# Programowanie współbieżne
## Lista 5

1. Na wykładzie 4 (str. 42-43) był przedstawiony program producent/konsument z ograniczonym buforem cyklicznym.
   a) Przepisz ten program, wykorzystując zamiast klasy BoundedBuffer klasę biblioteczną java.util.concurrent.ArrayBlockingQueue.
   b) W programie z podpunktu a) utwórz kilka producentów i konsumentów. Nadaj im unikatowe nazwy, np. Producer1, Consumer1 itd. W jednym z testów utwórz dwa producenty i trzy konsumenty. Dlaczego program się nie kończy?
   c) Z programu w podpunkcie b) usuń definicje klas Producer i Consumer. Wykorzystaj ExecutionContext do wykonywania odpowiadających im zadań. W jednym z testów utwórz dwa producenty i trzy konsumenty. Dlaczego program się kończy?

2. Zdefiniuj funkcję
   ```
   def pairFut[A, B] (fut1: Future[A], fut2: Future[B]): Future[(A, B)] = ???
   ```
   a) Wykorzystaj metodę zip (wykład 5, str. 38)
   b) Wykorzystaj for (wykład 5, str. 39)

3. Do typu Future[T] dodaj metodę exists:
   ```
   def exists(p: T => Boolean): Future[Boolean] = ???
   ```
   Wynikowy obiekt Future ma zawierać wartość true wtedy i tylko wtedy, obliczenia obiektu oryginalnego kończą się pomyślnie i predykat p zwraca wartość true, w przeciwnym razie wynikowy obiekt Future ma zawierać wartość false. Wykorzystaj klasę implicytną i mechanizm niejawnych konwersji (wykład 5, str. 42).
   a) Wykorzystaj promesę
   b) Nie korzystaj z promesy (użyj map)

4. Należy policzyć liczbę słów w każdym pliku tekstowym zadanego folderu i wydrukować wynik w postaci par (nazwa pliku, liczba słów), posortowany rosnąco względem liczby słów. Możemy założyć dla uproszczenia, że słowa są oddzielone spacjami (wykorzystaj metodę split). Obliczenia należy przeprowadzać asynchronicznie. Program ma być napisany funkcyjnie.

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import scala.util.{Success, Failure}
import scala.io.Source

object WordCount {
    def main(args: Array[String]) {
       val path = "ścieżka do folderu; można ją oczywiście wczytać"
       val promiseOfFinalResult = Promise[Seq[(String, Int)]]

 //  Tu oblicz  promiseOfFinalResult

       promiseOfFinalResult.future onComplete {
          case Success(result) => result foreach println
          case Failure(t)          => t.printStackTrace
       }
       Thread.sleep(5000)
```

```
    }       // koniec metody main

  // Oblicza liczbę słów w każdym pliku z sekwencji wejściowej
  private def processFiles(fileNames: Seq[String]): Future[Seq[(String, Int)]] = ???
       // Wskazówka.  Wykorzystaj Future.sequence(futures)

// Oblicza liczbę słów w podanym pliku i zwraca parę: (nazwa pliku, liczba słów)
  private def processFile(fileName: String): Future[(String, Int)] = ???

// Zwraca sekwencję nazw plików  (w naszym przypadku Array[String])
  private def scanFiles(docRoot: String): Future[Seq[String]] =
     Future { new java.io.File(docRoot).list.map(docRoot + _) }
}
```

Wszystkie definicje umieść w pliku Lista5.scala.