Лабораторная работа №1

Исследование способов реализации рекурсивных алгоритмов в функциональном программировании

1. Цель работы

Исследовать различные способы реализации рекурсивных алгоритмов в функциональном программировании. Реализовать и исследовать рекурсивные функции обработки списков на языке Scala.

2. Основные положения

2.1. Установка Java Development Kit

Для выполнения лабораторных работ понадобится установленный Java Development Kit (JDK). Если он еще не установлен на компьютере, то скачать установщик можно на официальном сайте компании Oracle (http://www.oracle.com/technetwork/java/javase/downloads/index.html). Актуальная версия JDK на момент написания данных методических указаний – Java SE Development Kit 8u102 (jdk1.8.0_102).

Внимание: Для выполнения лабораторных работ необходим именно **Java Development Kit (JDK)**, а не Java Runtime Enviroment (JRE)!

2.2. Установка среды разработки IntelliJ IDEA Community Edition

IntelliJ IDEA Community Edition является бесплатной средой разработки для языков семейства Java (и не только). Скачать ее можно бесплатно на официальном сайте компании IntelliJ (https://www.jetbrains.com/idea/#chooseYourEdition).

Рассмотрим подробно процесс установки IntelliJ IDEA. После запуска установочного файла, появится начальное окно установки (рисунок 2.1).

Рисунок 2.1 – Начальное окно установки IntelliJ IDEA

Нажимаем «Далее» для продолжения установки.

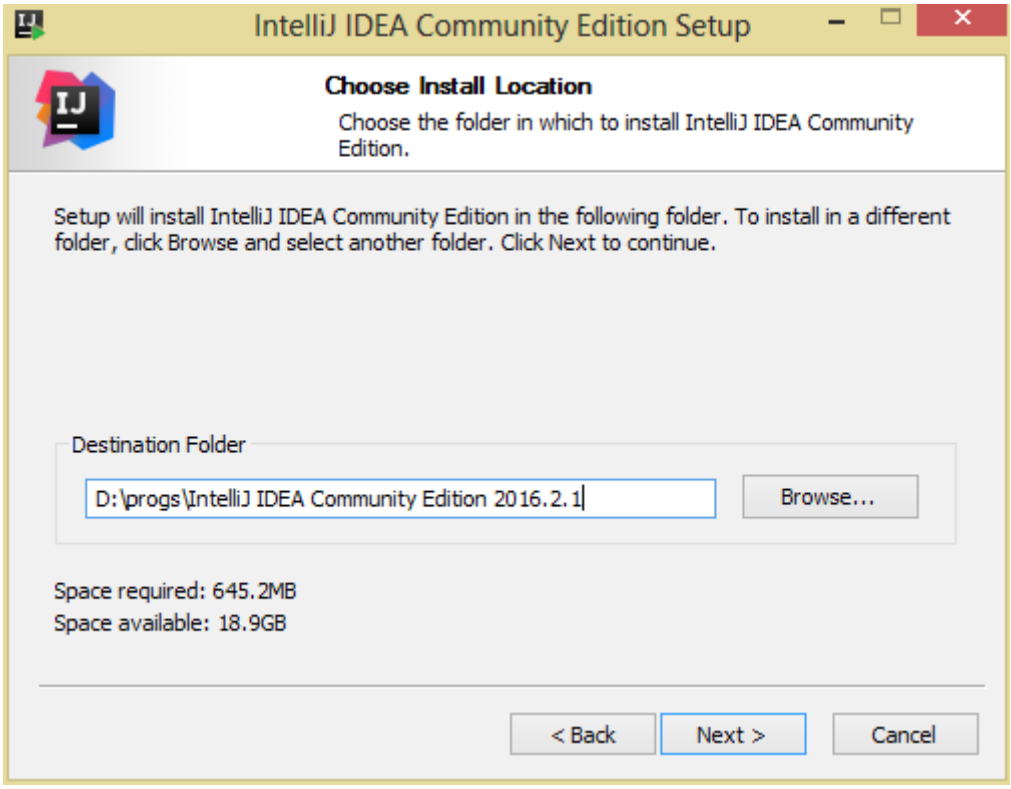

Рисунок 2.2 – Выбор пути установки

Выбираем путь установки (рисунок 2.2) и нажимаем «Далее».

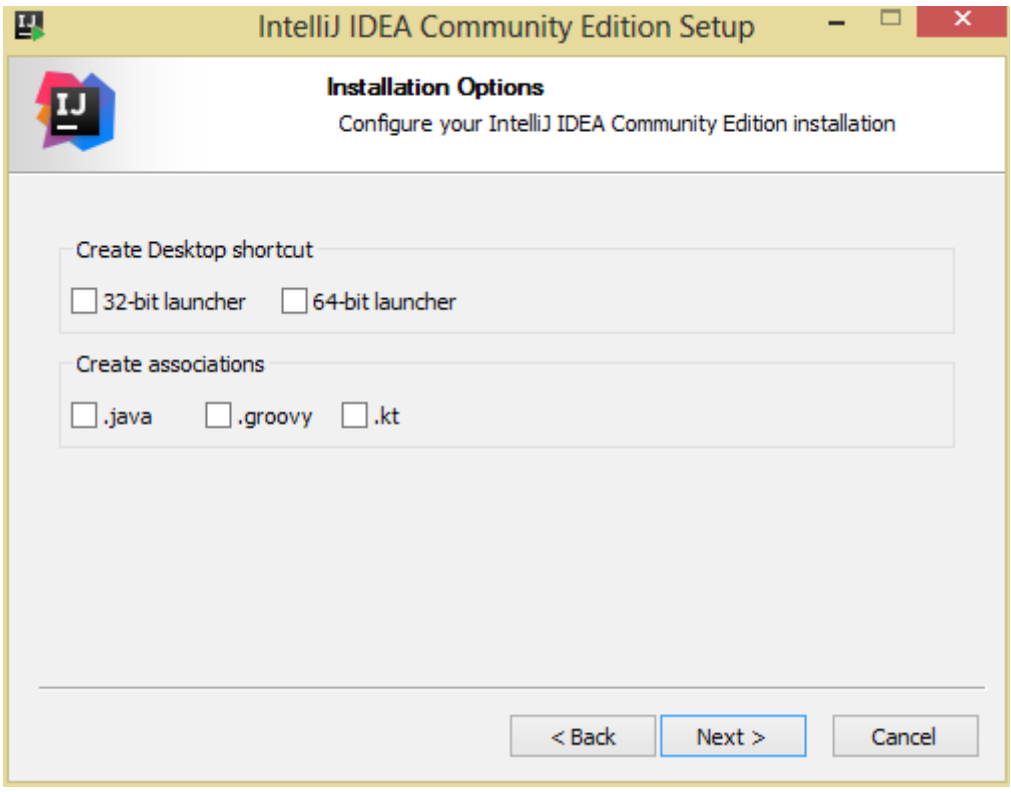

Рисунок 2.3 – Дополнительные настройки

Выбираем дополнительные настройки (рисунок 2.3). Если не знаете или не уверены, что надо выбирать, а что нет – поставьте галочки во всех чекбоксах.

Затем еще несколько раз нажимаем кнопку «Далее», пока не завершится установка.

2.3. Первый запуск и начальные настройки

При запуске IntelliJ IDEA впервые после установки, вы увидите следующее окно (рисунок 2.4).

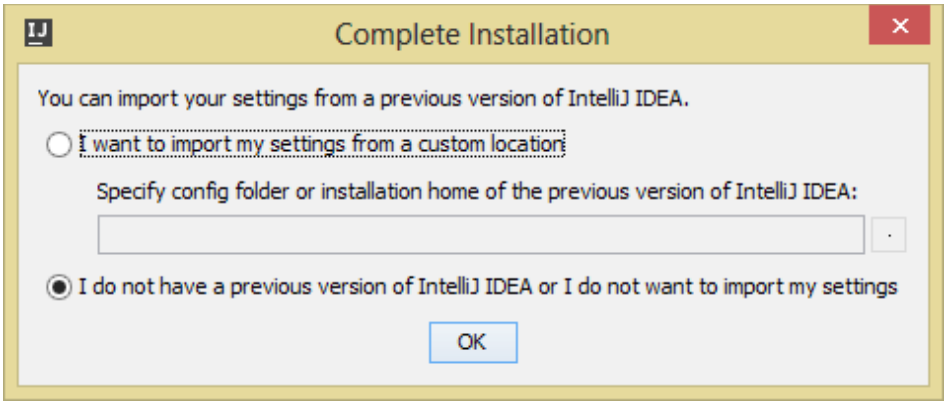

Рисунок 2.4 – Импорт настроек после установки

Выбираем нижний пункт и нажимаем «Ок».

Далее соглашаемся с соглашением политики конфиденциальности (рисунок 2.5).

