

**Федеральное государственное автономное образовательное учреждение
высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королёва»**

Факультет информатики

Кафедра технической кибернетики

Лабораторная работа №1
по дисциплине
«Интеллектуальный анализ данных»

Введение в Apache Spark

Оглавление

Введение.....	3
Цель работы.....	3
Работа в консоли spark-shell.....	4
Основные операции взаимодействия с распределённой файловой системой.....	4
Считывание данных.....	6
Обработка текста.....	7
Операции с множествами.....	9
Общие переменные.....	10
Широковещательные переменные.....	10
Аккумулирующие переменные.....	11
Пары ключ-значение.....	12
Анализ данных: десять наиболее популярных номеров такси, на основе количества поездок.....	12
Создание проекта на локальном компьютере.....	15
Разработка с использованием системы сборки SBT и языка Scala.....	15
Анализ данных велопарковок.....	23
Трансформации с использованием внутреннего представления.....	27
Настройка RDD хранения.....	31
Приложение А.....	33
Краткое описание файловой системы HDFS.....	33
Приложение Б.....	34
Основные понятия java.time.....	34

Введение

Apache Spark — программный каркас с открытым исходным кодом для реализации распределённой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop. В отличие от классического обработчика из ядра Hadoop, реализующего концепцию MapReduce с дисковым вводом и выводом, Spark использует специализируется на обработке в оперативной памяти, благодаря чему позволяет получать значительный выигрыш в скорости работы для некоторых классов задач. В частности, возможность многократного доступа к загруженным в память пользовательским данным делает библиотеку привлекательной для алгоритмов машинного обучения.

Главной абстракцией Spark фреймворка является распределённая коллекция элементов Resilient Distributed Dataset (RDD). К RDD можно применить трансформации (transformation) и действия (action). В первом случае в качестве результата возвращается ссылка на новый RDD, а во втором, вычисленное значение цепочки трансформаций.

В папке с заданием содержатся следующие наборы данных:

- warandpeace.txt - текст «Война и мир» Л.Н. Толстого,
- nyctaxi.csv,
- nyctaxisub.csv,
- trips.csv,
- stations.csv.

Цель работы

- повторить операции загрузки и выгрузки данных в HDFS,
- ознакомиться с базовыми операциями Apache Spark, объединением данных оператором Join, оптимизацией трансформаций,
- создать простейший проект по обработке данных в IDE,
- провести анализ данных велопарковок на локальном компьютере.

Работа в консоли spark-shell

Примечание. В первой части задания вы работаете с 3 файловыми системами:

- файловой системой Windows, в которой находятся файлы с заданиями,
- файловой системой Linux узла кластера, к которому вы подключаетесь по ssh и
- распределённой файловой системой MapR-FS (краткое описание приведено в приложении А).

Основные операции взаимодействия с распределённой файловой системой

В лабораторной будет использоваться MapR-FS --- API совместимая с HDFS реализация распределённой файловой системы (РФС) от MapR.

Для импорта/экспорта данных в РФС используйте команды `hadoop fs -put` и `hadoop fs -get`:

```
$ hadoop fs -put путь-в-локальной-системе путь-в-hdfs
```

```
$ hadoop fs -get путь-в-hdfs путь-в-локальной-системе
```

Список остальных команд для взаимодействия с РФС вы можете посмотреть, выполнив `hadoop fs` без дополнительных ключей. Мы рассмотрим примеры работы с наиболее полезными командами далее.

Вызовите в папке с файлами лабораторной powershell, либо git bash. Переместите наборы данных в РФС и проверьте, есть произошло ли это в действительности.

```
$ hadoop fs -put *
```

```
$ hadoop fs -ls
```


Следует обратить внимание на то, что в командах не указывалась директория, в которую перемещались данные и список файлов которой вывелся второй командой.

Попробуйте другие команды, например `-mkdir`, `-cat`, `-df`, `-du`. После того как вы освоитесь с перемещением данных в РФС, создайте подключение к кластеру по `ssh` и запустите в нём `spark-shell`.

```

vlpr@master:~
Windows PowerShell
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

PS C:\Users\vlpr> ssh master.ssau.ru
Last login: Sun Sep  8 00:47:05 2019 from 91.222.131.144
(base) [vlpr@master ~]$ spark-shell
Warning: Unable to determine $DRILL_HOME
Spark context Web UI available at http://master.ssau.ru:4040
Spark context available as 'sc' (master = yarn, app id = application_1567703169591_0009).
Spark session available as 'spark'.
Welcome to

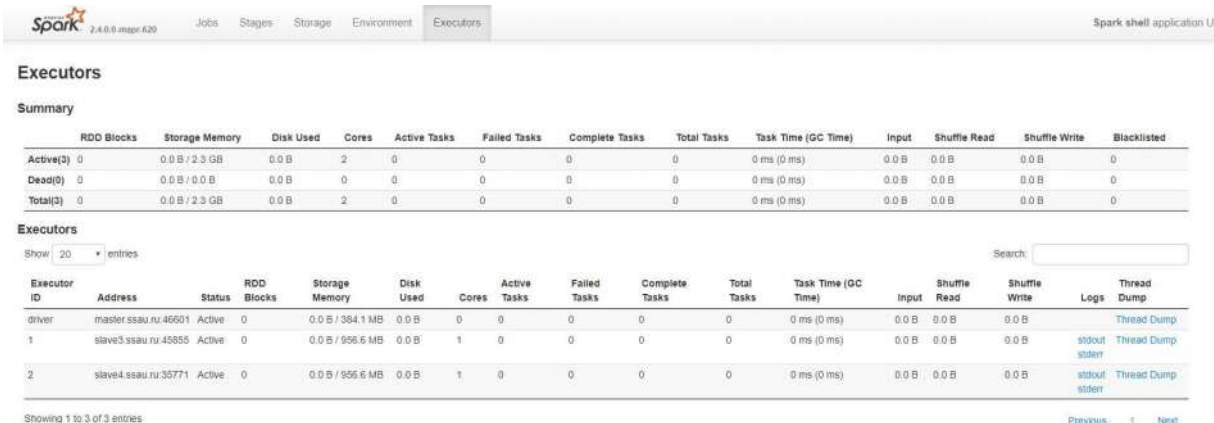
 version 2.4.0.0-mapr-620

Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_191)
Type in expressions to have them evaluated.
Type :help for more information.

scala>

```

Зайдите на веб-страницу запущенной сессии `spark-shell`.



Executors Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(3)	0	0.0 B / 2.3 GB	0.0 B	2	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(3)	0	0.0 B / 2.3 GB	0.0 B	2	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	master.ssau.ru:46601	Active	0	0.0 B / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump
1	slave3.ssau.ru:40805	Active	0	0.0 B / 956.6 MB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump
2	slave4.ssau.ru:35771	Active	0	0.0 B / 956.6 MB	0.0 B	1	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump

Примечание. Попытка привязки веб интерфейса начинается с номера порта 4040 и продолжается с увеличением на единицу пока не будет найден свободный порт.

Также зайдите на веб-страницу менеджера ресурсов YARN <https://slave2.ssau.ru:8090/cluster>. Ваше приложение должно присутствовать в списке в статусе `RUNNING`. Spark может быть запущен в нескольких режимах: локальном, standalone (собственный менеджер ресурсов), yarn и mesos (внешние менеджеры ресурсов).

Считывание данных

Создайте RDD для текстового файла `warandpeace.txt`. Для подробного списка операций считывания файлов обращайтесь к документации класса

SparkContext

<https://spark.apache.org/docs/2.4.0/api/scala/index.html#org.apache.spark.SparkContext>.

Примечание. При наборе команд используйте TAB --- функцию автодополнения.

```
val warandpeace = sc.textFile("warandpeace.txt")
```

В данной команде указывается относительный путь, который начинается с вашей папки в HDFS.

Выведите количество строк файла.

```
warandpeace.count
```

Примечание. При отсутствии у функции аргументов, в scala скобки можно опускать.

Попробуйте считать несуществующий файл, например *nil*, а затем вывести количество его строк на экран

```
val nilFile = sc.textFile("nil")
nilFile.count
```

```
scala> val nilFile = sc.textFile("nil")
16/03/01 08:57:40 INFO storage.MemoryStore: ensureFreeSpace(293456) called with curMem=323658, maxMem=278019440
16/03/01 08:57:40 INFO storage.MemoryStore: Block broadcast_4 stored as values in memory (estimated size 286.6 KB, free 264.6 MB)
16/03/01 08:57:40 INFO storage.MemoryStore: ensureFreeSpace(25484) called with curMem=617114, maxMem=278019440
16/03/01 08:57:40 INFO storage.MemoryStore: Block broadcast_4_piece0 stored as bytes in memory (estimated size 24.9 KB, free 264.5 MB)
16/03/01 08:57:40 INFO storage.BlockManagerInfo: Added broadcast_4_piece0 in memory on localhost:6370 (size: 24.9 KB, free: 265.1 MB)
16/03/01 08:57:40 INFO spark.SparkContext: Created broadcast 4 from textFile at <console>:21
nilFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5] at textFile at <console>:21

scala> nilFile.count
org.apache.hadoop.mapred.InvalidInputException: Input path does not exist: hdfs://master.ssau.ru:8020/user/vlpr/nil
    at org.apache.hadoop.mapred.FileInputFormat.singleThreadedListStatus(FileInputFormat.java:287)
```

Заметьте, что первая команда выполняется успешно, а вторая выводит сообщение, что такого файла нет. Это происходит потому, что выполнение обработки в Spark является ленивым и не запускается, до встречи команды действия(action). count - первая команда действия, с которой вы познакомились.

