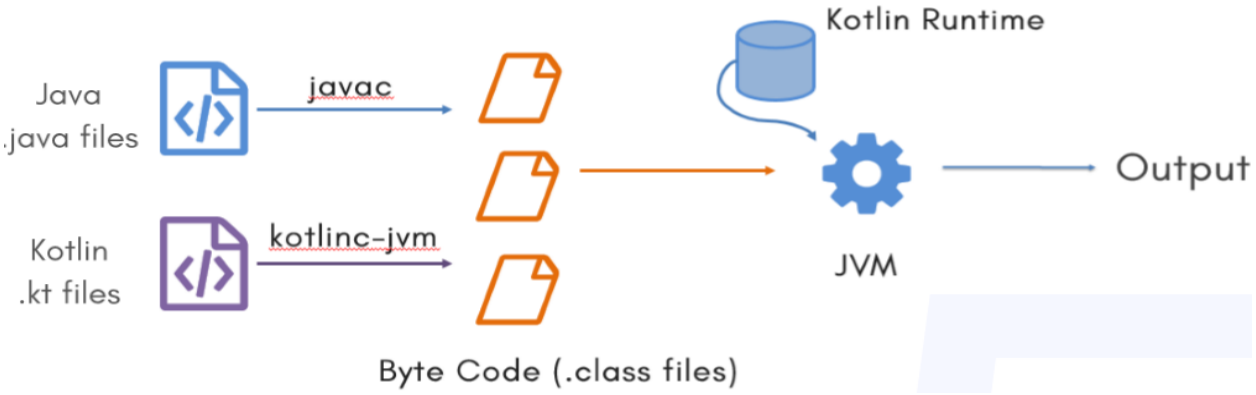
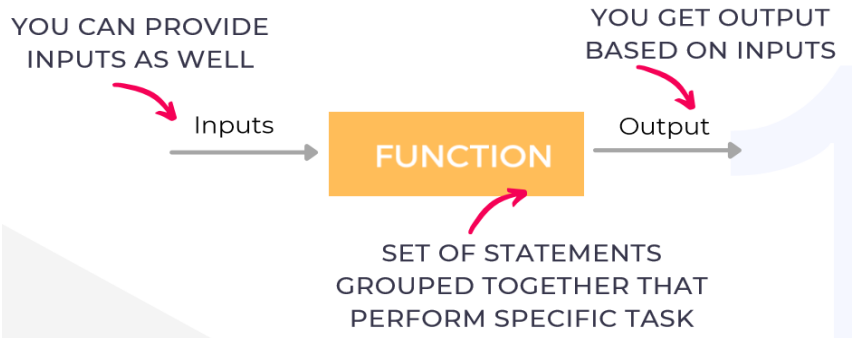
Kotlin





MAIN FUNCTION/METHOD

- When you run your program, JVM looks for this main method and starts executing this method. Main is the entry point of your program.

- If you have multiple Kotlin file, JVM looks for a file where Main method is defined and executes that method.

```
fun main()
{
    println("Welcome To CheezyCode")
}
```

VARIABLES

Variable is a simple box where you store your data. It has a name by which you can access this data.

VAL & VAR

- In Kotlin, variables are either - var and val.

- var can be re-assigned - i.e. new value can be re-assigned in your program.

- val cannot be re-assigned - i.e. once value is assigned to them you cannot change their value.

```
var score = 5;
val playerName = "John";
```

DATA TYPES

As said above, variables are boxes then we need to define the size of the box and the type of data we will be storing in it - Numbers, Strings, Booleans, Characters etc.

```
INTEGER (Byte, Short, Int, Long)
FLOATING POINT ( Float, Double)
BOOLEAN ( True, False)
CHARACTER (Char, String)
```

OPERATORS

Arithmetic Operators

Relational Operators

Logical Operators

ARITHMETIC OPERATORS

```
val i = 13
val j = 2

println( i + j ) // 15
println( i - j ) // 11
println( i * j ) // 26
println( i / j ) // 6 - if both int, result is int
println( i.toFloat() / j ) // 6.5
println( i % j ) // 1 - Remainder
```

RELATIONAL OPERATORS

```
val i = 13
val j = 2

println( i > j ) // 13 > 2 → true
println( i < j ) // 13 < 2 → false
println( i ⩾ j ) // 13 ⩾ 2 → true
println( i ⩽ j ) // 13 ⩽ 2 → false
println( i == j ) // 13 == 2 → false
println( i ≠ j ) // 13 ≠ 2 → true
```

LOGICAL OPERATORS

AND | OR | NOT

&& - AND OPERATOR - If both the conditions are true, then only the result is true.

|| - OR OPERATOR - If any of the condition is true, then result is true.

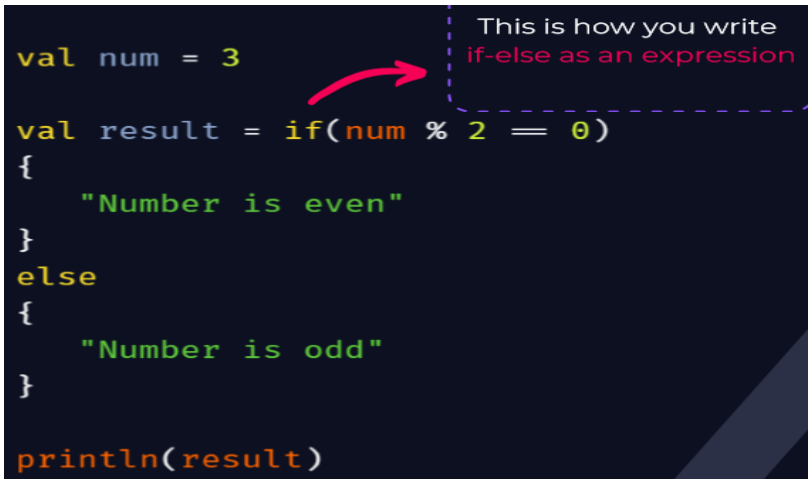
! - NOT OPERATOR - If condition is true, result is false & if condition is false, result is true.

```
val condition1 = true
val condition2 = false

println(condition1 && condition2) // false
println(condition1 || condition2) // true
```

IF-ELSE STATEMENT

```
val num = 3

if(num % 2 == 0)
{
    println("Number is even")
}
else
{
    println("Number is odd")
}
```

IF-ELSE AS EXPRESSION



SHORTER VERSION OF IF-ELSE EXPRESSION

```
val num = 3
val result = if(num % 2 == 0) "Number is even" else "Number is odd"
println(result)
```

Feels like Ternary operator. There is
No Ternary Operator in Kotlin.

