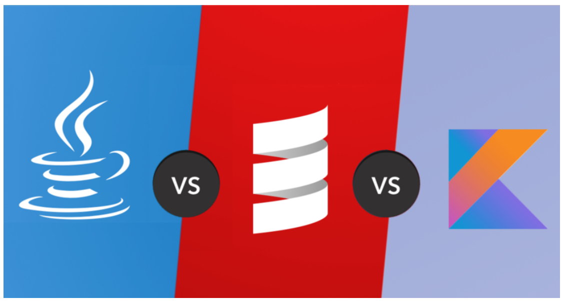
Project Report for COMP1815 - JVM Programming Languages

University Timetable Clash Detection System

Francesco Romano
001086733-3
`fr7846m`



FACULTY OF LIBERAL ARTS AND SCIENCES
COMPUTER SCIENCE BSC HONS

December 16, 2021

Contents

1 Programming Languages and Paradigms

This section will briefly introduce and critically compare the programming languages and paradigms taught in this module as well as highlight some of their strengths and weaknesses.

1.1 Objected Oriented Programming

One of the two programming paradigms that were present in this module is Object Oriented Paradigm (OOP), which is widely used as it makes the code easy to read, maintainable, and reusable. OOP relies on classes and objects: The first is a sort of blueprint that defines a series of properties and functions called methods that will define the behavior of the single instances of itself, which are called objects.(Wegner 1990)
OOP also requires the user to describe step by step how the state of the computer will change; this is called imperative programming. There are four principles of OOP (Rentsch 1982), that are:

- Inheritance: A class can "inherit" its methods and attributes from another class, avoiding duplicate code
- Encapsulation: Information about the objects can be hidden and only accessed through methods, hiding the data.
- Polymorphism: This occurs when many classes inherit from the same parent class, but have different implementations.
- Abstraction: It involves "generalising", which means hiding the logic and only showing the functionality of the classes.

The coursework required us to work with two languages that support the OOP paradigm, which are Kotlin and Java.

Kotlin is an open source multi-paradigm programming language that supports both OOP and functional programming. One of the strenghts of Kotlin is that it is fully compatible with Java, therefore it is possible to use Java classes in Kotlin and vice versa. (Mateus & Martinez 2019)
This was very beneficial as it was possible to share methods between the two languages as well as data structures and entire classes.
Furthermore, Kotlin has a much simpler syntax than Java, allowing to write the same functionalities using less code. This turned out to be helpful as it increased productivity and reduced the odds of typos.
Moreover, Kotlin is statically typed, meaning that the variables types must be explicitly declared, allowing full control over them.
Finally, one of the best features of Kotlin is lambda functions, that allowed the creation of simple anonymous functions to filter and retrieve data under particolar conditions (such as detecting clashes) without using nested loops or long series of if statements.
One of the weaknesses of this language is that although its syntax is simpler than Java, it still took some time to get used to it.

Java is a very well known high-level programming language when it comes to OOP.
It has many advantages that make it a very valid language for the creation of both simple projects and large scalable software. It includes automatic memory management, so the user is not required to handle the allocation of memory and garbage collection. It is extremely popuplar, as it runs of more than 10 billion devices as of today thanks to its portability, that allows it to function on any device capable of running a JVM. (Hugunin 1997)
It is also an extremely rich language with a large number of external libraries, very well-written documentation, and a big community and number of projects, so it is fairly easy to find resources online.
Despite its numerous upsides, Java has quite some disadvantages. First of all, its syntax is quite verbose and long, and this slows down the process of writing code.
For example, a semicolon is required at the end of each line, and this can be rather tedious when writing hundreds of lines of code.
Also, even if Java is considerably faster than interpreted languages, it is still not as fast as languages like C because it runs on the JVM, which slows its performance down.

Objected Oriented Programming has also some downsides, such as:

- Steep learning curve: Learning to create complex software: it needs the programmer to have enough knowledge about design patterns, classes, objects, and the programming language used.
- Difficulty of OOP principles: Writing code while taking into considerations every OOP principle is quite hard especially when it comes to inheritance and polymorphism. These are not very straightforward to implement and require good planning and programming skills.
- Coupling and Cohesion: It can become difficult to assign specific responsibilities to classes (coupling) and design them so that they do not depend on each other (cohesion).

In general, OOP has its advantages and disadvantages, but it is an extremely powerful paradigm that allows the creation of complex software that supports the implementation of new features, updates to existing ones, etc.
Also, the modularity of this programming paradigms allows different people to work on the same project at the same time, which explains the reason OOP is such a powerful tool for big companies that have many teams working on the same project.

1.2 Functional Programming

Functional programming (FP) is the other programming paradigm used during the development of the coursework. This way of coding does not require the creation of objects and classes, but emphasises on the creation of variables and functions. (Hughes 1989) Functional Programming, just like OOP, is based on a series of principles (Peyton Jones & Wadler 1993), that are:

- Immutability: Once a value is declared, it cannot be changed.
- Pure Functions: Given the same input, a function must always return the same output.
- No side-effects: A function can not rely on or modify parameters outside the function itself.
- Lazy Evaluation: It is an evaluation strategy used in FP that consists in waiting to evaluate an expression until needed. It reduces time complexity as expressions that are not related to the final result are not considered.
- Declarative Programming: It is necessary for programmers to express what they need from a particular function, instead of telling the program what to do step-by-step.
- It uses recursion rather than loops to iterate through collections of data.
- It supports high-order functions: functions that take or return functions as parameters.

The FP language used in this coursework is Scala, which just like Kotlin runs on the JVM and is highly compatible with Java. The peculiarity of Scala is that it is an object oriented language, but at the same time supports functional programming. This allows the creation of a sort of hybrid style that combines OOP with FP. (Odersky et al. 2004) As Odersky & Rompf (2014) affirms in his paper, "Scala's integration of functional and object-oriented concepts leads to a scalable language, in the sense that the same concepts work well for very small, as well as very large, programs." Also, this combination allows programmers to save many lines of code, as functional programming is more concise.

2 Developement

This section will be about the main phases of the development process of this project: Data generation, backend, frontend.

2.1 Data Generation

The mock data was generated using the website "www.mockaroo.com", which allows users to create personalised datasets using pre-defined data types, or even create new ones using formulas written in the programming language "Ruby". The format chosen for this project is "JSON", which is a highly-compatible data format that has the following characteristics (Bourhis et al. 2017):

- It us human-readable text to represent data.
- Data is grouped in pairs of keys and values separated by commas.
- Curly brackets hold objects, and square brackets hold arrays.
- It supports seven types of values: objects, arrays, strings, numbers, true, false, null.

