Building Console Applications

CS 346 Application Development

Getting started

Where do we begin?

Let's build an application!

Consider the features needed for a simple console application.

- Command-line only.
  - **Keyboard driven**. Support for keyboards but no pointing devices.
  - **Non-graphical**. Intended to be run from a shell/terminal.
  - **Standard text I/O**. Read/write from the terminal, or file system.
  - **Stores data**. Filesystem only.
- We'll avoid advanced features.
  - No graphical user interface.
  - No mouse support.
  - No networking/cloud.
  - No database.

We want the ability to add these later!

Starting here will allow us to build up the structure and then expand on it later.

Creating a Project

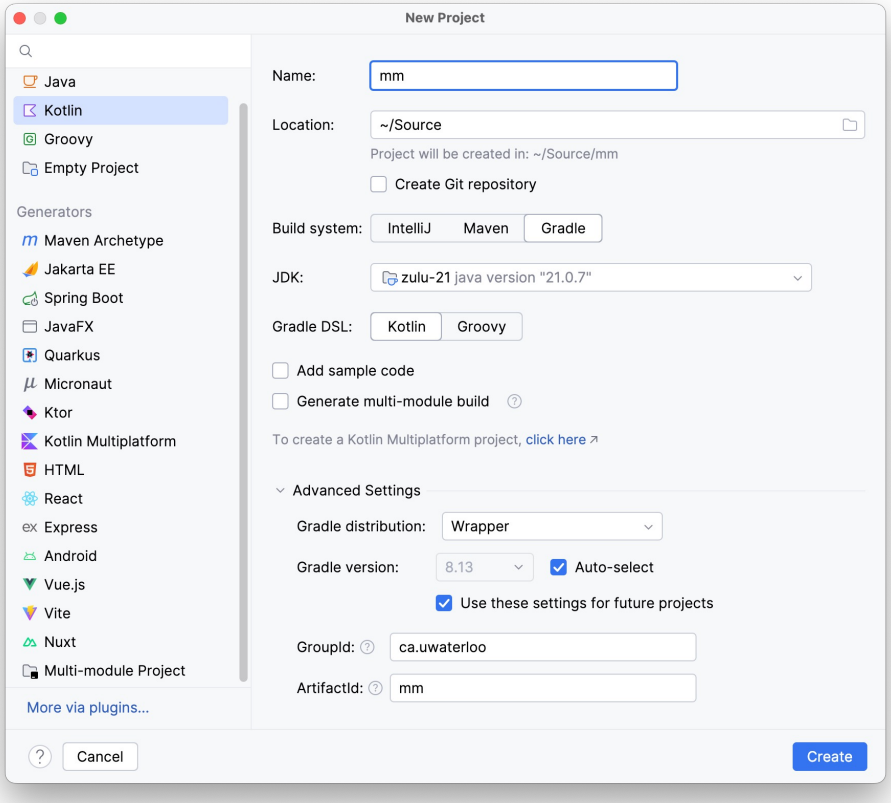
`File > New > Project > Kotlin`

Project settings

- Build system: Gradle
- JD: 21.0.X
- Gradle DSL: Kotlin
- GroupID: unique identifier
- ArtifactID: matches project name

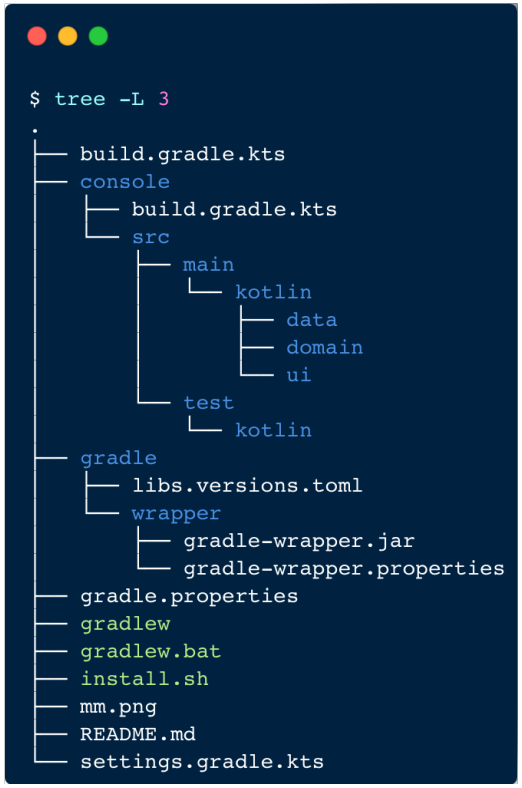
Note: "Console" and "desktop" projects are built the same way! One just has additional UI dependencies added to the project.