NESTING & ELSE-IF LADDER

```
val rating = 4
if(rating > 3){}
else if(rating == 3){}
else{}
```

Else-If Ladder

```
val rating = 4
if(rating > 3){
  if(rating == 5){}
  else{}
}
```

Nested If-Else

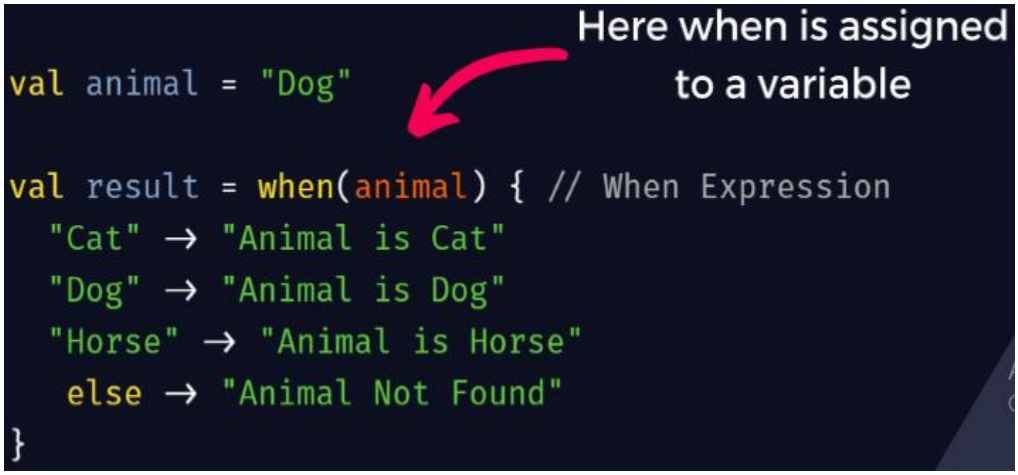
USING IF-ELSE

```
val animal = "Dog"

if(animal == "Cat") {
  println("Animal is Cat")
}
else if(animal == "Dog") {
  println("Animal is Dog")
}
else if(animal == "Horse") {
  println("Animal is Horse")
}
else {
  println("Animal Not Found")
}
```

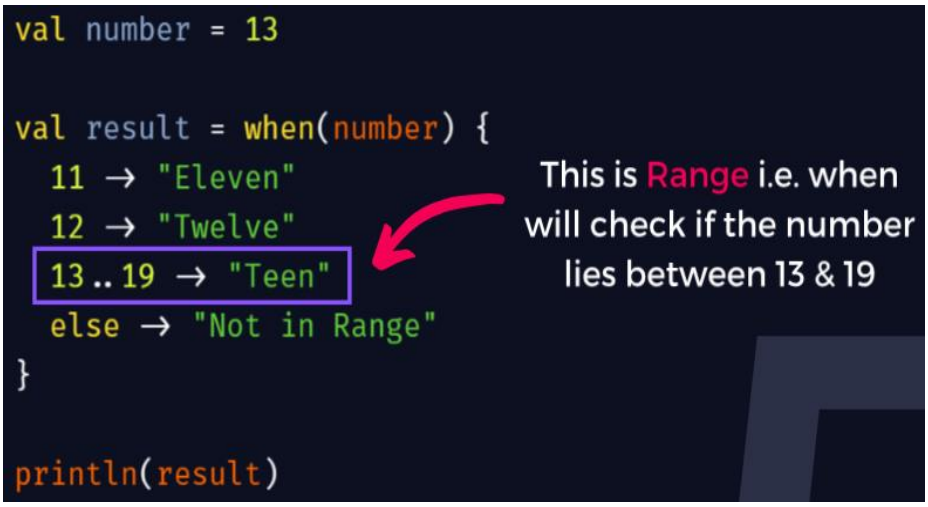
WHEN STATEMENT

```
val animal = "Dog"

when(animal)
{
  "Cat" → println("Animal is Cat")
  "Dog" → println("Animal is Dog")
  "Horse" → println("Animal is Horse")
   else → println("Animal Not Found")
}
```

