**Jakub Benduch, nr albumu: 281242**

**Sprawozdanie Scala**

**Laboratorium 1**

**1. Napisz funkcję rysującą prostokąt o zadanych wymiarach, który składa się z samych gwiazdek. Na przykład po wywołaniu tej funkcji z parametrami 3 i 8 powinniśmy otrzymać:**

**\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\***

Kod źródłowy:

rectangle(3,8)

**def** rectangle(row:Int, col:Int): Unit = **for** {i <- 1 to row} println("\*"\*col)

**2. Napisz program obliczający wartość wyrażenia:**

**Wskazówki:**

* **Liczbę 750000000 można zapisać w postaci Val x = 7.5e8**
* **Importować bibliotekę scala.math.\_**

Kod źródłowy:

**import** scala.math.\_

**val** w = pow(2.0\*log(521.0) + 7.5e8/(128.0\*422.0),1.0/3.0)

printf("Wartość wyrażenia: %f",w) // Wartość wyrażenia: 24,042336

**3. Napisz program, który znajduje wszystkie trzycyfrowe liczby, których suma sześcianów poszczególnych cyfr jest równa danej liczbie.**

**Wskazówki: Skorzystaj z operatorów arytmetycznych % i / dla argumentów całkowitych, gdzie:**

* **% - operator modulo – reszta z dzielenia. Na przykład wartością wyrażenia 937%10 jest liczba 7**
* **/ - operator dzielenia całkowitego. Na przykład wartością wyrażenia 937/10 jest liczba 93 (ale wartością 937/10.0 jest liczba 93.7)**
* **Proszę spróbować nie korzystać z instrukcji pętli tylko z innych własności scali**

Kod źródłowy:

**val** x = List.range(100,1000)

check(x) // 153, 370, 371, 407

**def** check(x:*List*[Int]): Any = {

**if** (!x.isEmpty){

**val** v:Int = x.head

**if** (v == (pow((v/100)%10,3) + pow((v/10)%10,3) + pow(v%10,3))){

println(v)

}

check(x.tail) // zamiast pętli, wykorzystano mechanizm rekursji

}

}

**Laboratorium 2**

**1. Utworzyć kilka przykładowych klas w scali. Sprawdzić możliwości definiowania pól oraz metod w klasach oraz możliwości dostępu do zmiennych klasy.**

Modyfikator *public* – ustawiony domyślnie, pola i metody są dostępne zarówno na zewnątrz klasy, jak i w klasach zagnieżdżonych wewnątrz danej klasy

Modyfikator *protected* – pola i metody dostępne w klasie bazowej, a także dla wszystkich klas zagnieżdżonych wewnątrz niej oraz tych zewnętrznych, które po niej dziedziczą

Modyfikator *private* – pola i metody dostępne tylko wewnątrz danej klasy i klasach zagnieżdżonych wewnątrz, nie są dostępne i nie mogą być modyfikowane poza klasą, w której są zdefiniowane, nie ulegają mechanizmowi dziedziczenia

Kod źródłowy:

//przykładowa klasa z polami xc i yc, oraz metodą move(Int, Int)

**class** Point(xc: Int, yc: Int) {

**var** x: Int = xc

**var** y: Int = yc

**def** move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Point x location : " + x);

println ("Point y location : " + y);

}

}

//modyfikator private

**class** Outer1 {

**class** Inner1 {

**private** **val** x = 0

**private** **def** f() { println("f") }

**class** InnerMost1 {

println(x) //ok - dostęp do pola prywatnego z poziomu klasy zagnieżdżonej

f() //ok - dostęp do metody prywatnej z poziomu klasy zagnieżdżonej

}

}

**val** foo1 = **new** Inner1

//println(foo1.x) //błąd - brak dostępu z zewnątrz do pola prywatnego

//foo1.f() //błąd - brak dostępu z zewnątrz do metody prywatnej

}

**class** Super1 {

**private** **val** x = 0

**private** **def** f() { println("f") }

}

**class** Sub1 **extends** Super1 {

//println(x) //błąd - brak dostępu do pola prywatnego z poziomu klasy dziedziczącej