Считайте первые 10 строк файла warandpeace.txt.

```
warandpeace.take(10)
```

Эта команда не требует считывания и передачи на главный узел всех данных RDD.

Узнайте на сколько частей разделились данные в кластере.

```
warandpeace.partitions
```

Если используется определённый метод распределения вы можете получить данные о нём командой *partitioner*. Начиная с версии 1.6.0 доступна команда *warAndPeaceFile.getNumPartitions* для получения информации о количестве разделов.

Создайте распределённую коллекцию из нескольких элементов и для каждого элемента верните ip адрес, на котором он расположен:

```
Sc.parallelize(Array(1,2,3)).map(x =>  
java.net.InetAddress.getLocalHost).collect
```

Обработка текста

Найдите строки, в которых содержится слово "война".

```
val linesWithWar = warAndPeaceFile.filter(x =>  
x.contains("война"))
```

Примечание. Аргументом *filter* является лямбда функция - функция без имени. До обозначения *=>* в скобках через запятую следуют переменные аргументов функции, затем следует команда языка Scala. При использовании фигурных скобок язык позволяет описывать лямбда функции с цепочкой команд в теле, аналогично именованным функциям.

Запросите первую строку. Строкой в данном файле является целый абзац, так как только по завершению абзаца содержится символ переноса строки.

```
linesWithWar.first
```

Данные могут быть перемещены в кэш. Этот приём очень полезен при повторном обращении к данным, для запросов "горячих" данных или запуска итеративных алгоритмов.

Перед подсчётом количества элементов вызовите команду кэширования `cache()`. Трансформации не будут обработаны, пока не будет запущена одна из команд - действий.

```
linesWithWar.cache()  
linesWithWar.count()  
linesWithWar.count()
```

Можете воспользоваться следующим блоком кода для замера времени выполнения команды.

```
def time[R](block: => R): R = {  
  val t0 = System.nanoTime()  
  val result = block      // call-by-name  
  val t1 = System.nanoTime()  
  println("Elapsed time: " + (t1 - t0) + "ns")  
  result  
}
```

```
scala> val linesWithWar = warandpeace.filter(x => x contains "война")  
linesWithWar: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at filter at <console>:25  
  
scala> time { linesWithWar.count }  
Elapsed time: 214709872ns  
res22: Long = 54  
  
scala> time { linesWithWar.count }  
Elapsed time: 136522537ns  
res23: Long = 54  
  
scala> linesWithWar.cache()  
res24: linesWithWar.type = MapPartitionsRDD[3] at filter at <console>:25  
  
scala> time { linesWithWar.count }  
Elapsed time: 152316423ns  
res25: Long = 54  
  
scala> time { linesWithWar.count }  
Elapsed time: 55745366ns  
res26: Long = 54  
  
scala> time { linesWithWar.count }  
Elapsed time: 57074725ns  
res27: Long = 54
```

При выполнении команды `count` второй раз вы должны заметить небольшое ускорение. Кэширование небольших файлов не даёт большого преимущества, однако для огромных файлов, распределённых по сотням или тысячам узлов, разница во времени выполнения может быть существенной. Вторая команда `linesWithWar.count()` выполняется над результатом от предшествующих команде `cache` трансформаций и на больших объёмах данных будет ускорять выполнение последующих команд.

Найдите гистограмму слов:

```
val wordCounts = linesWithWar.flatMap(line => line.split("
")).map(word => (word, 1)).reduceByKey((a, b)
=> a + b)
```

Spark существенно упростил реализацию многих задач, ранее решаемых с использованием MapReduce. Эта однострочная программа --- WordCount --- является наиболее популярным примером задачи, эффективно распараллеливаемой в Hadoop кластере. Её реализация в MapReduce занимает около 130 строк кода.

Сохраните результаты в файл, а затем, найдите данные в HDFS и выведите данные в linux консоли с помощью команды `hadoop fs -cat warandpeace_histogram.txt /*` (здесь используется относительный путь).

```
wordCounts.saveAsTextFile("warandpeace_histogram.txt")
```

```
$ hadoop fs -cat warandpeace_histogram.txt /*
```

Упражнение. Улучшите процедуру, убирая из слов лишние символы и трансформируя все слова в нижний регистр. Используйте регулярные выражения. Например, по регулярному выражению `"\\w*".r` следующий код

```
"\\w*".r.findAllIn("a b c").toArray.foreach(println)
```

выведет на каждой строке по букве. Кроме Scala консоли для тестирования регулярных выражений вы можете использовать сайты:

- <https://regex101.com/>,
- <https://www.debuggex.com/>.

Операции с множествами

Инициализируйте два множества

```
val a = sc.parallelize(Array(1,2,3,4))
val b = sc.parallelize(Array(3,4,6,7))
```

Найдите объединение `a` и `b` и соберите данные на главный узел с помощью функции `collect`.

```
a.union(b).collect
```

Обратите внимание, что общие элементы дублируются, поэтому результат не является классическим множеством на самом деле. Такое поведение делает эту операцию очень дешёвой, так как обновляется только информация о местонахождении данных для данного RDD. Уберите дубликаты с помощью `distinct`.

```
a.union(b).distinct().collect
```

Найдите пересечение множеств.

```
a.intersection(b).collect
```

Найдите разность множеств.

```
a.subtract(b).collect
```

Примечание. При запуске `collect` на центральный узел - *driver* передаются все данные из распределённого RDD. При работе с большим объемом данных выполнение данной команды может заполнить всю оперативную память *driver* узла.

Упражнение. Найдите в исходном коде Spark определение функции `distinct`. Объясните почему реализация этой операции действительно убирает дубликаты.

Общие переменные

В Apache Spark общими переменными являются широковещательные (broadcast) переменные и аккумулярующие (accumulator) переменные.

Широковещательные переменные

Общие переменные удобны если вы обращаетесь к небольшому объёму данных на всех узлах. Например, это могут быть параметры алгоритмов обработки, небольшие матрицы.

В консоли, с которой вы работали в предыдущем разделе, создайте широковещательную переменную. Наберите:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

Для получения значения обратитесь к полю value:

```
broadcastVar.value
```

Аккумулирующие переменные

Аккумулирующие переменные являются объектами, которые могут быть изменены только ассоциативной операцией добавления. Они используются для эффективной реализации счётчиков и суммарных значений. Вы можете также использовать свой тип, над которым определена ассоциативная операция при необходимости.

Особенностью использования переменной является возможность доступа к значению только на узле в driver процессе.

Потренируйтесь в создании аккумулирующих переменных:

```
val accum = sc.longAccumulator
```

Следующим шагом запустите параллельную обработку массива и в каждом параллельном задании добавьте к аккумулирующей переменной значение элемента массива:

```
sc.parallelize(Array(1,2,3,4)).foreach(x =>
  accum.add(x) )
```

Для получения текущего значения вызовите команду:

```
accum.value
```

Результатом должно быть число 10.

Пары ключ-значение

Создайте пару ключ-значение из двух букв:

```
val pair = ('a', 'b')
```

Для доступа к первому значению обратитесь к полю _1:

```
pair._1
```

Для доступа к второму значению к полю _2:

```
pair._2
```

Если распределённая коллекция состоит из пар, то они трактуются как для ключ-значение и для таких коллекций доступны дополнительные операции. Наиболее распространённые, это: группировка по ключу, агрегирование значений с одинаковыми ключами, объединение двух коллекций по ключу.

Вы можете выйти из консоли нажатием сочетания клавиш CTRL+D.

К текущему моменту вы познакомились со следующими командами действий: count, first, take, saveAsTextFile, collect, foreach. Полный список команд действий вы можете найти в документации соответствующей версии Spark <http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions>.

Анализ данных: десять наиболее популярных номеров такси, на основе количества поездок.

В данной части с помощью Spark вы проанализируете данные о поездках такси в Нью-Йорке и определите 10 наиболее популярных номеров такси, на основе количества поездок.

В первую очередь будет необходимо загрузить данные в MapRFS. Создайте новую папку в MapRFS:

Создайте RDD на основе загруженных данных nyctaxi.csv:

```
val taxi = sc.textFile("nyctaxi.csv")
```

Выведите первые 5 строк из данной таблицы:

```
taxi.take(5).foreach(println)
```

Обратите внимание, что первая строка является заголовком. Её как правило нужно будет отфильтровать. Одним из эффективных способов является следующий:

```
taxi.mapPartitionsWithIndex{(idx,iter)=> if (idx==0)  
iter.drop(1) else iter }
```

Примечание. Для анализа структурированных табличных данных рассматривайте в качестве альтернативы использование DataFrame и DataSet API.