This allowed the creation of a personalised format that included all the information that was needed: programmes, modules, and activities, and each one with their own attributes. (Figure 1)

2.2 Backend

The backend was written in two languages: Kotlin and Scala. The first one is responsible for the majority of the tasks, such as creating data classes and modifying them; the second one is only responsible for checking for clashes between activities.
There are four classes written in Kotlin, and they are: JsonHandler, DataFactory, GUICommands, and the main class.

JsonHandler is only responsible for retrieving and saving the data to and from the JSON file. The "readJSONFile" function parses the content of the JSON file into a string and it returns it; saveJsonFile, instead, takes a string as a parameters and saves it to "data.json". (Figure 2)

DataFactory is the class that has creates the data structure the holds all the information and it contains methods to filter the data and return what other classes need. This class uses a library called "Klaxon" to parse the JSON file (converted into a String by JsonHandler) into three data classes as well as turning the same classes back into a JSON String using the "Gson" library. The three data classes used to hold all the data are are "Programme", "Method", and "Activities". (Figure 3)
Some methods of DataFactory include functions to create, delete, or retrieve specified programmes, modules, and activities (Figure 4), and a function to detect clashes between activities.
In order to facilitate getting specific data, some functions were created to filter them, such

as methods to get an activity's module or programme, or even return all the activities in the same programme, year, and term, in order to check for clashes. (Figure 5)
Instead of using nested loops and if statements, a series of lambda functions were adopted, which include *filter, any, flatmap,* and *foreach.*

The clash detection function has two versions: one in Kotlin using an OOP approach, and one in Scala that uses a functional approach to the task.
The first one takes a programme, a year of study, and a term as parameters (e.g. Computer Science, year 1, term 1), and it uses a for loop in order to discover clashes between activities of the programme passed to the method. Before returning all the clashes, the duplicate ones are removed using the function removeDuplicateClashes, that will delete from the ArrayList of clashes that are present inversely (e.g. A clashes with B and B clashes with A). (Figure 6).
The clashes function in scala, instead, uses a functional approach to work out the clashes, without creating temporary lists and needing any sort of flow control. (Figure 7)

GUICommands is a class that interacts with the Java GUI, adding activities or removing them from the timetable as well as finding free slots to solve existing clashes. This class should have been written entirely in Java, but it was decided to use Kotlin in order to test the level of compatibility achievable between Java and Kotlin, which is surprisingly high.

The main class has no functionality and it only creates an instance of the GUI.

2.3 Frontend

The User Interface was developed entirely in Java and it contains three different windows: Timetable, AdminMenu, and ClashesWindow.
When the program is launched, a small window opens, which asks the user which language they want to use for clash detection. (Figure 8) Their choice will be saved in a variable and used when checking for clashes.

After the first window, the Timetable will show up. Initially, the grid will be empty and no activity will be on the timetable (Figure 9), and it will be necessary to click on the "Menu" button to open the admin menu to select a programme to display.

The admin menu allows the user to add and remove programmes, modules, and activities by typing the name of them and choosing the attributes they want them to have, such as undergraduate or postgraduate, compulsory or optional, etc. From this panel it is also possible to fill the timetable by selecting the programme, the year, and the term, and pressing on the "View Programme" button. (Figure 10)

When pressed, the button will perform these operations in the background:

- Retrieve the selected programme's activities in the same term and year.
- Get the clashes that occur between those activities
- Show the activities and the clashes on the Timetable

The clashes will appear as red cells on the GUI (Figure 11), and, by clicking on the "Clashes" button, it is possible to open "ClashesWindow" (Figure 12), which will show every clash and a button to fix them. When pressed, the button "Fix Clash" will call a function of GUICommands to find a free timeslot in order to solve the clash and it will correctly place both activities on the timetable.

2.4 Reflection

This project provided a really positive experience. The coursework was quite interesting and challenging to work on as it required us to use three different languages and different approaches for solving the tasks provided.
Working in a team was beneficial as we were able to share ideas and also solve problems and overcome hurdles and solve bugs more easily.
My role within the team involved working mainly on the backend, which included creating the classes and methods in Kotlin and Scala. During the development of the coursework I worked with the other members in charge of creating the GUI in order to provide them with useful methods suitable to their needs.

The result is great as we managed not only to implement the required functionalities, but also to include an extra feature, which is the option to fix the present clashes. However, there are some flaws in the code that we were not able to take care of due to time constraints. First of all, we did not manage to include some unit testing, but relied on manual tests to experiment with the code. Also, it would have been nice to include more abstraction and encapsulation, as well as use inheritance in a better way to achieve neater and more compact code. Finally, if we had more time to develop the project we could have made the timetable interactive, letting the user drag and drop the activities on it in order to edit the timetable.

References

Bourhis, P., Reutter, J. L., Suárez, F. & Vrgoč, D. (2017), Json: data model, query languages and schema specification, *in* 'Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems', pp. 123–135.

Hughes, J. (1989), 'Why functional programming matters', *The computer journal* **32**(2), 98–107.

Hugunin, J. (1997), Python and java: The best of both worlds, *in* 'Proceedings of the 6th international Python conference', Vol. 9, Citeseer, pp. 2–18.

Mateus, B. G. & Martinez, M. (2019), 'An empirical study on quality of android applications written in kotlin language', *Empirical Software Engineering* **24**(6), 3356–3393.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E. & Zenger, M. (2004), 'An overview of the scala programming language'.

