Day 1 - Concepts

1. Variables

```scala
// immutable variable
val numberOfDaysInWeek: Int = 7
//numberOfDaysInWeek = 8 // invalid

// mutable variable
var numberOfHoursInDay: Int = 24
numberOfHoursInDay = 25 // valid
println(s"Days in week: $numberOfDaysInWeek, Hours in day: $numberOfHoursInDay")

// lazy evaluation
lazy val totalHoursInWeek: Int = numberOfDaysInWeek * numberOfHoursInDay
println(s"Total hours in week: $totalHoursInWeek")

// variable type inference
val name = "Scala" // type inferred as String
```

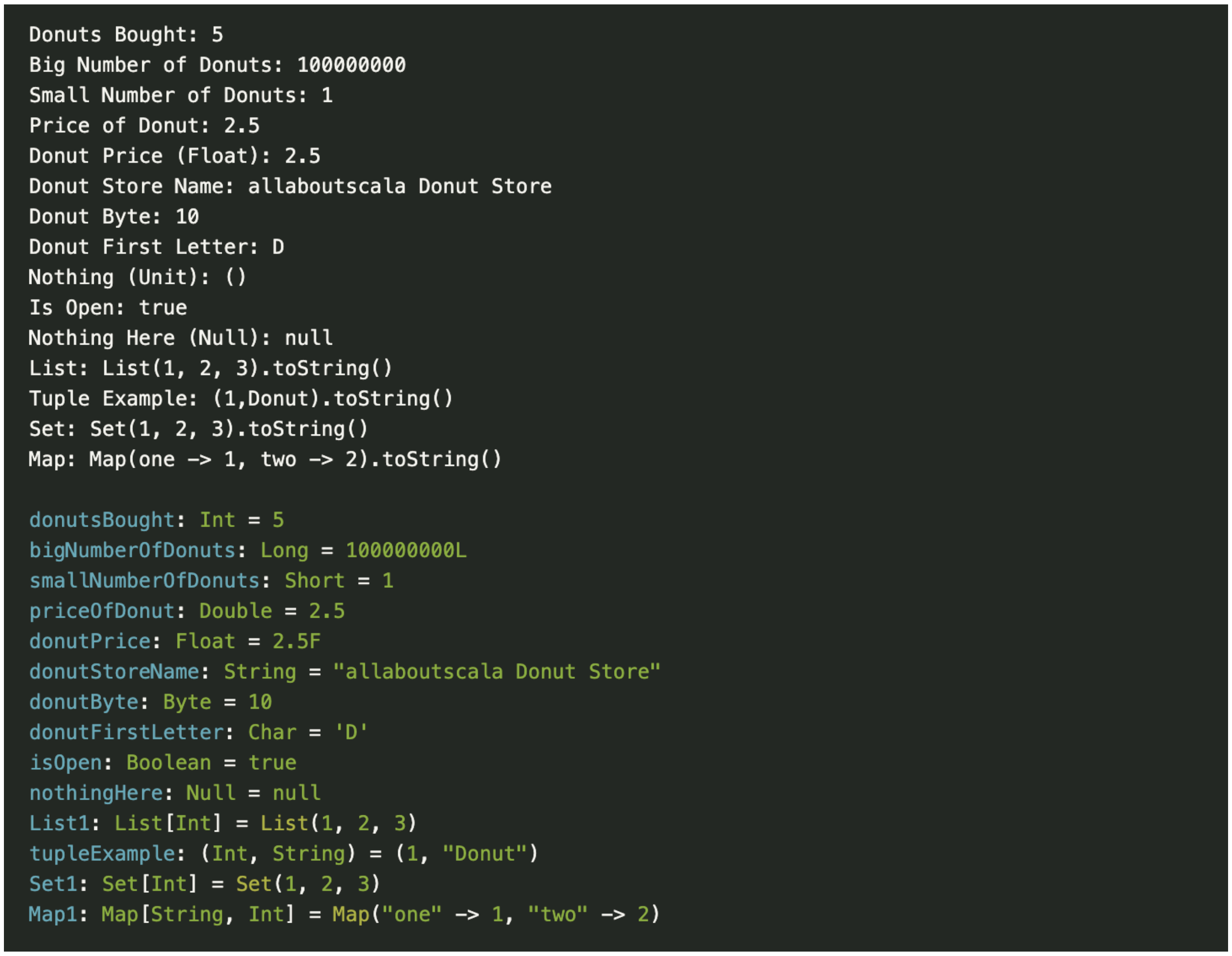
Output

```
Days in week: 7, Hours in day: 25
Total hours in week: 175

numberOfDaysInWeek: Int = 7
numberOfHoursInDay: Int = 25
totalHoursInWeek: Int = <lazy>
name: String = "Scala"
```