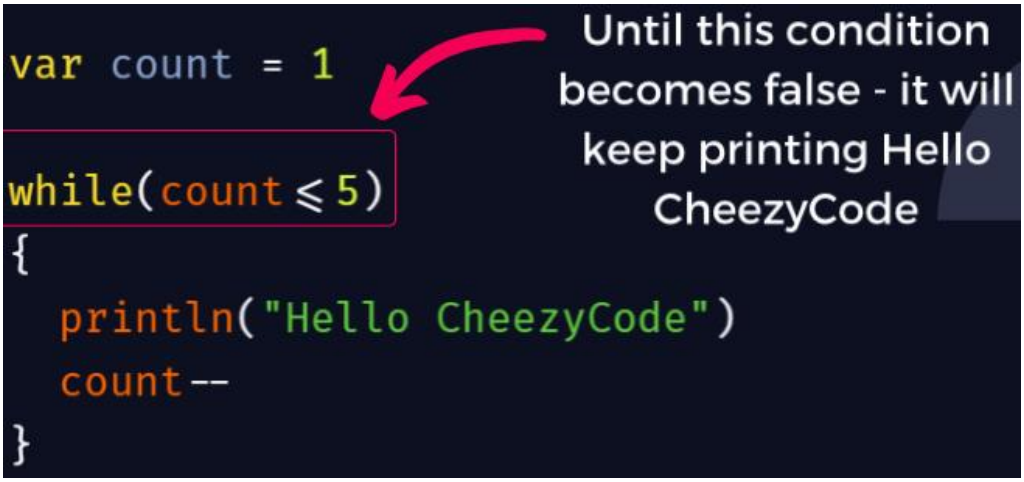
WHEN EXPRESSION



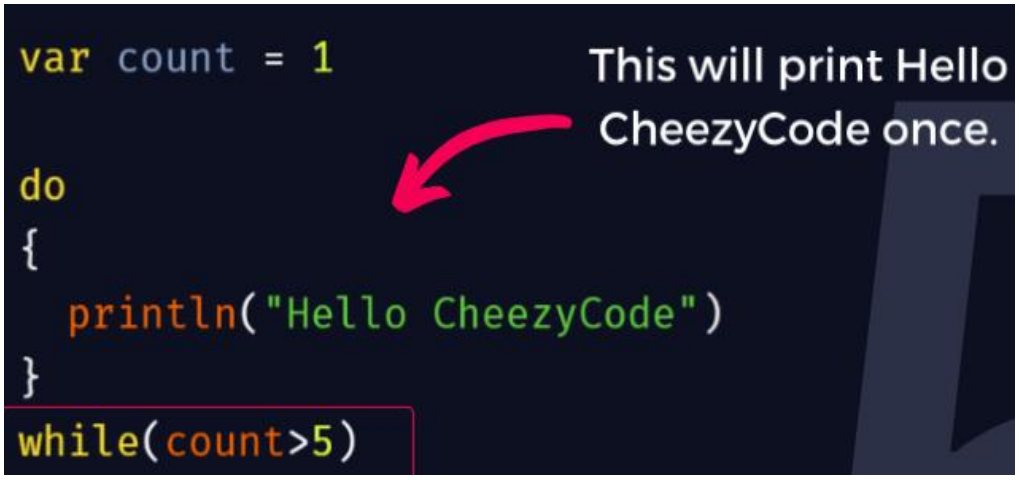
WHEN WITH RANGE



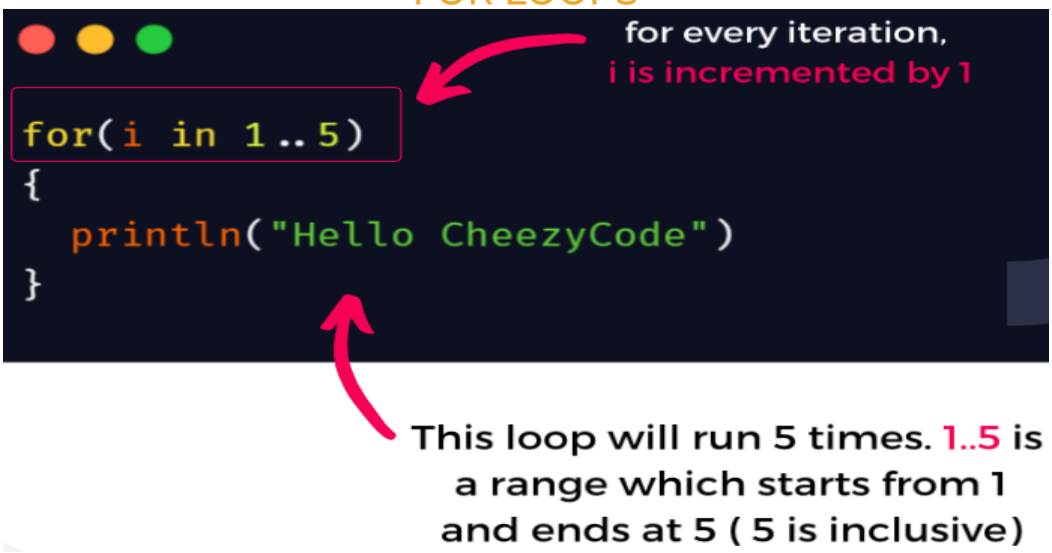
WHILE LOOP



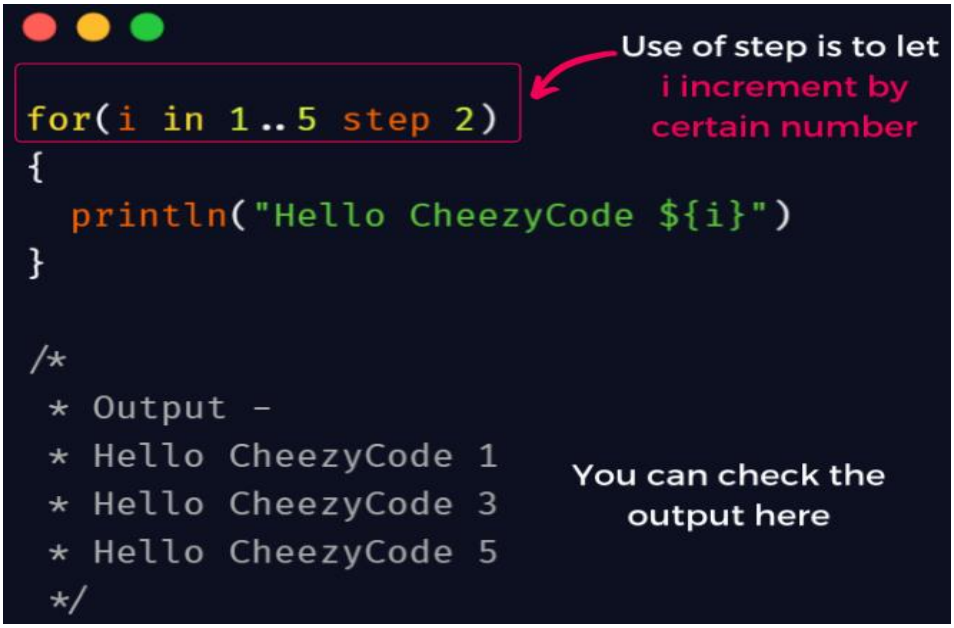
DO - WHILE LOOP



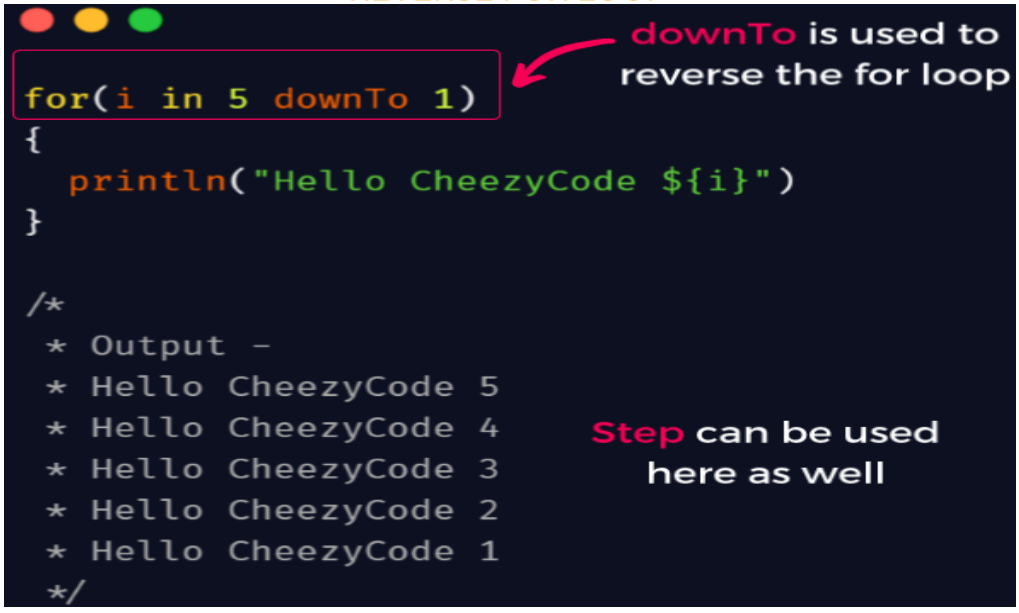
FOR LOOPS



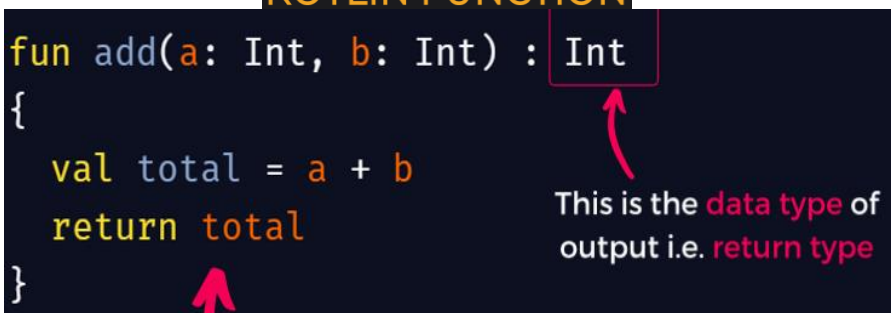
FOR LOOP WITH STEP



REVERSE FOR LOOP



KOTLIN FUNCTION



SINGLE LINE FUNCTIONS

```
fun add(a: Int, b: Int) : Int = a + b
fun add(a: Int, b: Int) = a + b
```

