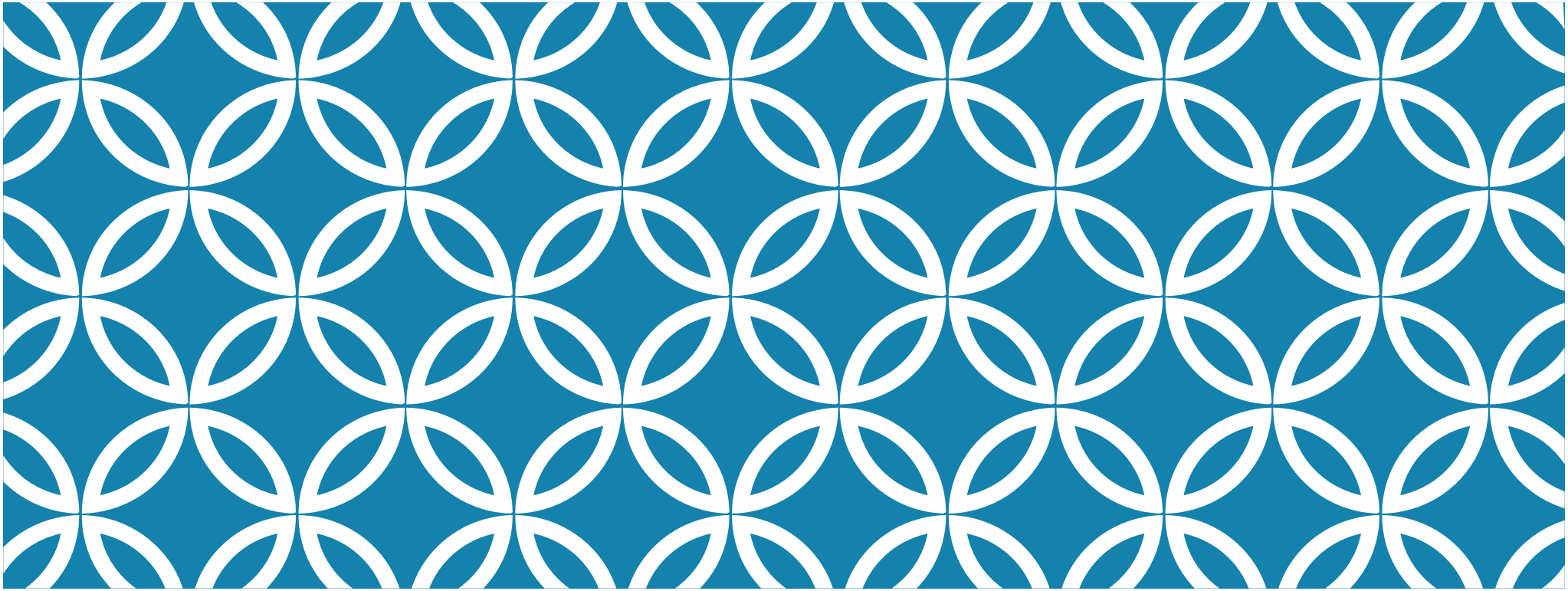


BUILDING AN APPLICATION

CS 346: Application Development

GETTING STARTED

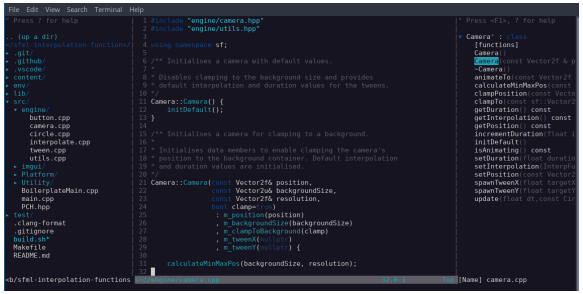
Let's consider a simple application as our starting point.

Command-line only.

- **Keyboard driven**. Not intended to support mice, trackpads.
- **Non-graphical**. Intended to be run from a shell/terminal, with limited character graphics.
- **Standard text I/O**. Read/write from the terminal, or file system.

We'll avoid advanced features.

- No graphics.
- No networking.