Для разбора значений потребуется создать RDD, где каждая строка разбита на массив подстрок. Используйте запятую в качестве разделителя. Наберите:

```
val taxiParse = taxi.map(line=>line.split(", "))
```

Теперь преобразуем массив строк в массив пар ключ-значение, где ключом будет служить номер такси (6 колонка), а значением единица.

```
val tmk = taxiParse.map(row => (row(6), 1))
```

Следом мы можем найти количество поездок каждого номера такси:

```
val tmc = tmk.reduceByKey((v1, v2) => v1+v2)
```

Выведем полученные результаты в отсортированном виде:

```
tmc.map(_._swap).top(10).map(_._swap).foreach(println)
```

Примечание. Нотация `_._swap` является объявлением анонимной функции от одного аргумента, аналог записи `x => x.swap`.

Являются ли обе `map` операции распределёнными? Найдите в документации Spark в классах RDD или PairRDDFunctions метод `top`.

Вы также можете сгруппировать все описанные выше трансформации, преобразующие исходные данные в одну цепочку:

```
val taxiCounts =  
taxi.map(line=>line.split(", ")).map(row=>(row(6), 1)).re  
duceByKey(_ + _)
```

Примечание. Нотация `_ + _` является объявлением анонимной функции от двух аргументов, аналог более многословной записи `(a,b) => a + b`.

Попробуйте найти общее количество номеров такси несколько раз, предварительно объявив RDD `taxiCounts` как сохраняемую в кэше:

```
taxiCounts.cache()
```

Сравните время, которое трансформации выполняются первый раз и второй. Чем больше данные, тем существеннее разница.

```
taxiCounts.count()  
taxiCounts.count()
```

Создание проекта на локальном компьютере

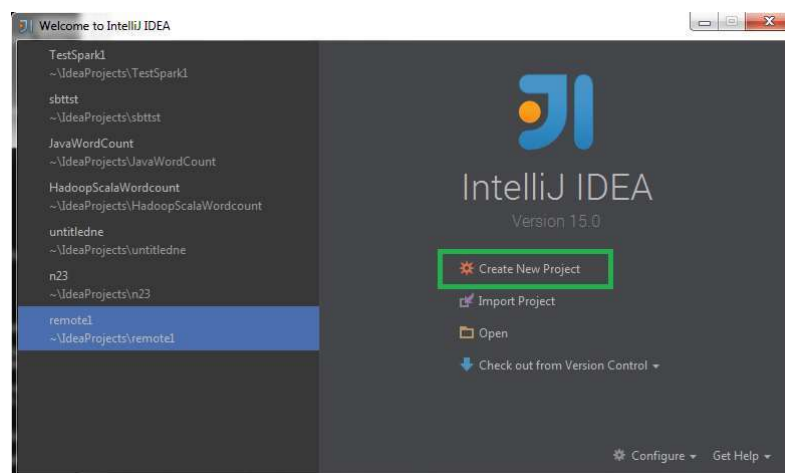
Разработка программы на Spark может быть выполнена на нескольких языках: Python, R, Scala, Java. В данном руководстве рассмотрим разработку на последних двух, так как они имеют самую полную поддержку API обработки данных.

Разработка приложения может производиться в любом текстовом редакторе, затем быть собрана системой сборки в отдельное приложение и запущена на Spark кластере с помощью консольной команды `spark-submit`.

В данной лабораторной работе мы будем использовать IntelliJ IDEA. IDE предоставляет набор возможностей для упрощения разработки: автодополнения, индексация проекта и используемых библиотек, проверка кода, подсветка синтаксиса, интеграция с системами контроля версий и системами сборки.

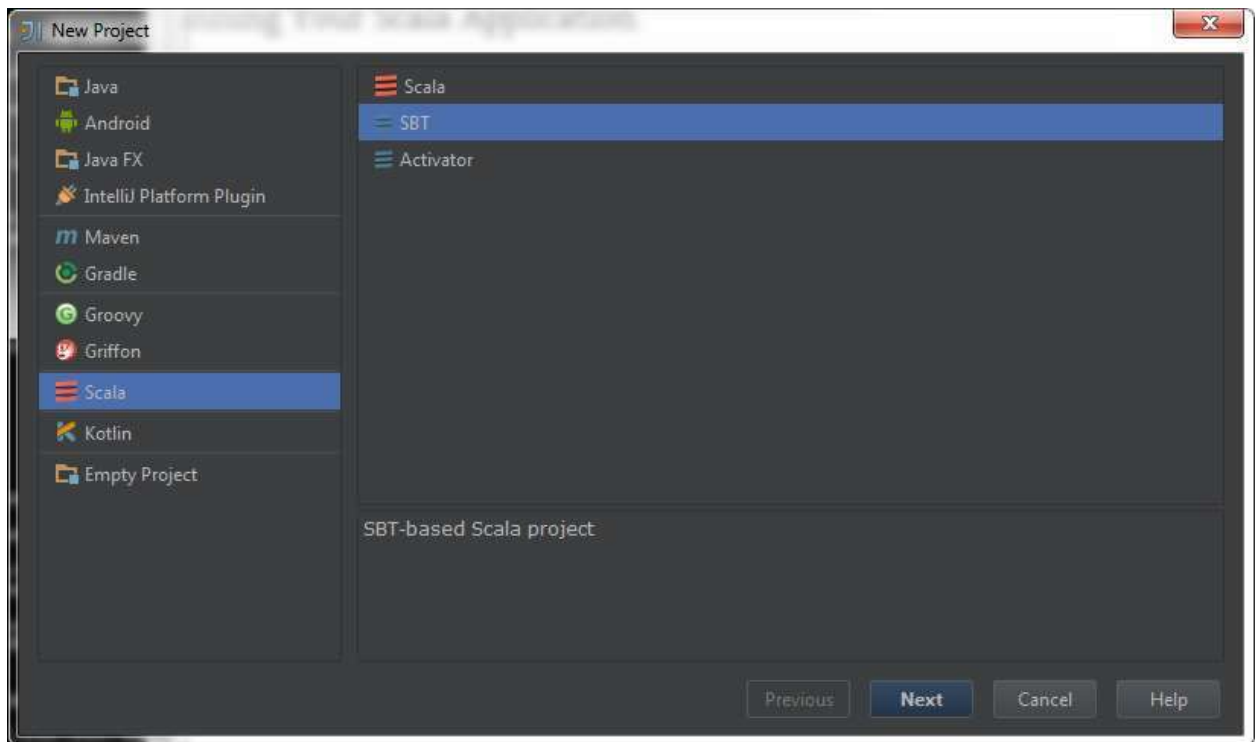
Для работы необходима установленная последняя версия IDE IntelliJ IDEA Community Edition (на момент написания версия 15.0). Данная среда разработки доступна для скачивания по адресу <https://www.jetbrains.com/idea/>.

Для создания проекта в IntelliJ IDEA запустите среду разработки, выберите Create New Project.



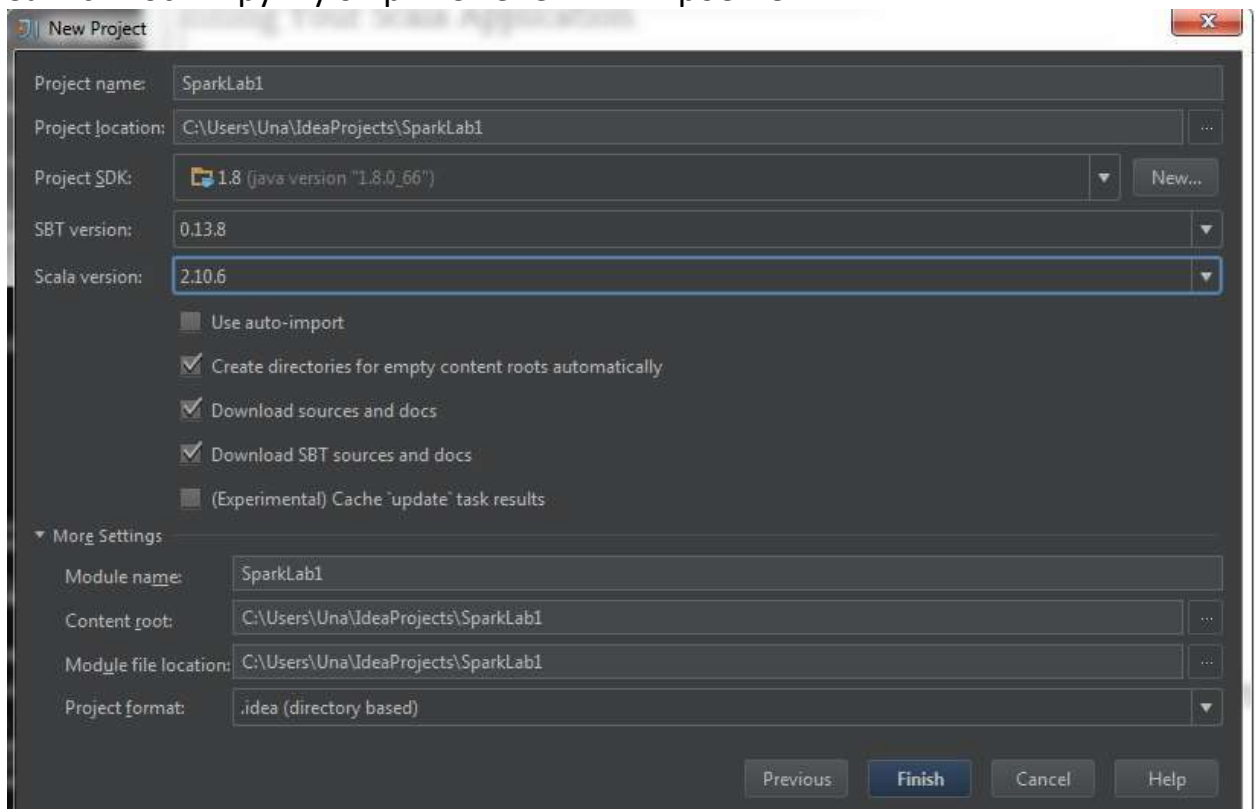
Разработка с использованием системы сборки SBT и языка Scala

Для создания Scala + SBT проекта выберите слева в меню Scala и затем SBT.