2. Data Types

```scala
// Data Types and Variables in Scala
val donutsBought: Int = 5
val bigNumberOfDonuts: Long = 100000000L
val smallNumberOfDonuts: Short = 1
val priceOfDonut: Double = 2.50
val donutPrice: Float = 2.50f
val donutStoreName: String = "allaboutscala Donut Store"
val donutByte: Byte = 0xa
val donutFirstLetter: Char = 'D'
val nothing: Unit = ()
val isOpen: Boolean = true
val nothingHere: Null = null
val List = List(1, 2, 3)
val tupleExample: (Int, String) = (1, "Donut")
val Set = Set(1, 2, 3)
val Map = Map("one" -> 1, "two" -> 2)

println(s"Donuts Bought: $donutsBought")
println(s"Big Number of Donuts: $bigNumberOfDonuts")
println(s"Small Number of Donuts: $smallNumberOfDonuts")
println(s"Price of Donut: $priceOfDonut")
println(s"Donut Price (Float): $donutPrice")
println(s"Donut Store Name: $donutStoreName")
println(s"Donut Byte: $donutByte")
println(s"Donut First Letter: $donutFirstLetter")
println(s"Nothing (Unit): $nothing")
println(s"Is Open: $isOpen")
println(s"Nothing Here (Null): $nothingHere")
println(s"List: $List")
println(s"Tuple Example: $tupleExample")
println(s"Set: $Set")
println(s"Map: $Map")
```

Output



```
Donuts Bought: 5
Big Number of Donuts: 100000000
Small Number of Donuts: 1
Price of Donut: 2.5
Donut Price (Float): 2.5
Donut Store Name: allaboutscala Donut Store
Donut Byte: 10
Donut First Letter: D
Nothing (Unit): ()
Is Open: true
Nothing Here (Null): null
List: List(1, 2, 3).toString()
Tuple Example: (1,Donut).toString()
Set: Set(1, 2, 3).toString()
Map: Map(one -> 1, two -> 2).toString()

donutsBought: Int = 5
bigNumberOfDonuts: Long = 100000000L
smallNumberOfDonuts: Short = 1
priceOfDonut: Double = 2.5
donutPrice: Float = 2.5F
donutStoreName: String = "allaboutscala Donut Store"
donutByte: Byte = 10
donutFirstLetter: Char = 'D'
isOpen: Boolean = true
nothingHere: Null = null
List1: List[Int] = List(1, 2, 3)
tupleExample: (Int, String) = (1, "Donut")
Set1: Set[Int] = Set(1, 2, 3)
Map1: Map[String, Int] = Map("one" -> 1, "two" -> 2)
```

3. Decision Making Statements

```
//decision making statements

val age = 25
if(age < 18){
   println("Hello Kiddy!")
}
else if(age >= 18 && age < 65){
   println("Hello Adult!")
}
else{
   println("Hello Senior Citizen!")
}

val day = "Saturday"
val daytype = if (day == "Saturday" || day == "Sunday") {
   "Weekend"
} else {
   "Weekday"
} //if as expression

println(s"$day is a $daytype.")
```

Output:

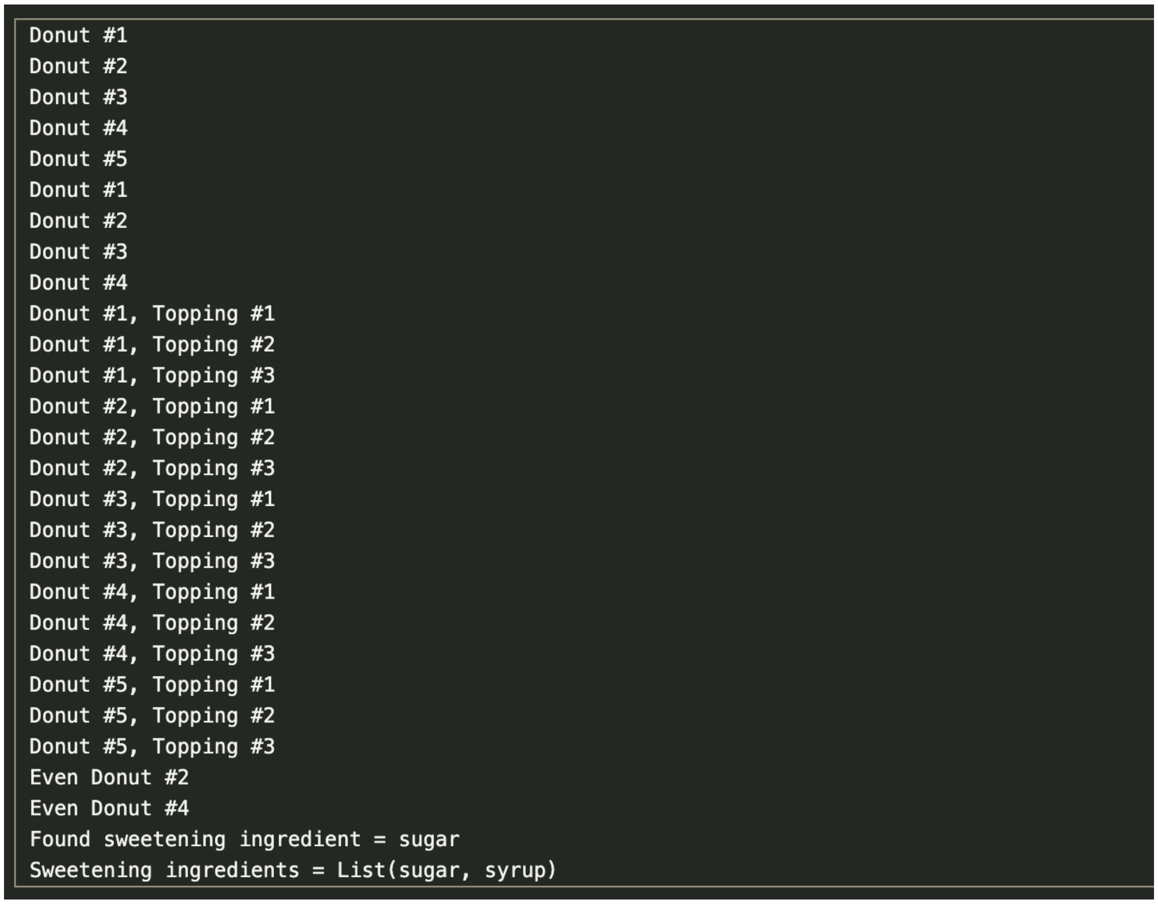
```
Hello Adult!
Saturday is a Weekend.

age: Int = 25
day: String = "Saturday"
daytype: String = "Weekend"
```

4. Control Structures

1. For comprehension

```
// for comprehension

for(i <- 1 to 5){
   println(s"Donut #$i") //prints including 5
}

for(i <- 1 until 5){
   println(s"Donut #$i") //prints excluding 5
}

for(i <- 1 to 5; j <- 1 to 3){
   println(s"Donut #$i, Topping #$j") // nested loops
}

for(i <- 1 to 5 if i % 2 == 0){
   println(s"Even Donut #$i") // prints only even numbers
}

val donutIngredients: List[String] = List("flour", "sugar", "egg yolks",
"syrup", "flavouring") // List of ingredients
for(ingredient <- donutIngredients if ingredient == "sugar"){
 println(s"Found sweetening ingredient = $ingredient") // loop to iterate
through list and filter only the ingredient "sugar"
}

val sweeteningIngredients = for (
 ingredient <- donutIngredients
 if (ingredient == "sugar" || ingredient == "syrup") // using or condition to
filter multiple ingredients
) yield ingredient // yield to create a new collection with the filtered
ingredients
println(s"Sweetening ingredients = $sweeteningIngredients")
```

Output:



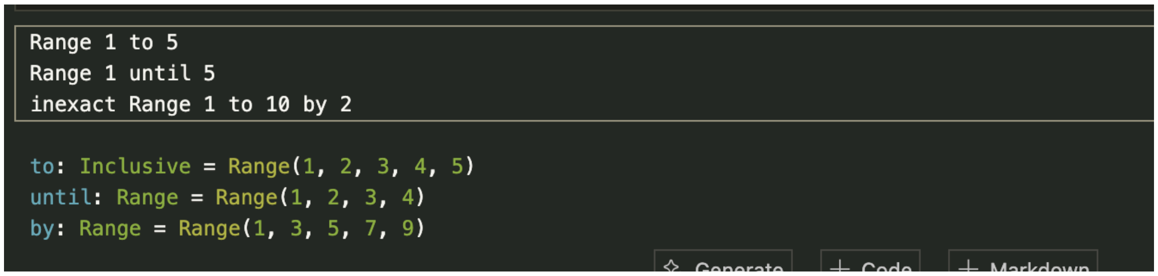
```
Donut #1
Donut #2
Donut #3
Donut #4
Donut #5
Donut #1
Donut #2
Donut #3
Donut #4
Donut #1, Topping #1
Donut #1, Topping #2
Donut #1, Topping #3
Donut #2, Topping #1
Donut #2, Topping #2
Donut #2, Topping #3
Donut #3, Topping #1
Donut #3, Topping #2
Donut #3, Topping #3
Donut #4, Topping #1
Donut #4, Topping #2
Donut #4, Topping #3
Donut #5, Topping #1
Donut #5, Topping #2
Donut #5, Topping #3
Even Donut #2
Even Donut #4
Found sweetening ingredient = sugar
Sweetening ingredients = List(sugar, syrup)
```