UNIT DATA TYPE

When the function does not return any value.

The return type for that function is Unit

```
fun add(a: Int, b: Int) : Unit
{
  val total = a + b
  println(total)
}
```

Since there is no return value, return type is Unit

This can be simplified as well - No need to write Unit explicitly. Kotlin can infer on its own.

```
fun add(a: Int, b: Int)
{
  val total = a + b
  println(total)
}
```

No return type

DEFAULT VALUE

```
fun printMessage(count = 1)
{
  for(i in 1..count)
  {
    println("Hello CheezyCode)
  }
}
```

Default value of count is 1

SOME FACTS

```
fun main()
{
  val result = add(1,2)
}

fun add(a: Int, b: Int) : Int
{
  return a + b
}
```

1 & 2 are arguments

a & b are parameters

These parameters (a, b) in Kotlin are VAL
You cannot change the value of these parameters inside this add method.

FUNCTION OVERLOADING

Multiple methods with the same name but different parameters.

```
fun add(a: Int, b: Int): Int
{
    return a + b
}

// Type is different but count is same
fun add(a: Double, b: Double): Double
{
    return a + b
}

// Count is different
fun add(a: Double, b: Double, c: Double): Double
{
    return a + b + c
}
```

Parameter Type is different

Parameters Count is different

NAMED ARGUMENTS

When you call a function, you can name the arguments i.e. explicitly defining which argument is for which parameter.

```
fun main()
{
    add(a = 2, b = 2)

    add(b = 3, a = 1)
}

fun add(a: Int, b: Int): Int
{
    return a + b
}
```

We have specified the arguments name.

You can change the sequence as well.

FUNCTION REFERNCE OPERATOR

In Kotlin, we can store functions in a variable. Just like we store values like int, double or string, we can also store functions inside a variable.

```
fun main()
{
    val fn = ::add
    val result = fn(1,2)
    println(result) //3
}

fun add(a: Int, b: Int): Int
{
    return a + b
}
```

Add function is stored in a variable fn. Now this add function can be called using fn as well.

You can use Reference Operator (: :) to store function in a variable

FUNCTION TYPE

If you can store function in a variable then what is the type of that variable? This is where Function Type comes into picture.

```
fun main()
{
  val fn = ::add
  val gn:(a: Int, b: Int) → Int = ::add
}

fun add(a: Int, b: Int): Int
{
    return a + b
}
```

This is how you define a function type.

Type - Function with 2 Int parameters and return type as Int is accepted

```
fun main()
{
  var gn:(a: Int, b: Int) → Int = ::add
  gn = ::multiply
  println(gn(2,3)) //6
}

fun add(a: Int, b: Int): Int
{
    return a + b
}

fun multiply(a: Int, b: Int): Int
{
    return a * b
}
```

Since the multiply function has the same signature as the add function, it can be assigned to this variable.

ARRAY

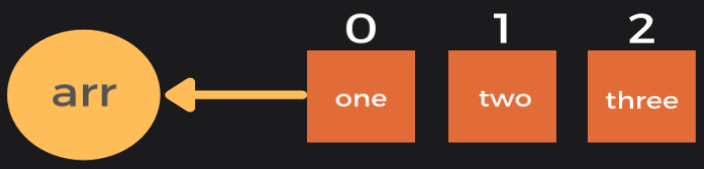
Simple Object that can store multiple values of same type and it has a fixed size.

```
val arr = arrayOf("one", "two", "three")

println(arr[0]) // one
println(arr[1]) // two
println(arr[2]) // three
```

Elements are stored starting from index 0 i.e. first element is stored at location 0, second element is stored at location 1 and so on.

MEMORY REPRESENTATION

val arr = arrayOf ("one", "two", "three")



Collection of items stored at contiguous memory locations i.e. elements are stored sequentially with every items being indexed and index starts from 0

LOOPING THROUGH ARRAY

```
fun main()
{
    val arr = arrayOf("Laptop", "Mouse", "Mic")

    for(el in arr)
    {
        println(el)
    }

    for((i , el) in arr.withIndex())
    {
        println(el + " - " + i)
    }
}
```

Simple For-In Loop

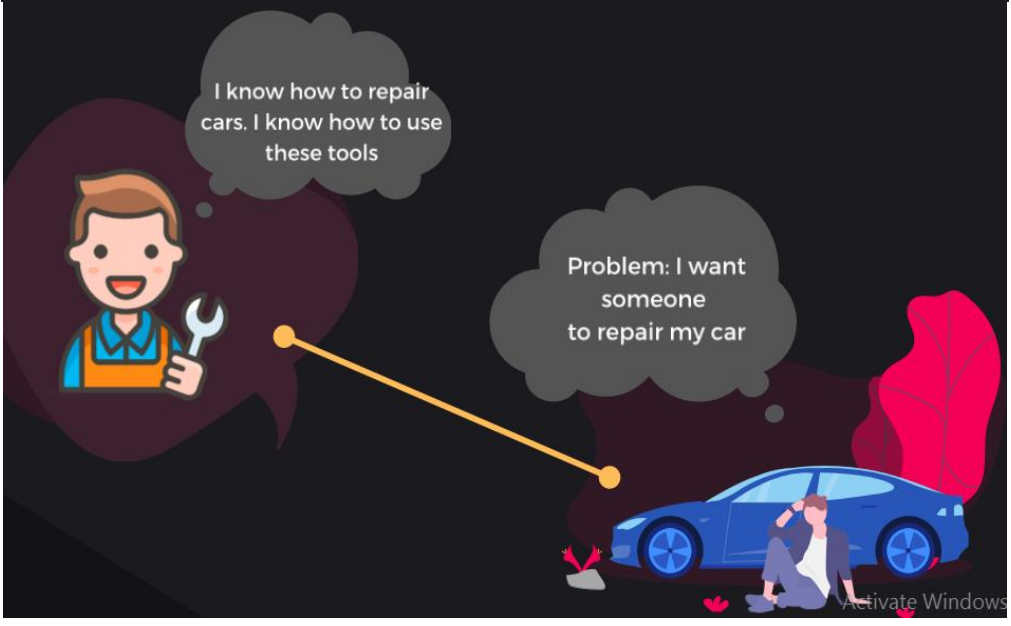
You can access the index of the element as well using withIndex()