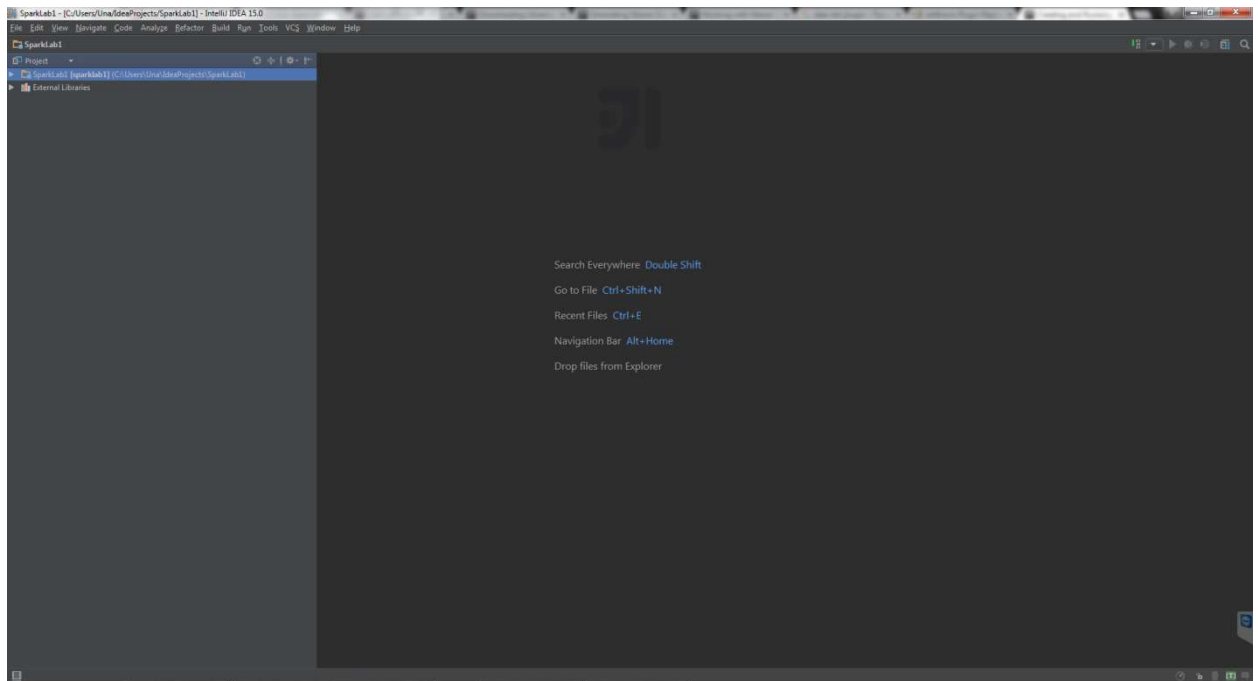


Далее укажите имя проекта, версию Java, версию SBT и версию Scala компилятора. Для разработки на Spark **нужно** выбрать версию Scala 2.10.

Примечание. Установите флаг Use auto-import для того, чтобы не обновлять зависимости вручную при изменениях в проекте.



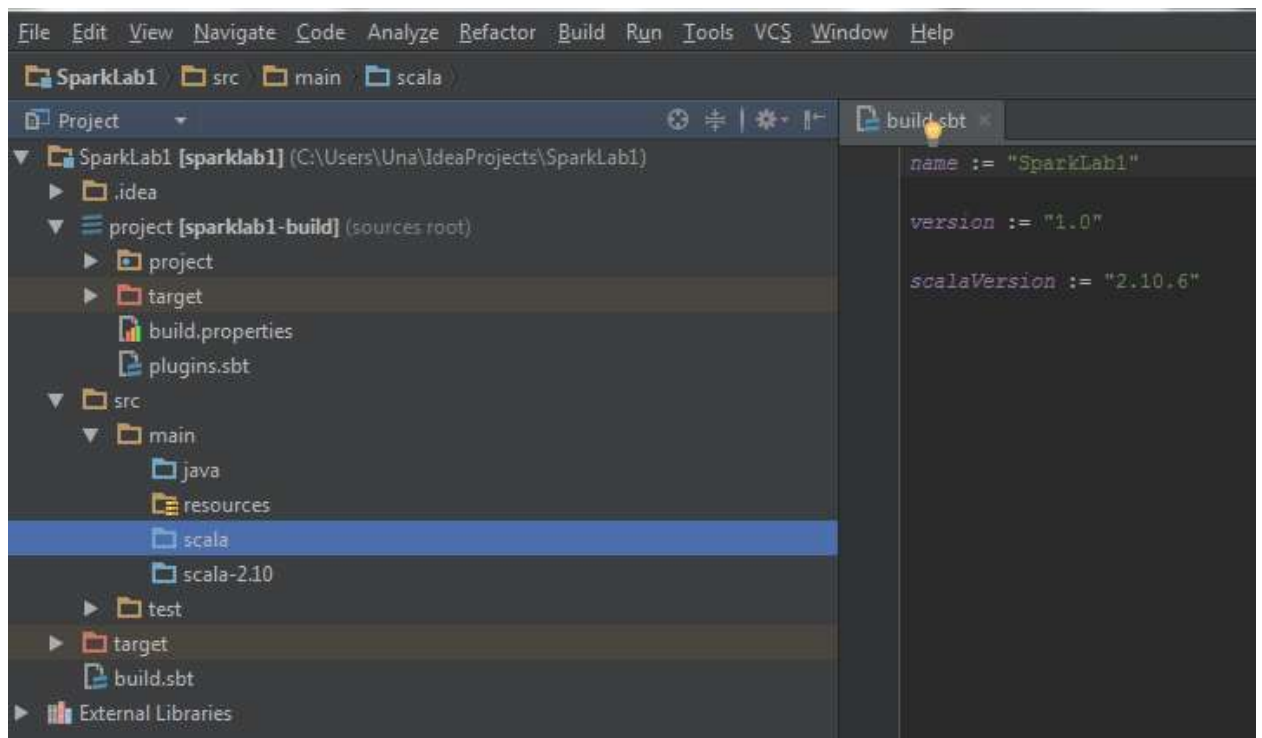
После нажатия Finish откроется главное окно среды разработки.



Подождите, когда SBT скачает зависимости.

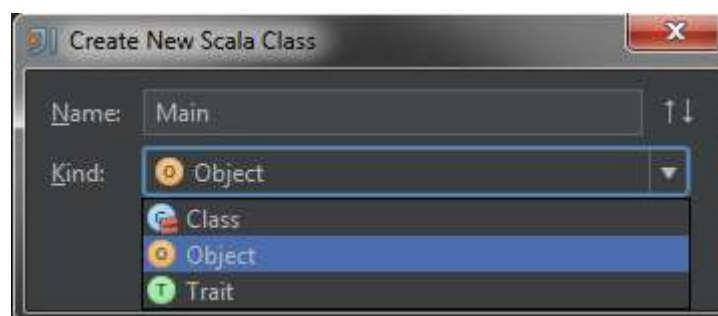
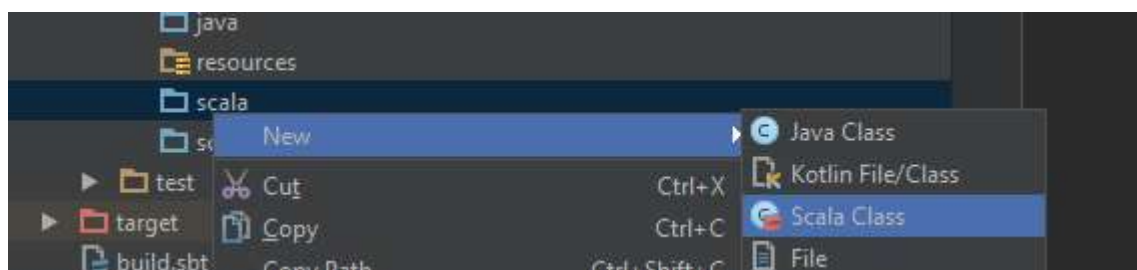
В дереве проекта должен появиться файл `build.sbt`, являющийся основным файлом для настройки сборки и указания зависимостей проекта SBT. В файле на момент создания указаны: имя проекта, версия проекта, версия языка Scala.