//f() //błąd - brak dostępu do metody prywatnej z poziomu klasy dziedziczacej

}

**class** Other1 {

**val** foo1 = **new** Super1

//println(foo1.x) //błąd - brak dostępu z zewnątrz do pola prywatnego

//foo1.f() //błąd - brak dostępu z zewnątrz do metody prywatnej

}

//modyfikator protected

**class** Outer2 {

**class** Inner2 {

**protected** **val** x = 0

**protected** **def** f() { println("f") }

**class** InnerMost2 {

println(x) //ok - dostęp do pola chronionego z poziomu klasy zagnieżdżonej

f() //ok - dostęp do metody chronionej z poziomu klasy zagnieżdżonej

}

}

**val** foo2 = **new** Inner2

//println(foo2.x) //błąd - brak dostępu z zewnątrz do pola chronionego

//foo2.f() //błąd - brak dostępu z zewnątrz do metody chronionej

}

**class** Super2 {

**protected** **val** x = 0

**protected** **def** f() { println("f") }

}

**class** Sub2 **extends** Super2 {

println(x) //ok - dostęp do pola chronionego z poziomu klasy dziedziczącej

f() //ok - dostęp do metody chronionej z poziomu klasy dziedziczacej

}

**class** Other2 {

**val** foo1 = **new** Super2

//println(foo2.x) //błąd - brak dostępu z zewnątrz do pola chronionego

//foo2.f() //błąd - brak dostępu z zewnątrz do metody chronionej

}

//modyfikator public

**class** Outer3 {

**class** Inner3 {

**val** x = 0

**def** f() { println("f") }

**class** InnerMost3 {

println(x) //ok - dostęp do pola publicznego z poziomu klasy zagnieżdżonej

f() //ok - dostęp do metody publicznego z poziomu klasy zagnieżdżonej

}

}

**val** foo3 = **new** Inner3

println(foo3.x) //ok - dostęp z zewnątrz do pola publicznego

foo3.f() //ok - dostęp z zewnątrz do metody publicznej

}

**class** Super3 {

**val** x = 0

**def** f() { println("f") }

}

**class** Sub3 **extends** Super3 {

println(x) //ok - dostęp do pola publicznego z poziomu klasy dziedziczącej

f() //ok - dostęp do metody publicznej z poziomu klasy dziedziczacej

}

**class** Other3 {

**val** foo3 = **new** Super3

println(foo3.x) //ok - dostęp z zewnątrz do pola publicznego

foo3.f() //ok - dostęp z zewnątrz do metody publicznej

}

**2. Pokazać różnice pomiędzy klasą, a obiektem. Wskazać przypadki, w których należy definiować klasy oraz takie, w których muszą być używane obiekty.**

* Obiekt Scali używany jest w przypadku, gdy nie może istnieć więcej niż jedna jego instancja (czyli są jest Singleton’em)
* Obiekt Scali nie przyjmuje parametrów podczas inicjalizacji
* Obiekt Scali jest w pewnym sensie odpowiednikiem słowa kluczowego „static” z języka Java, stąd metoda main musi koniecznie być umieszczona w obiekcie Scali, a nie klasie (można utworzyć klasę z metodą main, jednak wówczas nie jest ona statyczna i JVM nie jest w stanie znaleźć punktu startowego programu)

Kod źródłowy:

//object Person(firstName: String, lastName:String, age: Int) {

//def greet() {

//println(s"Hello, I'm $firstName $lastName and I am $age years old. Nice to meet you.")

//}

//} //bląd - obiekt nie może przyjmować parametrów

**object** JakubBenduch{

**val** firstName = "Jakub"

**val** lastName = "Benduch"

**def** fullName: *String* = firstName + " " + lastName

**def** greet():*String* = {

"Hello, my name is " + fullName + "."

}

}

**class** Person(firstName: *String*, lastName: *String*, age: Int) {

**def** greet() {

println(s"Hello, I'm $firstName $lastName and I am $age years old. Nice to meet you.")