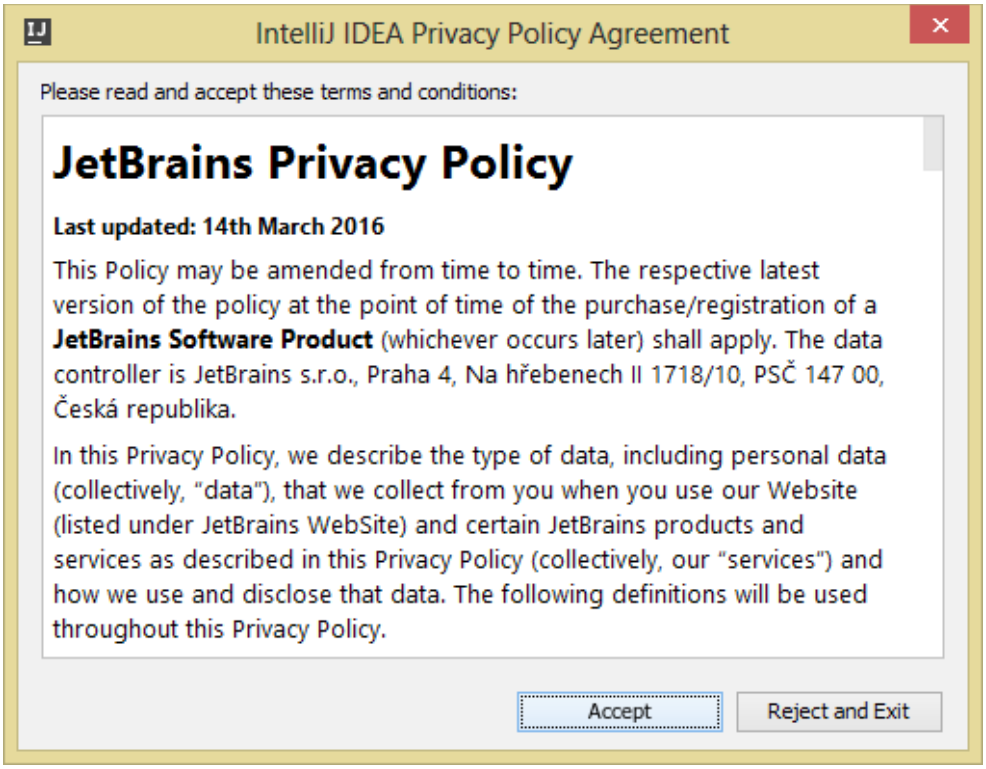

Рисунок 2.5 – Соглашение политики конфиденциальности.

Затем выбираем визуальную тему по вкусу и нажимаем нижнюю правую кнопку (на рисунке 2.6 это кнопка «Next: Default plugins»)

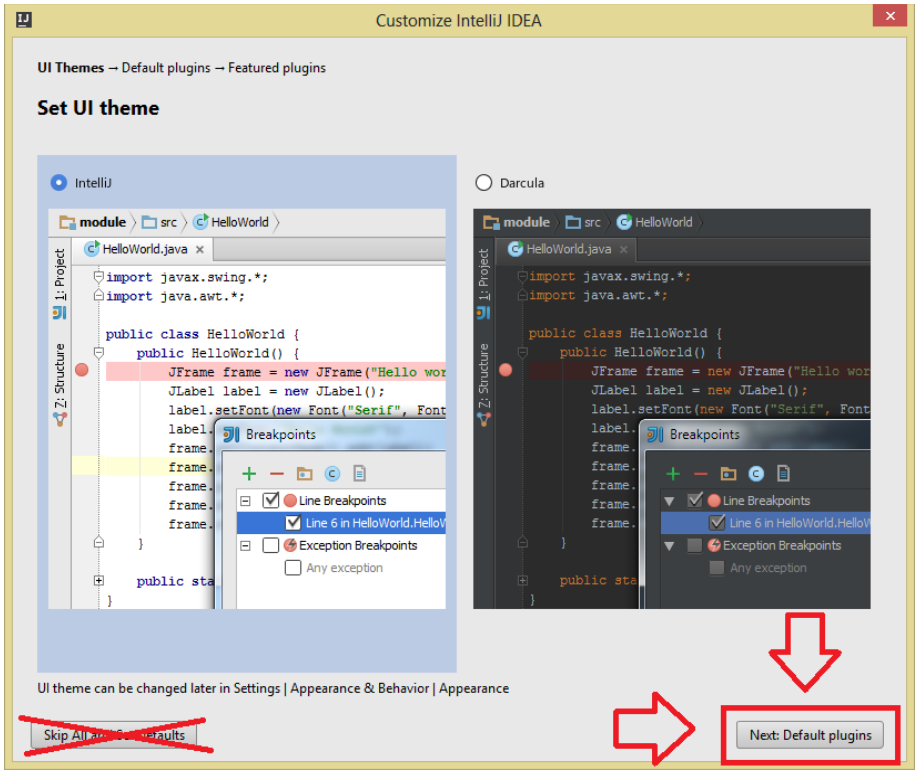

Рисунок 2.6 – Выбор визуальной темы

На следующем экране (рисунок 2.7) оставляем все как есть и опять нажимаем нижнюю правую кнопку.

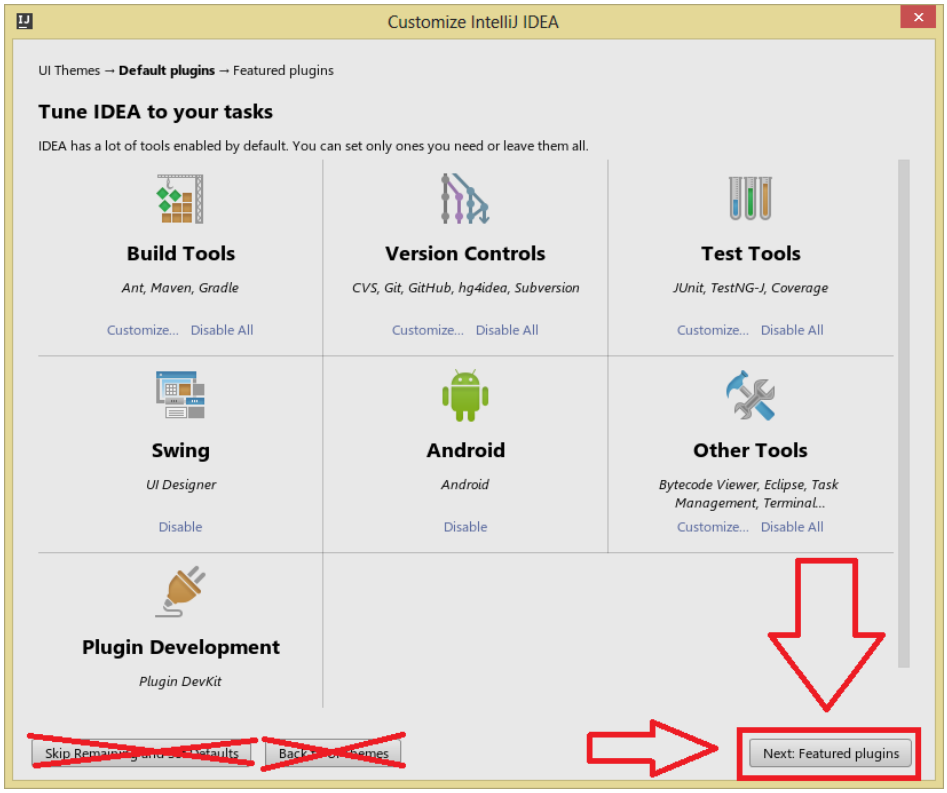

Рисунок 2.7 – Различные настройки

На следующем экране (рисунок 2.8) нажимаем кнопку для установки плагина для поддержки языка Scala.