COMMAND-LINE EXECUTION

Typically, command-line applications should use this calling convention, or something similar:

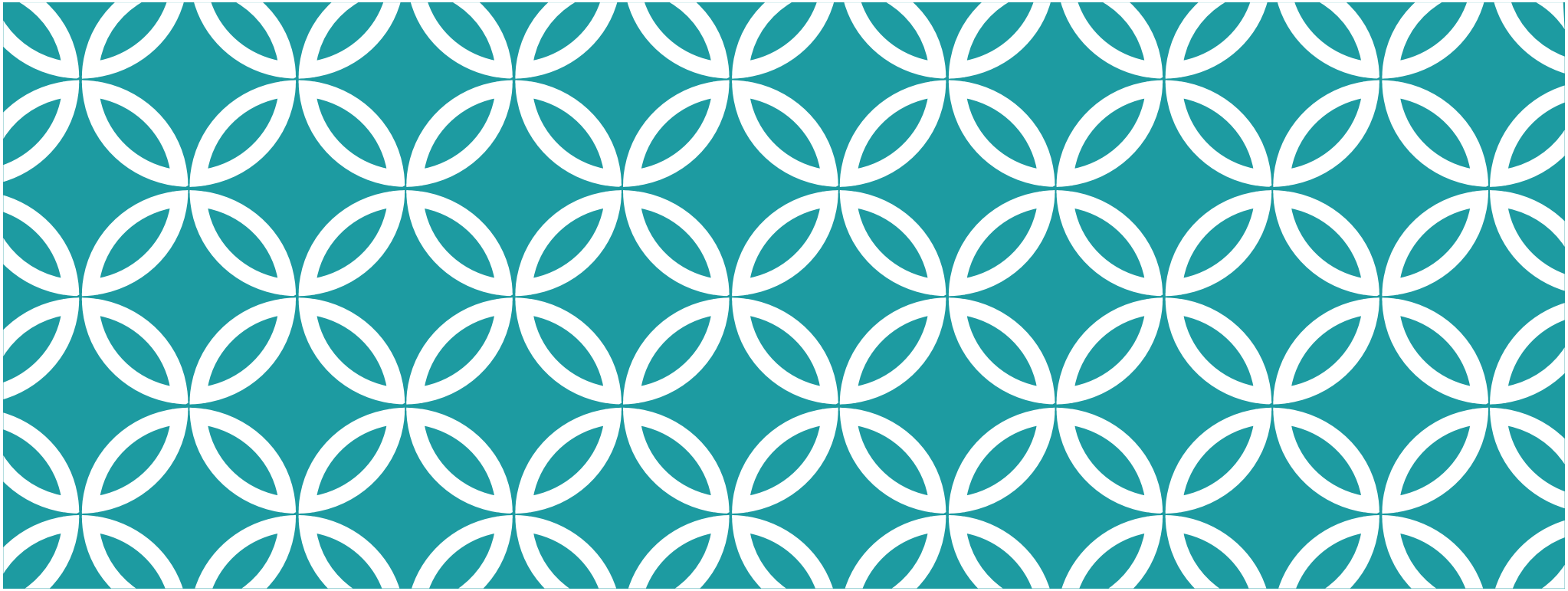
```
$ program_name -option value parameter
```

What these mean?

- **program_name** is the name of your program. It should describe what your program does.
- **options** represent a value that tells your program how to operate. Options are normally prefixed with a dash ("-") to distinguish them from parameters and may require values.
- **parameter** represents data that your program would act upon (e.g. the name of a file containing data). If multiple parameters are required, separate them with whitespace.

Running a program with insufficient arguments should display information on how to successfully execute it.

```
$ rename
 Usage: rename [source] [dest]
```



FEATURES

CS 346: Application Development

CREATING A PROJECT

`File > New > Project > Kotlin > Kotlin/JVM` to create a new project.

Add the `application` plugin to the `build.gradle.kts`. This adds the `Tasks > application > run` task.

Command	What does it do?
`Tasks` > `build` > `clean`	Removes temp files (deletes the `/build` directory)
`Tasks` > `build` > `build`	Compiles your application
`Tasks` > `application` > `run`	Executes your application (builds it first if necessary)
`Tasks` > `application` > `installDist`	Creates a distribution package
`Tasks` > `application` > `distZip`	Creates a distribution package

MAIN METHOD

As you would expect from similar languages, Kotlin applications require a main method as an entry point.

