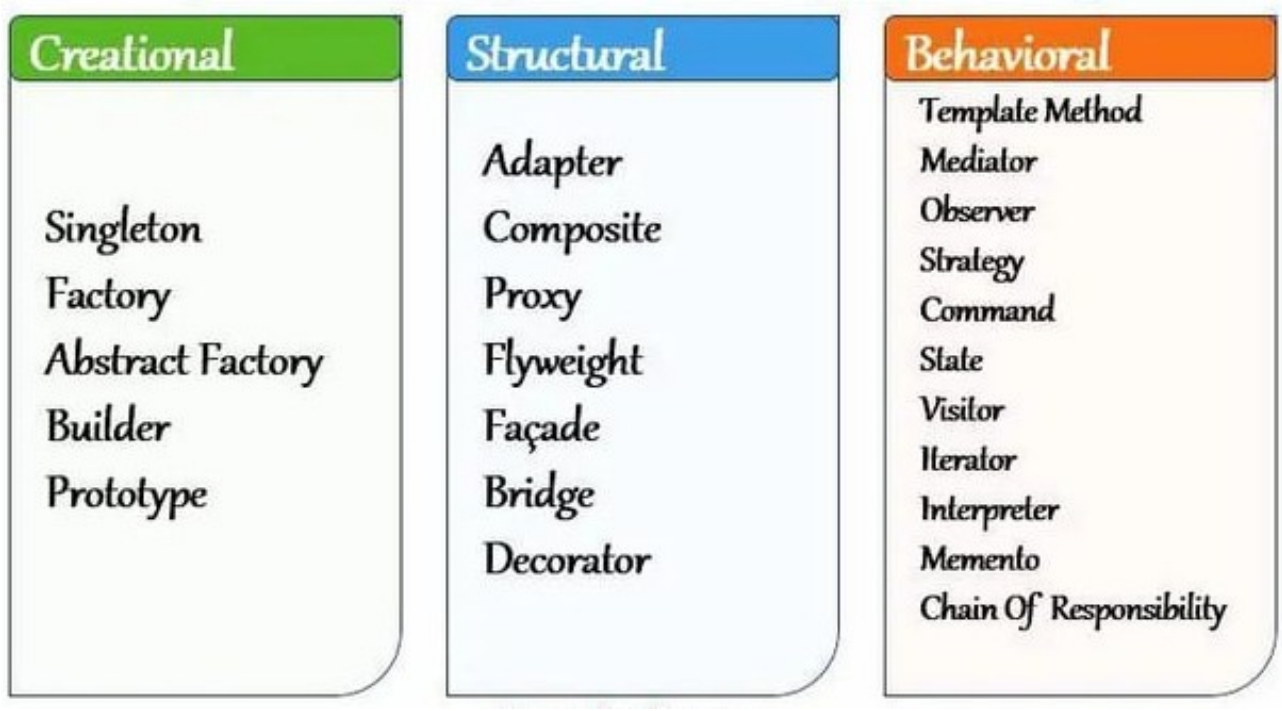
Design patterns

Design patterns are the best solutions to the common problems encountered in software design.

Creational Patterns:
These patterns focus on object creation mechanisms

Structural Patterns:
Structural patterns deal with object composition and class relationships, helping to define how objects are connected to one another.

Behavioral Patterns:
Behavioral patterns focus on how objects interact and communicate with each other



Creational

Singleton

The singleton pattern ensures that only one instance of a class is ever created.

1 – Using "**object**" keyword

```
object MySingleton {

    fun doSomeThing(){
        // your code
    }
}
```

2 – With " **parameters**"

```
class MySingleton private constructor(private val param: String) {

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

        @Synchronized
        fun getInstance(param: String): MySingleton {
           return INSTANCE ?: MySingleton(param).also { INSTANCE = it }
        }
    }
}
```

3 – Using **“ Application Class “** & **“ lazy initializer “**

```
class MyApp: Application() {

    companion object {
        lateinit var appModule: AppModule
    }

    override fun onCreate() {
        super.onCreate()
        appModule = AppModuleImpl(this)
    }
}

interface AppModule {
    val authApi: AuthApi
    val authRepository: AuthRepository
}

class AppModuleImpl(private val appContext: Context): AppModule {
    override val authApi: AuthApi by lazy {
        Retrofit.Builder()
            .baseUrl("https://my-url.com")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create()
    }
    override val authRepository: AuthRepository by lazy {
        AuthRepositoryImpl(authApi)
    }
}
```