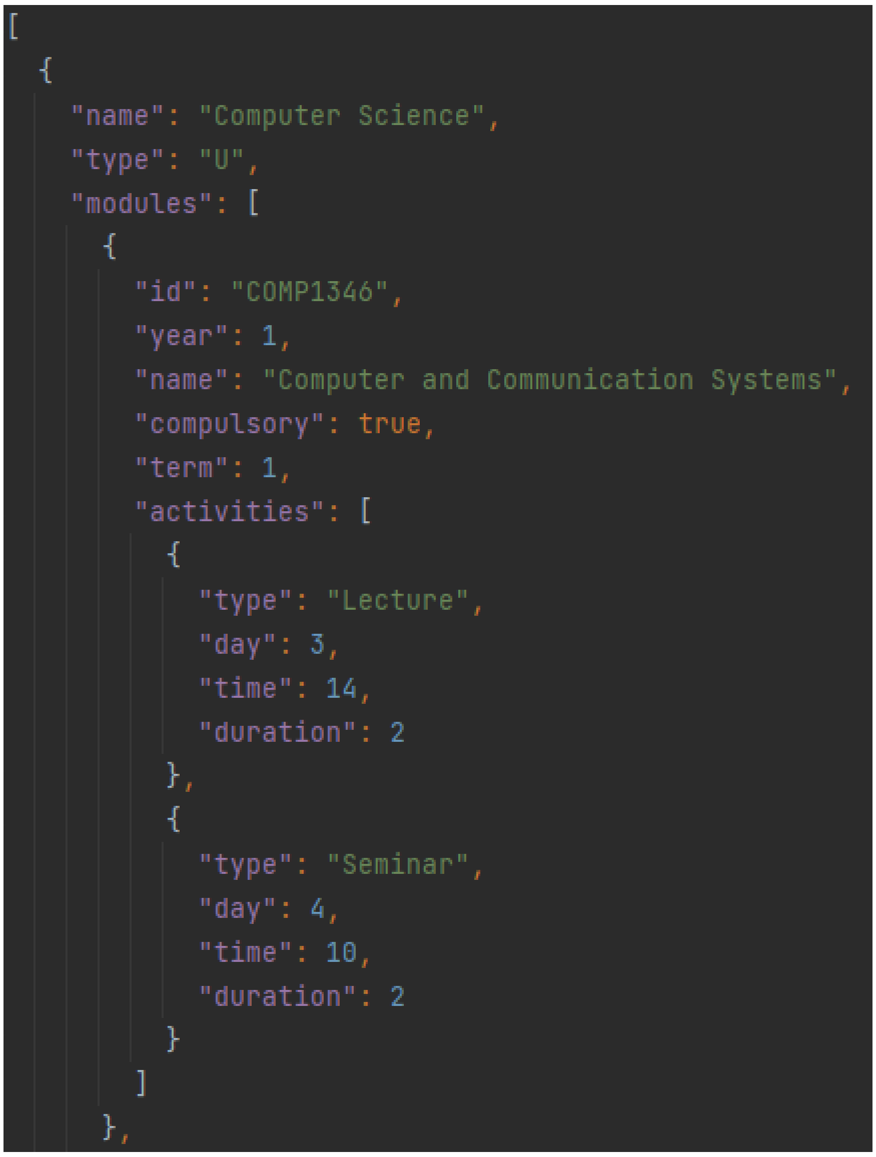
Odersky, M. & Rompf, T. (2014), 'Unifying functional and object-oriented programming with scala', *Communications of the ACM* **57**(4), 76–86.

Peyton Jones, S. L. & Wadler, P. (1993), Imperative functional programming, *in* 'Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages', pp. 71–84.

Rentsch, T. (1982), 'Object oriented programming', *ACM Sigplan Notices* **17**(9), 51–57.

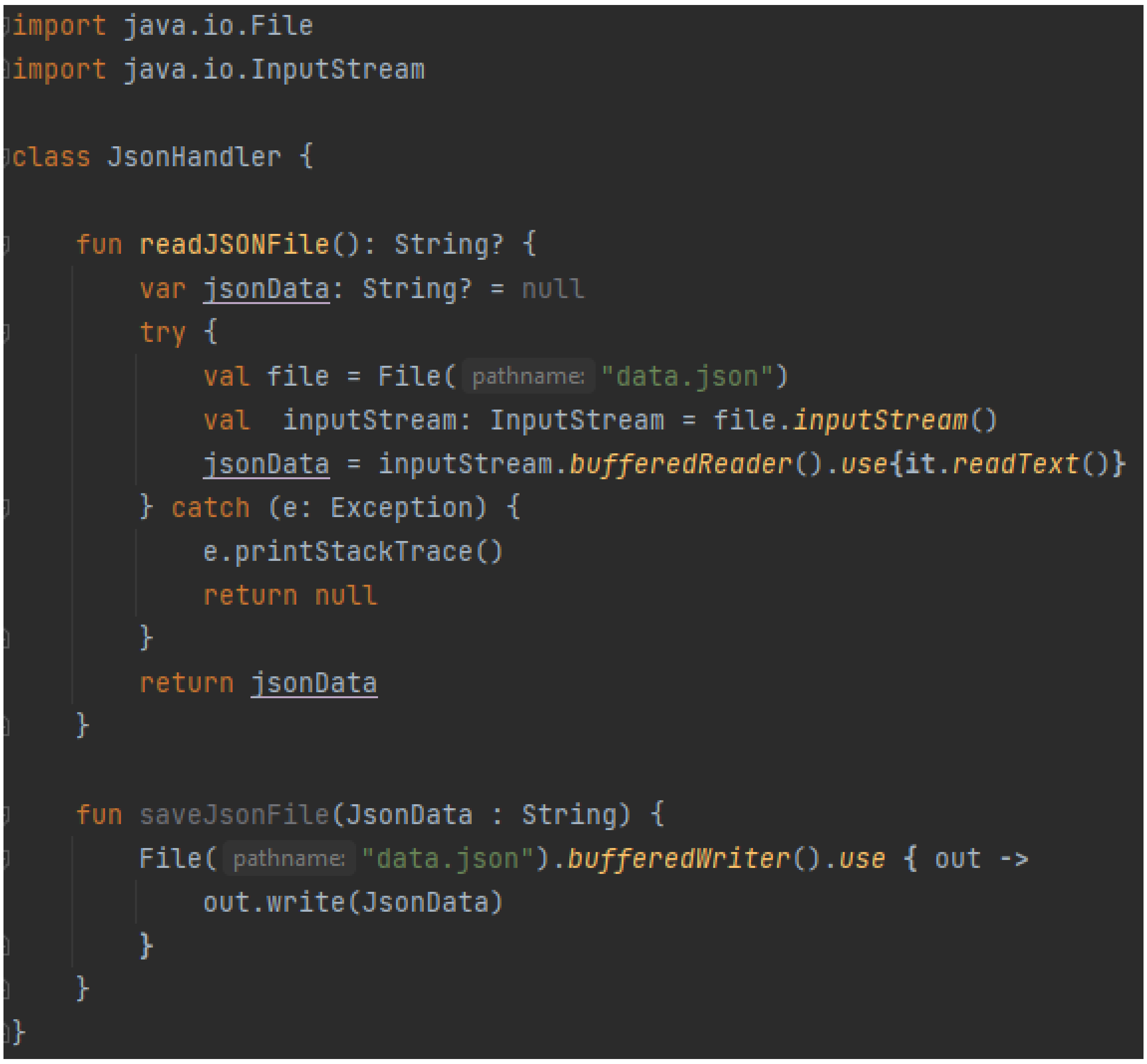
Wegner, P. (1990), 'Concepts and paradigms of object-oriented programming', *ACM Sigplan Oops Messenger* **1**(1), 7–87.