Arguments are optional. If provided, we can iterate over args or extract data from the array directly.

```
fun main(args: Array<String>) {
    print("${args.size} arguments passed in")
    for (arg in args) {
        println(arg)
    }
}
```

https://pl.kotl.in/Y_7mTzYYW

ARGUMENTS

Arguments are passed in as an array to the main method.

```
fun main(args: Array<String>) {
    if (args.size == 0) {
        println("Usage: rename [old] [new]")
        println(" [old] is the source filename")
        println(" [new] is the target filename")
    } else {
        for (arg in args) {
            if File.exists(arg) {
                // do something useful here
            }
        }
    }
}
```

STDIO, ERRORS

The Kotlin Standard Library includes classes and functions for interacting with the console: kotlin-stdlib / kotlin.io

- **readln()** :: reads and returns a line of input from the stdin (not including CRLF) or throws a RuntimeException if EOF has already been reached.
- **readlnOrNull()** :: Reads a line of input from the standard input stream and returns it, or return null if EOF has already been reached
- **println()** :: directs output to stdout, with CRLF.
- **print()** :: directs output to stdout without CRLF.

```
// read single value from stdin
val str:String ?= readLn()
if (str != null) {
   println(str.toUpperCase())
}
```

IO CLASSES

File :: class representing a file or directory in the filesystem (extends `java.io.File` with new methods).

```
File(filename).writeText("Jeff was here")

val lines: List<String> = File(filename).readLines()
val contents: String = File(filename).readText()
println(contents)

2020-06-06T14:35:44, 1001, 78.22, CDN
2020-06-06T14:38:18, 1002, 12.10, CDN
```

FileTreeWalk :: class to iterate over Files in the filesystem.

Reader or Writer :: classes to create a buffered file reader or writer… i.e. support for Java classes.

```
// Java approach with streams
val reader = someStream.bufferedReader()
reader.useLines {
    it.map { line -> // do something with line }
}
```

ERROR HANDLING

Kotlin uses exceptions to indicate that an operation has failed. The mechanism is similar to other languages: if an error is detected, an exception is created and 'thrown', and then 'caught' and consumed by error handling code further up the stack.

Exceptions mechanisms can be either checked or unchecked:

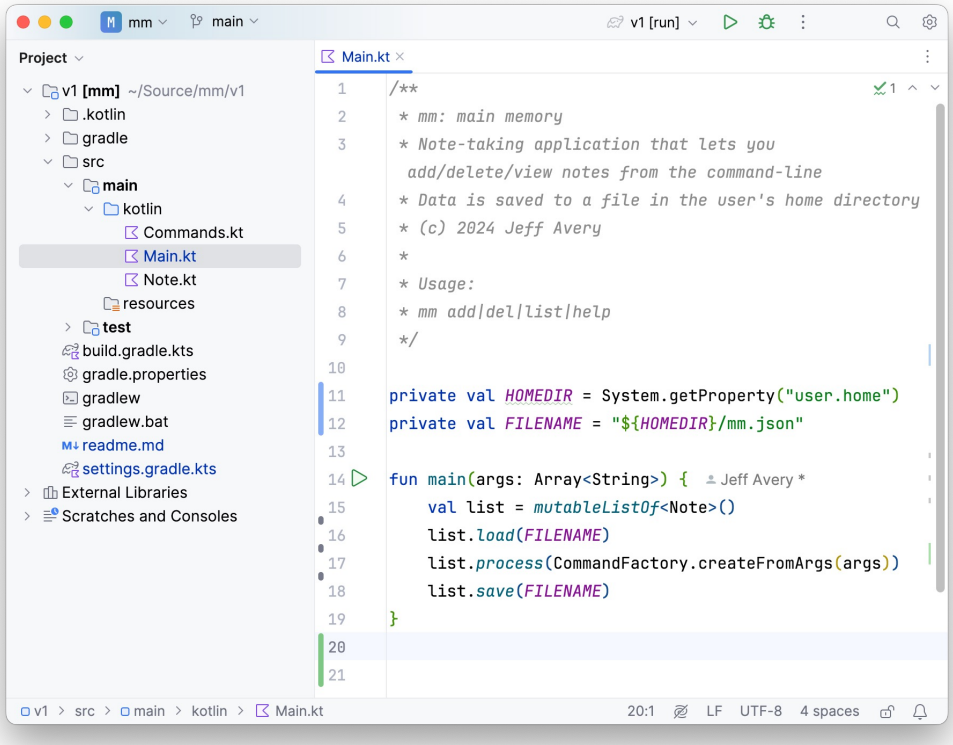
- **Unchecked** means that exceptions are **not** checked at compile-time. If an exception is thrown by some function, it is passed up the call stack and may or may not be handled by a corresponding 'catch' block. e.g. C++
- **Checked** exceptions are checked at compile-time. Exceptions are declared with each function in the call-chain and must be handled by a corresponding 'catch' block.