We'll discuss desktop projects soon.



https://git.uwaterloo.ca/cs346/demos/mm-console

Step 1: Create the directory structure



build.gradle.kts – the main build configuration file

src/main/kotlin – source code goes here
src/test/kotlin – unit tests go here

libs.versions.toml– version catalog that lists dependencies

gradlew – script to run gradle tasks

Step 2: Add plugins, customize build

build.gradle.kts

```
plugins {
  application
  alias(libs.plugins.kotlin.jvm)
  alias(libs.plugins.kotlin.serialization)
  alias(libs.plugins.versions)
}

group = "ca.uwaterloo.cs346"
version = "1.0"

dependencies {
  implementation(libs.clikt)
  implementation(libs.json)
}
```

Plugins describe the supported Gradle tasks.

- **application** – supports running JVM applications.
- **kotlin.jvm** – JVM applications build tasks
- **kotlin.serialization** - reading/writing files later.
- **versions** – version catalog.

group is the top-level package name.
version is a unique label (see semantic versioning).

Dependencies from the version catalog.

- **libs.clikt** – the clikt library for command-line args
- **libs.json** – supports reading/writing JSON files.

Step 3: Add dependencies (e.g., clikt)

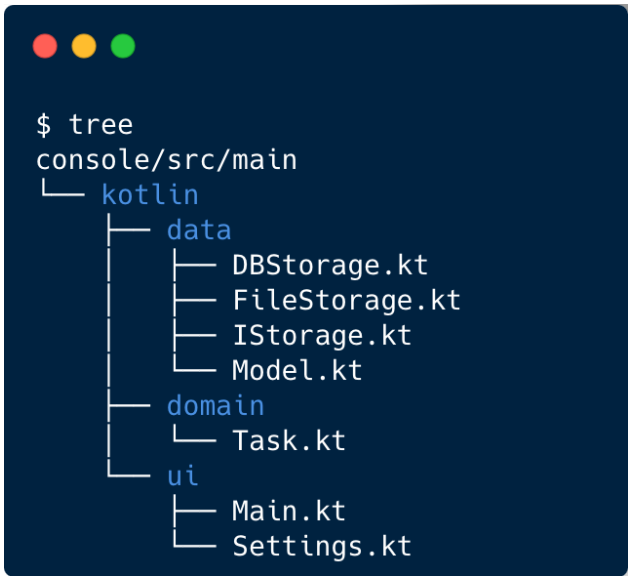
libs.version.toml

```
[versions]
clikt-version = "4.4.0"


[plugins]


[libraries]
  clikt = { module = "com.github.ajalt.clikt:clikt", version.ref = "clikt-version"
}
```

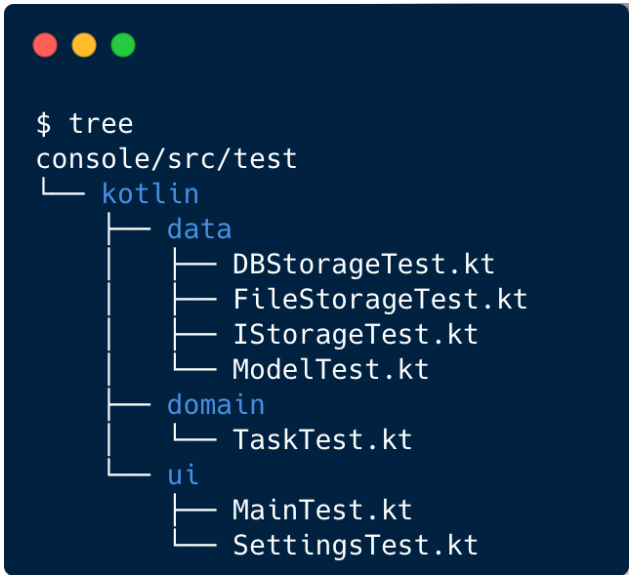
build.gradle.kts

```
dependencies {
    implementation(libs.clikt)
}
```

Step 4: Create source files



data – model, storage layer
domain – business logic layer
ui – user interface layer (including `main`)



Unit tests for each source files belong in a corresponding file with the "Test" suffix e.g., MainTest.kt.