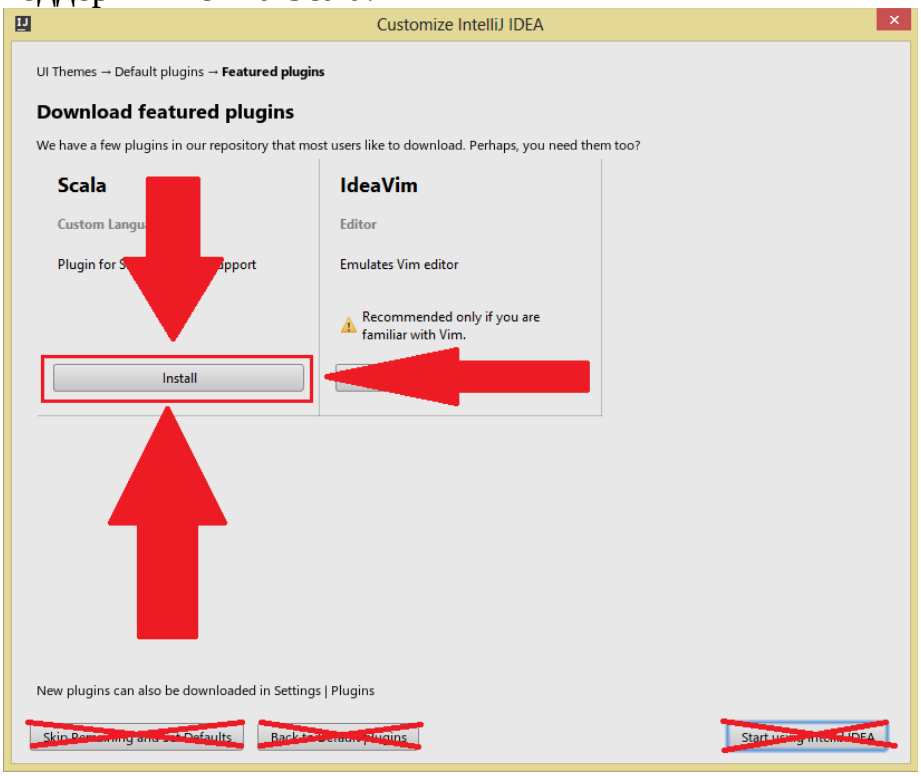

Рисунок 2.8 – Установка плагина для поддержки языка Scala

После завершение установки плагина нажимаем нижнюю правую кнопку (рисунок 2.9)

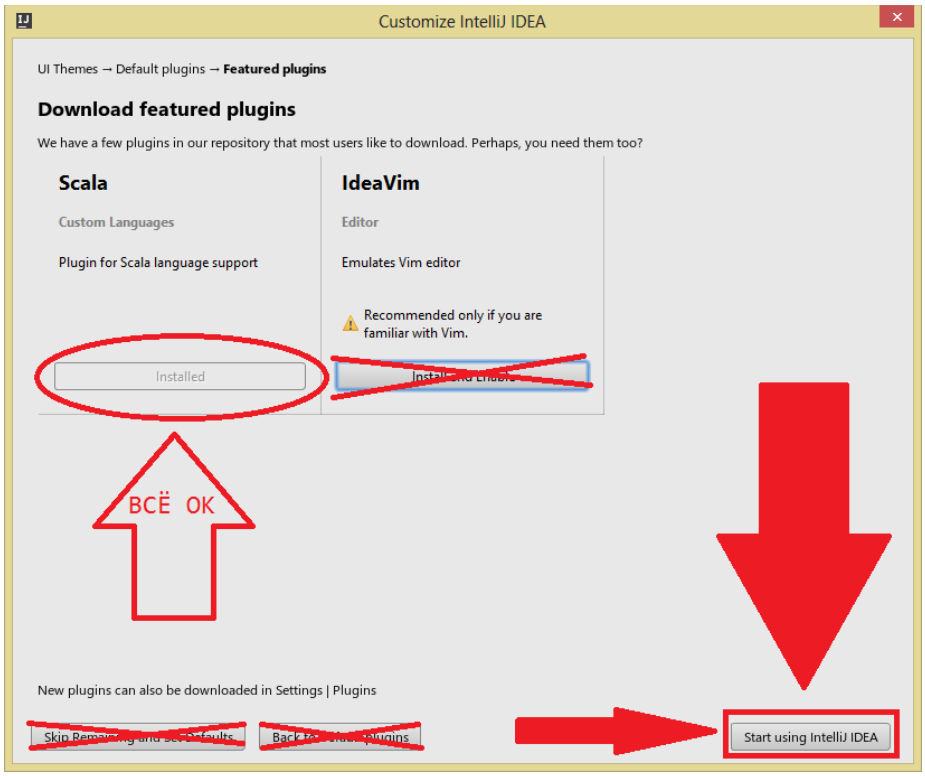

Рисунок 2.9 – Завершение первоначальной настройки

Создание Scala-проекта в IntelliJ IDEA

После запуска IntelliJ IDEA появится следующее окно (рисунок 2.10).

Рисунок 2.10 – Начальное окно IntelliJ IDEA

Выбираем создание нового проекта. В появившемся окне (рисунок 2.11) выбираем Scala.

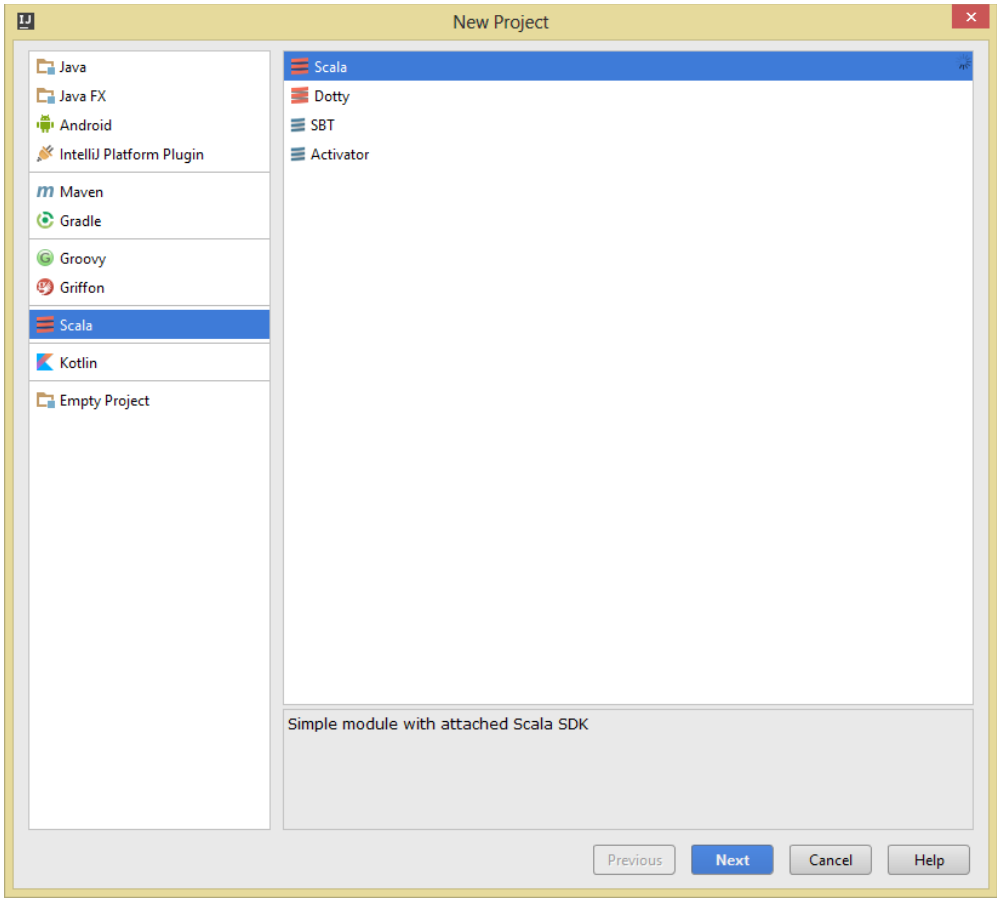

Рисунок 2.11 – Окно выбора типа нового проекта

Далее (рисунок 2.12) можно изменить имя проекта (1) и директорию проекта (2). Затем необходимо указать путь к JDK, для этого нажимаем кнопку «New» (3) и выбираем пункт «JDK» (4).

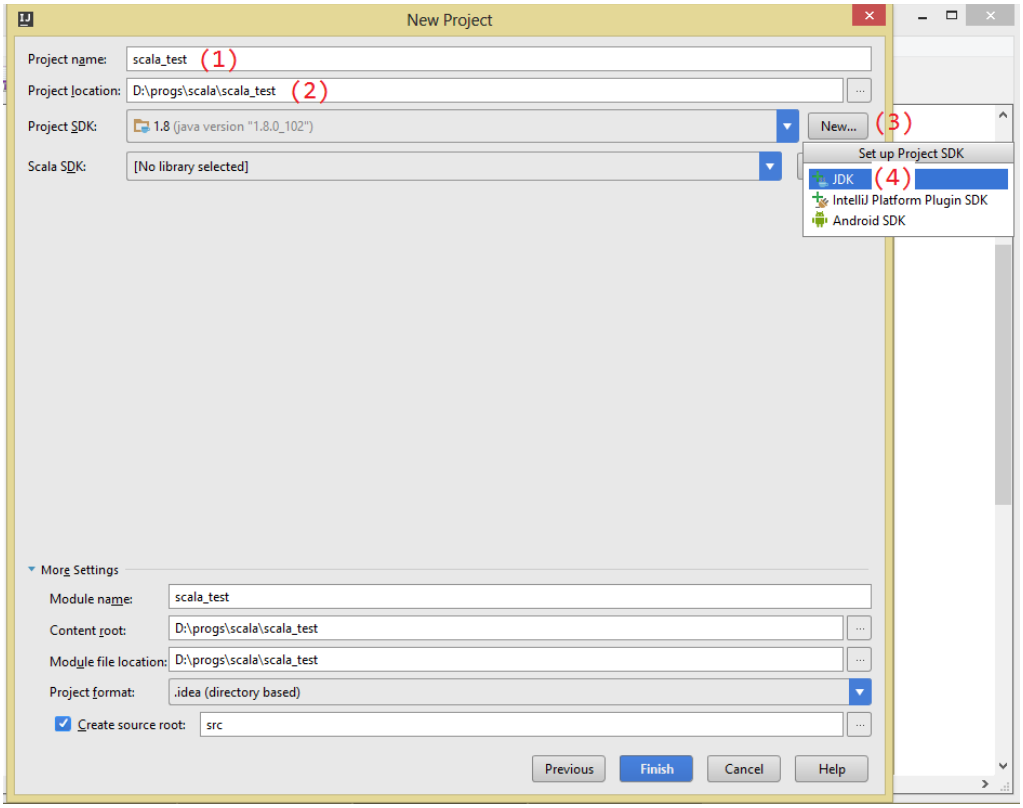

Рисунок 2.12 – Настройки проекта.