Like C++, Kotlin supports unchecked exceptions.

DEMO: MM APPLICATION

TODO Application

The “simplest thing” I could think of, which still demonstrates key functionality.



EXAMPLE: SEQUENTIAL PROCESSING

```
/**
 * mm: main memory
 * Note-taking application that lets you add/delete/view notes
 * Data is saved to a file in the user's home directory
 * (c) 2024 Jeff Avery
 *
 * Usage:
 * mm add|del|list|help
 */

private val HOMEDIR = System.getProperty("user.home")
private val FILENAME = "${HOMEDIR}/mm.json"

fun main(args: Array<String>) {
    val list = mutableListOf<Note>()
    list.load(FILENAME)
    list.process(CommandFactory.createFromArgs(args))
    list.save(FILENAME)
}
```

Pipes-and-filter

EXAMPLE: SEPARATION OF CONCERNS

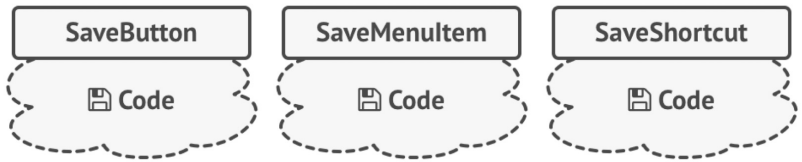
```kotlin
/**
 * Note.kt
 * Our primary data class, used for storing and displaying notes.
 */

@Serializable
data class Note(
    val id: String = UUID.randomUUID().toString(),
    var index: Int = 0,
    val title: String? = null,
    val content: String? = null
)

// save to a file
fun MutableList<Note>.save(filename: String) {
    val output = Json.encodeToString(this)
    File(filename).writeText(output)
}
```

Data is completely managed by a single data class.

DESIGN PATTERN: COMMAND

Problem: Imagine that you are writing a user interface, and you want to support a common action like Save. You might invoke Save from the menu, or a toolbar, or a button. Where do you put the code, without duplicating it?

The **command pattern** is a behavioural design pattern that turns a request into a stand-alone object that contains all information about the request (a command could also be thought of as an action to perform).



Several classes implement the same functionality.

EXAMPLE: ARGS VIA COMMAND-PATTERN

```
// Factory pattern
// generate a command based on the arguments passed in
object CommandFactory {
    fun createFromArgs(args: Array<String>): Command = if (args.isEmpty()) {
        HelpCommand(args)
    } else {
        when (args[0]) {
            "add" -> AddCommand(args)
            "del" -> DelCommand(args)
            "list" -> ListCommand(args)
            else -> HelpCommand(args)
        }
    }
}
```

```kotlin
// Command pattern
// represents all valid commands that can be issued by the user
// any functionality for a given command should be contained in that class
interface Command {
    fun execute(items: MutableList<Note>)
}

class AddCommand(val args: Array<String>) : Command {
    override fun execute(items: MutableList<Note>) {
        items.add(Note(index = items.size, title = args[1], content = args[2]))
    }
}

class DelCommand(val args: Array<String>) : Command {
    override fun execute(items: MutableList<Note>) {
        items.removeIf { it.id == args[1] }
    }
}

class ListCommand(val args: Array<String>) : Command {
    override fun execute(items: MutableList<Note>) {
        items.forEach { println("[${it.index}] ${it.title} ${it.content}") }
    }
}
```