Примечание. Появление предупреждений о конфликте имён в SBT 0.13.8 является известной ошибкой <https://github.com/sbt/sbt/issues/1933>. Одно из решений — использование более ранней версии или скрытие всех предупреждений установкой степени логирования `logLevel := Level.Error`.



Код Scala помещается в папку `src/main/scala` или `src/main/scala-2.10`.

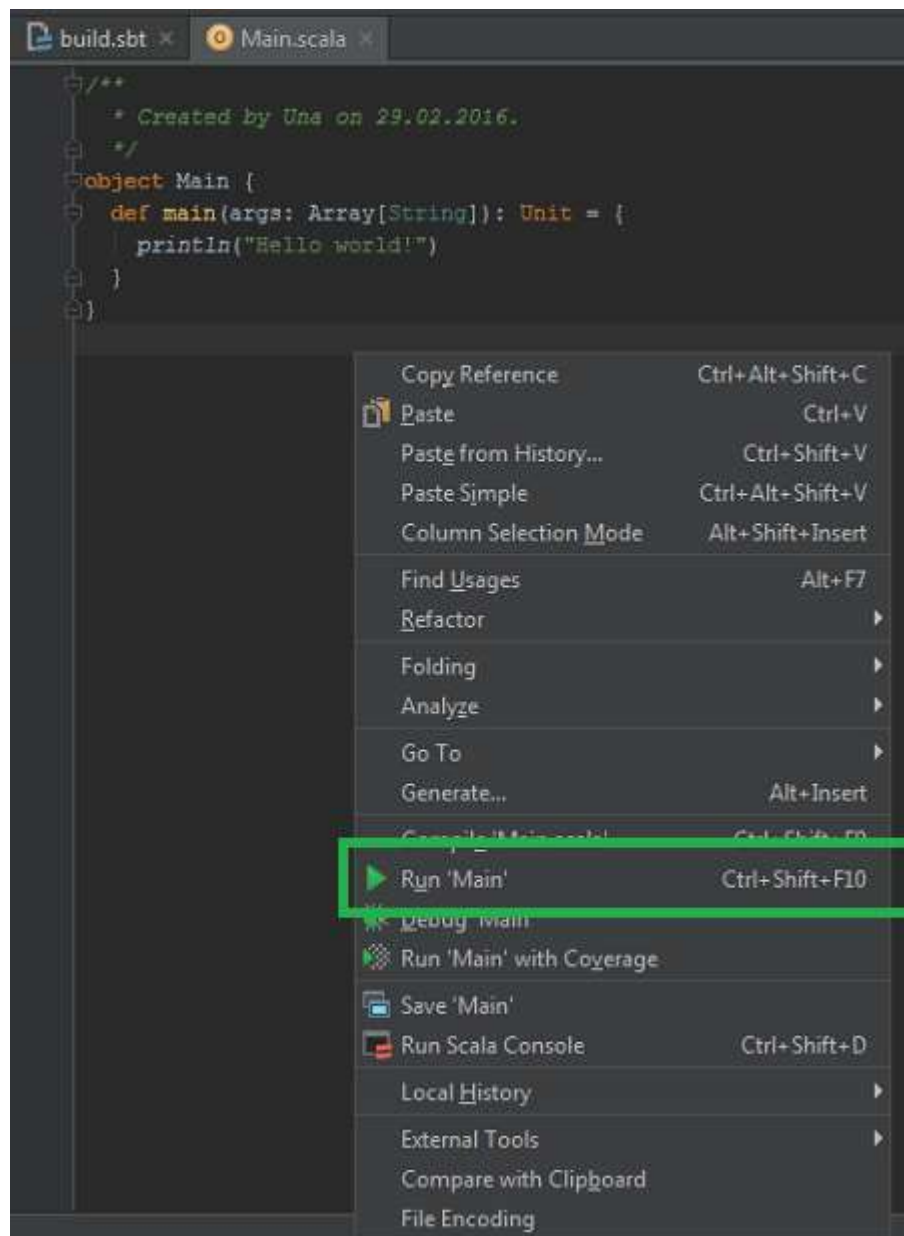
Создайте в папке `scala` объект `Main` с методом `main`. Данный метод будет точкой входа в программу.



Примечание. Аналогом объекта `object` в Java является паттерн `Singleton`. Выполнения тела объекта происходит при его загрузке в память, аналогично инициализации в конструкторе, методы `object` доступны без создания объекта оператором `new`, аналогично публичным статическим методам.

```
object Main {  
    def main(args: Array[String]) {  
        println("Hello world")  
    }  
}
```

В контекстном меню выберите Run 'Main', либо нажмите сочетание клавиш Ctrl+Shift+F10.



После выполнения в консоли должно появиться приветствие.

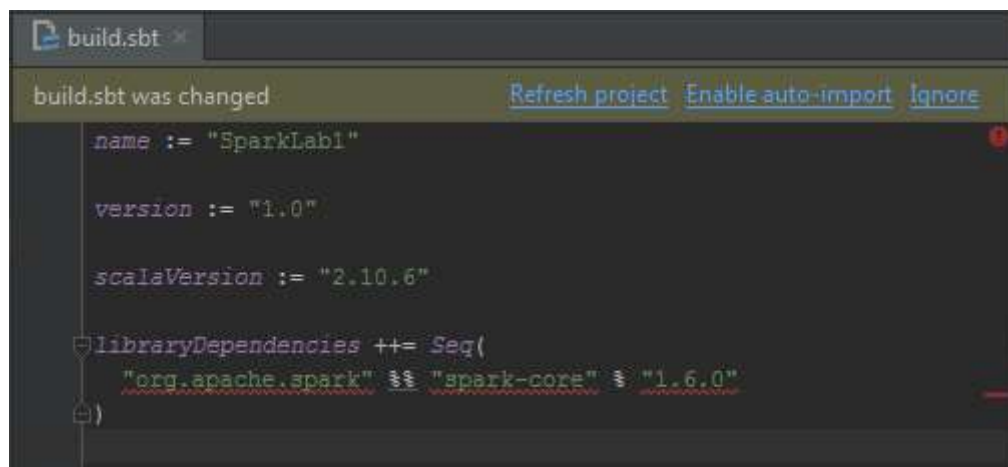
```
"C:\Program Files\Java\jdk1.8.0_66\bin\java" ...
Hello world!

Process finished with exit code 0
```

Добавьте к проекту зависимость Spark, записав следующие строки в конце файла build.sbt:

```
libraryDependencies += Seq(
  "org.apache.spark" %% "spark-core" % "1.6.0"
)
```

Сохраните изменения и обновите проект.



Подождите, когда SBT скачает все зависимые библиотеки.

Измените код Main.scala и создайте простейшую Spark программу. Импортируйте классы пакета *org.apache.spark*.

```
import org.apache.spark._
```

Создайте конфигурацию Spark с помощью класса SparkConf. Укажите обязательные параметры: имя запускаемой задачи (имя контекста задачи) и режим запуска (список режимов <http://spark.apache.org/docs/latest/submitting-applications.html#master-urls>). В нашем случае в качестве режима будет указан параметр local[2], означающий запуск с двумя потоками на локальной машине. В качестве режима может быть указан адрес главного узла.

```
val cfg = new SparkConf()
```

```
.setAppName("Test").setMaster("local[2]")
```

Примечание. В Scala различаются два вида переменных: `val` и `var`. Переменные `val` являются неизменяемыми и инициализируются один раз, в отличие от `var`, которой можно присваивать новые значения.

Инициализируйте контекст Spark в главном методе.

```
val sc = new SparkContext(cfg)
```

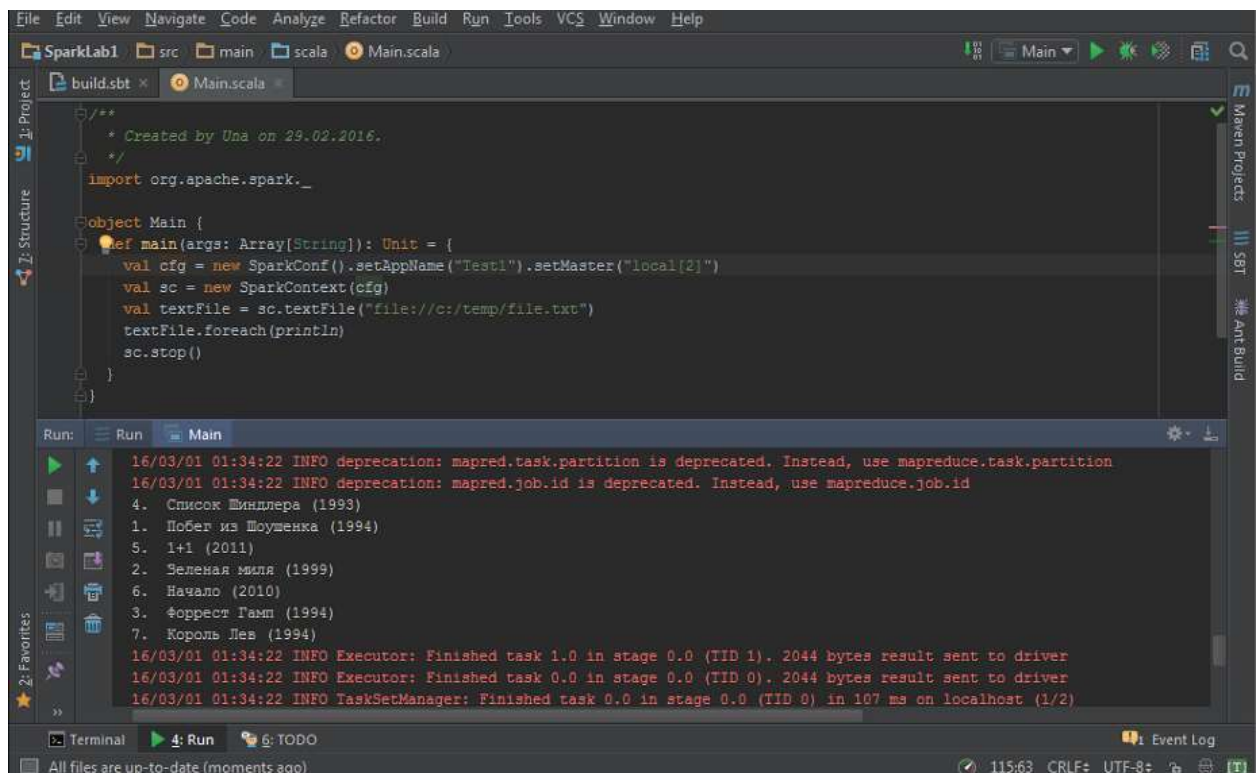
Добавьте в конец файла команду остановки контекста

```
sc.stop()
```

