приложении. Например, предположим, другая команда владеет одним проектом. Они очень заняты для того, чтобы мигрировать, например. Но нам не нужно стопорить всю миграцию.



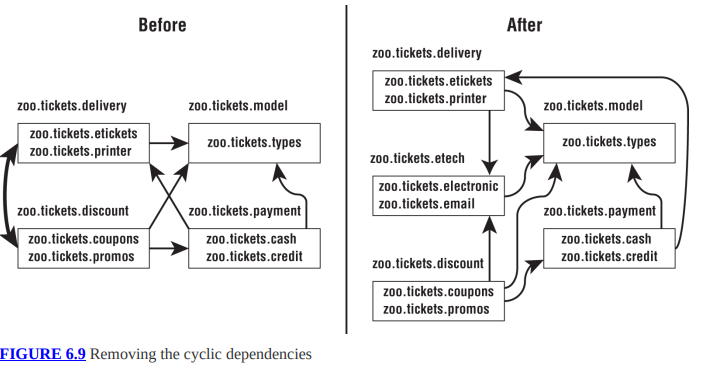


**Разделение большого проекта на модули.**

Предполагаем, что у нас есть проект с нексолькими пакетами. Первый шаг это разделить их на логические группы и начертить зависимости между ними.



На данном графике можно увидеть циклическую зависимость, она не допустима. Для того, чтобы избавиться от нее, необходимо выделить еще один модуль, содержащий общий код для циклически зависимых модулей.

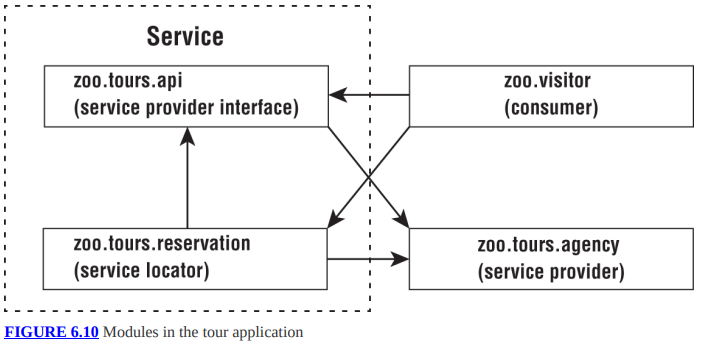


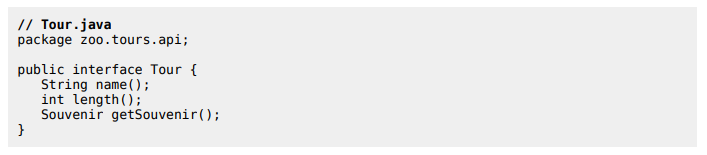
Использование модульной системы является приемуществом с точки зрения защиты от циклических зависимостей. Если два модуля ссылаются друг на друга, то при попытке скомпилироваться будет выброшена ошибка.

**Создание сервиса.**

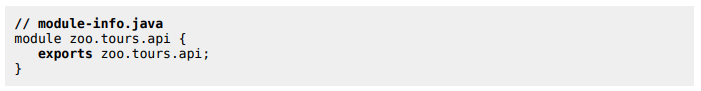
Сервис это интерфейс, любой класс, ссылающийся на интерфейс и пусть поиска для имплементации этого интерфейса. Имплементация не является частью сервиса.

Пример модульной системы:







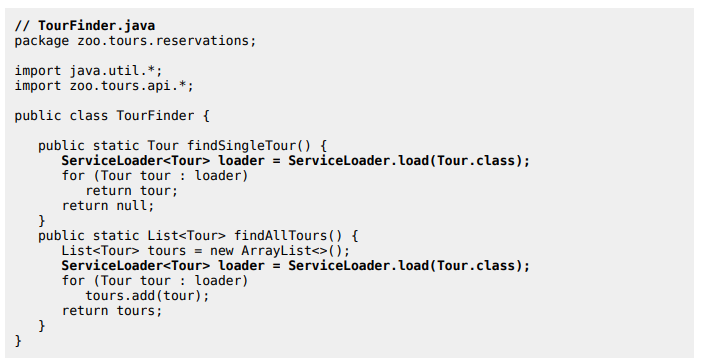


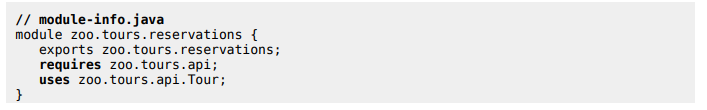
//реализация модуля zoo.tours.api

Далее он компилируетя и собирается в jar

Теперь создадим сервис локатор

В java уже есть удобный класс ServiceLoadre<T> который по class файлу может найти все реализации интерфейсов, которые используются у нас в программе.





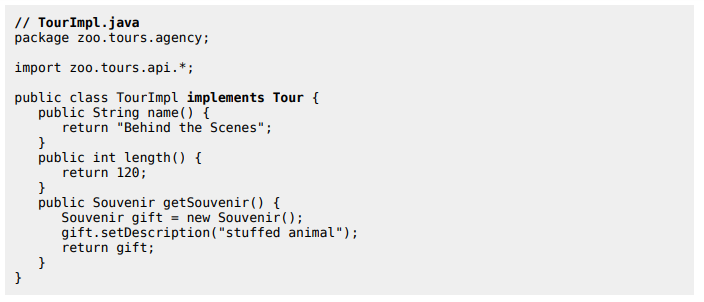
//реализация модуля zoo.tours.reservations

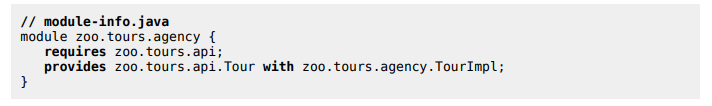
Важно заметить, что нам требуется и requires и uses директивы. Далле мы компилируем и собираем в jar файл так же как всегда.