В появившемся окне (рисунок 2.13) указываем путь к JDK, установленному в пункте 2.1 настоящих методических указаний. По-умолчанию это C:\Program Files\Java\jdk… или C:\Program Files (x86)\Java\jdk…

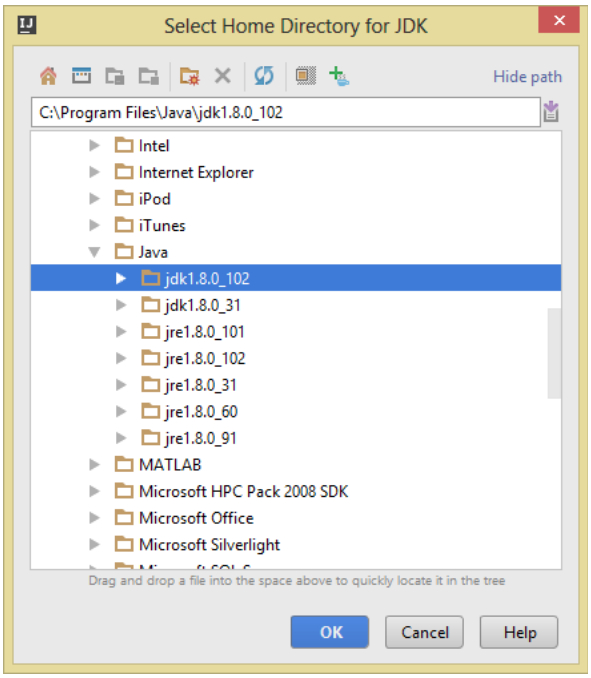

Рисунок 2.13 – Окно выбора пути к JDK

Затем надо указать Scala SDK. Для этого нажимаем кнопку «Create» напротив Scala SDK. В появившемся окне (рисунок 2.14) нажимаем кнопку «Download» и выбираем (рисунок 2.15) самую последнюю версию Scala (на

момент написания настоящих методических указаний актуальной версией является 2.11.8).

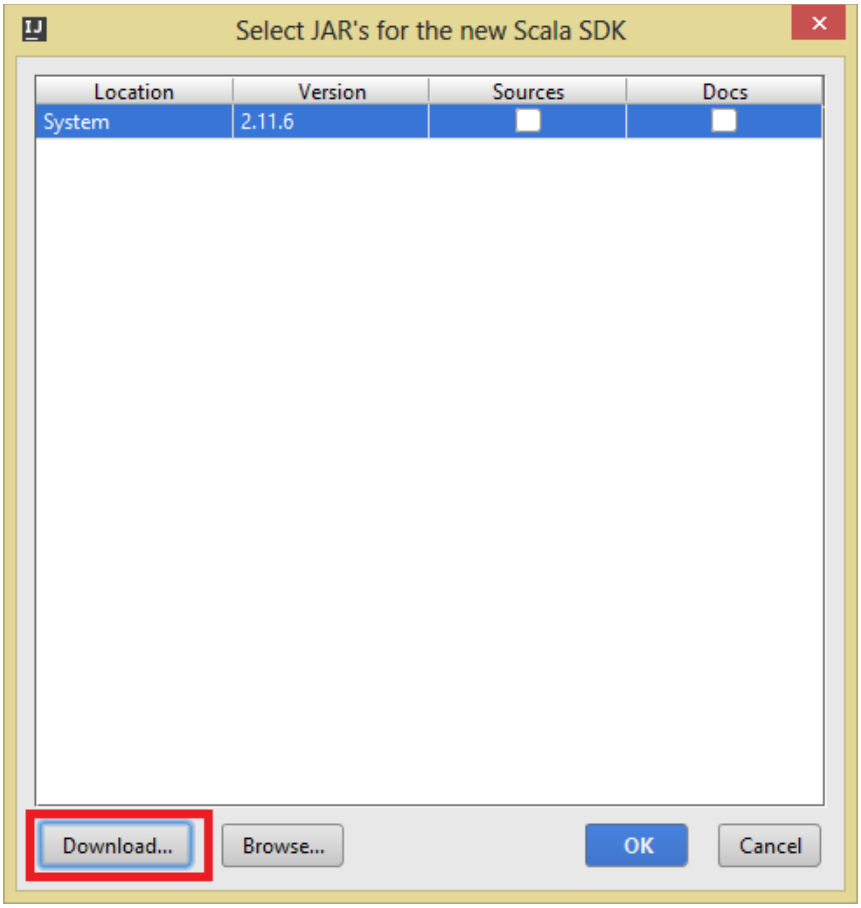

Рисунок 2.14 – Окно выбора Scala SDK

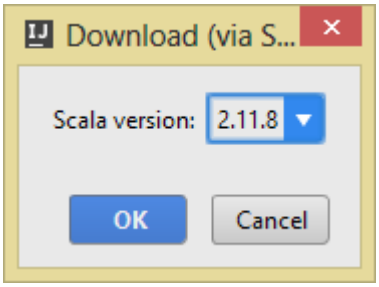

Рисунок 2.15 – Окно выбора версии Scala

После этого начнется процесс загрузки необходимых файлов (рисунок 2.16), который может занять времени больше, чем хотелось бы.

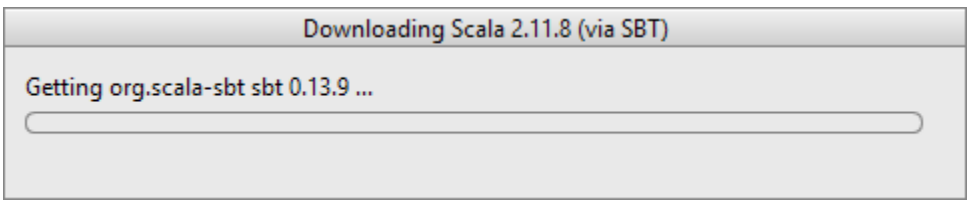

Рисунок 2.16 – Процесс установки необходимых файлов

После завершения установки нажимаем кнопку «Finish» (рисунок 2.17).

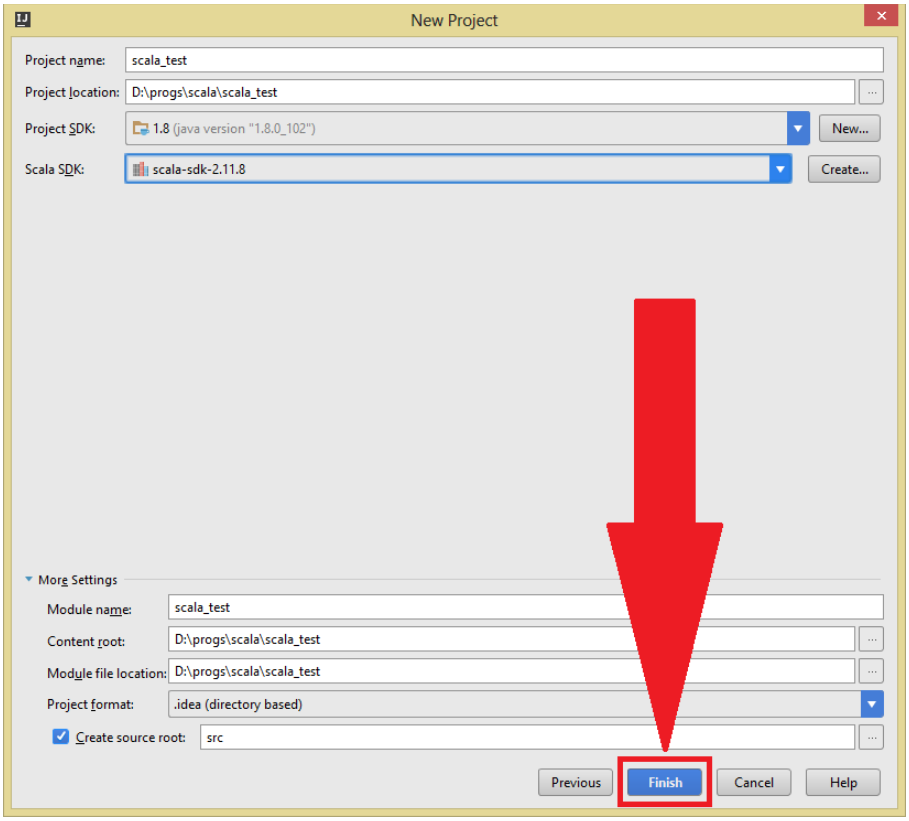

Рисунок 2.17 – Завершение настройки проекта

Откроется основное окно IntelliJ IDEA с созданным проектом (рисунок 2.18).