После инициализации контекста вы можете обращаться к командам Spark. Считайте любой текстовый файл из локальной файловой системы и выведите его по строкам в консоль.

Примечание. Путь к файлу в локальной файловой системе имеет определённый формат, имеющий префикс `"file://"`.

```
val textFile = sc.textFile("file:///c:/temp/file.txt")
textFile.foreach(println)
```



Примечание. При работе без winutil.exe запись в файловую систему будет порождать ошибку. Известным решением является скачивание данного файла из проекта Hadoop в файловую систему в папку с названием bin и указанием переменной Spark `hadoop.home.dir`. В переменной `hadoop.home.dir` хранится путь к папке bin. Установить переменную среды JVM вы можете кодом `System.setProperty(key, value)`.

<https://issues.apache.org/jira/browse/SPARK-2356>.

Анализ данных велопарковок

Тестовыми данными являются список поездок на велосипедах `trips.csv` и список велостоянок проката велосипедов `stations.csv`.



Создайте по одному RDD на основе каждого файла `stations.csv`, `trips.csv`. Считайте данные в переменную, затем запомните заголовок. Объявите новую переменную с данными, в которых не будет заголовка, а строка преобразована в массив в соответствии с разделителем - запятой.

Примечание. Существует более эффективный, но громоздкий способ исключить заголовок из данных с использованием метода `mapPartitionWithIndex`. Пример присутствует в первой лабораторной работе в разделе нахождения десяти популярных номеров такси.

```
val tripData = sc.textFile("file:///z:/data/trips.csv")

// запомним заголовок, чтобы затем его исключить
val tripsHeader = tripData.first
val trips = tripData.filter(row=>row!=tripsHeader)
                      .map(row=>row.split(",",-1))

val stationData = sc.textFile("file:///z:/data/stations.csv")
```

```
val stationsHeader = stationData.first
val stations = stationData.filter(row=>row!=stationsHeader)
                           .map(row=>row.split(",",-1))
```

Примечание. Использование в качестве второго параметра -1 в `row.split(",",-1)` позволяет не отбрасывать пустые строки. Например `"1,2".split(",")` вернёт `Array("1","2")`, а `"1,2".split(",",-1)` вернёт `Array("1","2","")`.

Выведите заголовки таблиц и изучите колонки csv файлов.

```
stationsHeader
tripsHeader
```

Выведите несколько элементов данных в `trips` и `stations`.

Объявите `stationsIndexed` так, чтобы результатом был список пар ключ-значение с целочисленным ключом из первой колонки. Таким образом вы создаёте индекс на основе первой колонки - номера велостоянки

```
val stationsIndexed = stations.keyBy(row=>row(0).toInt)
```

Примечание. Обращение к массиву в Scala производится в круглых скобках. Например `Array(1,2,3)(0)` вернёт первый элемент.

Выведите часть данных нового RDD.

Аналогичное действие проделайте для индексирования коллекции `trips` по колонкам `Start Terminal` и `End Terminal` и сохраните результат в переменные, например `tripsByStartTerminals` и `tripsByEndTerminals`.

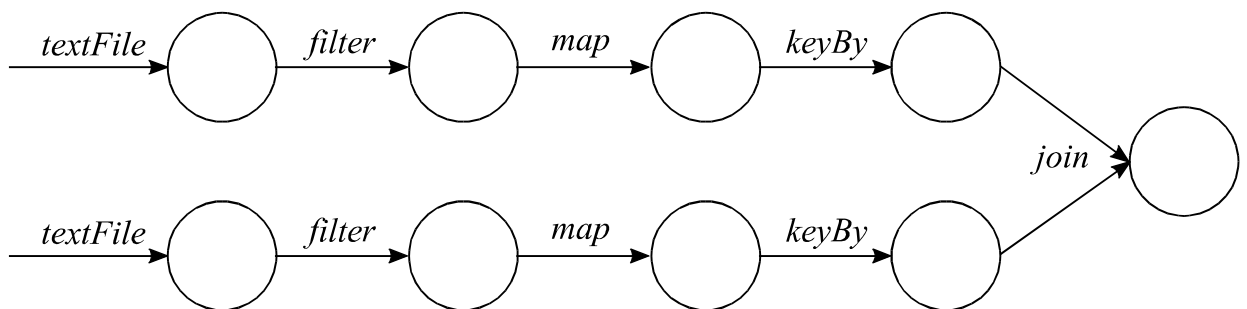
Выполните операцию объединения коллекций по ключу с помощью функции `join`. Объедините `stationsIndexed` и `tripsByStartTerminals`, `stationsIndexed` и `tripsByEndTerminals`.

```
val startTrips =
stationsIndexed.join(tripsByStartTerminals)
val endTrips =
stationsIndexed.join(tripsByEndTerminals)
```

Объявление последовательности трансформаций приводит к созданию ациклического ориентированного графа. Вывести полученный граф можно для любого RDD.


```
startTrips.toDebugString
endTrips.toDebugString
```

```
scala> startTrips.toDebugString
res39: String =
(2) MapPartitionsRDD[18] at join at <console>:20 []
| MapPartitionsRDD[17] at join at <console>:20 []
| CoGroupedRDD[16] at join at <console>:20 []
+- (2) MapPartitionsRDD[10] at keyBy at <console>:15 []
| | MapPartitionsRDD[9] at map at <console>:14 []
| | MapPartitionsRDD[8] at filter at <console>:14 []
| | MapPartitionsRDD[3] at textFile at <console>:12 []
| | file:///d:/station_data.csv HadoopRDD[2] at textFile at <console>:12 []
+- (2) MapPartitionsRDD[14] at keyBy at <console>:15 []
| MapPartitionsRDD[7] at map at <console>:14 []
| MapPartitionsRDD[6] at filter at <console>:14 []
| MapPartitionsRDD[5] at textFile at <console>:12 []
| file:///d:/trip_data.csv HadoopRDD[4] at textFile at <console>:12 []
```



Выполните объявленные графы трансформаций вызовом команды count.

```
startTrips.count()
endTrips.count()
```

Если вы знаете распределение ключей заранее, вы можете выбрать оптимальный способ хеширования ключей по разделам Partition. Например, если один ключ встречается на порядки чаще, чем другие ключи, то использование HashPartitioner будет не лучшим выбором, так как данные связанные с этим ключом будут собираться в одном разделе. Это приведёт к неравномерной нагрузке на вычислительные ресурсы.

Выбрать определённую реализацию класса распределения по разделам можно с помощью функции RDD partitionBy. Например, для RDD stationsIndexed выбирается HashPartitioner с количеством разделов равным количеству разделов trips RDD.

```
stationsIndexed.partitionBy(new  
HashPartitioner(trips.partitions.size))
```

Также можно создавать свои классы для распределения ключей. Узнать какой класс назначен для текущего RDD можно обращением к полю `partitioner`.

```
stationsIndexed.partitioner
```

Трансформации с использованием внутреннего представления

Для более эффективной обработки и получению дополнительных возможностей мы можем объявить классы сущностей предметной области и преобразовать исходные строковые данные в объявленное представление.

В Scala часто для объявления структур данных используется конструкция `case class`. Особенностью такого объявления класса являются: автоматическое создание методов доступа `get` для аргументов конструктора, автоматическое определение методов `hashCode` и `equals`, возможность `case` классов быть разобранными по шаблону (`pattern matching`). Например, для определения

```
case class IntNumber(val value:Integer)
```

выполнение

```
new IntNumber(4).value
```

вернёт значение 4.