3 Appendix



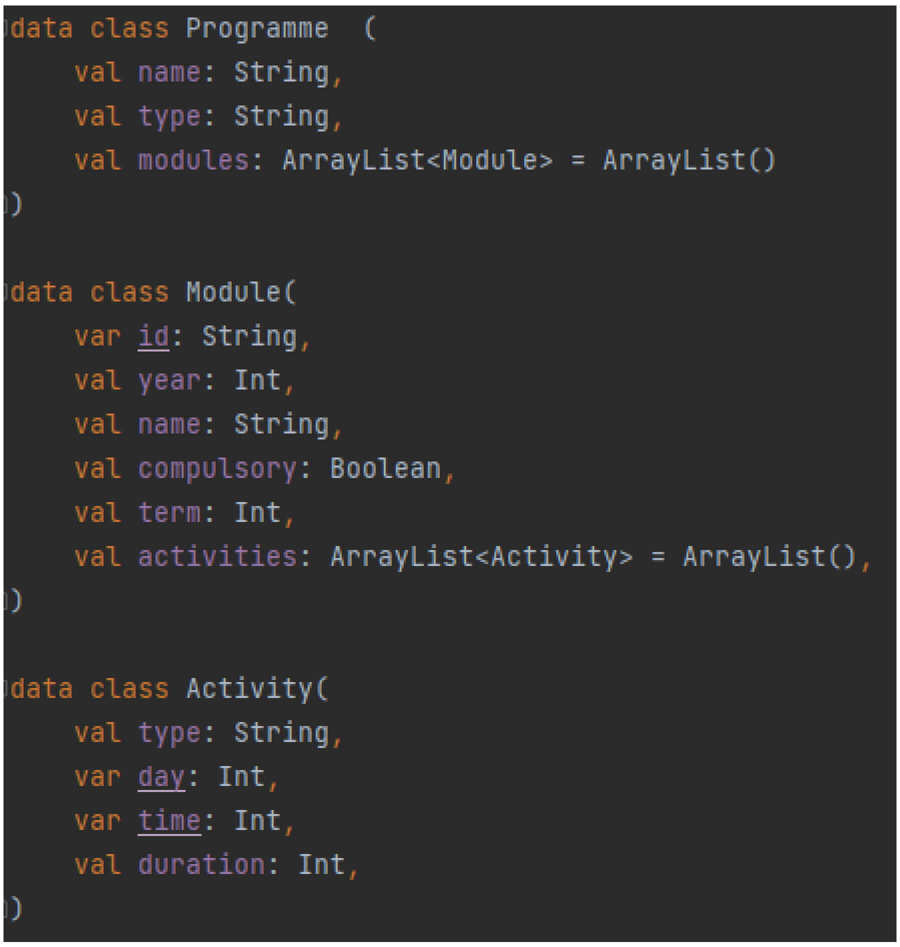
```
[
  {
    "name": "Computer Science",
    "type": "U",
    "modules": [
      {
        "id": "COMP1346",
        "year": 1,
        "name": "Computer and Communication Systems",
        "compulsory": true,
        "term": 1,
        "activities": [
          {
            "type": "Lecture",
            "day": 3,
            "time": 14,
            "duration": 2
          },
          {
            "type": "Seminar",
            "day": 4,
            "time": 10,
            "duration": 2
          }
        ]
      },
```

Figure 1: This image shows the structure of the JSON file. A programme contains an array of modules, which contain each an array of activities.
The schema used for the generation of the dataset used for this paper is available at: https://www.mockaroo.com/18552e10
File: data.json



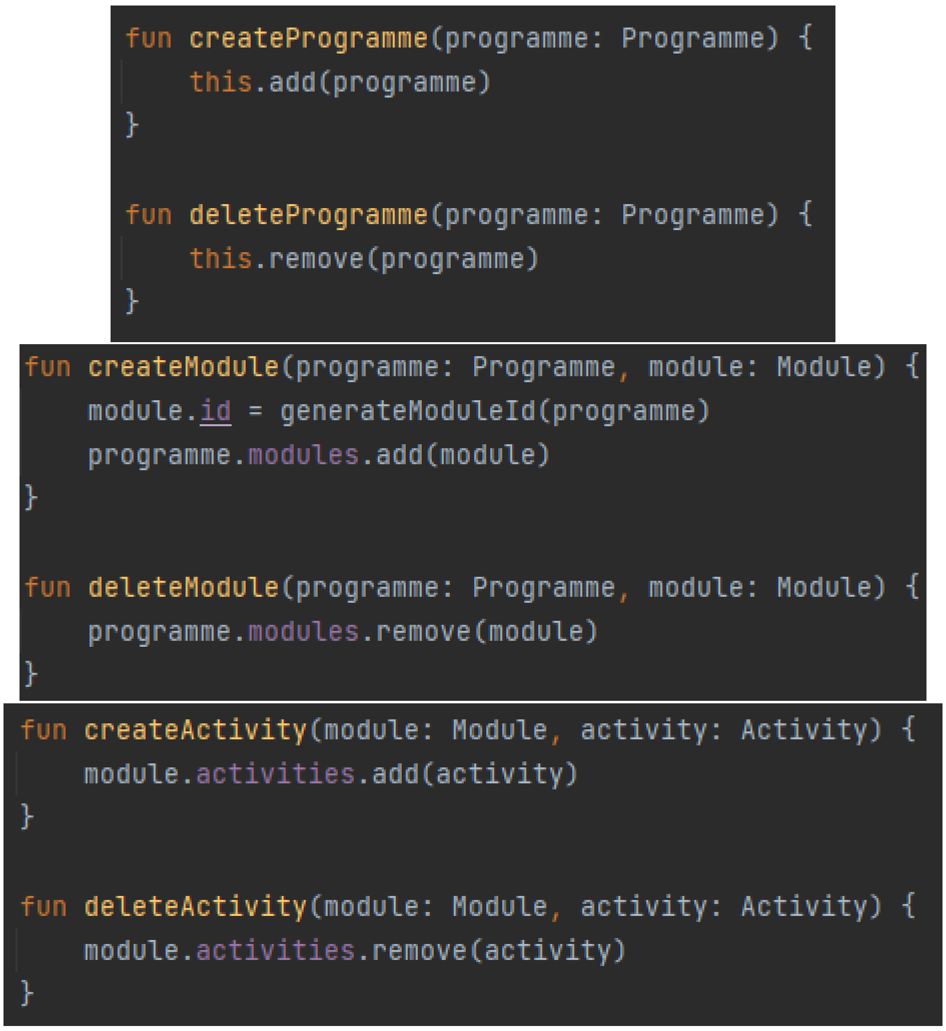
```kotlin
import java.io.File
import java.io.InputStream

class JsonHandler {

    fun readJSONFile(): String? {
        var jsonData: String? = null
        try {
            val file = File( pathname: "data.json")
            val  inputStream: InputStream = file.inputStream()
            jsonData = inputStream.bufferedReader().use{it.readText()}
        } catch (e: Exception) {
            e.printStackTrace()
            return null
        }
        return jsonData
    }

    fun saveJsonFile(JsonData : String) {
        File( pathname: "data.json").bufferedWriter().use { out ->
            out.write(JsonData)
        }
    }
}
```