2. Range

```
val to = 1 to 5 // inclusive range
val until = 1 until 5 // exclusive range
val by = 1 to 10 by 2 // range with step
```

```
println(to)     // Output: Range 1 to 5
println(until) // Output: Range 1 until 5
println(by)     // Output: Range 1 to 10 by 2
```

Output:

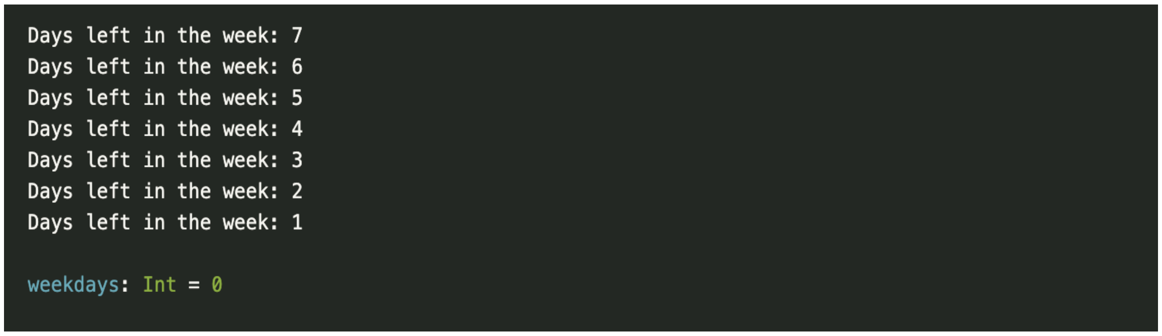


```
Range 1 to 5
Range 1 until 5
inexact Range 1 to 10 by 2

to: Inclusive = Range(1, 2, 3, 4, 5)
until: Range = Range(1, 2, 3, 4)
by: Range = Range(1, 3, 5, 7, 9)
```

3. While loop

```
var weekdays = 7

while (weekdays > 0) {
   println(s"Days left in the week: $weekdays")
   weekdays -= 1
}
```

Output:

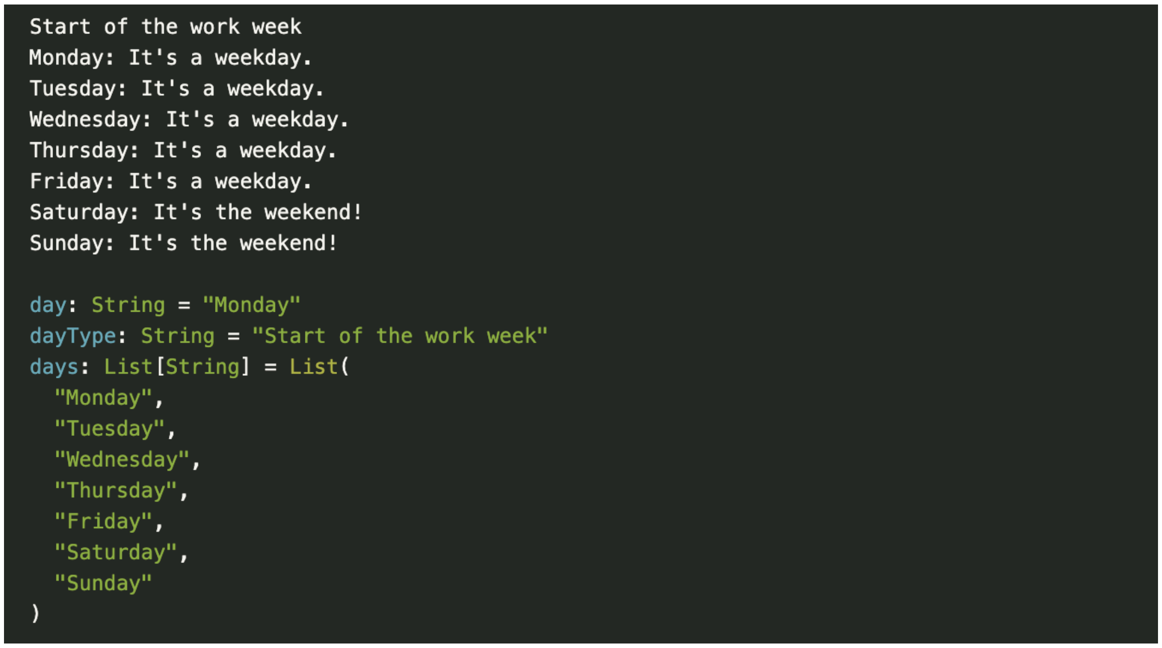


```
Days left in the week: 7
Days left in the week: 6
Days left in the week: 5
Days left in the week: 4
Days left in the week: 3
Days left in the week: 2
Days left in the week: 1

weekdays: Int = 0
```