Объявите `case` классы для представления строк таблиц в соответствии с именами заголовков.

```
case class Station(  
  stationId:Integer,  
  name:String,  
  lat:Double,  
  long:Double,  
  dockcount:Integer,  
  landmark:String,  
  installation:String,  
  notes:String)  
  
case class Trip(  
  tripId:Integer,  
  duration:Integer,  
  startDate:LocalDateTime,  
  startStation:String,  
  startTerminal:Integer,  
  endDate:LocalDateTime,  
  endStation:String,  
  endTerminal:Integer,
```

```
bikeId: Integer,  
subscriptionType: String,  
zipCode: String)
```

Для конвертации времени будем использовать пакет `java.time`. Краткое введение в работу с пакетом находится в Приложении Б. Объявим формат данных.

```
val timeFormat = DateTimeFormatter.ofPattern("M/d/yyyy  
H:m")
```

Объявим `trips` с учётом преобразования во внутреннее представление.

```
val tripsInternal = trips.map(row=>  
    new Trip(tripId=row(0).toInt,  
             duration=row(1).toInt,  
             startDate= LocalDate.parse(row(2),  
timeFormat),  
             startStation=row(3),  
             startTerminal=row(4).toInt,  
             endDate=LocalDate.parse(row(5), timeFormat),  
             endStation=row(6),  
             endTerminal=row(7).toInt,  
             bikeId=row(8).toInt,  
             subscriptionType=row(9),  
             zipCode=row(10)))
```

Изучите полученные данные. Например, вызовом следующих команд

```
tripsInternal.first  
tripsInternal.first.startDate
```

То же можно проделать и для `station` RDD

```
val stationsInternal = stations.map(row=>  
    new Station(stationId=row(0).toInt,  
                name=row(1),  
                lat=row(2).toDouble,  
                long=row(3).toDouble,  
                dockcount=row(4).toInt,  
                landmark=row(5),  
                installation=row(6)  
                notes=null))
```

Примечание. Восьмая колонка не присутствует в таблице, так как в данных она пустая. Если в будущем она не будет использоваться имеет смысл её убрать из описания case класса.

Примечание. В данных присутствуют различные форматы времени.

Посчитаем среднее время поездки, используя *groupByKey*.

Для этого потребуется преобразовать trips RDD в RDD коллекцию пар ключ-значение аналогично тому, как мы совершали это ранее методом *keyBy*.

```
val tripsByStartStation =  
tripsInternal.keyBy(row=>row.startStation)
```

Рассчитаем среднее время поездки для каждого стартового парковочного места

```
val avgDurationByStartStation = tripsByStartStation  
  .mapValues(x=>x.duration)  
  .groupByKey()  
  .mapValues(col=>col.reduce((a,b)=>a+b)/col.size)
```

Выведем первые 10 результатов

```
avgDurationByStartStation.take(10).foreach(println)
```

Выполнение операции *groupByKey* приводит к интенсивным передачам данных. Если группировка делается для последующей редукции элементов лучше использовать трансформацию *reduceByKey* или *aggregateByKey*. Их выполнение приведёт сначала к локальной редукции над разделом *Partition*, а затем будет произведено окончательное суммирование над полученными частичными суммами.

Примечание. Выполнение *reduceByKey* логически сходно с выполнением *Combine* и *Reduce* фазы MapReduce работы.

Функция *aggregateByKey* является аналогом *reduceByKey* с возможностью указывать начальный элемент.

Рассчитаем среднее значение с помощью *aggregateByKey*. Одновременно будут вычисляться два значения для каждого стартового терминала: сумма времён и количество поездок.

```
val avgDurationByStartStation2 = tripsByStartStation
  .mapValues(x=>x.duration)
  .aggregateByKey((0,0)) (
    (acc, value) => (acc._1 + value, acc._2 + 1),
    (acc1, acc2) => (acc1._1+acc2._1, acc1._2+acc2._2))
  .mapValues(acc=>acc._1/acc._2)
```

В первых скобках передаётся начальное значение. В нашем случае это пара нулей. Первая анонимная функция предназначена для прохода по коллекции раздела. На этом проходе значение элементов помещаются средой в переменную value, а переменная «аккумулятора» асс накапливает значения. Вторая анонимная функция предназначена для этапа редукции частично посчитанных локальных результатов.

Сравните результаты avgDurationByStartStation и avgDurationByStartStation2 и их время выполнения.

Теперь найдём первую поездку для каждой велостоянки. Для решения также потребуется группировка. Ещё одним недостатком groupByKey данных является то, что для группировки данные должны поместиться в оперативной памяти. Это может привести к ошибке OutOfMemoryException для больших объёмов данных.

Сгруппируем поездки по велостоянкам и отсортируем поездки в группах по возрастанию даты.

```
val firstGrouped = tripsByStartStation
  .groupByKey()
  .mapValues(list =>
    list.toList.sortWith((trip1, trip2) =>
      trip1.startDate.compareTo(trip2.startDate)<0))
```

```
(Mountain View City Hall,Trip(4081,218,2013-08-29T09:38,Mountain View City Hall,27,2013-08-29T09:41,Moun
(California Ave Caltrain Station,Trip(4375,880,2013-08-29T12:26,California Ave Caltrain Station,36,2013-
(San Jose Civic Center,Trip(4510,166,2013-08-29T13:31,San Jose Civic Center,3,2013-08-29T13:34,San Salva
(Yerba Buena Center of the Arts (3rd @ Howard),Trip(4355,2044,2013-08-29T12:18,Yerba Buena Center of the
(Commercial at Montgomery,Trip(4086,178,2013-08-29T09:42,Commercial at Montgomery,45,2013-08-29T09:45,Co
scala>
```

Лучшим вариантом с точки зрения эффективности будет использование трансформации reduceByKey

```
val firstGrouped = tripsByStartStation
  .reduceByKey((trip1,trip2) =>
```

```
if (trip1.startDate.compareTo(trip2.startDate)<0)
    trip1 else trip2)
```

В данном случае «передаваться дальше» будет то из значений, которое меньше.

```
(Mountain View City Hall,Trip(4081,218,2013-08-29T09:38,Mountain View City Hall,27,2013-08-29T09:41,Mo
(California Ave Caltrain Station,Trip(4375,880,2013-08-29T12:26,California Ave Caltrain Station,36,201
(San Jose Civic Center,Trip(4510,166,2013-08-29T13:31,San Jose Civic Center,3,2013-08-29T13:34,San Sal
(Yerba Buena Center of the Arts (3rd @ Howard),Trip(4355,2044,2013-08-29T12:18,Yerba Buena Center of t
```

Настройка RDD хранения

В данной части будет рассмотрена настройка способов хранения RDD. Вы сравните различные способы хранения, включая: хранение в сериализованном виде, в исходном, с репликацией.

Рассчитайте среднее время поездки для каждого конечного парковочного места, аналогично расчёту для стартового. Используйте `aggregateByKey`. Запишите результат в переменную `avgDurationByEndStation`.

Выведите на экран полученные данные

```
avgDurationByStartStation.collect
avgDurationByEndStation.collect
```

Теперь вызовите функцию `cache()` или `persist(StorageLevel.MEMORY_ONLY)` для переменной `trips`. Функция `cache()` в действительности вызывает данную функцию установки способа хранения

```
trips.persist(StorageLevel.MEMORY_ONLY)
```

Запустите действия `collect` ещё раз.


Теперь измените способ хранения на хранение в памяти в сериализованном виде

```
trips.unpersist(true)
trips.persist(StorageLevel.MEMORY_ONLY_SER)
```

Запустите действия ещё раз. Что изменилось? Зайдите в Spark UI на адрес узла с которым вы работаете на порт 4040.

Примечание. UI пытается привязаться к порту 4040 при запуске spark-shell и выбирает другой порт (следующий) до тех пор, пока не найдёт свободный.

Во вкладке Storage вы должны заметить, что последний запуск использует меньше памяти. Почему?

1.2.1

Jobs

Stages

Storage

Environment

Executors

Zeppelin application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
204	Memory Serialized 1x Replicated	2	100%	68.7 MB	0.0 B	0.0 B
186	Memory Deserialized 1x Replicated	2	100%	164.3 MB	0.0 B	0.0 B

Другими способами хранения являются:

- MEMORY_AND_DISK,
- MEMORY_AND_DISK_SER,
- DISK_ONLY,
- MEMORY_ONLY_2,
- MEMORY_AND_DISK_2,
- OFF_HEAP.

Поэкспериментируйте с данными вариантами.