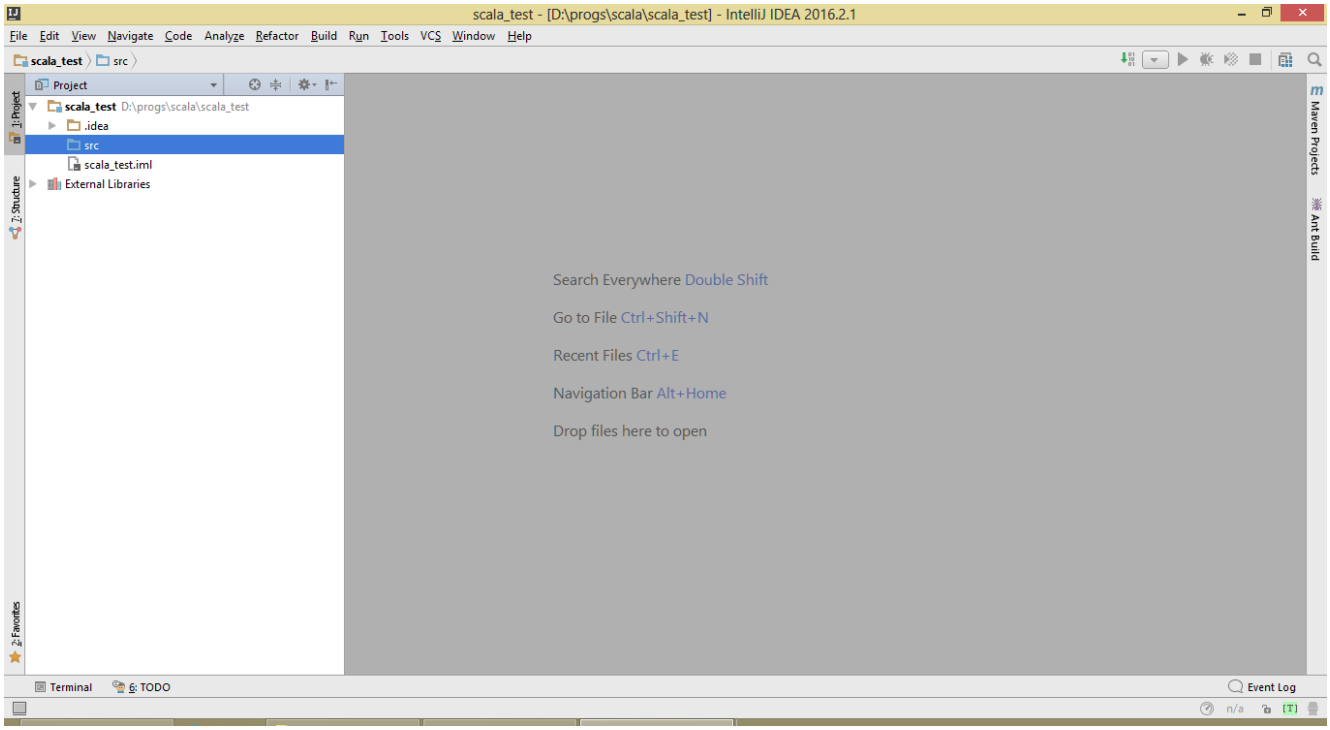

Рисунок 2.18 – Основное окно IntelliJ IDEA

В папке *src* создадим пакет *main*, внутри которого создадим пакет *scala* (рисунок 2.19).

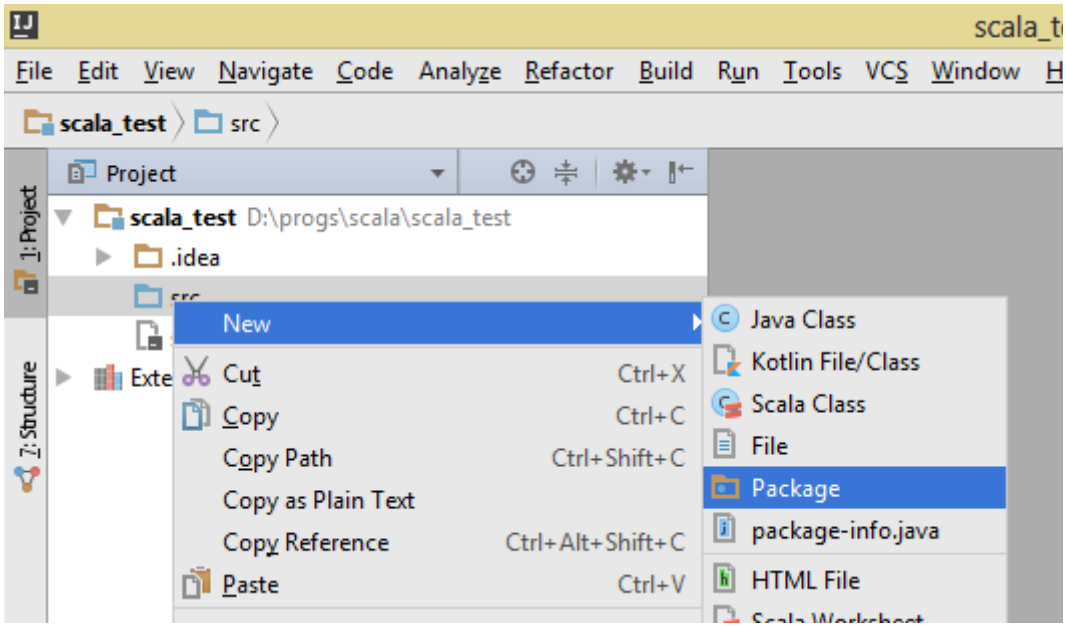

Рисунок 2.19 – Создание пакета

Затем в пакете *main.scala* создадим объект приложения (рисунки 2.20 и 2.21).

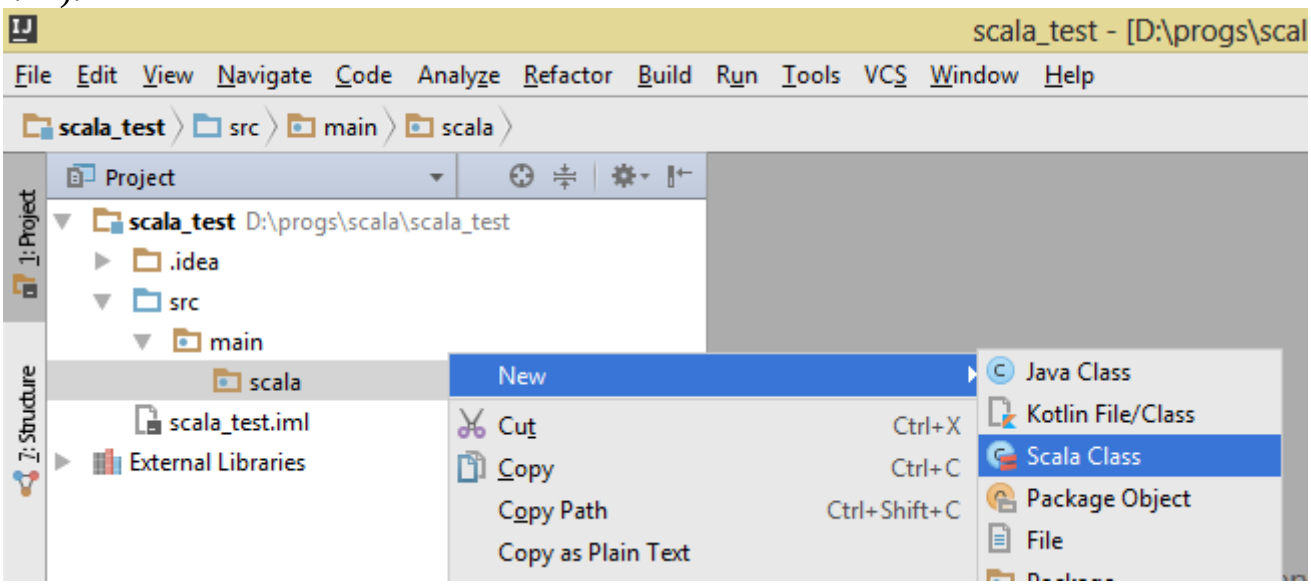

Рисунок 2.20 – Создание файла с исходным кодом Scala

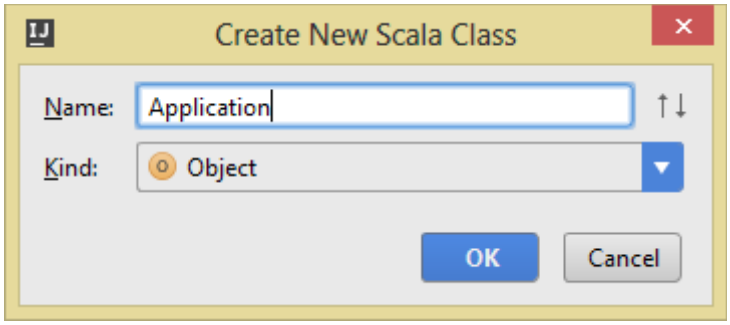

Рисунок 2.21 – Создание объекта приложения

В открывшемся файле введём код, представленный на рисунке 2.22.