4. Pattern Matching

```
val day = "Monday"

val dayType = day match {
   case "Monday" => "Start of the work week"
   case "Friday" => "End of the work week"
   case "Saturday" | "Sunday" => "Weekend"
   case _ => "Midweek day"
}

println(dayType)

val days: List[String] = List("Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday")

// Iterate through the list and print whether it's a weekend or a weekday
days.foreach { day =>
   day match {
       case "Saturday" | "Sunday" => println(s"$day: It's the weekend!")
       case _ => println(s"$day: It's a weekday.")
   }
}
```

Output:



```
Start of the work week
Monday: It's a weekday.
Tuesday: It's a weekday.
Wednesday: It's a weekday.
Thursday: It's a weekday.
Friday: It's a weekday.
Saturday: It's the weekend!
Sunday: It's the weekend!

day: String = "Monday"
dayType: String = "Start of the work week"
days: List[String] = List(
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
)
```

5. Arrays

```
// arrays


val donuts: Array[String] = Array("Glazed", "Chocolate", "Sprinkles")
```

```scala
// for loop for iteration
for (donut <- donuts) {
   println(donut)
}

// for each loop for iteration
donuts.foreach(donut => println(donut))

// using indices
for (i <- donuts.indices) {
   println(s"Donut at index $i is ${donuts(i)}")
}

println("Array length: " + donuts.length) //length of the array
println("First donut: " + donuts(0)) //accessing first element
println("First using head: " + donuts.head) //accessing first element using head
println("Using last: " + donuts.last) //accessing last element using last
println("Using tail: " + donuts.tail.mkString(", ")) //accessing all elements except first using tail
println("Using init: " + donuts.init.mkString(", ")) //accessing all elements except last using init
println("Reversed array: " + donuts.reverse.mkString(", ")) //reversing the array
println("Sorted array: " + donuts.sorted.mkString(", ")) //sorting the array
println("Contains 'Chocolate': " + donuts.contains("Chocolate")) //checking if array contains an element
println("Index of 'Sprinkles': " + donuts.indexOf("Sprinkles")) //finding index of an element

// new keyword
val moreDonuts: Array[String] = new Array[String](3)

println(moreDonuts.mkString(", "))

val countFlavouredDonuts: Array[Int] = new Array[Int](5)
println(countFlavouredDonuts.mkString(", "))
```

Output:

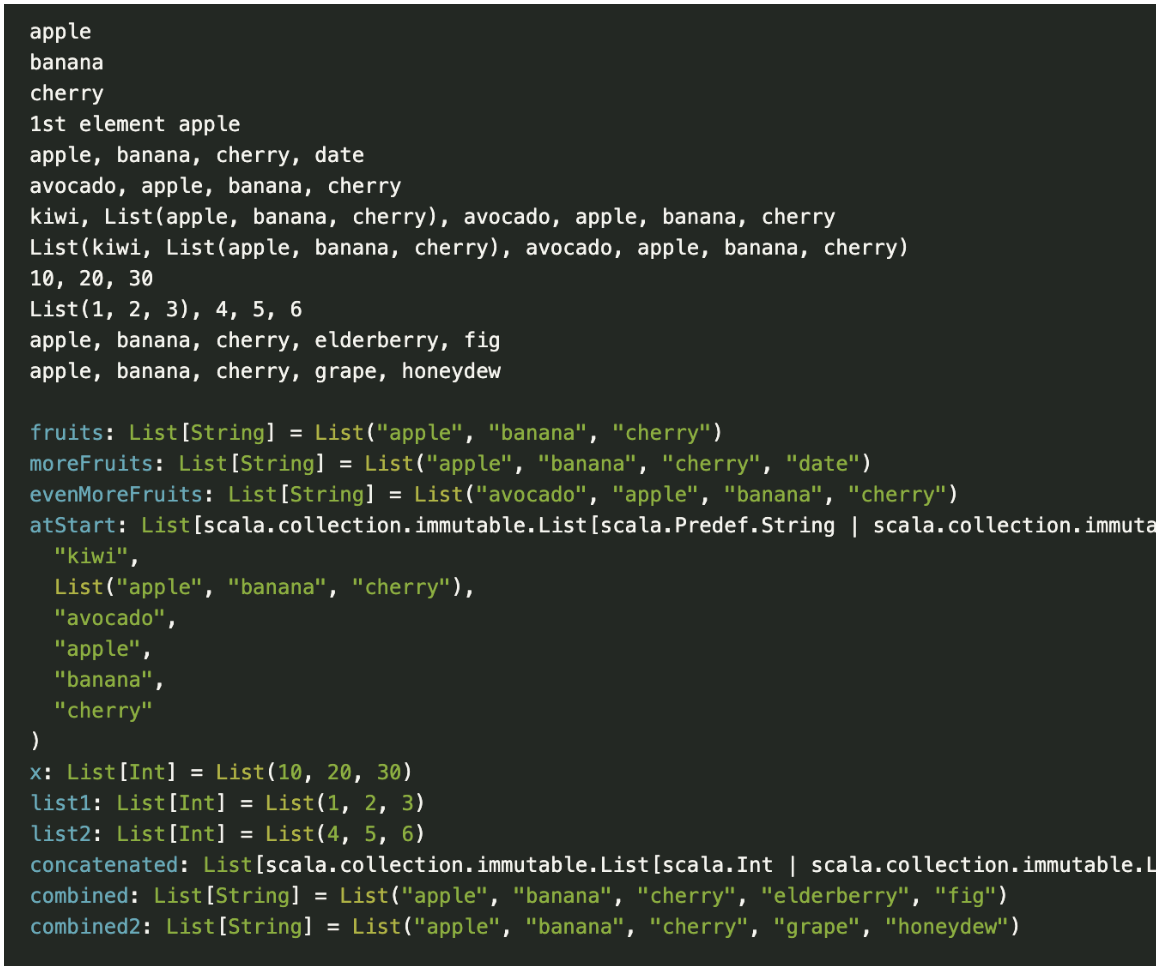
```
Glazed
Chocolate
Sprinkles
Glazed
Chocolate
Sprinkles
Donut at index 0 is Glazed
Donut at index 1 is Chocolate
Donut at index 2 is Sprinkles
Array length: 3
First donut: Glazed
First using head: Glazed
Using last: Sprinkles
Using tail: Chocolate, Sprinkles
Using init: Glazed, Chocolate
Reversed array: Sprinkles, Chocolate, Glazed
Sorted array: Chocolate, Glazed, Sprinkles
Contains 'Chocolate': true
Index of 'Sprinkles': 2
null, null, null
0, 0, 0, 0, 0

donuts: Array[String] = Array("Glazed", "Chocolate", "Sprinkles")
moreDonuts: Array[String] = Array(null, null, null)
countFlavouredDonuts: Array[Int] = Array(0, 0, 0, 0, 0)
```

5. Collections

1. List

```scala
val fruits: List[String] = List("apple", "banana", "cherry")
fruits.foreach(println)

println(s"1st element ${fruits(0)}") // access by index

//append at end
val moreFruits = fruits :+ "date" // using :+ operator
println(moreFruits.mkString(", "))

//append at beginning
val evenMoreFruits = "avocado" +: fruits // using +: operator
println(evenMoreFruits.mkString(", "))

val atStart = "kiwi" +: fruits +: evenMoreFruits // multiple prepends
println(atStart.mkString(", "))
println(atStart)

val x = 10 :: 20 :: 30 :: Nil // constructing a List using ::
println(x.mkString(", "))

// List concatenation
val list1 = List(1, 2, 3)
val list2 = List(4, 5, 6)
val concatenated = list1 :: list2 // using :: operator - this creates a List of
two Lists
println(concatenated.mkString(", "))

//adding two lists
val combined = fruits ++ List("elderberry", "fig") // using ++ operator - this
creates a single list of all elements
println(combined.mkString(", "))

//adding two lists using :::
val combined2 = fruits ::: List("grape", "honeydew") // using ::: operator -
this creates single list of all elements
println(combined2.mkString(", "))

//empty list
val emptyList: List[Int] = List.empty[Int]
println(s"Empty list size: ${emptyList.size}")  // prints 0

val emptyList2: List[String] = Nil
println(s"Empty list2 size: ${emptyList2.size}")  // prints 0
```