ARRAY OPERATIONS

```
fun main()
{
    val arr = arrayOf("one", "two", "three")

    // Length of the array
    println(arr.size) //----> 3

    // Get First Element
    println(arr.get(0)) //----> one

    // Set First Element
    arr.set(0, "zero") //----> ("zero, "two", "three)

    // Check If Array Contains Particular Element
    println(arr.contains("one")) //----> false

    // Index Of Particular Element
    println(arr.contains("three")) //---->true

    //If you try to access index that does not exist
    println(arr[10]) //--->java.lang.ArrayIndexOutOfBoundsException
}
```

OBJECT ORIENTED PROGRAMMING

Object oriented programming is a paradigm to solve problems using objects. Objects interact with each other to solve a problem. This is similar to how we solve problems in real life.



Problem - Need to repair Car

Objects -

- Car
- Person Object(Me & Car Mechanic)
- Car Repairing Tools

BUT HOW DO WE CREATE OBJECTS?

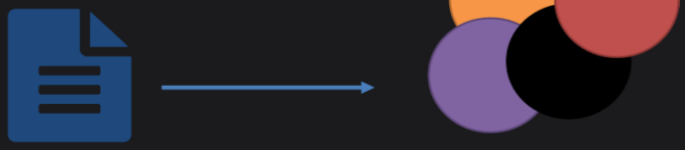
To create objects, we first need to define a blueprint or template which will define how the objects will look like.

These are the actual objects

This defines a template or blueprint for the objects i.e. cookies

CLASS & OBJECTS

- Class is a blueprint or template.
- Objects are the real thing.



Class (Template)

Different Objects

WHAT ARE OBJECTS?

Object has 2 things - Properties and Methods

- Properties or Fields (What Object Knows)
- Methods (What Object Does)

CLASS & OBJECTS IN KOTLIN

```kotlin
class Car (val name: String, val type : String) //properties
{

    fun driveCar(){ //methods
        println("$name Car is driving")
    }

    fun applyBrakes(){
        println("Applied Brakes")
    }
}

fun main() {
    val mustang = Car("mustang", "petrol")
    val beetle = Car("beetle", "diesel")

    println(mustang.name) //---> mustang
    println(beetle.name) //---> beetle

    mustang.driveCar() // ---> mustang Car is driving
    beetle.driveCar() // --->beetle Car is driving
}
```

We can access properties inside the methods. Properties define the state of the object and methods behave according to state.

This is how you create Car Objects

Two Objects in Memory will be created. Both are separate entities have their own properties.

USER DEFINED DATA TYPE

We have data types to store Integers, Floating Points, Strings but what if we want to store Student Information or Car Information. We don't have data types to store these information.

In Kotlin, You Can Create User Defined Data Type Using Classes

```
fun main()
{
    val num : Int = 10

    //If we define a Car Class
    //i.e. User Defined Data Type
    //We can do something like this

    val car = Car("Beetle", "Petrol", 100)
}
```

This is how we store Integers Using Int This is a predefined data type.

We can define our own data types i.e. User Defined Data Type to store information we want.

METHODS BEHAVIOR IS DEPENDENT ON OBJECT'S STATE

```
fun main() {
    val mustang = Car("Mustang", 20)
    mustang.startEngine()
    val beetle = Car("Beetle", 0)
    beetle.startEngine()
}

class Car(val name: String, var fuel: Int)
{
  fun startEngine() {
    if(fuel > 0) {
      println("$name - Engine Started")
    }
    else {
      println("$name - Re-fill your fuel tank")
    }
  }
}
```

Engine will only start if it has fuel

Output

CONSTRUCTORS

In objects, we have 2 things - Properties & Methods

To provide default values to these properties, constructors are used

PRIMARY CONSTRUCTOR

```
fun main()
{
    var car = Automobile("Car", "Petrol", 2)
    var car2 = Automobile("Car2",  "Petrol", 4)
}

class Automobile(val name: String, val tyres: Int, val maxSeating: Int)
{
    fun drive(){}
    fun applyBrakes(){}
}
```

These values are passed as Default Values for the object

This is how we define Primary Constructor in Kotlin. Values passed during the creation process are used to initialize these properties.

```
/*
    Here constructor is empty. We have defined properties
    inside the class
*/

class Person()
{
    var name: String = ""
    var age: Int = 0
}
```

In both of these cases, Person has only 2 properties - name & age

```
/*
    Just like methods, you can define parameters to
    initialize values of the properties
*/

class Person(nameParam: String, ageParam: Int){
    val name: String = nameParam
    val age: Int = ageParam
}
```

These are simple parameters. If you don't use val or var - they are not considered as properties

```
fun main() {
    val person1 = Person("John", 20)
    println(person1.name) //John
    println(person1.canVote) //true
}

class Person(val name: String, ageParam : Int)
{
    val canVote = ageParam > 18
}
```

Here we have defined name as val which is a property whereas ageParam is just a parameter. This parameter is used to define another property canVote

INITIALIZER BLOCK

Primary Constructor does not have any code.
To write initialization code, we can use init block

```
fun main()
{
    val person1 = Person("John", 20)
}

class Person(val name: String, val age : Int)
{
    init
    {
      println("Object Created - Init Block #1")
    }

    // We can define multiple init blocks
    // They are executed in order
    init
    {
      println("Init Block #2")
    }
}
```

SECONDARY CONSTRUCTOR

When you want to provide different ways of creating an object - you can use secondary constructor.

```
fun main()
{
    val car1 = Automobile("Car1", "Petrol")
    val car2 = Automobile("Car2", 2, "Diesel")
}

class Automobile(val name: String, val seats: Int, val engineType: String)
{
    constructor(nameParam: String, engineParam: String) :
            this(nameParam, 4, engineParam)

    fun drive(){}
    fun applyBrakes(){}
}
```

2 ways to create Automobile objects

This is how you define secondary constructor.
Make sure to call primary constructor using this

LATEINIT IN KOTLIN

When you don't know the initial value for a variable, you can mark the property as lateinit meaning - Late Initialization.

```
class Person
{
  var gender: String //Error - Property must be initialized or be abstract

  //if you don't know the value at compile time
  //you can mark the property as lateinit

  lateinit var gender2: String // This will work fine
}
```

Informing Kotlin that this property will be initialized later. You don't know the value at this point.