Architecture

Components and structure.

Architecture

We DO want to support best-practices.

- Flexibility to add or expand features later.
- Benefits of separation of concerns, low coupling.

How will this work?

- Start with a layered architecture:
  - **User interface**: standard IO only.
  - **Domain layer**: intermediate classes (“business logic”).
  - **Data layer**: store our data (file or database).
- Map desired functionality to each layer.
- Add unit tests as we go.

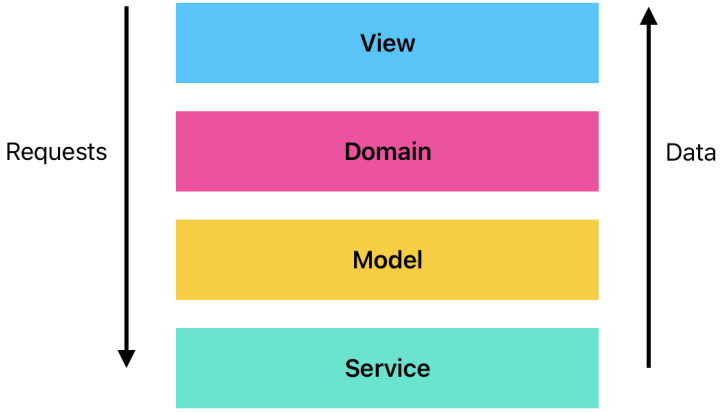
Recall: Layered Architecture

Layered architecture:

- Each layer has specific functionality that it presents to the layer above
- Requests flow down, and data flows up.
- *This also means that dependencies extend down*.

For a console application:

- The View will be pretty “thin” since we don’t have a graphical user interface.
- Domain, Model are still present.
- Service is customized based on functionality.



Each of these is a layer which may consist of multiple classes and/or functions.

mm-console

A simple, layered console application to get us started.

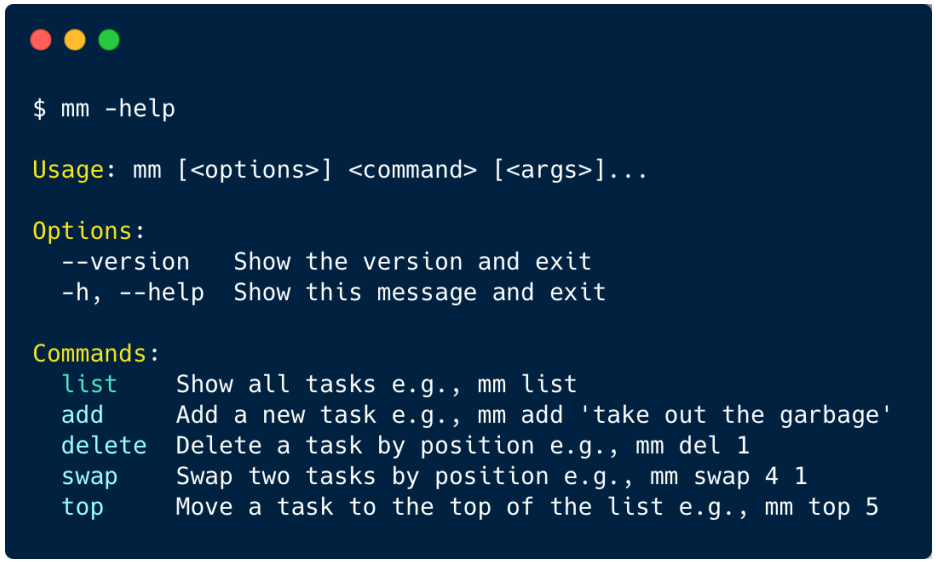
Example: TODO application

Display a “TODO list”:

- User can add/delete/show items.
- Standard options like `help`, `version`.
- Color output.

