IT2010 – Mobile Application Development
BSc (Hons) in Information Technology
2nd Year
Faculty of Computing
SLIIT
2023 - Tutorial

Kotlin Coroutines

Kotlin coroutines are a design pattern that you can use in Kotlin programming to simplify code that executes asynchronously. Coroutines allow you to write asynchronous code in a sequential manner, making it easier to read and understand.

Key Concepts:

- Coroutines: Essentially lightweight threads. They are launched with launch or async builders and are often used to perform long-running or CPU-bound work without blocking the main thread.
- Suspending Functions: Functions that can be paused and resumed later. They are defined with the suspend modifier and can only be called from another suspending function or within a coroutine.
- CoroutineScope: An interface that defines a scope for new coroutines. Every coroutine builder (like launch and async) is an extension on CoroutineScope and inherits its coroutine context from it.
- Dispatchers: They determine the thread on which the coroutine will run. Some of the standard dispatchers available in Kotlin are Dispatchers.Main, Dispatchers.IO, and Dispatchers.Default.
- Jobs and Deferred: Job is a cancellable thing with a lifecycle that culminates in its completion. It is conceptually very similar to Future but has its own API and does not implement Future. Deferred is a non-blocking cancelable future — it is a Job with a result.

Key Benefits:

- Simplicity: Write asynchronous code just like synchronous code, using direct style programming.
- Efficiency: Coroutines do not require context switching on the thread scheduler, reducing overhead.
- Flexibility: You can decide where coroutines should run (main thread, background, etc.) using dispatchers.

Usage:

Coroutines are widely used for various use-cases like:

- Network API calls
- Database transactions
- File I/O operations
- Any other operations that might block the main thread.

Notes:

- Coroutines are not bound to any threading model. They can be run on a single thread, multiple threads, or even different hardware architectures.
- To use coroutines in your Kotlin project, you need to add the appropriate dependencies to your build file.

Example

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)

      val button:Button = findViewById(R.id.button)
      val textView:TextView = findViewById(R.id.textView)

      CoroutineScope(Dispatchers.Main).launch {
         counter(textView)
      }

      button.setOnClickListener {
         Toast.makeText(this,"Hello, World!",Toast.LENGTH_LONG).show()
      }


   }

   private suspend fun counter(view:TextView){
      var count = 0;
      while(true){
         delay(1000)
         view.text = count.toString()
         count ++
      }
   }
}
```

Now add the following snippet into the Button click and observe the output.