GETTERS & SETTERS

If you want to execute any logic before setting or getting any property value, you can use getters and setters.

```
fun main() {
    var p1 = Person("John", 21)
    p1.age = -8
}

class Person(nameParam: String, ageParam: Int)
{
    var name: String = nameParam
    var age: Int = ageParam
}
```

Here we can set a negative age as well - which is incorrect.

```
fun main() {
    var p1 = Person("John", 21)
    p1.age = -8
}

class Person(nameParam: String, ageParam: Int)
{
    var name: String = nameParam
    var age: Int = ageParam
    set(value)
    {
        if(value > 0)
        {
            field = value
        }
        else{
            println("Age can't be negative")
        }
    }
}
```

We can define a setter like this. Before setting the value, this logic will be executed which prevents incorrect values.

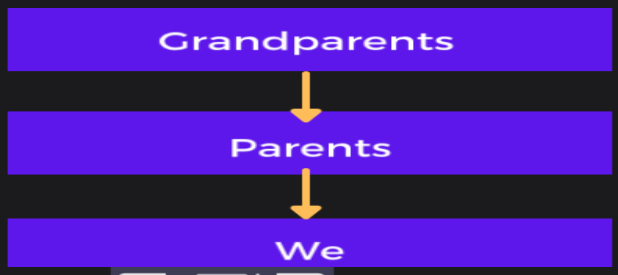
```
fun main() {
    var p1 = Person("John")
    println(p1.name) // Output - - > JOHN
}

class Person(nameParam: String)
{
    var name: String = nameParam
    get(){
        return field.toUpperCase()
    }

}
```

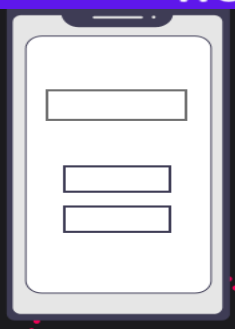
This is how you can use getter if you want to manipulate values while accessing them.

WHAT IS INHERITANCE

We inherit features and characteristics from our parents and grandparents

We can apply this same logic in our classes as well.

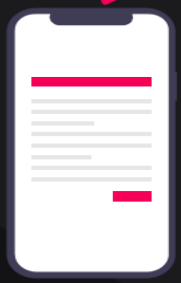




Basic Phone

Basic Phone functionality is written in a class. This class will be inherited by SmartPhones and Tablets.

SmartPhone



Tablet

IS-A RELATIONSHIP

Inheritance is also known as IS-A Relationship. For e.g. Car is a Vehicle, Truck is a Vehicle then Vehicle can be defined as a parent class.


You need to mark class as
OPEN for Inheritance
open class Parent
{
var parentProperty: String = "Parent"
fun parentFn(){
println("Parent Function Is Called")
}
}
This is how you inherit from
parent class
class Child : Parent()
{
var childProperty: String = "Child"
fun childFn(){
println("Child Function Is Called")
}
}
fun main()
{
val child = Child()
You can access the properties
of parent class - just like your
own properties
println(child.parentProperty) // ---> Parent Property
child.parentFn() // ---> Parent Function Is Called
child.childFn() // ---> Child Function Is Called
}

SOME FACTS

- Parent's Constructor is called before Child's constructor.

- You can only inherit from a single class.

- Mark your classes open if you want to inherit them.

OVERRIDING

We can override the properties and behaviors we get from our parent class.

```
fun main()
{
    val square = Square()
    square.display() //⟶ Square is displayed
}

open class Shape
{
  open fun display()
  {
    println("Shape is displayed")
  }
}

class Square : Shape()
{
  override fun display()
  {
    println("Square is displayed")
  }
}
```

Method or Property should be marked as open if you want them to be overridden.

To override method or property in the child class, we need to use Override Keyword

SUPER KEYWORD

You can still call the parent's method from the child class if you want to - using super

```
open class Shape
{
  open fun display()
  {
    println("Shape is displayed")
  }

}

class Square : Shape()
{
  override fun display()
  {
    super.display()
    println("Square is displayed")
  }
}
```

To call the Shape class's display method, super is used.

ANY CLASS

Every Kotlin class has Any as a superclass.

TOSTRING( ) METHOD

```
fun main()
{
    val shape = Shape()
    println(shape.toString())
    println(shape) // behind the scene - it calls toString
}

open class Shape
{
  open fun display()
  {
    println("Shape is displayed")
  }

  override fun toString() : String
  {
    return "I am toString from Shape"
  }

}
```

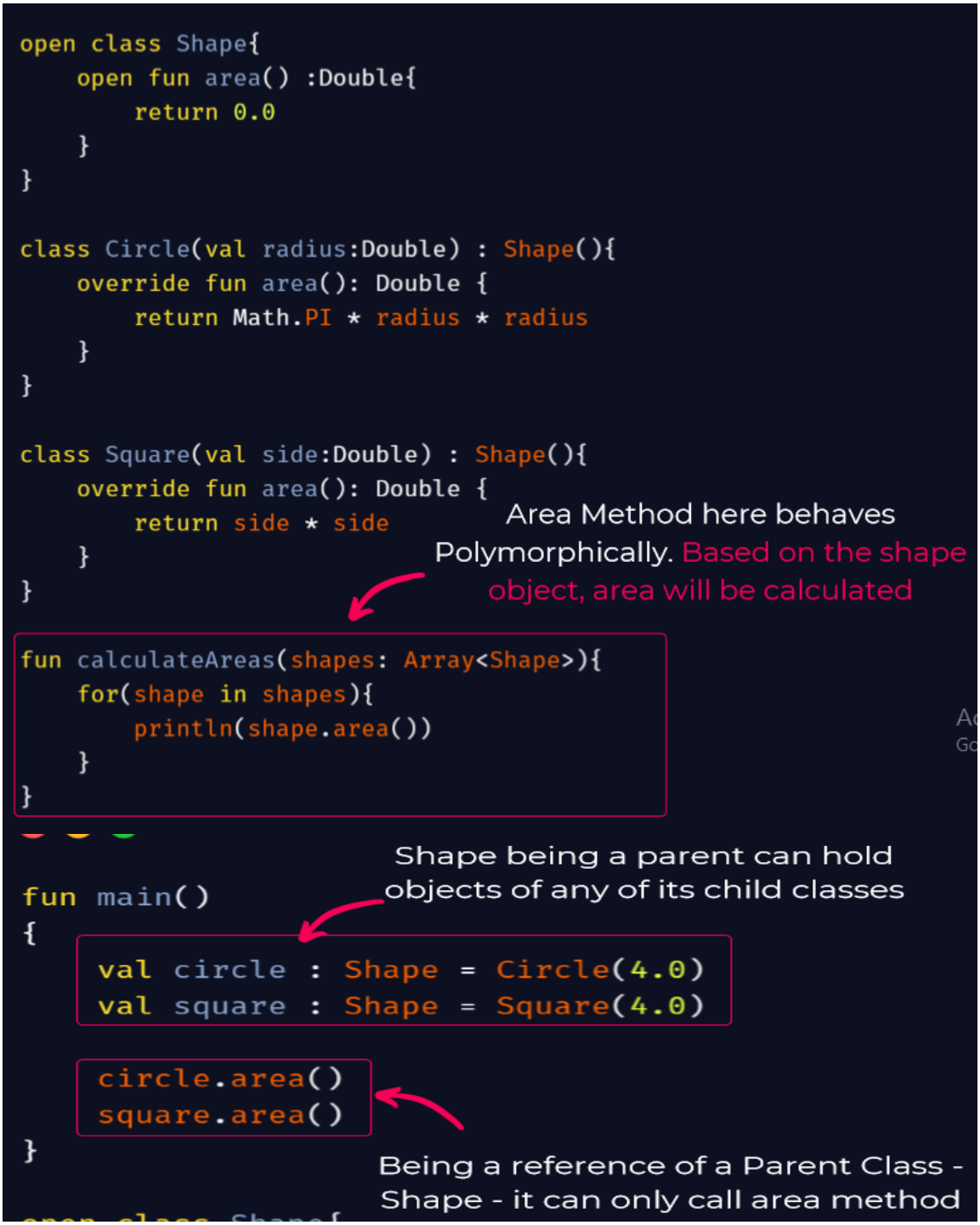
We can override this method - used for printing the string representation of the object