Figure 2: JsonHandler class.File: JsonHandler.kt (Kotlin)



```kotlin
data class Programme  (
    val name: String,
    val type: String,
    val modules: ArrayList<Module> = ArrayList()
)

data class Module(
    var id: String,
    val year: Int,
    val name: String,
    val compulsory: Boolean,
    val term: Int,
    val activities: ArrayList<Activity> = ArrayList(),
)

data class Activity(
    val type: String,
    var day: Int,
    var time: Int,
    val duration: Int,
)
```

Figure 3: These are the three classes that contain data. Their purpose is to only hold data as they do not have any method.
File: DataFactory.kt (Kotlin)

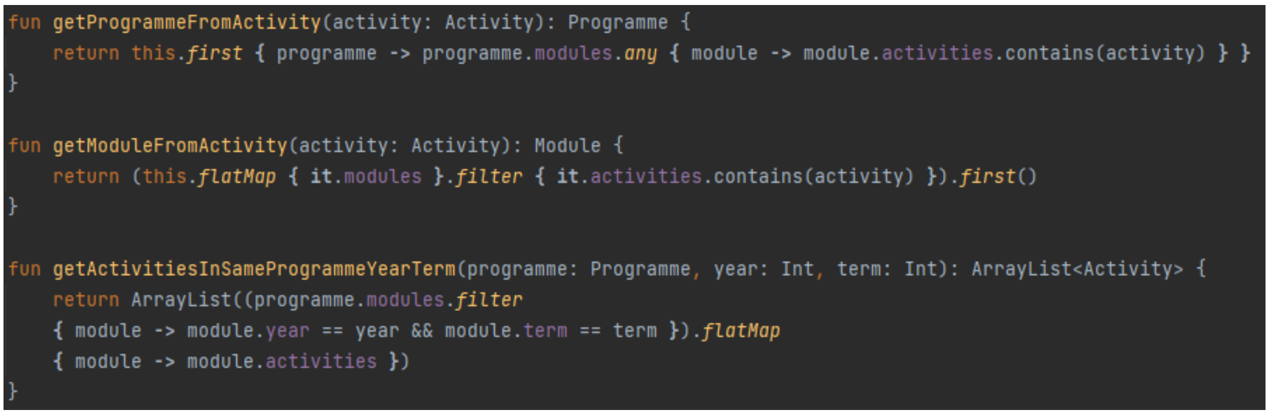


```kotlin
fun createProgramme(programme: Programme) {
    this.add(programme)
}

fun deleteProgramme(programme: Programme) {
    this.remove(programme)
}
```

```kotlin
fun createModule(programme: Programme, module: Module) {
    module.id = generateModuleId(programme)
    programme.modules.add(module)
}

fun deleteModule(programme: Programme, module: Module) {
    programme.modules.remove(module)
}
```

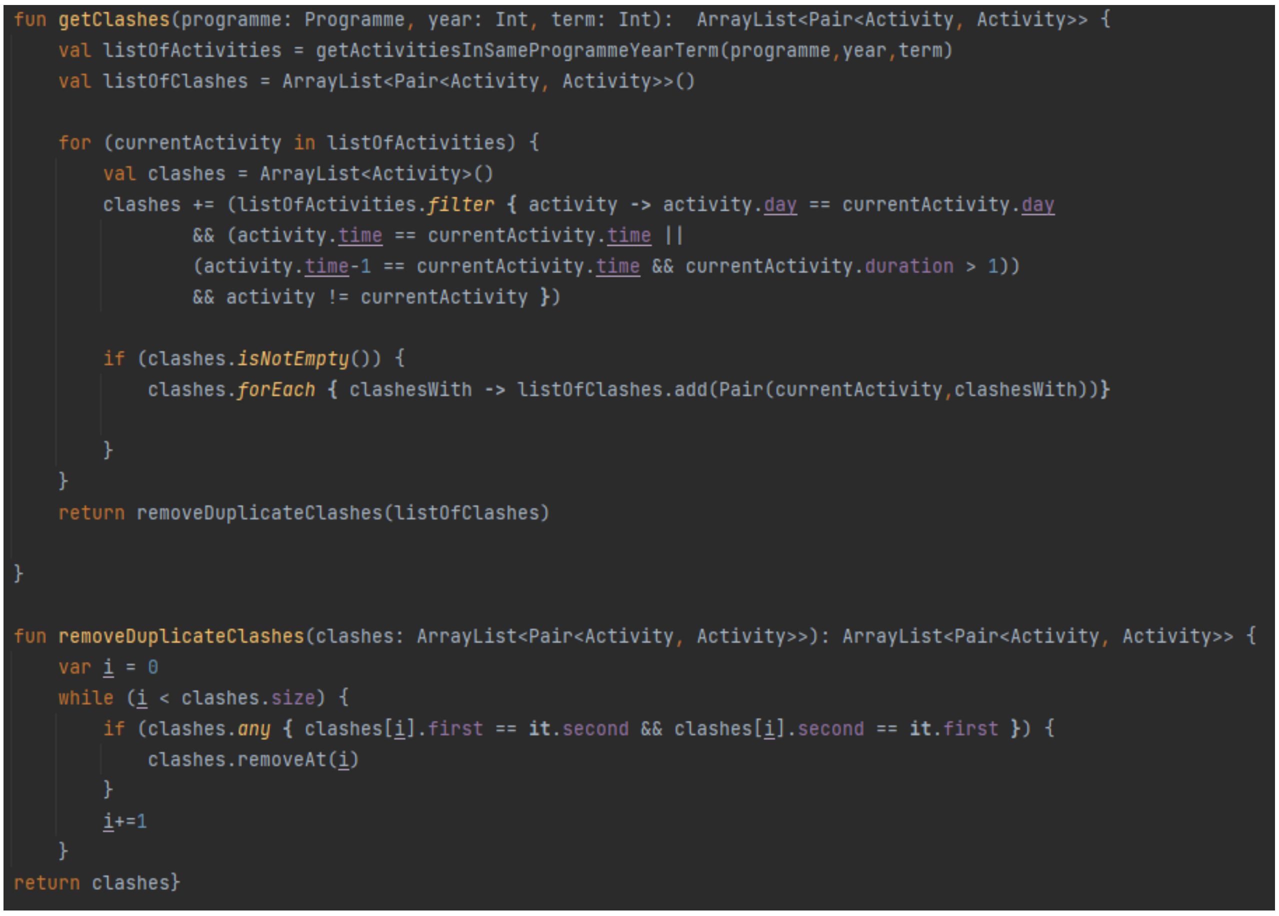
```kotlin
fun createActivity(module: Module, activity: Activity) {
    module.activities.add(activity)
}

fun deleteActivity(module: Module, activity: Activity) {
    module.activities.remove(activity)
}
```