To use it, run `mm`:

- `mm` by itself will display the list. ★
- `mm <command>` will run command.
- `mm --<option>` will run that option.



The help screen displayed when you run `mm –help`.

★ Standard behavior should probably display `help` if you don’t provide arguments, but displaying the list by default makes the application more usable.

Example: TODO design

- Standard console application.
  - Win, Mac, Linux.
  - stdin, stdout, stderr.
- Data files in the user's home directory.
  - Plain-text.
  - Human-readable.
- External libraries to support color.
- Layered architecture.
  - Option to swap text-file for DB later.
  - Ability to move to a graphical UI later.
- Testable, with unit tests.

```
$ mm -help

Usage: mm [<options>] <command> [<args>]...

Options:
  --version   Show the version and exit
  -h, --help  Show this message and exit

Commands:
  list    Show all tasks e.g., mm list
  add     Add a new task e.g., mm add 'take out the garbage'
  delete  Delete a task by position e.g., mm del 1
  swap    Swap two tasks by position e.g., mm swap 4 1
  top     Move a task to the top of the list e.g., mm top 5
```

The help screen displayed when you run `mm –help`.

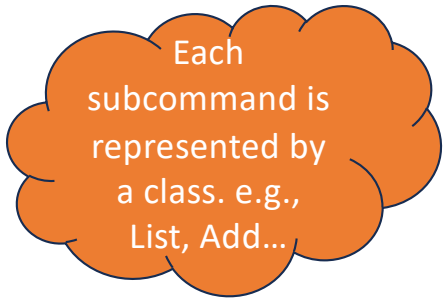
https://git.uwaterloo.ca/cs346/demos/mm-console

Kotlin applications require a main method as an entry point.
Our main method delegates to the mm top-level class (Clikt class).

```
/**
 * Main.kt
 * Main entry point for our console application.
 * Delegates to the MegaMind (Clikt) class.
 */

fun main(args: Array<String>) =
    MegaMind()
        .versionOption(version = "1.0")
        .subcommands(List(), Add(), Delete(), Swap(), Top())
        .main(args)
```

https://pl.kotl.in/f9MngUzGF

```
class MegaMind : CliktCommand(name = "mm", invokeWithoutSubcommand = true) {
    override fun run() {
        val subcommand = currentContext.invokedSubcommand
        if (subcommand == null) {
            val storage = DBStorage(".mm.db")
            val model = Model(storage)
            model.tasks.sortedBy { it?.position }.forEach {
                echo("${it?.position}. ${it?.title}")
            }
        }
    }
}

class List : CliktCommand(help = "Show all tasks e.g., mm list") {
    override fun run() {
        val storage = DBStorage(".mm.db")
        val model = Model(storage)
        model.tasks.sortedBy { it?.position }.forEach {
            echo("${it?.position}. ${it?.title}")
        }
    }
}
```



Model class

```
class Model(private val storage: IStorage) {
    var tasks = mutableListOf<Task?>()

    init {
        tasks = storage.readAll().toMutableList()
    }

    fun add(contents: String) {
        val task = Task(
                position = tasks.size + 1,
                title = contents,
                description = "",
                dueDate = "",
                tags = "")

        storage.create(task)
        tasks.add(task)
    }
```

Storage interface

```
interface IStorage {
    // canonical operations
    fun create(task: Task): Int
    fun read(id: Int): Task?
    fun readAll(): List<Task?>
    fun update(task: Task)
    fun delete(task: Task)
    fun deleteAll()

    // extended operations
    fun upsert(task: Task)
}
```

Storage classes

```
// storage classes implement the IStorage interface
// and each implements methods like add(), delete().
class DBStorage: IStorage { }
class FileStorage: IStorage { }

// our code instantiates the correct storage class.
// changing from DB to file storage is a single line change.
val storage = DBStorage()
val model = Model(storage)
model.tasks.sortedBy { it?.position }.forEach {
      echo("${it?.position}. ${it?.title}")
}
```

Testing

Considerations when writing unit tests for console applications.