Output

```
I am toString from Shape
I am toString from Shape
```

POLYMORPHISM

- Poly means **Many** & Morph means **Forms**
- In computer science, it is defined as – same method name but it will behave differently based on the object.



ABSTRACT CLASS

Another way to think about this is - Let's say I want you to calculate the area of a shape. Until you don't know the actual shape, you can't calculate its area.

WHAT ARE ABSTRACT METHODS?

Protocol is fine but being a Parent Class, I don't know how to calculate my child shape classes' area. Even I don't know how to display them as well.

Abstract Methods come to rescue

When you don't want to define the body of the functions in the parent class (either you don't know the exact behavior or it does not make sense to have a body in parent class - **You Mark That Method As Abstract Method.**


```
fun main()
{
    val circle = Circle(4.0)
    println(circle.area())
    circle.display()
}

abstract class Shape
{
    abstract fun area() :Double
    abstract fun display()
}

class Circle(val radius:Double) : Shape()
{
  override fun area(): Double =   Math.PI * radius * radius
  override fun display()
  {
      println("Circle is getting displayed")
  }
}
```

This is how we define an abstract class with abstract methods.

SOME FACTS

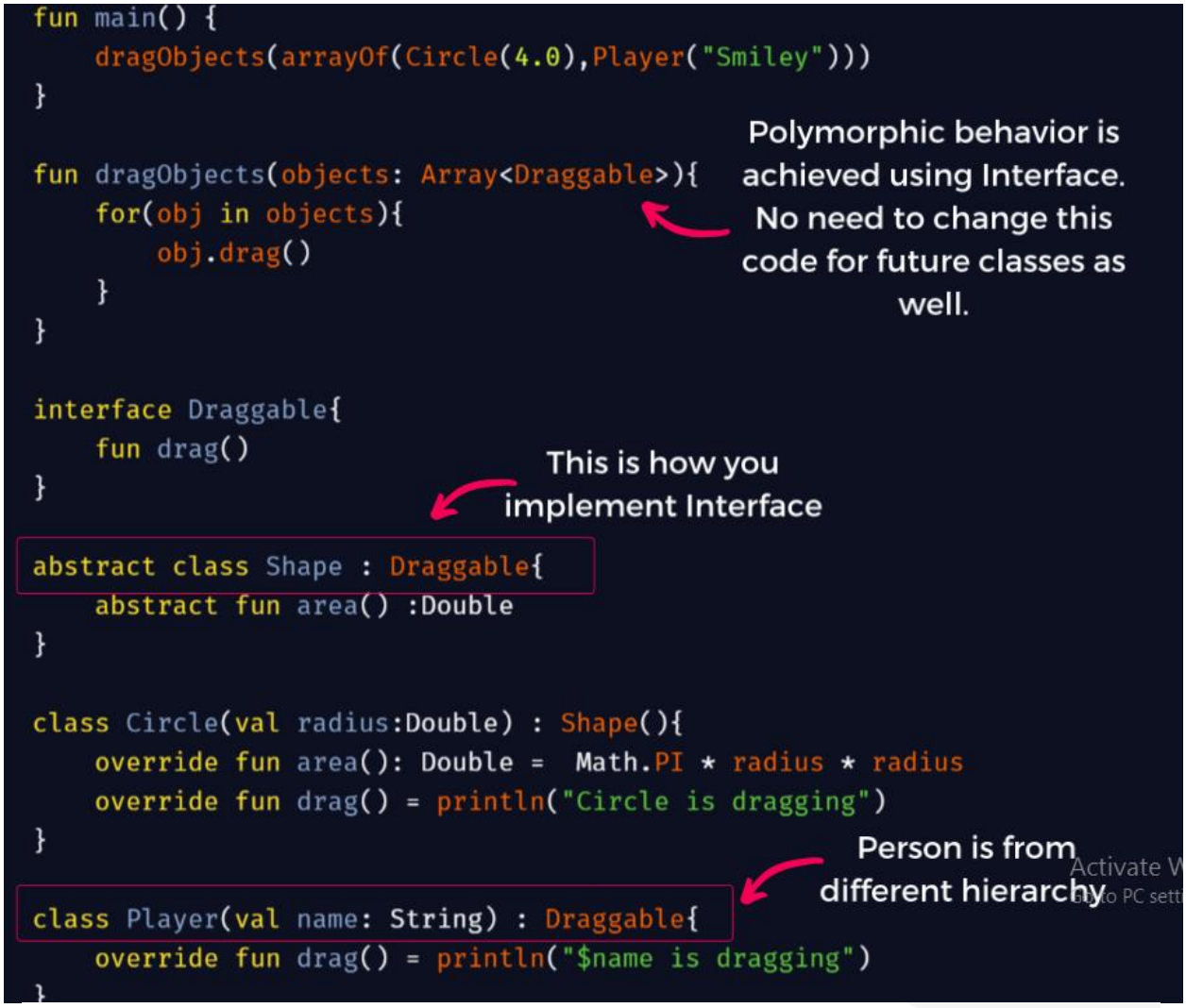
1. Abstract Methods are those methods that do not have any implementation.
2. A class that has an abstract method must be marked abstract.
3. Abstract classes can be defined without any abstract method as well.
4. Abstract classes are meant to be inherited - they are by default open and the same applies to Abstract Method as well. They are meant to be overridden.