}

}

//object JakubBenduch{} //błąd - nie mogą istnieć dwa obiekty Scali o takiej samej nazwie

**val** a = **new** Person("Jakub", "Benduch", 23)

**val** b = **new** Person("Małgosia", "Dąbrowa", 21)

**val** c = **new** Person("Martin", "Odersky", 50)

a.greet()

b.greet()

c.greet()

**object** ex1{

**def** main(args: Array[*String*]) {

println("Hello, world")

}

}

**class** ex2{

**def** main(args: Array[*String*]) {

println("Hello, world")

}

} //jest możliwe utworzenie klasy z metodą main, jednak po kompilacji program nie zadziała

**3. Wskazać różnice pomiędzy regularnymi klasami oraz case class.**

Tworzenie instancji zwykłej klasy wymaga użycia słowa kluczowego „new”, co nie jest wymagane, jeśli mowa o case class.

Kod źródłowy:

**class** Point(xc: Int, yc: Int) {

**val** x: Int = xc

**val** y: Int = yc

}

**case** **class** **Note**(name: *String*, duration: *String*, octave: Int)

**val** point1 = **new** Point(1,1)

//val point2 = Point(1,1) //błąd, bo nie ma słowa "new"

**val** note1 = **Note**("C", "Quarter", 3) //brak "new" ale nie zwraca komunikatu o błędzie

**val** note2 = **new** **Note**("C", "Quarter", 3) //stosowanie "new" nie jest zabronione, ale jest niepraktykowane

Klasy regularne wprowadzają pojęcie tożsamości, które nie istnieje w case class. Te same definicje obiektów klasy regularnej prowadzą do różnych wartości, podczas gdy te same definicje case class prowadzą do równych wartości. W Scali domyślnie porównywanie obiektów będzie porównywać ich tożsamość, ale w przypadku instancji klasy przypadków równość zostanie ponownie zdefiniowana w celu porównania wartości zagregowanej informacji.

Kod źródłowy:

**val** point3 = **new** Point(1,1)

println(point1 == point3) //zwraca fałsz

println(note1 == note2) //zwraca prawdę

Case class może być wykorzystane w mechanizmie dopasowywania wzorca, zwykła klasa nie.

Kod źródłowy:

//point1 match {

//case Point(xc, yc) => s"The xc of point1 is $xc" //klasa regularna nie może być wykorzystana w mechnizmie Pattern Matching

//}

note1 **match** {

**case** **Note**(name, duration, octave) => s"The duration of note1 is $duration" //case class jest wykorzystywana w Pattern Matching

}

Klasa regularna może rozszerzyć kolejną klasę, podczas gdy case class nie może rozszerzyć innej case class, ponieważ nie byłoby możliwe prawidłowe wdrożenie ich równości.

Kod źródłowy:

**class** Point3D(xc: Int, yc: Int, zc: Int) **extends** Point(xc, yc) {

**val** x: Int = xc

**val** y: Int = yc

**val** z: Int = zc

}

//case class BigNote(name: String, duration: String, octave: Int, size: Int) extends Note(name: String, duration: String, octave: Int) //błąd – zabronione

**4. Wskazać różnice między obiektami klas abstrakcyjnych oraz cech (trait).**

Klasa może dziedziczyć z wielu cech, ale tylko z jednej klasy abstrakcyjnej.

Kod źródłowy:

**abstract** **class** Xcor(xc: Int) {

**val** x: Int = xc

}

**abstract** **class** Ycor(yc: Int) {

**val** y: Int = yc

}

**abstract** **class** Zcor(zc: Int) {

**val** z: Int = zc

}

**trait** Rang {

**val** R\_angle: Int

}

**trait** Pang {

**val** P\_angle: Int

}

**trait** Yang {

**val** Y\_angle: Int

}

//class LocationByAbstractClasses(xc: Int, yc: Int, zc: Int) extends Xcor(xc: Int), Ycor(yc: Int), Zcor(zc: Int){