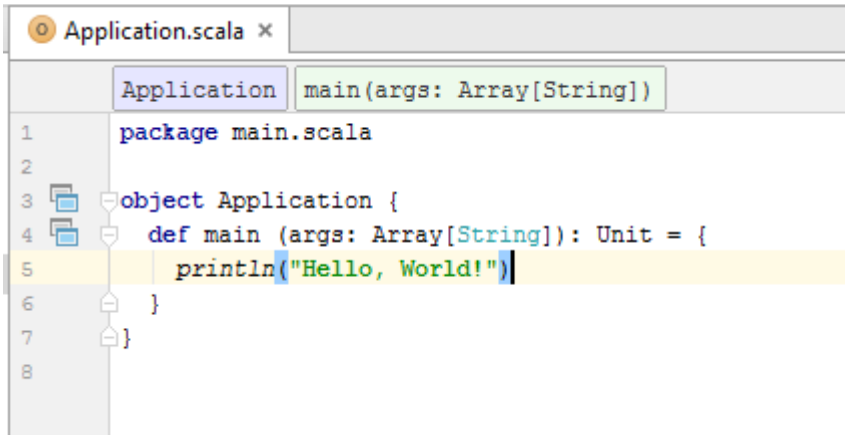

Рисунок 2.22 – Тестовый код для проверки работоспособности окружения

Сохраним файл и запустим его. Для этого нажмём правой кнопкой на вкладке с файлом и выберем «Run 'Application'» (рисунок 2.23).

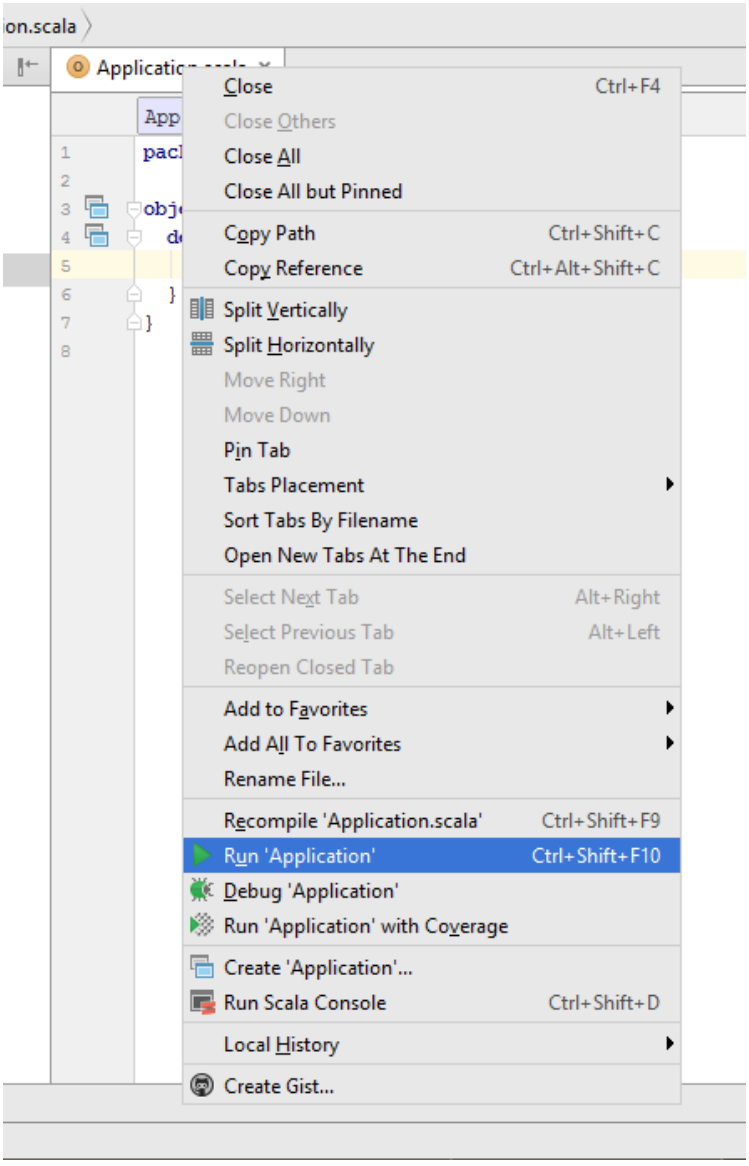

Рисунок 2.23 – Запуск программы

В результате в нижней части окна откроется консоль с результатами работы программы. (рисунок 2.24)

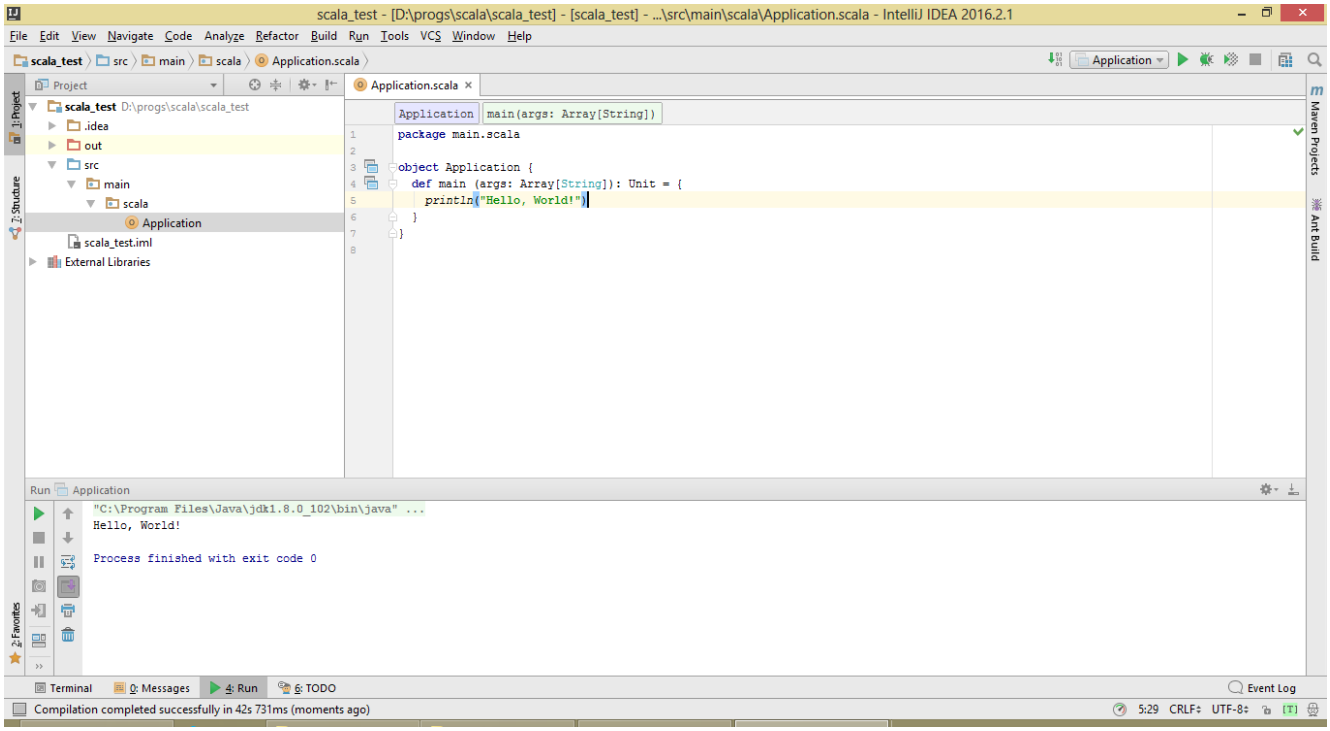

Рисунок 2.24 – Результаты работы программы

2.5. Основы синтаксиса языка Scala

2.5.1. Объявление переменных

В языке скала существует два способа объявления переменных: с помощью ключевого слова *val* и ключевого слова *var*. Значения, объявленные с помощью *val* являются константами, они принимают своё значение один единственный раз – при объявлении. Значения, объявленные с помощью var являются обычными переменными, привычными по императивным языкам. Пример:

```
val i = 0
var j = 0
j = j + 1    // OK
i = i + 1    // Ошибка Reassignment to val
```

На примере выше видно, что:

1) Однострочный комментарий в языке Scala начинается с // (многострочный комментарий начинается с /* и заканчивается */)
2) В конце инструкции не обязательно ставить точку с запятой (;). Однако, если на одной строке идут подряд несколько инструкций, между ними необходимо ставить точку с запятой

3) Объявлять тип переменной при её определении не обязательно (несмотря на то, что Scala – язык со статической типизацией). Компилятор в силах сам вывести тип переменной, стоящей слева от знака присваивания, основываясь на выражении, стоящем справа от знака присваивания.