Output:



```
apple
banana
cherry
1st element apple
apple, banana, cherry, date
avocado, apple, banana, cherry
kiwi, List(apple, banana, cherry), avocado, apple, banana, cherry
List(kiwi, List(apple, banana, cherry), avocado, apple, banana, cherry)
10, 20, 30
List(1, 2, 3), 4, 5, 6
apple, banana, cherry, elderberry, fig
apple, banana, cherry, grape, honeydew

fruits: List[String] = List("apple", "banana", "cherry")
moreFruits: List[String] = List("apple", "banana", "cherry", "date")
evenMoreFruits: List[String] = List("avocado", "apple", "banana", "cherry")
atStart: List[scala.collection.immutable.List[scala.Predef.String | scala.collection.immuta
  "kiwi",
  List("apple", "banana", "cherry"),
  "avocado",
  "apple",
  "banana",
  "cherry"
)
x: List[Int] = List(10, 20, 30)
list1: List[Int] = List(1, 2, 3)
list2: List[Int] = List(4, 5, 6)
concatenated: List[scala.collection.immutable.List[scala.Int | scala.collection.immutable.L
combined: List[String] = List("apple", "banana", "cherry", "elderberry", "fig")
combined2: List[String] = List("apple", "banana", "cherry", "grape", "honeydew")
```

2. Map

```
//initialize using tuple syntax
val initialCalories = Map(
   ("Glazed", 200),
   ("Sugar", 250),
   ("Chocolate", 300)
)
println(initialCalories("Chocolate")) // 300

var donutCalories: Map[String, Int] = Map(
   "Glazed" -> 200,
   "Sugar" -> 250,
   "Chocolate" -> 300
)

println(donutCalories("Sugar")) // 250
```

```
// accessing using get method
println(donutCalories.get("Vanilla")) // None
println(donutCalories.getOrElse("Vanilla", 0)) // 0
donutCalories += ("Vanilla" -> 220) // adding new key-value pair at end of map
println(donutCalories) // Map(Glazed -> 200, Sugar -> 250,
// Chocolate -> 300, Vanilla -> 220)
donutCalories -= "Glazed" // removing key-value pair from map
println(donutCalories) // Map(Sugar -> 250, Chocolate -> 300, Vanilla -> 220)

donutCalories = donutCalories.updated("Chocolate", 320) // updating value for a
key
println(donutCalories) // Map(Sugar -> 250, Chocolate -> 320, Vanilla -> 220)

// remove multiple key-value pairs
donutCalories = donutCalories -- List("Sugar", "Vanilla")
println(donutCalories) // Map(Chocolate -> 320)



// merge two maps using ++=
val moreDonutsMap = Map(
   "Boston Cream" -> 350,
   "Cinnamon" -> 275
)

val lotMore = donutCalories ++ moreDonutsMap
println(lotMore) // Map(Sugar -> 250, Chocolate -> 300, Vanilla -> 220, Boston
Cream -> 350, Cinnamon -> 275)
```

Output:

```
300
250
None
0
Map(Glazed -> 200, Sugar -> 250, Chocolate -> 300, Vanilla -> 220)
Map(Sugar -> 250, Chocolate -> 300, Vanilla -> 220)
Map(Sugar -> 250, Chocolate -> 320, Vanilla -> 220)
Map(Chocolate -> 320)
Map(Chocolate -> 320, Boston Cream -> 350, Cinnamon -> 275)

initialCalories: Map[String, Int] = Map(
  "Glazed" -> 200,
  "Sugar" -> 250,
  "Chocolate" -> 300
)
donutCalories: Map[String, Int] = Map("Chocolate" -> 320)
moreDonutsMap: Map[String, Int] = Map("Boston Cream" -> 350, "Cinnamon" -> 275)
lotMore: Map[String, Int] = Map(
  "Chocolate" -> 320,
  "Boston Cream" -> 350,
  "Cinnamon" -> 275
)
```

3. Set

```
var donuts: Set[String] = Set("Glazed", "Sugar", "Chocolate")
println(donuts) // Set(Glazed, Sugar, Chocolate)

// elements existence
println(donuts.contains("Sugar")) // true
println(donuts("Vanilla")) // false
println(donuts.exists(d => d.startsWith("C"))) // true



// adding elements
donuts += "Vanilla"
println(donuts) // Set(Glazed, Sugar, Chocolate, Vanilla)
donuts = donuts + "Strawberry"
println(donuts) // Set(Glazed, Sugar, Chocolate, Vanilla, Strawberry)
donuts = donuts ++ Set("Blueberry", "Lemon")
println(donuts) // Set(Glazed, Sugar, Chocolate, Vanilla, Strawberry,
// Blueberry, Lemon)
// removing elements
donuts -= "Sugar"
println(donuts) // Set(Glazed, Chocolate, Vanilla, Strawberry, Blueberry,
Lemon)
donuts = donuts - "Lemon"
```

```scala
println(donuts) // Set(Glazed, Chocolate, Vanilla, Strawberry, Blueberry)
donuts = donuts -- Set("Glazed", "Vanilla")
println(donuts) // Set(Chocolate, Strawberry, Blueberry)
```