Builder

The Builder pattern allows you to create complex objects step-by-step using methds instead of the constructor overloading

Using Kotlin best practices

```
class InputFilterUtility private constructor(private val maxLength: Int?) {
    private val implementedFilter: ArrayList<InputFilter> = ArrayList()
    fun getFilters() = implementedFilter.toTypedArray()

    init {
        if (maxLength != null) {
            implementedFilter.add(InputFilter.LengthFilter(maxLength))
        }
    }

    class Builder {
        private var maxLength: Int? = null
        fun setMaxLength(maxLength: Int) = apply { this.maxLength = maxLength }

        fun build() = InputFilterUtility(maxLength)
    }
}
```

Usage

```
        val inputFilter = InputFilterUtility.Builder()
            .setMaxLength(2)
            .build()

        inputFilter.getFilters()
```

Facade

The facade pattern is used to define a simplified interface to a more complex subsystem functionalities like libraries

Here’s an example for a permission utility library

```
//Facade
class PermissionsUtility(activity: AppCompatActivity) {
    val checkPermission = CheckPermission(activity)
    val settingsOpener = SettingsOpener(activity)

    private var listener: OnPermissionListener? = null

    private val requestPermission =

 activity.registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions(
)) {
     if (it[Manifest.permission.CAMERA] == true) {
        listener?.onPermissionListener(PermissionsIdentifier.CAMERA, true)
        } else if (it[Manifest.permission.CAMERA] == false) {
         listener?.onPermissionListener(PermissionsIdentifier.CAMERA, false)
            }
        }

    fun requestCamera() {
        requestPermission.launch(
            arrayOf(Manifest.permission.CAMERA)
        )
    }

    fun setPermissionsCallBack(implementer: OnPermissionListener) {
        this@PermissionsUtility.listener = implementer
    }
}
```

```
//Helper classe
class CheckPermission(private val context: Context) {

    fun camera(): Boolean {
        return ActivityCompat.checkSelfPermission(
          context, Manifest.permission.CAMERA
        ) ==  PackageManager.PERMISSION_GRANTED
    }
}
```

```
//Helper classe
class SettingsOpener(private val context: Context) {

      fun cameraSettings() {
        val builder = AlertDialog.Builder(context)
        builder.setTitle("Camera permission")
        builder.setMessage(
          "Camera access is denied\nYou can enable it from the settings."
                           )
        builder.setPositiveButton("Go to settings") { dialog, _ ->
            settingsIntent()
            dialog.dismiss()
        }
        builder.setNegativeButton("Cancel") { dialog, _ ->
            dialog.dismiss()
        }
        val dialog = builder.create()
        dialog.show()
    }

    private fun settingsIntent() {
        val intent = Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS)
        val uri: Uri = Uri.fromParts("package", context.packageName, null)
        intent.data = uri
        context.startActivity(intent)
    }
}
```

```
//Helper classes
interface OnPermissionListener {
    fun onPermissionListener(permission: PermissionsIdentifier, granted: Boolean)
}

enum class PermissionsIdentifier {
    CAMERA,
}
```

Usage

```
//Usage
class MainActivity : AppCompatActivity(), OnPermissionListener {
    private lateinit var binding: ActivityMainBinding
    private lateinit var permissionsUtility: PermissionsUtility

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        initPermissionsUtility()

        binding.apply {
            btnCamera.setOnClickListener {
                if (permissionsUtility.checkPermission.camera()) {
                    Toast.makeText(
                      this@MainActivity, "Camera already granted",
                      Toast.LENGTH_SHORT
                    ).show()
                } else {
                    permissionsUtility.requestCamera()
                }
            }
        }
    }

    private fun initPermissionsUtility() {
        permissionsUtility = PermissionsUtility(this@MainActivity)
        permissionsUtility.setPermissionsCallBack(this@MainActivity)
    }

    override fun onPermissionListener(
      permission: PermissionsIdentifier, granted: Boolean
    ) {
        when (permission) {
            PermissionsIdentifier.CAMERA -> {
                if (granted) {
                    Toast.makeText(
                      this@MainActivity, "Camera granted", Toast.LENGTH_SHORT
                    ).show()
                } else {
                    permissionsUtility.settingsOpener.cameraSettings()
                }
            }
        }
    }
}
```