//val x: Int = xc

//val y: Int = yc

//val z: Int = zc

//} //Nie jest możliwe dziedziczenie od wielu klas abstrakcyjnych

**class** OrientationByTraits **extends** Rang **with** Pang **with** Yang {

**val** R\_angle = 20

**val** P\_angle = 10

**val** Y\_angle = 150

} //klasa może korzystać z wielu cech

**abstract** **class** Location(xc: Int, yc: Int, zc: Int) {

**val** x: Int = xc

**val** y: Int = yc

**val** z: Int = zc

}

**class** LocationAndOrientation(xc: Int, yc: Int, zc: Int) **extends** Location(xc: Int, yc: Int, zc: Int) **with** Rang **with** Pang **with** Yang {

**val** x: Int = xc

**val** y: Int = yc

**val** z: Int = zc

**val** R\_angle = 20

**val** P\_angle = 10

**val** Y\_angle = 150

} //klasa może rozszerzać jedną klasę abstrakcyjną oraz wiele cech

Klasy abstrakcyjne mogą mieć parametry konstruktora, a także parametry typu. Cechy mogą mieć tylko parametry typu.

Kod źródłowy:

**abstract** **class** ClassWithoutConstructorParameters(){}

**abstract** **class** ClassWithConstructorParameters(parameter1: Any, parameter2: Any){}

**trait** TraitWithoutConstructorParameters {}

//trait TraitWithConstructorParameters(parameter: Any) {}; //błąd - cecha nie może przyjmować parametrów konstruktora

**5. Zbudować drzewo dziedziczenia, w którym wszystkie obiekty będą miały wspólnego przodka oraz obiekty na kolejnych poziomach dziedziczenia będą rozbudowywane o kolejne cechy. Wskazać różnice pomiędzy dziedziczeniem, a kompozycją oraz pokazać przykłady polimorfizmu.**

Kod źródłowy:

**class** A {

**val** firstAInt = 1

**val** secondAInt = 2

**private** **val** thirdADouble = 1.23

**def** greetClass():Unit = {

println("Hello from class A")

}

}

**trait** firstTrait {

**val** firstTraitDouble = 77.7

**val** firstTraitString = "I'm first trait"

**def** greetTrait():Unit = {

println("Hello from first Trait")

}

}

**class** B **extends** A **with** firstTrait {

**override** **val** firstAInt = 32

//val thirdA = 4.32

}

**trait** secondTrait **extends** firstTrait {

**val** secondTraitChar = 'K'

**override** **def** greetTrait():Unit = {

println("Hello from second Trait")

}

}

**class** C **extends** A **with** firstTrait **with** secondTrait {

**val** firstCDouble = 4.5234

**override** **def** greetClass():Unit = {

println("Hello from class C")

}

}

**trait** thirdTrait {

**val** thirdTraitLong = 7.5e31

**def** greetTrait():Unit = {

println("Hello from third Trait")

}

}

//class D extends C with thirdTrait { //błąd - dwa razy ta sama metoda greetTrait() - jedna od secondTrait z klasy C, druga od thirdTrait

//def greetClass():Unit = {

//println("Hello form class D")s

//} //błąd

//}

**class** D **extends** C **with** firstTrait {

**val** firstDLong = 4.21e21

**override** **def** greetClass():Unit = {

println("Hello from class D")

}

}

**class** E **extends** C {}

**class** F **extends** A {}

**class** G **extends** B {}

**val** a = **new** B

println(a.firstAInt) //wyświetla 32

//println(a.thirdA) //błąd - thirdA jest polem prywatnym klasy A

a.greetClass() //"wyświetla "Hello from class A"

a.greetTrait() //wyświetla "Hello from first Trait"

println("------------------------------------------------------------------------------------------------------------------")

**val** b = **new** C

println(b.secondTraitChar) //wyświetla K

b.greetClass() //wyświetla "Hello from class A"

b.greetTrait() //wyświetla "Hello from second Trait"

println("------------------------------------------------------------------------------------------------------------------")