Output:

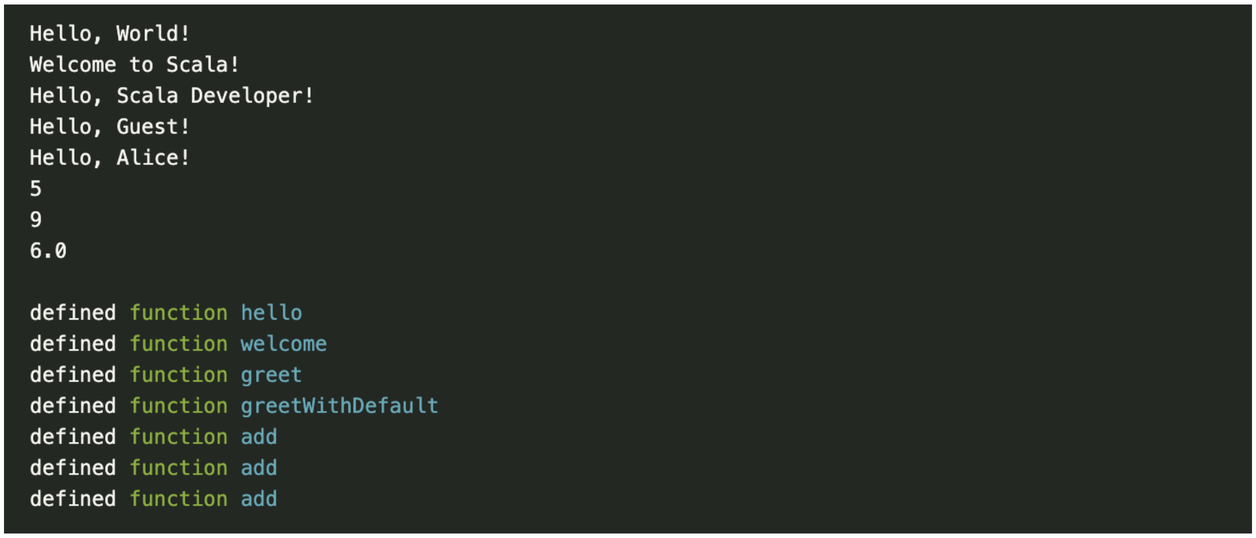
```
Set(Glazed, Sugar, Chocolate)
true
false
true
Set(Glazed, Sugar, Chocolate, Vanilla)
HashSet(Vanilla, Glazed, Chocolate, Sugar, Strawberry)
HashSet(Lemon, Vanilla, Glazed, Chocolate, Blueberry, Sugar, Strawberry)
HashSet(Strawberry, Lemon, Vanilla, Glazed, Chocolate, Blueberry)
HashSet(Strawberry, Vanilla, Glazed, Chocolate, Blueberry)
HashSet(Strawberry, Chocolate, Blueberry)

donuts: Set[String] = HashSet("Strawberry", "Chocolate", "Blueberry")
```

6. Functions

```scala
// function without return type
def hello(): Unit = {
   println("Hello, World!")
}
hello() // Output: Hello, World!

// function with string return type

def welcome(): String = {
   "Welcome to Scala!"
}

println(welcome()) // Output: Welcome to Scala!

// function with parameters and return type

def greet(name: String): String = {
   s"Hello, $name!"
}

println(greet("Scala Developer")) // Output: Hello, Scala Developer!
```

```scala
// function with default parameter value
def greetWithDefault(name: String = "Guest"): String = {
   s"Hello, $name!"
}

println(greetWithDefault()) // Output: Hello, Guest!
println(greetWithDefault("Alice")) // Output: Hello, Alice!

//function Overloading
def add(a: Int, b: Int): Int = {
   a + b
}
def add(a: Int, b: Int, c: Int): Int = {
   a + b + c
}
def add(a: Double, b: Double): Double = {
   a + b
}
println(add(2, 3)) // Output: 5
println(add(2, 3, 4)) // Output: 9
```

Output:



```
Hello, World!
Welcome to Scala!
Hello, Scala Developer!
Hello, Guest!
Hello, Alice!
5
9
6.0

defined function hello
defined function welcome
defined function greet
defined function greetWithDefault
defined function add
defined function add
defined function add
```