```
button.setOnClickListener {
   Toast.makeText(this,"Hello, World!",Toast.LENGTH_LONG).show()
   runBlocking {
      delay(3000)
   }
}
```

Coroutine Scope

A CoroutineScope in Kotlin is an interface that provides a way to manage the lifecycle of coroutines. It is used to launch coroutines and to manage their cancellation.

A CoroutineScope is a context in which coroutines run. It provides a set of rules that define how coroutines should behave within that context. The scope is responsible for creating a coroutine context and managing its lifecycle.

When a coroutine is launched, it is tied to the context of the scope in which it was launched. This means that when the scope is cancelled, all the coroutines launched within that scope are cancelled as well.

Dispatchers

In Kotlin coroutines, Dispatchers is a class that provides a set of thread pools and thread contexts that are used to determine on which thread a coroutine should be executed.

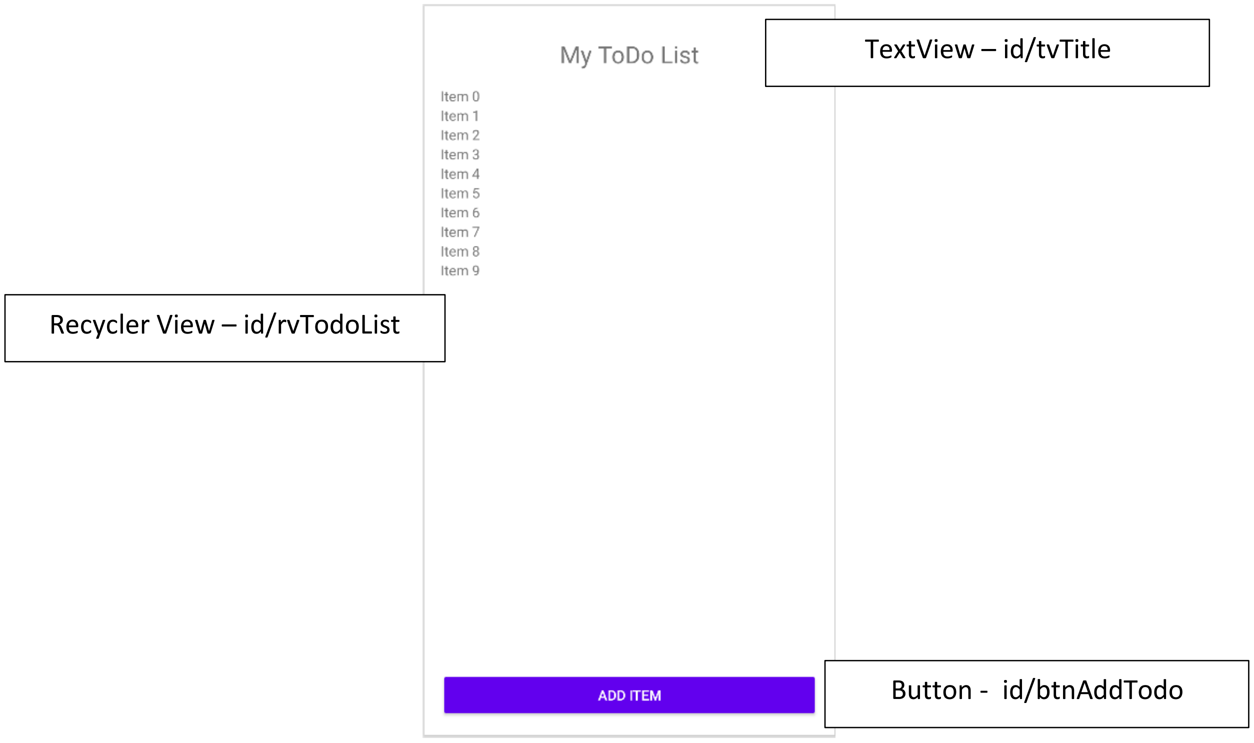
A coroutine running on a background thread can be used to perform long-running or CPU-intensive tasks without blocking the UI thread, while a coroutine running on the UI thread can be used to update the user interface.

Here are the common dispatchers available in Kotlin coroutines:

- **Dispatchers.Default**: This dispatcher is used for CPU-intensive tasks. It uses a shared thread pool and creates as many threads as there are CPU cores available.
- **Dispatchers.IO:** This dispatcher is used for IO-bound tasks, such as reading from or writing to files or making network requests. It uses a shared thread pool that is optimized for IO operations and can create more threads than the number of available CPU cores.
- **Dispatchers.Main:** This dispatcher is used for performing operations on the UI thread, such as updating the user interface or launching coroutines that will interact with UI elements.
- **Dispatchers.Unconfined:** This dispatcher is used for coroutines that do not have a specific thread context, and can run on any thread, including the UI thread.

Database implementation with Room

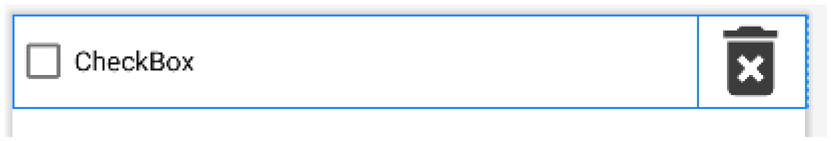
1. Remove all the previous codes and viewsDesign the following UI



2. Create a new package named ‘adapters’ in the main package
3. Create a class named TodoAdapter in it.
4. Extend the class with RecyclerView.Adapter class. And override the following methods