Figure 4: Functions to create and delete programmes, modules, and activities.
File: DataFactory.kt (Kotlin)



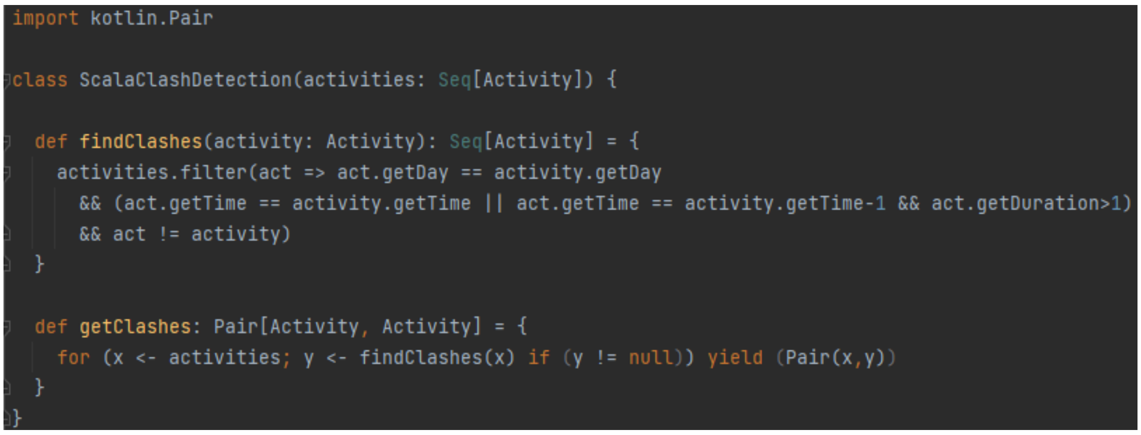
```kotlin
fun getProgrammeFromActivity(activity: Activity): Programme {
    return this.first { programme -> programme.modules.any { module -> module.activities.contains(activity) } }
}

fun getModuleFromActivity(activity: Activity): Module {
    return (this.flatMap { it.modules }.filter { it.activities.contains(activity) }).first()
}

fun getActivitiesInSameProgrammeYearTerm(programme: Programme, year: Int, term: Int): ArrayList<Activity> {
    return ArrayList((programme.modules.filter
    { module -> module.year == year && module.term == term }).flatMap
    { module -> module.activities })
}
```

Figure 5: Methods that filter the data to return the requested information.
File: DataFactory.kt (Kotlin)



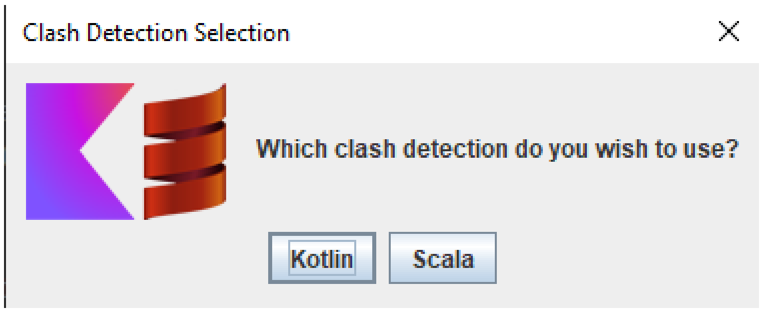
```
fun getClashes(programme: Programme, year: Int, term: Int):  ArrayList<Pair<Activity, Activity>> {
    val listOfActivities = getActivitiesInSameProgrammeYearTerm(programme,year,term)
    val listOfClashes = ArrayList<Pair<Activity, Activity>>()

    for (currentActivity in listOfActivities) {
        val clashes = ArrayList<Activity>()
        clashes += (listOfActivities.filter { activity -> activity.day == currentActivity.day
                && (activity.time == currentActivity.time ||
                (activity.time-1 == currentActivity.time && currentActivity.duration > 1))
                && activity != currentActivity })

        if (clashes.isNotEmpty()) {
            clashes.forEach { clashesWith -> listOfClashes.add(Pair(currentActivity,clashesWith))}

        }
    }
    return removeDuplicateClashes(listOfClashes)

}

fun removeDuplicateClashes(clashes: ArrayList<Pair<Activity, Activity>>): ArrayList<Pair<Activity, Activity>> {
    var i = 0
    while (i < clashes.size) {
        if (clashes.any { clashes[i].first == it.second && clashes[i].second == it.first }) {
            clashes.removeAt(i)
        }
        i+=1
    }
return clashes}
```

Figure 6: Clash detection function in Kotlin.File: DataFactory.kt (Kotlin)



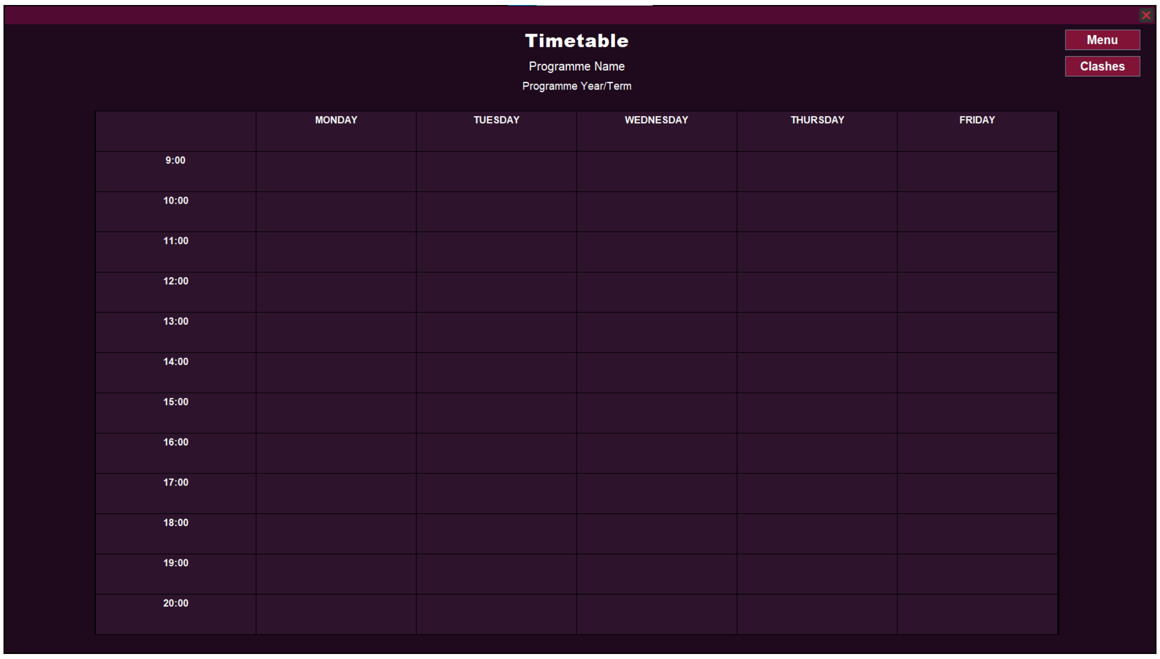
```
import kotlin.Pair

class ScalaClashDetection(activities: Seq[Activity]) {

  def findClashes(activity: Activity): Seq[Activity] = {
    activities.filter(act => act.getDay == activity.getDay
      && (act.getTime == activity.getTime || act.getTime == activity.getTime-1 && act.getDuration>1)
      && act != activity)
  }

  def getClashes: Pair[Activity, Activity] = {
    for (x <- activities; y <- findClashes(x) if (y != null)) yield (Pair(x,y))
  }
}
```