Однако, если очень хочется, можно явно указать тип следующим образом:

```
val i: Double = 1    // аналог на Java – double i = 1;
```

2.5.2. Основные типы

Как и в языке Java, в Scala семь числовых типов – Byte, Char, Short, Int, Long, Float и Double. Также есть логический тип Boolean. В отличии от Java, все эти типы являются классами. Соответственно, можно вызывать методы от чисел:

```
1.toString()    // вернёт строку “1”
1.to(5)        // вернёт объект класса Range(1, 2, 3, 4, 5)
```

2.5.3. Упрощённый вызов методов

Язык Scala поддерживает упрощённый вызов методов, например:

```
a.method(b)
a method b
```

Описанные выше две строки идентичны. Такая возможность вызова методов позволяет в некоторых случаях писать более понятный код и реализовывать DSL (Domain-specific language).

Ещё одной особенностью языка Scala в плане вызовов методов является возможность не указывать скобки при вызове метода, не имеющего параметры, например:

```
1.toString    // идентично вызову 1.toString()
// или даже так:
1 toString
```

Среди программистов на языке Scala существует соглашение, функции/методы без параметров, которые не имеют побочных эффектов, вызываются без скобок, а функции/методы без параметров, которые имеют побочные эффекты, вызываются со скобками.

2.5.4. Метод Apply

В Scala используется синтаксис, напоминающий вызовы функций. Например, если *str* – это строка, то выражение *str(i)* вернёт *i*-ый символ строки.

```
“Hello”(4)    // вернёт ‘o’
```

Это можно считать перегруженной формой оператора (). Но на самом деле это метод с именем *apply*. И поэтому вызов *“Hello”(4)* является краткой формой записи *“Hello.apply(4)”*.

2.5.5. Условные выражения

В Scala имеется условное выражение if/else, как и в языке Java, но в Scala это не просто оператор, а именно выражение (со своим значением). Например, выражение

```
if (x > 0) 1 else -1
```

имеет значение 1 или -1, в зависимости от значения x. И всё это выражение имеет тип Int. По сути это аналог тернарного оператора из Java/C++/C# (*x > 0 ? 1 : -1*).

Пример инициализации значения переменной в зависимости от условия на языке Java:

```
int y = 0;
if (x > 0) y = 1 else y = -1;
```

То же самое на языке Scala

```
val y = if (x > 0) 1 else -1    // или на Java: int y = x > 0 ? 1 : -1;
```

Если вместо одного выражения идёт блок выражений, то конечным значением всего блока будет значение последнего выражения. Например:

```
val i = {1; 2}    // i имеет значение 2

val test = if (x > 0)
{    // начало первого блока
  2
  println(2+2)
  “String”
}    // значение первого блока – “String”
else
{    // начало второго блока
  3
  2+2
}    // значение второго блока - 4
```

В данном примере ветви выражения if/else имеют разный тип. В таком случае типом переменной temp будет являться общий предок типов ветвей

выражения. В данном случае тип ветви if – *String*, а тип ветви else – *Int*. Эти два типа не имеют другого общего предка, кроме класса *Any* (аналог класса *Object* в языке Java. Класс, от которого автоматически наследуются все остальные классы). Таким образом, переменная temp имеет тип *Any*.

2.5.6. Циклы

В Scala имеются циклы while и do-while, как в языках Java/C++/C#:

```
var n = -10
while (n < 0) {
  println(n)
  n = n + 1
}

var m = 0
do {
  println(m)
  m += 1    // в Scala нет оператора ++, но есть оператор +=
} while (m < 10)
```

В Scala нет прямого аналога оператора *for(<инициализация> ; <условие выхода> ; <изменение>)*, но есть своё собственное выражение for, которое будет рассмотрено в следующих лабораторных работах.

2.5.7. Функции и процедуры

Функции в языке Scala определяются следующим образом:

```
def <имя функции>(<список параметров>): <тип возвращаемого значения> = {
  <тело функции>
}
```

Например, функция, которая перемножает свои аргументы, на языке Scala определяется следующим образом:

```
def mult(a: Int, b: Int): Int = {
  a*b
}
```

Обратите внимание на отсутствие оператора return. Как уже упоминалось выше (пункт 2.5.5) значением блока является последнее выражение. В данном случае значением, которое возвращает функция, является значение последнего вычисленного в ней выражения. Так же, как и для переменных, для функций не обязательно задавать тип возвращаемого значения. Компилятор в силах сам

вычислить тип (за исключением рекурсивных функций, в которых всегда необходимо явно указывать тип). Таким образом, нижеизложенные определения функций полностью идентичны и законны в рамках синтаксиса языка Scala:

```
def fun1(a: Int, b: Int): Int = { a+b }
def fun2(a: Int, b: Int) = { a+b }
def fun3(a: Int, b: Int) = a+b
```

Процедуры на языке Scala определяются так же, как и функции, с одной лишь особенностью: между сигнатурой функции и открывающей скобкой блока кода отсутствует знак ‘=’. Пример:

```
def funkciya() = { 2+3 } // это функция, которая всегда возвращает 5

def procedura() { 2+3 } // это процедура, которая всегда возвращает ничего
```

Для обозначения «ничего» в языке Scala существует класс Unit (аналог void в Java/C++/C#). Таким образом, все процедуры имеют тип Unit.

Если тело функции состоит всего из одного выражения, то указывать фигурные скобки не обязательно. Пример:

```
def sum(a: Int, b: Int): Int = a + b
```

2.5.8. Рекурсивные функции

Рекурсивная функция – это функция, которая в своём теле содержит вызов самой себя. Продемонстрируем рекурсивную функцию на примере вычисления факториала n-го числа:

```
def fact(n: Int): Int = {
  if (n == 0) 1
  else n * fact(n-1)
}
```

Первое правило рекурсии – прежде чем входить в рекурсию, подумай, как из неё выйти. В данном случае мы сразу определили граничное условие завершения рекурсии (n == 0). А далее следует тривиальное определение факториала ($fact_n=n*fact_{n-1}$).

Рассмотрим подробнее как происходит вычисление факториала. Для примера вычислим факториал от числа 3 (рисунок 2.25):

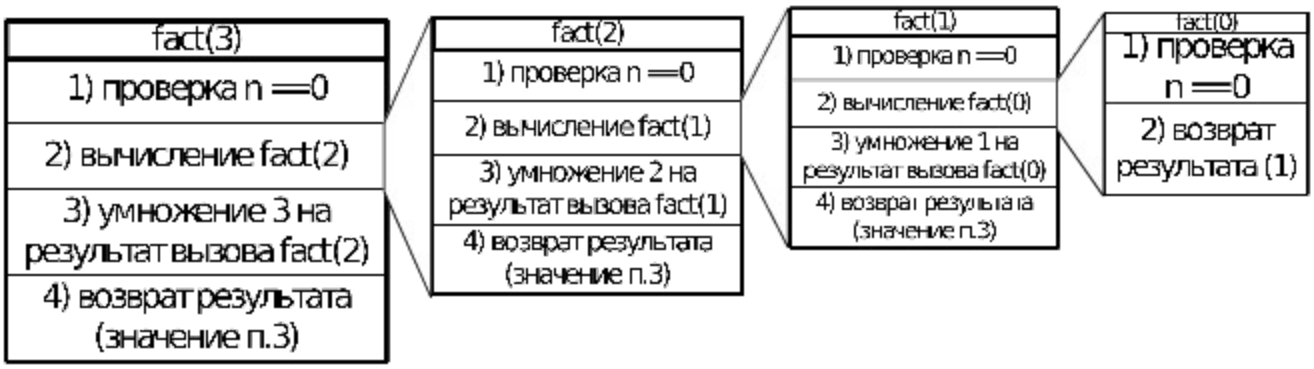

Рисунок 2.25 – Рекурсивные вызовы функции вычисления факториала

На рисунке 2.25 видно, что между рекурсивным вызовом (п. 2) и возвращением результата из функции (п. 4) есть еще действия, которые необходимо выполнить (п. 3). Таким образом, во время каждого нового рекурсивного вызова необходимо сохранять в стеке контекст предыдущего вызова, чтобы к нему можно было вернуться и продолжить вычисления. Это ведёт к накладным расходам и, в критическом случае, переполнению стека.