Подробнее о способах хранения вы можете узнать по адресу <http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence>

Дополнительные задачи для анализа:

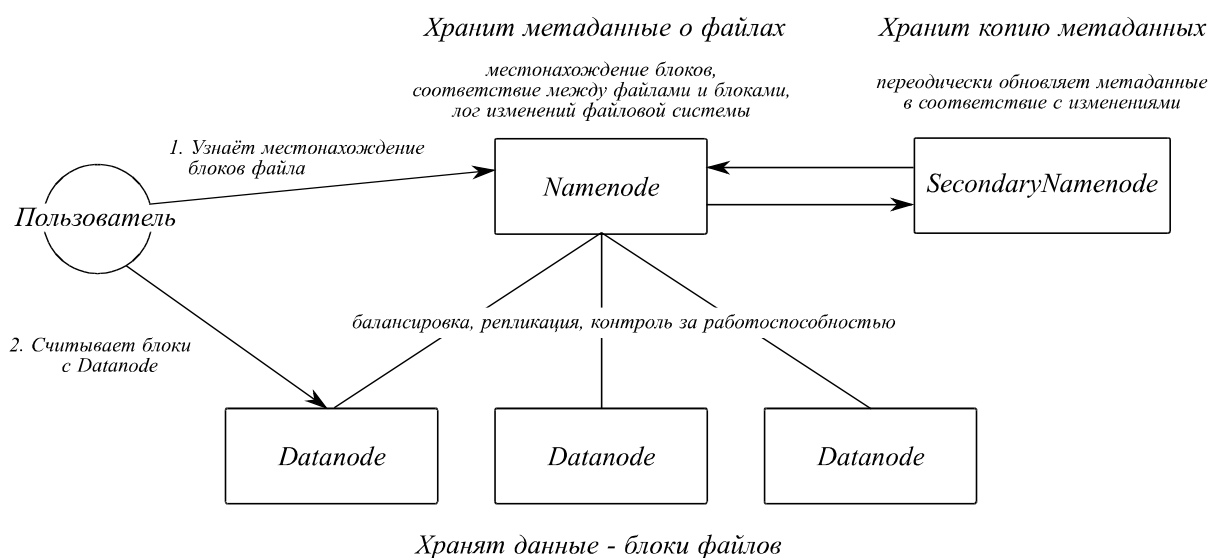
1. Найти велосипед с максимальным временем пробега
2. Найти велосипед с максимальным пробегом
3. Найти наибольшее расстояние между станциями
4. Найти путь велосипеда с максимальным пробегом через станции
5. Найти количество велосипедов в системе
6. Найти подписчиков, которые ездили больше 3 часов

Приложение А

Краткое описание файловой системы HDFS

HDFS — распределенная файловая система, используемая в проекте Hadoop. HDFS-кластер в первую очередь состоит из NameNode-сервера и DataNode-серверов, которые хранят данные. NameNode-сервер управляет пространством имен файловой системы и доступом клиентов к данным. Чтобы разгрузить NameNode-сервер, передача данных осуществляется только между клиентом и DataNode-сервером.

Архитектура HDFS



Развёртывание экземпляра HDFS предусматривает наличие центрального узла имён (англ. name node), хранящего метаданные файловой системы и метаинформацию о распределении блоков, и серии узлов данных (англ. data node), непосредственно хранящих блоки файлов. Узел имён отвечает за обработку операций уровня файлов и каталогов — открытие и закрытие файлов, манипуляция с каталогами, узлы данных непосредственно отрабатывают операции по записи и чтению данных. Узел имён и узлы данных снабжаются веб-серверами, отображающими текущий статус узлов и позволяющими просматривать содержимое файловой системы. Административные функции доступны из интерфейса командной строки.

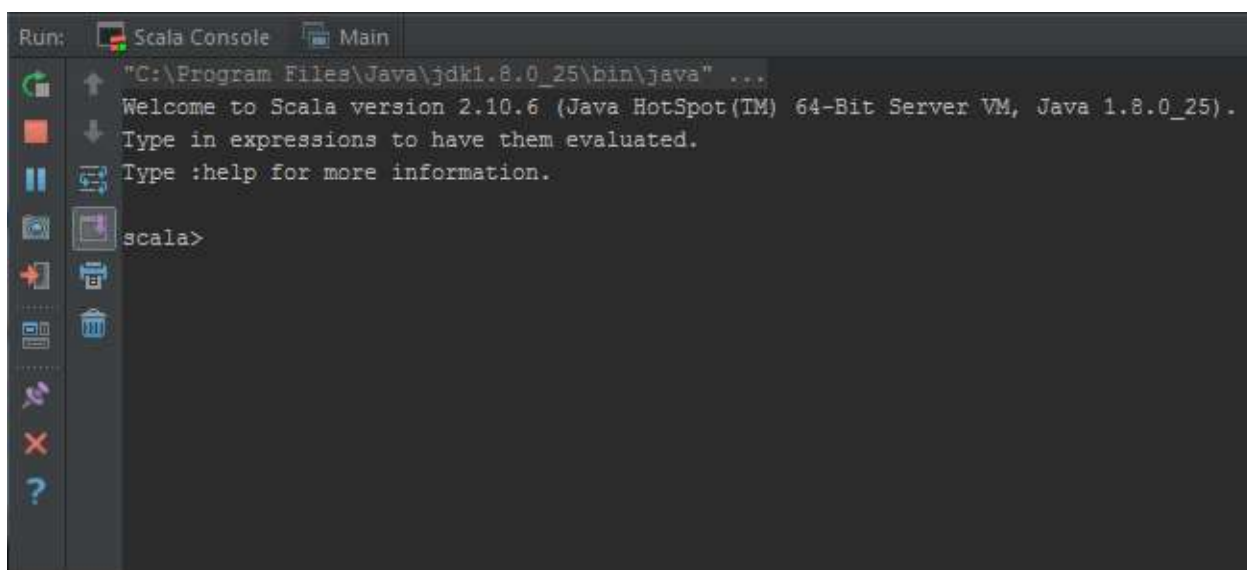
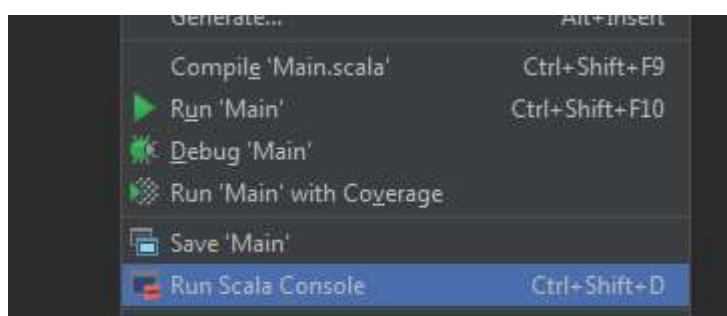
Приложение Б

Основные понятия `java.time`

Для представления времени в Java 8 рекомендуется использовать пакет `java.time`, реализующий стандарт JSR 310. Документация пакета `java.time` доступна по адресу <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>.

Далее приводится работа с основными классами представления времени `java.time`. Для экспериментов удобно использовать REPL консоль. Если вы находитесь в среде разработки IDEA Scala консоль может быть запущена нажатием `Ctrl+Shift+D`, либо через контекстное меню IntelliJ IDEA.

Примечание. REPL (от сокращения `read, eval, print, loop` - считать, выполнить, напечатать, повторять в цикле) – интерактивный цикл взаимодействия программной среды с пользователем.



Примечание. Консоль также можно запустить в командном окне операционной системы с помощью `sbt console`, находясь в папке с проектом.

В обоих вариантах зависимости проекта подключаются автоматически так, что вы можете работать со сторонними библиотеками.

В пакете `java.time` различаются представления времени:

- класс `Instant` — хранит числовую метку;
- класс `LocalDate` — хранит дату без времени;
- класс `LocalTime` — хранит время без даты;
- класс `LocalDateTime` — хранит время и дату;
- класс `ZonedDateTime` — хранит дату, время и часовой пояс.

Узнайте в консоли текущее время, вызвав статический метод `now()` у каждого класса, изучите возвращаемое представление. Например,

```
import java.time._
Instant.now()
```

Перед использованием классов объявите их импорт. Символ «`_`» импортирует все классы данного пакета.

Enter используется для переноса строки. Для выполнения нажмите сочетание клавиш `Ctrl+Enter`.

```
scala> import java.time._
Instant.now()
import java.time._
scala> res0: java.time.Instant = 2016-03-10T07:03:55.612Z
```

Создайте примечательную вам дату с помощью статического конструктора *of* классов `LocalDate`, `LocalDateTime`, `ZonedDateTime`. Воспользуйтесь подсказками среды разработки или документацией для определения количества аргументов метода *of* и их значения.

```
scala> of(year: Int, month: Int, dayOfMonth: Int)    LocalDate
of(year: Int, month: Month, dayOfMonth: Int)        LocalDate
ofEpochDay(epochDay: Long)                         LocalDate
ofYearDay(year: Int, dayOfYear: Int)                LocalDate
Press Ctrl+Period to choose the selected (or first) suggestion and insert a dot afterwards >>>
LocalDate.of|
```

```
scala> LocalDate.of(2015,1,1)
res7: java.time.LocalDate = 2015-01-01
```

Изучите создание времён и дат с помощью метода *parse* данных классов. Используйте форматирование, которое выдавалось системой при возвращении значения в консоль.

```
scala> LocalDate.parse("2015-09-01")
res8: java.time.LocalDate = 2015-09-01

scala> LocalTime.parse("00:00:00")
res9: java.time.LocalTime = 00:00

scala> LocalDateTime.parse("2015-09-01T00:30:00")
res10: java.time.LocalDateTime = 2015-09-01T00:30

scala> ZonedDateTime.parse("2015-09-01T00:00:00+04:00")
res11: java.time.ZonedDateTime = 2015-09-01T00:00+04:00
```

Для задания пользовательского формата считывания вы можете использовать класс `DateTimeFormatter`. Описание класса и символов шаблона располагается по адресу <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>.