**val** c = **new** D

println(c.firstAInt) //wyświetla 1

println(c.secondAInt) //wyświetla 2

println(c.firstCDouble) //wyświetla 4.5234

println(c.firstDLong) // wyświetla 4.21E21

c.greetClass() //wyświetla "Hello from class D"

c.greetTrait() //wyświetla "Hello from second Trait"

println("------------------------------------------------------------------------------------------------------------------")

**val** d = **new** E

d.greetClass() //wyświetla "Hello from class C"

d.greetTrait() //wyświetla "Hello from second Trait"

println("------------------------------------------------------------------------------------------------------------------")

**val** e = **new** F

e.greetClass() //wyświetla "Hello from class A"

//e.greetTrait() //błąd - klasa F nie rozszerza żadnej cechy

/\* A

\* / | \

\* / | \

\* B F C

\* | / \

\* G D E

\*/

Podstawowe różnice między kompozycją, a dziedziczeniem:

* Kompozycja jest relacją „ma”, a dziedziczenie jest relacją „jest”.
* W przypadku dziedziczenia występuje silne sprzężenie - zmiana w nadklasie, może spowodować nieprzewidziane zachowanie w klasie dziedziczącej.
* Rozszerzać można tylko jedną klasę, komponować da się z wielu elementów (np. używając kilku cech (traits))

Polimorfizm w programowaniu oznacza, że można wykonać tę samą operację na różnych typach

Kod źródłowy:

//polimorfizm

**trait** Animal {

**def** makeVoice(): Unit

}

**class** Cat **extends** Animal{

**def** makeVoice(): Unit = println("Meow")

}

**class** Dog **extends** Animal{

**def** makeVoice(): Unit = println("Woof")

}

**class** Box(animal:Animal) {

**def** makeSound() = animal.makeVoice()

}

**val** cat = **new** Cat

**val** dog = **new** Dog

**val** box1 = **new** Box(cat)

**val** box2 = **new** Box(dog)

box1.makeSound() //wyświetla "Meow"

box2.makeSound() //wyświetla "Woof"

**6. Wykorzystać klasy kolekcji (List, Vector, …) do przechowywania obiektów z polecenia 5. Zbadać możliwości dostępu oraz wykorzystania zgromadzonego zestawu obiektów.**

Kod źródłowy:

**val** a1 = **new** A

**val** a2 = **new** A

**val** b1 = **new** B

**val** b2 = **new** B

**val** c1 = **new** C

**val** c2 = **new** C

**val** d1 = **new** D

**val** d2 = **new** D

**val** e1 = **new** E

**val** e2 = **new** E

**val** f1 = **new** F

**val** f2 = **new** F

**val** g1 = **new** G

**val** g2 = **new** G

**val** list = List(a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2)

**val** vector = Vector(a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2)

**val** set = Set(a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2)

**val** map = Map(1->a1, 2-> a2, 3->b1, 4->b2, 5->c1, 6->c2, 7->d1, 8->d2, 9->e1, 10->e2, 11->f1, 12->f2, 13->g1, 14->g2)

**val** tuple = **Tuple14**(a1,a2,b1,b2,c1,c2,d1,d2,e1,e2,f1,f2,g1,g2)

Kilka przykładów możliwości:

* Wyszukiwanie wartości maksymalnej bądź minimalnej
* Filtrowanie
* Wyznaczanie części wspólnej zbiorów, różnicy, sumy
* Iteracja – sprawdzanie czy elementy zbioru spełniają określony warunek, metoda map()
* Wiele innych

**7. Pokazać różnice pomiędzy sekwencjami Sets oraz Maps.**

Set to zbiór różnych elementów tego samego typu. Innymi słowy, Set jest kolekcją, która nie zawiera żadnych duplikatów elementów. Tworząc Set podaje się tylko wartości

Map to zbiór par klucz/wartość. Każda wartość może być zwrócona na podstawie jej klucza. Klucze są unikalne na mapie, ale wartości nie muszą być unikalne. Mapy są również nazywane tabelami haszującymi. Tworząc Map podaje się pary klucz/wartość wg wzoru:

Map(key1->value1, key2->value2,…)

Kod źródłowy:

**val** s = Set("red","blue","orange","blue", "brown", "red") //powtarzające się stringi zostały usunięte

**val** m = Map(1 -> "red", 2 -> "blue", 3 -> "orange", 4 -> "blue", 5 -> "brown", 6 -> "red") //powtarzające się wartości, ale to dopuszczalne, bo ich klucze są różne

**val** m1 = Map(1 -> "red", 2 -> "blue", 3 -> "orange", 1 -> "blue") //nadpisanie pary 1 -> "red" parą 1 -> "blue"

println(s) //wyświetla Set(red, blue, orange, brown)

println(m) //wyświetla Map(5 -> brown, 1 -> red, 6 -> red, 2 -> blue, 3 -> orange, 4 -> blue)

println(m1) //wyświetla Map(1 -> blue, 2 -> blue, 3 -> orange)

**8. Zaprezentować przykłady wykorzystania związane z mechanizmami dopasowania wzorców w scali. Pokazać możliwości zastosowania tego mechanizmu do dopasowywania:**

1. **Wartości**
2. **Obiektów**
3. **Typów**

**a.**

**def** matchValue(x: Int): *String* = x **match** {

**case** 1 => "one"

**case** 2 => "two"

**case** \_ => "many"

}

println(matchValue(1) + " " + matchValue(2) + " " + matchValue(3)) //wyświetla one two many

**b.**

/\*class SomeClass(name: String, id: Int)

object SomeObject1

object SomeObject2

def matchObject(obj: Any): Any = obj match {

case SomeObject1 => println("Object1")

case SomeObject2 => println("Object2")

case \_ => println("What?")

}

matchObject(SomeObject1)

matchObject(SomeObject2)

matchObject("Boring")

matchObject(2.34)\*/

**abstract** **class** Notification

**case** **class** **Email**(sender: *String*, title: *String*, body: *String*) **extends** Notification

**case** **class** **SMS**(caller: *String*, message: *String*) **extends** Notification

**case** **class** **VoiceRecording**(contactName: *String*, link: *String*) **extends** Notification

**def** matchObject(notification: Notification): *String* = notification **match** {

**case** **Email**(email, title, body) =>

s"You got an email from: $email \nTitle: $title \nContent:\n$body"

**case** **SMS**(number, message) =>

s"You got an SMS from $number! Message: $message"

**case** **VoiceRecording**(name, link) =>

s"you received a Voice Recording from $name! Click the link to hear it: $link"

}

**val** someSms = **SMS**("Frank", "Are you there?")

**val** someEmail = **Email**("Tom", "WR OLYH LV WR GLH", "ZKHQ D PDQ OLHV KH PXUGHUV VRPH SDUW RI WKH ZRUOG \nWKHVH DUH WKH SDOH GHDWKV ZKLFK PHQ PLVFDOO WKHLU OLYHV \nDOO WKLV L FDQQRW EHDU WR ZLWQHVV DQB ORQJHU \nFDQQRW WKH NLQJGRP RI VDOYDWLRQ WDNH PH KRPH")

println(matchObject(someSms))

println(matchObject(someEmail))

**c.**

**def** matchType(x: Any): Any = x **match** {

**case** i: Int => println("Integer: " + i)

**case** s: *String* => println("String: " + s)

**case** f: Double => println("Double: " + f)

**case** other => println("other: " + other)

}

matchType(1)

matchType("Hello")

matchType(2.0)

matchType('k')

**9. Pokazać możliwości wykorzystania krotek (tuples) w scali. Sprawdzić czy i jak działają mechanizmy dopasowywania wzorców dla krotek.**

Krotka pozawala przechowywać elementy różnego typu:

Kod źródłowy:

**val** t1 = ("Hello", 3.4, 't', 5, Console)

**val** t2 = **new** **Tuple5**("Hello", 3.4, 't', 5, Console) //to samo co powyżej z użyciem konstruktora

**val** t3 = (1,2,3,4)

**val** sum = t3.\_1 + t3.\_2 + t3.\_3 + t3.\_4 //sumowanie elementów

println("Suma: " + sum)

t3.productIterator.foreach{ i => println("Value = " + i)} //iterowanie zawartości krotki

println("Połączone stringi: " + t2.toString()) //metoda toString()

**val** t4 = ("Hello","World")

println("Zmieniona kolejność: " + t4.swap) //zamiana kolejności, koniecznie ten sam typ danych

Pattern matching z wykorzystaniem krotek:

Kod źródłowy:

println("Pattern matching on Tuples")

**val** strawberryDonut = **Tuple3**("Strawberry Donut", "Very Tasty", 2.50)

**val** glazedDonut = **Tuple3**("Glazed Donut", "Very Tasty", 2.50)

**val** plainDonut = **Tuple3**("Plain Donut", "Tasty", 2)

**val** donutList = List(glazedDonut, strawberryDonut, plainDonut)

**val** priceOfPlainDonut = donutList.foreach { tuple => {

tuple **match** {

**case** ("Plain Donut", taste, price) => println(s"Donut type = Plain Donut, price = $price")

**case** d **if** d.\_1 == "Glazed Donut" => println(s"Donut type = ${d.\_1}, price = ${d.\_3}")

**case** \_ => **None**

}

}

}

**10. Zaprogramować pierwsze zadania z poprzedniego tygodnia wykorzystując mechanizmy funkcyjne oraz metody map, reduce dostępne dla kolekcji.**

Kod źródłowy:

//zad1

rectangle(4,4)

**def** rectangle(row:Int , col:Int):Unit = {

**val** list = List.range(1, row+1).map(n => println(List.fill(col)("\*").reduce(\_+\_)))

}

//zad2

**val** array = Array((2.0,log(521.0)),(128.0,422.0)) map { x => x.\_1 \* x.\_2 }

**val** list = 7.5::List.fill(8)(10.0)

**val** w = pow(List(array(0),List(list.reduce(\_\*\_),array(1)).reduce(\_/\_)).reduce(\_+\_),1.0/3.0)

println(w) //liczenie tego w ten sposób to chyba trochę sztuka dla sztuki

//zad3

**val** v = Vector.range(100,1000)

check(v)

**def** check(v: *Vector*[Int]):Any = {

**for** { i <- v

**val** list = List((i/100)%10, (i/10)%10, i%10).map(x=>pow(x,3)).reduce(\_+\_)

}

**if** (i == list) println(list)

}

**11. Zmodyfikować zadanie nr 10 tak, aby wykorzystać mechanizmy „for comprehensions”.**

Kod źródłowy:

//zad3

**import** scala.math.\_

**val** v = Vector.range(100,1000)

print(pow3sum(v))

**def** pow3sum(v:*Vector*[Int]):*Vector*[Int] = {

**var** result = Vector[Int]()

**for** {

n <- v

**if** (n == (pow((n/100)%10,3) + pow((n/10)%10,3) + pow(n%10,3)))

} result = result :+ n

result

}

**12. Pokazać przykład mechanizmu rekursji dla wybranych zadań przetwarzania sygnału (np. sumowanie wartości listy, obliczanie silni)**

Kod źródłowy:

println(sumIt(List(10, 20, 30, 40, 50))) //wynik: 150

**def** sumIt(toSum:*List*[Int], sum:Int= 0):Int = {

**if**(toSum.isEmpty)

sum

**else**

sumIt(toSum.tail, sum + toSum.head)

}

println(silnia(5)) //wynik: 120

**def** silnia(x:Int):Int = {

**if** (x < 2) {

**return** 1

}

**else** {

**return** x\*silnia(x-1)

}

}

**Laboratorium 3**

**Zadanie: Utworzenie programu okienkowego: konwerter temperatury ˚C <-> ˚F**

Kod źródłowy:

**import** scalafx.application.JFXApp

**import** scalafx.Includes.\_

**import** scalafx.scene.layout.\_

**import** scalafx.scene.control.\_

**import** scalafx.scene.\_

**import** scalafx.scene.paint.\_

**import** scalafx.scene.text.\_

**import** scalafx.event.ActionEvent

**import** scalafx.scene.Parent.\_

**import** scalafx.geometry.\_

**object** file3 **extends** JFXApp{

stage = **new** JFXApp.PrimaryStage{

title = "Simple temperature converter Program"

scene = **new** Scene(390, 400) {

resizable = **false**

fill = Color.grayRgb(0,0.0431)

**val** subScene1 = **new** SubScene(400, 60) {

**val** label1 = **new** Label {

text = "Celsius to Fahrenheit converter"

font = **new** Font(18)

alignment = **Pos**.Center

prefHeight = 10

prefWidth = 50

}

root = label1

}

**val** textField1 = **new** TextField {

layoutX = 50

layoutY = 80

prefWidth = 100

prefHeight = 20

}

**val** textField2 = **new** TextField {

layoutX = 255

layoutY = 80

prefWidth = 100

prefHeight = 20

editable = **false**

}

**val** button1 = **new** Button {

font = **new** Font(14)

layoutX = 155

layoutY = 140

text = "Convert"

prefWidth = 100

prefHeight = 30

}

button1.onAction = (e:ActionEvent) => {

**val** temp = textField1.getText()

**val** C = temp.toDouble

**val** F = 1.8\*C +32

textField2.setText(F.toString)

}

**val** subScene2 = **new** SubScene(400, 60) {

layoutY = 210

**val** label2 = **new** Label {

text = "Fahrenheit to Celsius converter"

font = **new** Font(18)

alignment = **Pos**.Center

}

root = label2

}

**val** textField3 = **new** TextField {

layoutX = 50

layoutY = 290

prefWidth = 100

prefHeight = 20

}

**val** textField4 = **new** TextField {

layoutX = 255

layoutY = 290

prefWidth = 100

prefHeight = 20

editable = **false**

}

**val** button2 = **new** Button {

font = **new** Font(14)

layoutX = 155

layoutY = 350

text = "Convert"

prefWidth = 100

prefHeight = 30

}

button2.onAction = (e:ActionEvent) => {

**val** temp = textField3.getText()

**val** F = temp.toDouble

**val** C = (F-32)/1.8

textField4.setText(C.toString)

}

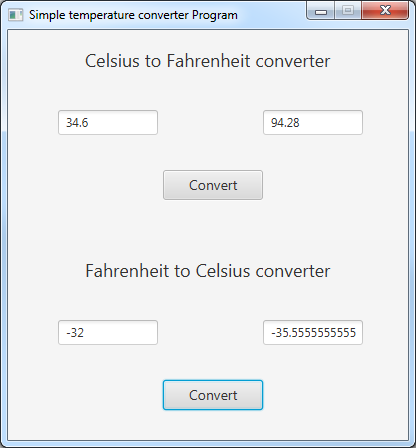
content = List(subScene1, textField1, textField2, button1, subScene2, textField3, textField4, button2)

}

}

}

Opis działania programu:



Program składa się z dwóch obiektów typu „Label” (etykieta), czterech obiektów typu „TextField” (pole tekstowe) oraz dwóch obiektów typu „Button” (przycisk). Aby przekonwertować temperaturę z jednej skali do drugiej należy wpisać w pole tekstowe znajdujące się po lewej stronie wartość temperatury w odpowiedniej skali, a następnie nacisnąć przycisk „Convert”. W przypadku wpisania ciągu znaków, który nie jest liczbą, stan pola tekstowego po prawej stronie nie ulegnie zmianie, jednakże wpisanie liczby ze znakiem „-„ nie zaburza poprawnego konwertowania. Dodatkowo program posiada zabezpieczenie uniemożliwiające edytowanie zawartości pól tekstowych znajdujących się po prawej stronie. W celu poprawnego wyśrodkowania tekstów etykiet umieszczono je wewnątrz obiektów typu „SubScene”. Okienko programu konwertera nie może zmienić sowich rozmiarów.