Выходом из данной ситуации является хвостовая рекурсия

2.5.9. Хвостовая рекурсия

Хвостовая рекурсия – это особый вид рекурсии. Отличие заключается в том, что в теле функции рекурсивный вызов находится в самом конце, перед возвратом значения из функции.

Это позволяет на уровне компиляции оптимизировать рекурсивные вычисления и преобразовать их в циклы. При таком подходе значение, которое необходимо вернуть из функции не вычисляется напрямую, (как в п.3 рисунка 2.25). Значением, возвращаемым из функции, является значение следующего рекурсивного вызова. Таким образом, нет нужды сохранять контекст выполнения вызывающей функции при рекурсивном вычислении и можно использовать одну и ту же ячейку стека для последующих вызовов.

Чтобы преобразовать обычную рекурсию в хвостовую, необходимо перенести вычисление результата (п.3 рис. 2.25) из тела функции в её параметр (т.н. аккумулятор). Преобразуем нашу функцию факториала к рекурсивному виду.

```
def fact(n: Int, acc: Int): Int = {
  if (n == 0) acc
  else fact(n-1, acc * n)
}
// Начальное значение аккумулятора = 1
fact(7, 1) // Запуск рекурсии
```

Как видно, при преобразовании функции к рекурсивному виду изменился ее интерфейс (был один входной параметр, а стало два). В случаях, когда надо

сохранить интерфейс хвостовую рекурсию оборачивают в функцию с желаемым интерфейсом:

```
def fact(n: Int): Int = {
  // внутри функции определим вспомогательную функцию,
  // которая реализует хвостовую рекурсию
  // (да, в Scala можно объявлять функции внутри функций)
  def factAcc(_n: Int, acc: Int): Int = {
    if (_n == 0) acc
    else factAcc(_n-1, acc * _n)
  }

  // и в качестве результата внешней функции вернём результат
  // работы функции с хвостовой рекурсией
  factAcc(n, 1)
}
```

Таким образом, мы добились двух вещей:

1) Преобразовали обычную рекурсию в хвостовую.
2) Сохранили интерфейс функции.

2.5.10. Списки

Список (List) – это коллекция данных, которая имеет определённую структуру. Список можно представить в виде цепочки, где каждое звено состоит из двух ячеек. В первой ячейке n-го звена содержится значение n-го элемента списка, а во второй ячейке звена содержится указатель на продолжение списка (на звено n+1). Структура списка «12, 3, 7, 4, 21, 9» представлена на рисунке 2.26.

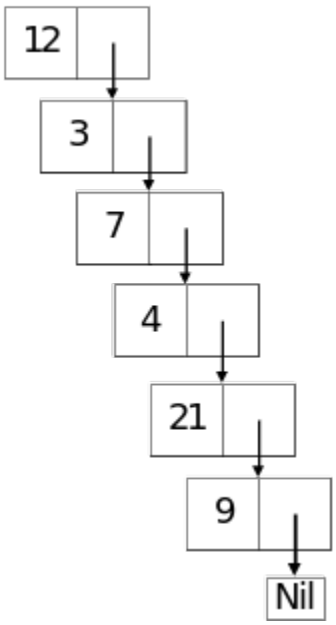

Рисунок 2.26 – Структура списка

Последним элементом списка обязательно должен является элемент, обозначающий конец списка (Nil).

В Scala списки можно создать следующим образом:

```
val spisok = 12 :: 3 :: 7 :: 4 :: 21 :: 9 :: Nil // spisok имеет тип List[Int]
// или
val spisok_dva = List(12, 3, 7, 4, 21, 9) // в данном случае добавлять
// Nil в конец списка не надо (но он там есть)
```

При объявлении переменной spisok_dva мы не указали тип содержимого списка, но компилятор в силах сам вывести тип, таким образом типом значения spisok_dva является List[Int].

Для работы со списком есть два основных свойства – head (для получения первого значения в списке) и tail (для получения продолжения списка).

```
val nums = 3 :: 7 :: 4
nums.head // вернёт 3
nums.tail // вернёт List(7, 4)
nums.tail.tail // вернёт List(4) – не значение, а список из одного значения
nums.tail.tail.tail // вернёт Nil
```

Стандартные списки в Scala неизменяемые (как и большинство значений в функциональном программировании). Это означает, что для изменения значения списка необходимо создать новый список с обновленным значением.

Этот факт налагает различную стоимость на операции со списком. Операции с головой списка производятся быстро. Можно сразу получить значение головы списка, также можно сразу удалить голову списка (вызвать tail) или заменить голову (присоединить новое значение к хвосту списка).

```
val lst = List(4, 9 ,6)
nums.tail // удаление головы (вернёт список без его головы)
2 :: nums.tail // изменение головы списка
```

Добавление в начало списка также проводится быстро:

```
val another_list = 375 :: 88 :: 42
// добавление числа 99 к голове списка
99 :: another_list // вернёт новый список List(99, 375, 88, 42)
```

Чтобы получить значение n-го элемента списка, необходимо перебрать n-1 элемент списка, которые предшествуют искомому значению, следовательно, чем дальше элемент, который мы хотим получить, тем дольше будет выполняться операция получения значения.

С добавлением элемент в конец списка тоже не всё так просто. При добавлении элемента в начало списка создается новое звено из двух ячеек. В первую ячейку помещается значение, которое мы хотим добавить к списку, а во вторую – указатель на имеющийся список. По аналогии, чтобы добавить значение к концу списка, необходимо создать новое звено, в первую ячейку помещается

добавляемое значение, а во вторую – Nil. Чтобы присоединить это звено к списку необходимо изменить указатель последнего звена имеющегося списка, чтобы вместо Nil он указывал на только что созданное новое звено. Но, т.к. объекты и значение неизменяемые, то для того, чтобы изменить последнее звено списка, нужно создать его заново (со старым значением и новым указателем на продолжение списка). Это влечёт за собой пересоздание предпоследнего звена списка, затем предпредпоследнего и т.д. Таким образом, при добавлении элемента в конец списка, происходит перестройка всего списка.

2.5.11. Обработка списков

Для обработки списков удобно применять рекурсивные функции шаблона «пока не Nil – обрабатываем голову – делаем рекурсивный вызов от хвоста списка». Например, продемонстрируем функцию, которая печатает каждый элемент списка:

```
def printList(lst: List[Int]): Unit = {
  if (lst == Nil) println("Конец списка")
  else {
    println(lst.head)
    printList(lst.tail)
  }
}
```

Ниже изображены еще два примера рекурсивных функций для обработки списков:

```
// Функция возвращает сумму ввсех его элементов
def sumList(lst: List[Int]): Int = {
  if (lst == Nil) 0
  else lst.head + sumList(lst.tail)
}

// Функция возвращает новый список, где каждый элемента
// в два раза больше соответствующего элемента исходного списка
def doubleList(lst: List[Int]): List[Int] = {
  if (lst == Nil) Nil
  else lst.head * 2 :: doubleList(lst.tail)
}
```

Перепишем эти функции используя хвостовую рекурсию:

```
// Функция возвращает сумму ввсех его элементов
def sumList(lst: List[Int]): Int = {
  def sumListAcc(__list: List[Int], acc: Int): Int = {
    if (__list == Nil) acc
    else sumListAcc(__list.tail, acc + __list.head)
  }
```

```
  sumListAcc(lst, 0)
}

// Функция возвращает новый список, где каждый элемента
// в два раза больше соответствующего элемента исходного списка
def doubleList(lst: List[Int]): List[Int] = {
  def doubleListAcc(__list: List[Int], acc: List[Int]): List[Int] ={
    if (__list == Nil) acc
    else doubleListAcc(__list.tail, __list.head * 2 :: acc)
  }

  doubleListAcc(lst, Nil).reverse
}
```

3. Задание на лабораторную работу

Написать функцию, которая преобразует список в соответствии с вариантом.

1. Для своего варианта задания написать два вида функции – обычную рекурсию и хвостовую рекурсию.
2. Убедиться, что на достаточно большом количестве элементов в списке обычная рекурсия выдаёт ошибку переполнения стека (stack overflow), а хвостовая рекурсия продолжает работать.
3. Зафиксировать длину списка (количество элементов), для которого обычная рекурсия перестаёт работать.

3.1. Варианты заданий:

1) Функция возвращает новый список, в котором каждый элемент является суммой предыдущего элемента нового списка и текущего элемента входного списка. Пример: List(1, 4, 2) преобразуется в List(1, 5, 7).
2) Функция возвращает новый список, в котором каждый элемент входного списка повторяется дважды. Пример: List(33, 9, 321) преобразуется в List(33, 33, 9, 9, 321, 321)
3) Функция возвращает новый список, в котором каждый элемент является список из чисел от 0 до модуля соответствующего элемента входного списка. Пример: List(2, -6, 10) преобразуется в List(List(0, 1, 2), List(0, 1, 2, 3, 4, 5, 6), List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10))