```
class TodoAdapter:RecyclerView.Adapter<ViewHolder>() {
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {

   }

   override fun getItemCount(): Int {

   }

   override fun onBindViewHolder(holder: ViewHolder, position: Int) {

   }

}
```

5. Create a new layout resource file in the layout directory named view_item
6. Convert the view to linear layout and design it like below



- Linear layout
    - Layout width – match parent
    - Layout height – 48dp
    - Id - todo_row_item
- Check Box
    - Layout width – wrap_content
    - Layout height – math_parent
    - Layout weight – 8
    - Id- cbTodo
- Image View
    - Layout width – wrap_content
    - Layout height – math_parent
    - Layout weight – 1
    - Id – ivDelete

7. Implement the RecyclerView ViewHolder as follows.

```
class ToDoViewHolder(view:View):ViewHolder(view) {
  val cbTodo:CheckBox
  val ivDelete:ImageView

  init {
    cbTodo = view.findViewById(R.id.cbTodo)
    ivDelete = view.findViewById(R.id.ivDelete)
  }

}
```

8. Let's implement the Recycler View Adapter

```
class TodoAdapter:RecyclerView.Adapter<TodoViewHolder>() {
  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val view = LayoutInflater.from(parent.context)
      .inflate(R.layout.view_item,parent,false)
    return ViewHolder(view)
  }
```

```
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.cbTodo.text = "Sample Test"
    }
    override fun getItemCount(): Int {
        return 1
    }
}
```

9. Implement the main activity code as follows and run the program

```
val recyclerView:RecyclerView = findViewById(R.id.rvTodoList)
val adapter = TodoAdapter()
recyclerView.adapter = adapter
recyclerView.layoutManager = LinearLayoutManager(this)
```

Database implementation with room

1. Add the following dependencies

```
// Room
implementation "androidx.room:room-runtime:2.5.0"
kapt "androidx.room:room-compiler:2.5.0"

// Kotlin Extensions and Coroutines support for Room
implementation "androidx.room:room-ktx:2.5.0"
```

And in plugins add the following. Then click Sync now

```
id 'kotlin-kapt'
```

2. Create a package named database in the main directory.
3. Create a class named TodoDatabase
4. Before implementing it create new package named entities in the database directory
5. In entities, create a data class as follows.

```
@Entity
data class Todo(
  var item: String?
) {
  @PrimaryKey(autoGenerate = true)
  var id: Int? = null
}
```

6. After that, create a new package named daos in the database directory.
7. In the daos package, create an interface named TodoDao.

```
@Dao
interface TodoDao {
  @Insert
  suspend fun insertTodo(todo: Todo)

  @Delete
  suspend fun deleteTodo(todo: Todo)

  @Query("SELECT * FROM Todo")
  fun getAllTodoItems():List<Todo>
}
```

8. Implement the TodoDatabase as follows

```
@Database(entities = [Todo::class], version = 1)
abstract class TodoDatabase:RoomDatabase(){

  abstract fun getTodoDao():TodoDao

  companion object{
    @Volatile
    private var INSTANCE: TodoDatabase? = null

    fun getInstance(context:Context):TodoDatabase{
      synchronized(this){
        return INSTANCE ?: Room.databaseBuilder(
          context.applicationContext,
          TodoDatabase::class.java,
                "todo_db"
            ).build().also {
                INSTANCE = it
            }
        }

    }
  }
}
```

9. Create a new package named repositories in the database directory.
10. Inside that create a class named TodoRepository and implement it as follows.