Figure 7: Clash detection function in Sca;a.File: ScalaClashDetection.scala (Scala)



Which clash detection do you wish to use?

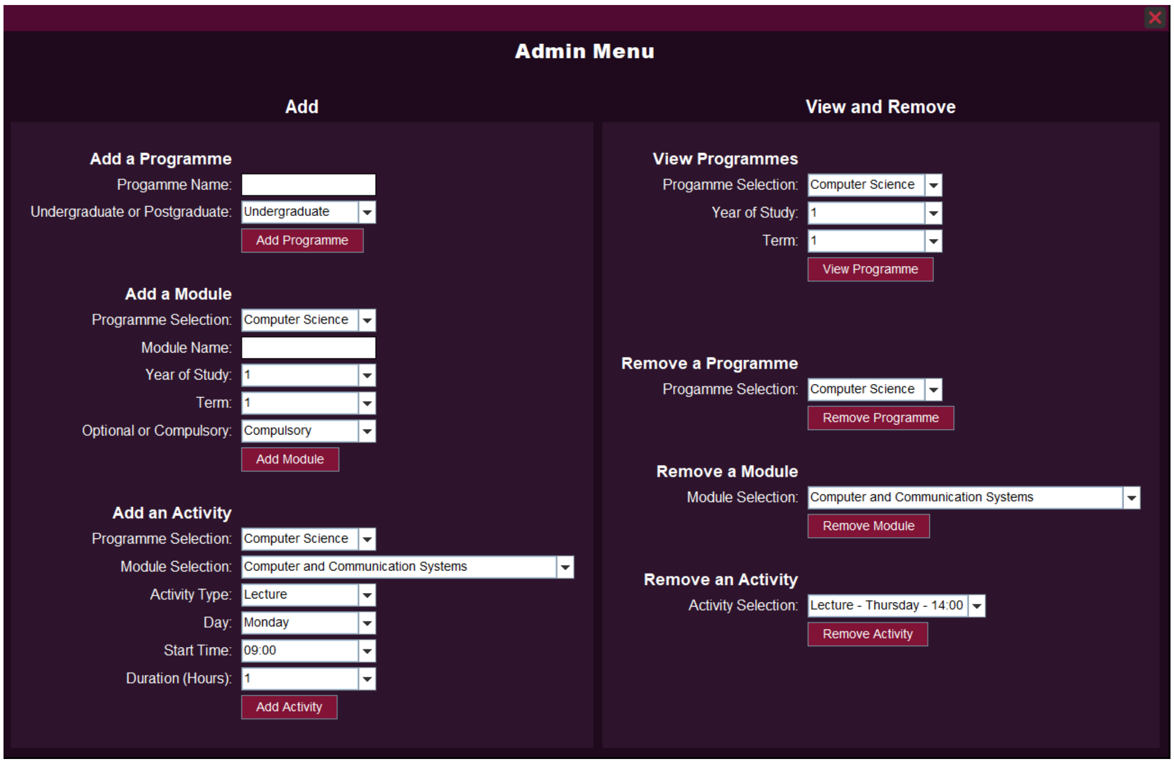
Figure 8: This window asks the user if they want to use Kotlin or Scala for Clash Detection.
File: Timetable.java (Java)



Timetable
Programme Name
Programme Year/Term

	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
9:00					
10:00					
11:00					
12:00					
13:00					
14:00					
15:00					
16:00					
17:00					
18:00					
19:00					
20:00					

Figure 9: This is the main window, which shows the entire timetable of the selected programme.
File: Timetable.java (Java)



Admin Menu

Add

Add a Programme

Progamme Name:	
Undergraduate or Postgraduate:	Undergraduate

Add Programme

Add a Module

Programme Selection:	Computer Science
Module Name:	
Year of Study:	1
Term:	1
Optional or Compulsory:	Compulsory

Add Module

Add an Activity

Programme Selection:	Computer Science
Module Selection:	Computer and Communication Systems
Activity Type:	Lecture
Day:	Monday
Start Time:	09:00
Duration (Hours):	1