Console == Baseline

Guidelines from the unit testing lecture are 100% applicable.

- Try and write your tests as you develop functionality.
- **Unit tests are not just about correctness! Tests should inform your design**.

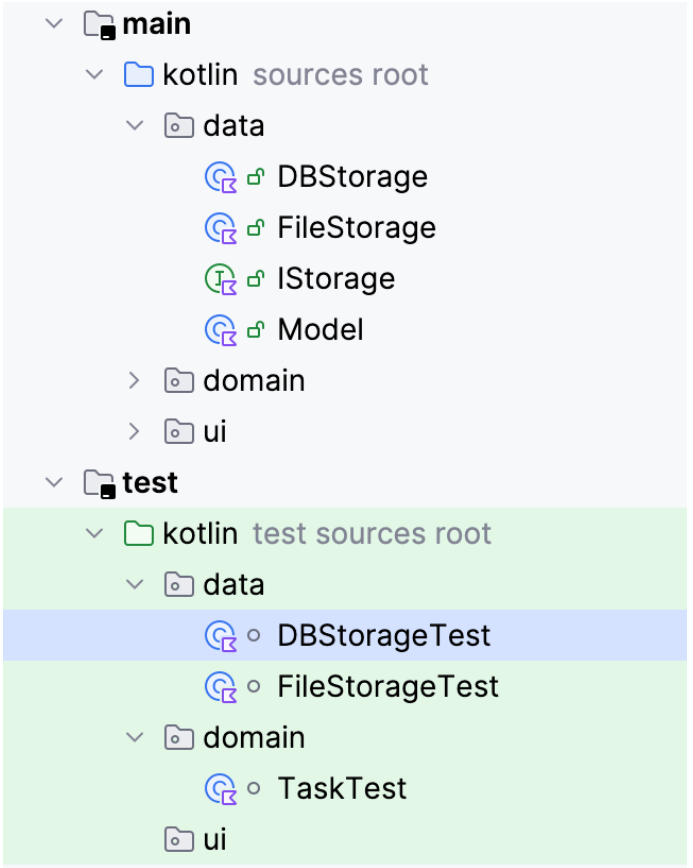
Things to focus on:

1. Test every layer.
2. Make tests independent of each other.
3. Use Dependency Injection – *critical*

Test every layer

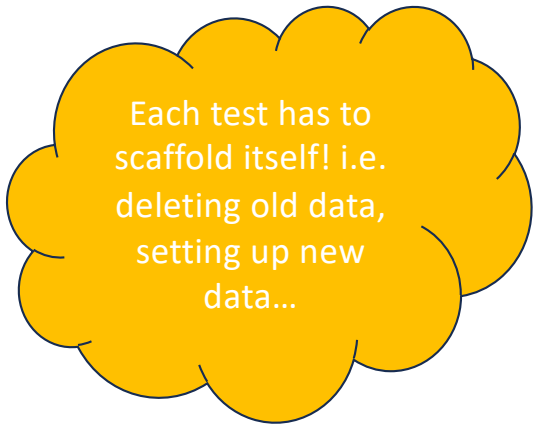
- Your test package structure should match the source code packages.
  - One test class for every class.
  - ClassName + “Test” suffix.

- This example?
  - Structure is correct.
  - Missing the `model` and `ui` tests.



Test independence

```
val task01 = Task(position = 1, title = "Test Task 1")

@Test
fun updateTest() {
  // setup
  val database = DBStorage(databaseName = ".mm-test.db")
  database.deleteAll()
  assert(database.readAll().isEmpty())

  // save test data
  val id1 = database.create(task01)
  assert(id1 == task01.id)

  // what we're actually testing!
  task01.title = "Revised"
  database.update(task01)
}
```



Use DI to avoid hard-coding dependencies

FileStorage.kt

```
class FileStorage (filename: String = ".mm.json"): IStorage {
    private var fullPath = System.getProperty("user.home") + "/$filename"

    init {
        // open file here
    }
}
```

FileStorageTest.kt

```
val filename = ".mm-test.json” // filename reused across tests

@Test
fun initTest() {
    val storage = FileStorage(filename)
    storage.deleteAll()
    assert(storage.readAll().isEmpty())
}
```