Далее используя директиву requires для сервис локатора и сервис провайдера мы может использовать этот поиск интерфейса для дальнейшего его использования.

Но нам неоходимо реализовать этот интерфейс и инициализировать.

Service provider это реализация service provider interface.





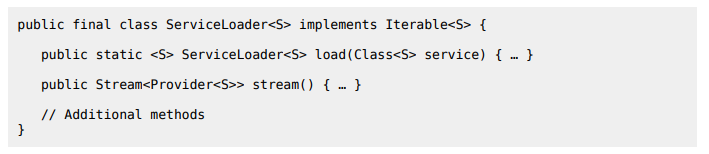
Так же важно заметить что provides ипользуется совместно reaquires.



Вообще примущество в том, что нам не нужно пересобирать все другие модули, мы просто создали провайдер, скомпилировали и собрали в jar при этом все работает и будет работать дальше.

**Совмещение сервис – локатора и потребителя.**

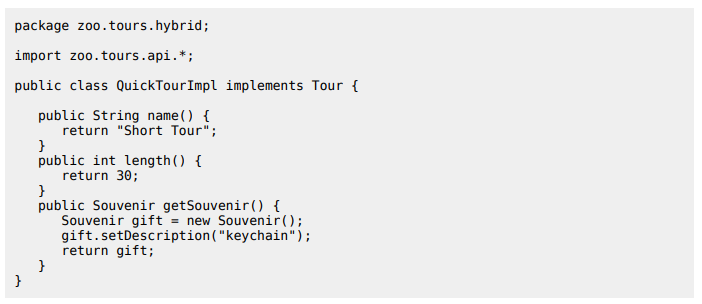
Класс ServiceLoader для загрузки текущих сервисов:

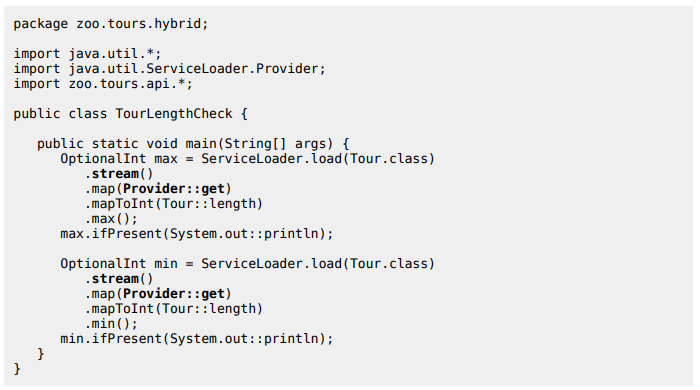


Load Ищет использованные сервисы по его имплементации и можно дальше проитерировать их через цикл и посмотреть какие сервисы используются.

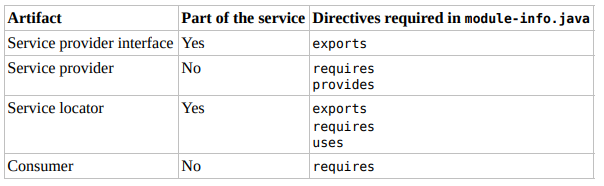
Но через метод stream() возвращается совершенно другй класс в стриме Provider<S>

Это сделано для контроля. По методу get() можно получить любой сервис который нужен:





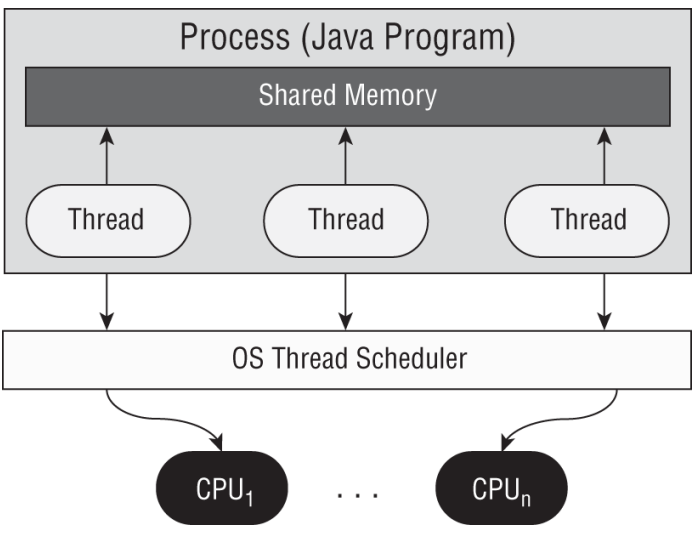
Обзор service provider interface, service provider, service locator и consumer:



**ГЛАВА 7. CONCURRENCY**

**Threads**

Модель потоков:



Task – это единичный юнит работы, подготовленной потоком.

Системный поток – поток, созданный JVM и запускается в бекграунде приложения.

Пользователе-определенный поток – это поток, созданный разработчиком для обработки специальной задачи.

Context switch – процесс записи состояния текуего потока и потом перезаписывает состояние потока для продолжения выполнения.