```
package com.talk.preperation

import java.time.{LocalDate, LocalDateTime, Period}

import com.talk.utils.Haversine

import scala.util.{Random, Success, Try}


object MainScala extends App {

  /**
   * 1. Scala Basics
   */

  /**
   * 1.1 OOP
   */
  class Person(name: String, bornOn: LocalDate) {
    def getName: String = name
  }

  class Student(name: String, bornOn: LocalDate) extends Person(name, bornOn) {
    override def toString: String = s"My name is ${name}, age ${Period.between(
bornOn, LocalDate.now()).getYears}"
  }

  class Professor(name: String, bornOn: LocalDate) extends Person(name, bornOn) {
    override def toString: String = s"Professor ${name}"
  }

  var jernej = new Student("Jernej", LocalDate.parse("1985-01-01"))
  var me = new Student("Oto", LocalDate.parse("1987-01-01"))
  var miha = new Professor("Miha", LocalDate.parse("1991-01-01"))

  var people: List[Person] = List(jernej, me, miha)
  people = people :+ new Professor("Pehta", LocalDate.parse("1930-01-01"))
  people.foreach { person => println(person) }

  /**
   * 1.2 JavaInterop
   */
  println(LocalDateTime.now())
  println(System.getProperty("java.version"))

  /**
   * 1.3 Collections
   * - Immutable
   * - Mutable
   * - Java compatible (conversions)
   */

  println("    Collections" + " ~" * 10)
  var letters = Array("A", "B", "C", "D")
  letters.appended("E", "F", "G")
  letters.foreach(println)

  var numbers: List[Int] = List(1, 2, 3, 4, 42)
```

```
  println(numbers.sorted.map(_ * 2).sum)

  var addressBook = Map[String, Person](("oto" -> me), "jernej" -> jernej, "oto"
 -> jernej)
  println(addressBook)
  println(addressBook.get("dodo"))

  /**
   * 1.4 Case classes
   * - Similar to new Java's "Record" type
   * - Companion object
   */
  println("    Case classes" + " ~" * 10)

  case class User(email: String, id: Int)

  case class Address(street: String)

  object Users {
    private[this] implicit val personToUser: Person => User = person => {
      val personNameToEmail: String = person.getName.toLowerCase.strip() + "@
gmail.com"
      User(personNameToEmail, new Random(1000).nextInt(42))
    }

    val ourUsers: List[User] = List(me, jernej, miha)

    def getUser(email: String): Option[User] =
      ourUsers.find(_.email == email)

    def getAddress(userOpt: Option[User]): Option[Address] = {
      userOpt match {
        case Some(user) =>
          if (user.email.startsWith("oto")) {
            Some(Address("Cankarjeva 1, 1000 Ljubljana"))
          } else None
        case _ => None
      }
    }
  }

  println(Users.getUser("oto@gmail.com"))

  Users.getUser("somebloke@gmail.com") match {
    case Some(user) =>
      println(s"Hello, ${user}")
    case None =>
      println("Noup... not here...")
  }

  // Composing
  val getUserAndAddress = Users.getUser _ andThen Users.getAddress
  println(getUserAndAddress("oto@gmail.com"))

  println(for {
    user <- Users.getUser("oto@gmail.com")
    address <- Users.getAddress(Some(user))
  } yield {
```

```
    (user, address)
  })

  /**
   * Bit more types
   * - Case class to ADT
   * - Distance
   */
  println("    A bit more types" + " ~" * 10)

  case class Coordinates(latitude: Double, longitude: Double)

  val maribor = Coordinates(46.554650, 15.645881)

  type Latitude = Double
  type Longitude = Double
  type CoordinatesT = (Latitude, Longitude)

  type Distance = Double
  type Meters = Distance
  val ljubljana: CoordinatesT = (46.056946, 14.505751)

  object Distance {
    def between[Point <: CoordinatesT](a: Point)(b: Point): Meters =
      Haversine.apply((a._2, a._1), (b._2, b._1))

    def toHome[Point <: CoordinatesT](current: Point)(implicit home: Point): Meters
 =
      between(current)(home)
  }

  implicit val coordinatesToCoordinatesT: Coordinates => CoordinatesT = c => (c.
latitude, c.longitude)

  println(Distance.between[CoordinatesT](ljubljana)(maribor))
  println(Distance.between[CoordinatesT](maribor)(ljubljana))

  println {
    Distance.between[CoordinatesT](maribor)(ljubljana) == Distance.between[
CoordinatesT](ljubljana)(maribor)
  }

  implicit val home: CoordinatesT = maribor
  type KiloMeters = Meters
  val distanceToHome: KiloMeters = Distance.toHome[CoordinatesT](ljubljana) /
1000.0
  println(f"$distanceToHome%1.2f km")

  /**
   * Intersection types, union types, type lambdas, match types, dependent function
types
   */


  /**
   * Errors and types
   */
```

```
  object Football {
    def isChampion(name: String): Try[Boolean] = {
      if (name.toLowerCase().startsWith("maribor"))
        Success(true)
      else
        throw new Exception("Boing     - Could not happen!")
    }

    def championBad(name: String): Boolean = {
      if (name.toLowerCase().startsWith("maribor"))
        true
      else
        throw new Exception("Crash       ")
    }
  }

  // Show how crash can happen despite right type
  // val isMBOLD: Boolean = Football.championBad("x")
  // println(isMBOLD)

  // Leveraging types
  val isMB: Try[Boolean] = Football.isChampion("Maribor")
  println(isMB.map(_ => "ŠAMPION"))

  println {
    Football.isChampion("Maribor").map(_ => "Maribor Šampion").getOrElse("not")
  }
}
```