Add Activity

View and Remove

View Programmes

Progamme Selection:	Computer Science
Year of Study:	1
Term:	1

View Programme

Remove a Programme

Progamme Selection:	Computer Science

Remove Programme

Remove a Module

Module Selection:	Computer and Communication Systems

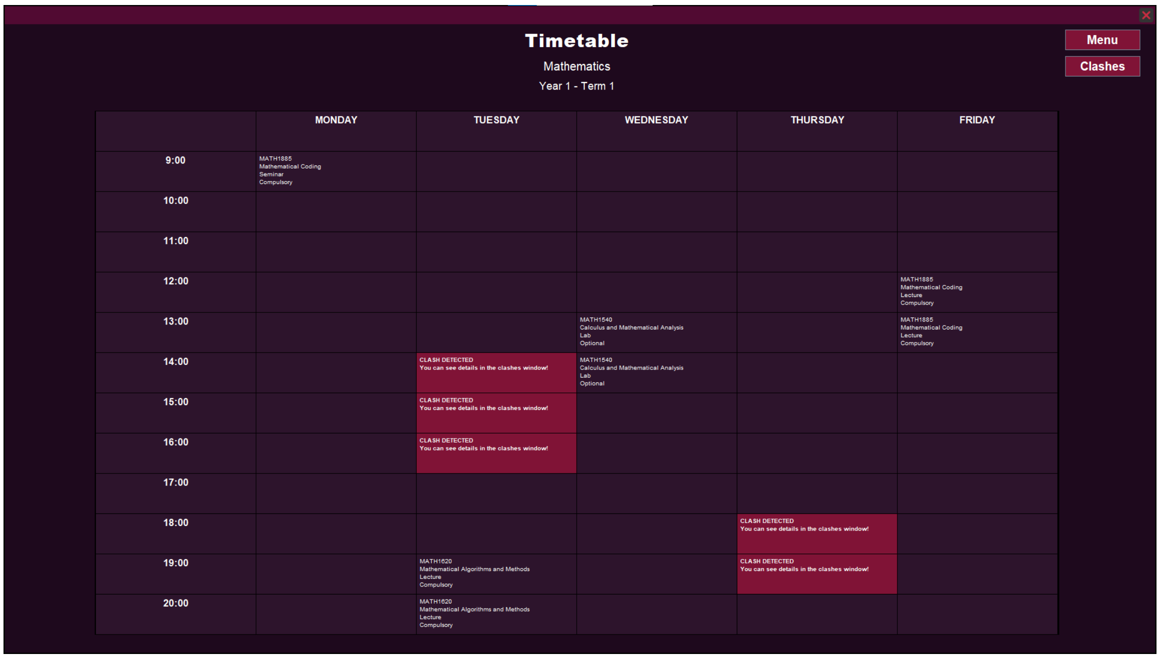
Remove Module

Remove an Activity

Activity Selection:	Lecture - Thursday - 14:00

Remove Activity

Figure 10: This is the admin menu, which allows the user to add or remove programmes, modules, and activities from the database.
File: Timetable.java (Java)



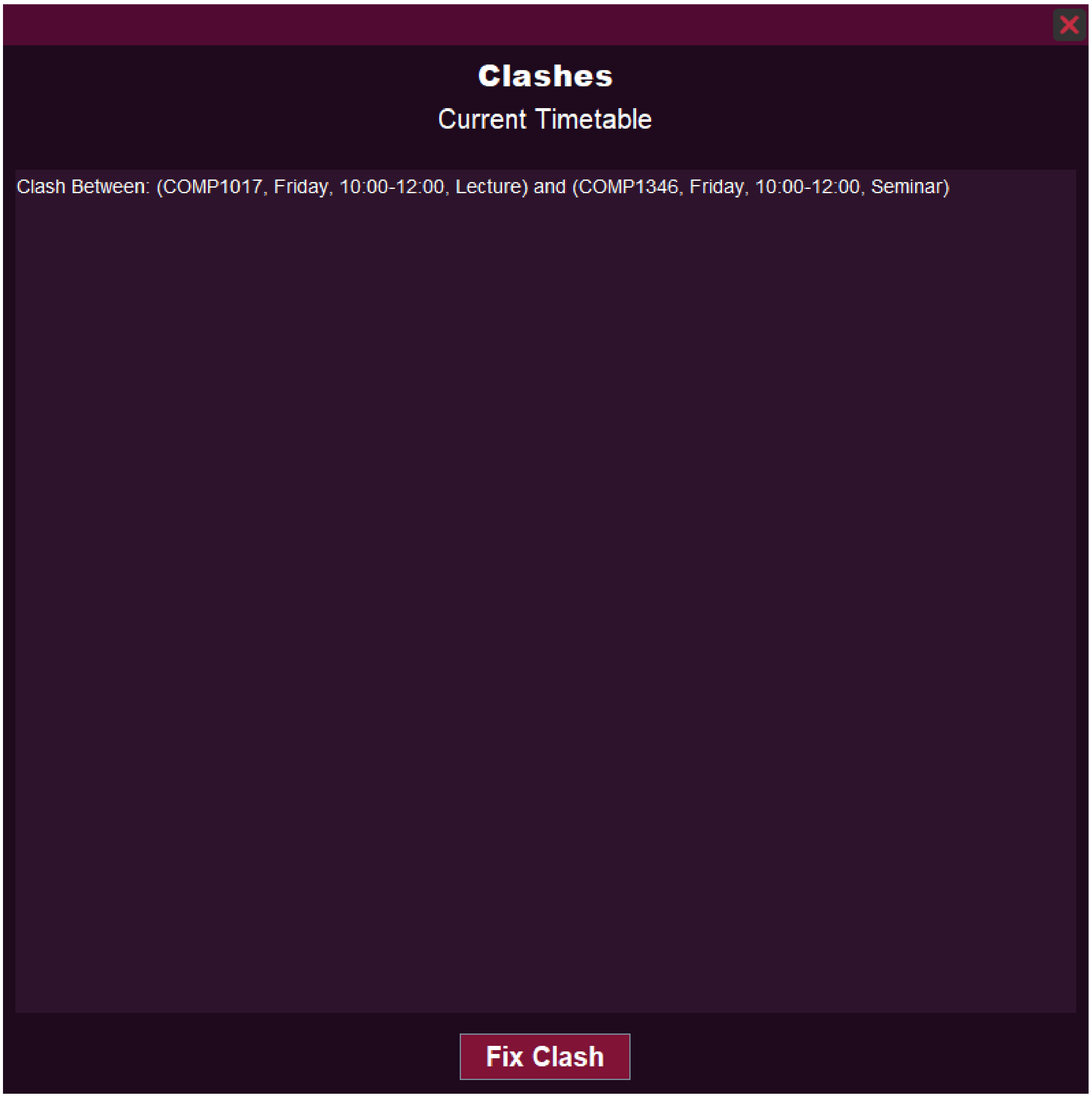
Timetable

Mathematics

Year 1 - Term 1

	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
9:00	MATH1885 Mathematical Coding Seminar Compulsory				
10:00					
11:00					
12:00					MATH1885 Mathematical Coding Lecture Compulsory
13:00			MATH1540 Calculus and Mathematical Analysis Lab Optional		MATH1885 Mathematical Coding Lecture Compulsory
14:00		CLASH DETECTED You can see details in the clashes window!	MATH1540 Calculus and Mathematical Analysis Lab Optional		
15:00		CLASH DETECTED You can see details in the clashes window!			
16:00		CLASH DETECTED You can see details in the clashes window!			
17:00					
18:00				CLASH DETECTED You can see details in the clashes window!	
19:00		MATH1620 Mathematical Algorithms and Methods Lecture Compulsory		CLASH DETECTED You can see details in the clashes window!	
20:00		MATH1620 Mathematical Algorithms and Methods Lecture Compulsory			

Figure 11: This figure shows the timetable filled with activities and clashes, which are highlighted in red.
File: Timetable.java (Java)



Clash Between: (COMP1017, Friday, 10:00-12:00, Lecture) and (COMP1346, Friday, 10:00-12:00, Seminar)

Figure 12: This is theh window that shows all the clashes with a "Fix Clash" button that allow.
File: Timetable.java (Java)