```
class TodoRepository(
  private val db:TodoDatabase
) {
  suspend fun insert(todo:Todo) = db.getTodoDao().insertTodo(todo)
  suspend fun delete(todo:Todo) = db.getTodoDao().deleteTodo(todo)
  fun getAllTodoItems():List<Todo> = db.getTodoDao().getAllTodoItems()

}
```

11. Create the ViewModel to manage the RecyclerView data as follows.

```
class MainActivityData:ViewModel() {

  private val _data = MutableLiveData<List<Todo>>()
  val data:LiveData<List<Todo>> = _data
  fun setData(data:List<Todo>){
    _data.value = data
  }

}
```

12. Now let’s update the adapter as follows:
13. Update the class constructer parameters and initialized the variables.

```
class TodoAdapter(items:List<Todo>, repository: TodoRepository,
viewModel:MainActivityData):Adapter<ToDoViewHolder>() {

  var context:Context? = null
  val items = items
  val repository = repository
```

```
   val viewModel = viewModel
```

14. Update the onCreateViewHolder method.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ToDoViewHolder {
   val view = LayoutInflater.from(parent.context)
      .inflate(R.layout.view_item,parent,false)
   context = parent.context
   return ToDoViewHolder(view)
}
```

15. Next, getItemCount method

```
override fun getItemCount(): Int {
   return items.size
}
```

16. Then update the onBindViewHolder

```
override fun onBindViewHolder(holder: ToDoViewHolder, position: Int) {

      holder.cbTodo.text = items.get(position).item
      holder.ivDelete.setOnClickListener {
         val isChecked = holder.cbTodo.isChecked
         if(isChecked){
            CoroutineScope(Dispatchers.IO).launch {
               repository.delete(items.get(position))
               val data = repository.getAllTodoItems()
               withContext(Dispatchers.Main){
                  viewModel.setData(data)
               }
            }
            Toast.makeText(context,"Item Deleted",Toast.LENGTH_LONG).show()
         }else{
            Toast.makeText(context,"Select the item to delete",Toast.LENGTH_LONG).show()
         }
      }

   }
```

17. Now let’s introduce the dialog box to enter data to the todo item as follows.

```
fun displayDialog(repository: TodoRepository){
    val builder = AlertDialog.Builder(this)

    // Set the alert dialog title and message
    builder.setTitle("Enter New Todo item:")
    builder.setMessage("Enter the todo item below:")

    // Create an EditText input field
    val input = EditText(this)
    input.inputType = InputType.TYPE_CLASS_TEXT
    builder.setView(input)

    // Set the positive button action
    builder.setPositiveButton("OK") { dialog, which ->
// Get the input text and display a Toast message
        val item = input.text.toString()
        CoroutineScope(Dispatchers.IO).launch {
            repository.insert(Todo(item))
            val data = repository.getAllTodoItems()
            runOnUiThread {
                viewModel.setData(data)
            }
        }
    }
    // Set the negative button action
    builder.setNegativeButton("Cancel") { dialog, which ->
        dialog.cancel()
    }
// Create and show the alert dialog
    val alertDialog = builder.create()
    alertDialog.show()
  }
```

18. Now implement the following in the onCreate method of the MainActivity.

```
private lateinit var adapter:TodoAdapter
private lateinit var viewModel:MainActivityData
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)

  val repository = TodoRepository(TodoDatabase.getInstance(this))
  val recyclerView:RecyclerView = findViewById(R.id.rvTodoList)
  viewModel = ViewModelProvider(this)[MainActivityData::class.java]
```

```
    viewModel.data.observe(this){
        adapter = TodoAdapter(it,repository, viewModel)
        recyclerView.adapter = adapter
        recyclerView.layoutManager = LinearLayoutManager(this)
    }

    CoroutineScope(Dispatchers.IO).launch {
       val data = repository.getAllTodoItems()
        runOnUiThread {
            viewModel.setData(data)
        }
    }
    val btnAddItem: Button = findViewById(R.id.btnAddTodo)

    btnAddItem.setOnClickListener {
        displayDialog(repository)
    }


}
```