INTERFACE

Grouping of classes can be done in 2 ways

- Based on the Hierarchy i.e. What They Are
- Based on the Work i.e. What They Do

```
- Classes grouped based on the hierarchy is known as Inheritance.
- Classes can also be grouped based on the work they do -
  that is done with the help of Interfaces.
```

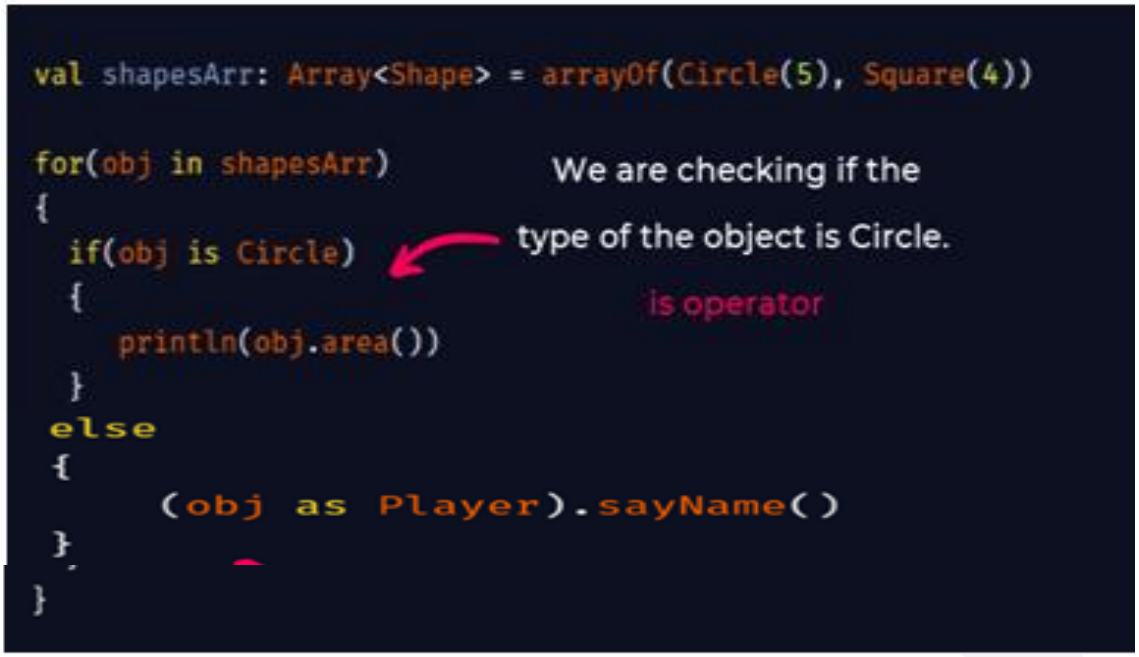


SOME FACTS

1. Interfaces cannot contain state i.e. you cannot define any property.
2. Interfaces are considered 100% abstract but in Kotlin you can define a function with a body that does not depend on the state.
3. A class can implement multiple interfaces but can only inherit from one parent class.

TYPE CHECKING

There are scenarios in which we need to check the type of the objects. For this, we need type checking.



VISIBILITY MODIFIERS

You define the visibility of the variables, methods, classes, interfaces, objects, etc by using these 4 modifiers.

Modifiers	Top Level Declarations	Class Members
public	Everywhere	Everywhere
internal	With in a module	With in a module
private	With in file	With in class
protected	N/A	Subclasses

ENCAPSULATION

- Bundling data and methods that work on that data within one unit is Encapsulation. You mark some fields or methods as private because you don't want others to look into the private matter of the class.

- The benefit of doing this, we can change the implementation or internal representation without breaking the public interface of the class.

THIS IS ALSO CALLED INFORMATION HIDING

OBJECT DECLARATION

- In Kotlin, you can create objects directly without a class i.e. you don't need to define a class to create objects.
- Object keyword is used to create objects directly. A single instance of that custom type is created for you.

```
object Logger
{
  val loggerURL = "https://api.logger.com"
  fun log(type: String, message: String)
  {

  }
}
```

SINGLETONS

- You cannot define a constructor but you can define an init block inside this object.
- You can inherit this object from a class or interface.
- Since a single instance will be created - you can implement Singleton Pattern using Object Keyword.

> **Singleton pattern restricts the instantiation of a class to one "single" instance.**

OBJECT EXPRESSION

- You can create an anonymous object using an object keyword and assign it to a variable.
- Object expressions are useful when you want to do a slight change in the class which is already defined. You just inherit the class and change the method using Object Keyword.

```
open class Test {
  open fun method1() = println("I am original")
}

fun main() {
    val mod = object : Test() {
        override fun method1() = println("I am modified")
    }
    mod.method1()
}
```

Anonymous Object inheriting Test class

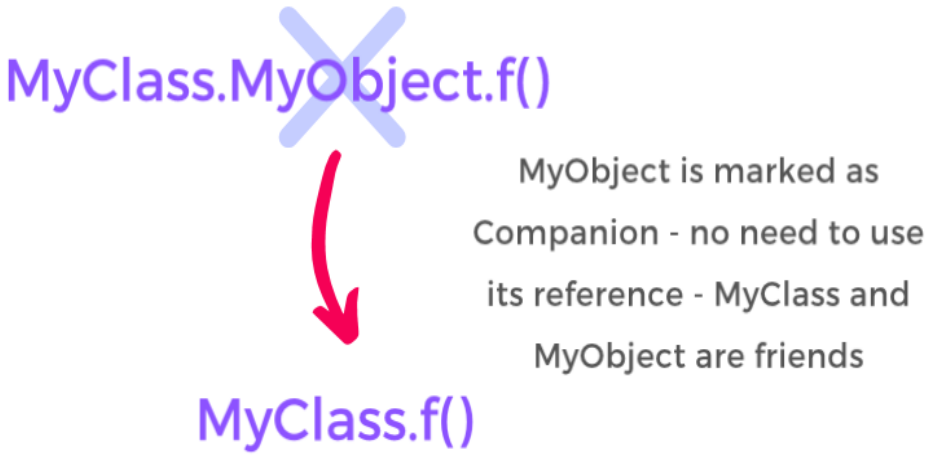
OBJECT EXPRESSION

You can implement the interface right away, no need to create a class and then implement the interface.

```
interface Cloneable
{
  fun clone()
}

fun main()
{
  var obj = object:Cloneable {
    override fun clone() {
        println("I am cloned")
    }
  }
}
```

COMPANION OBJECT

We can mark any one of these objects as a companion & can use its functionality without the object reference

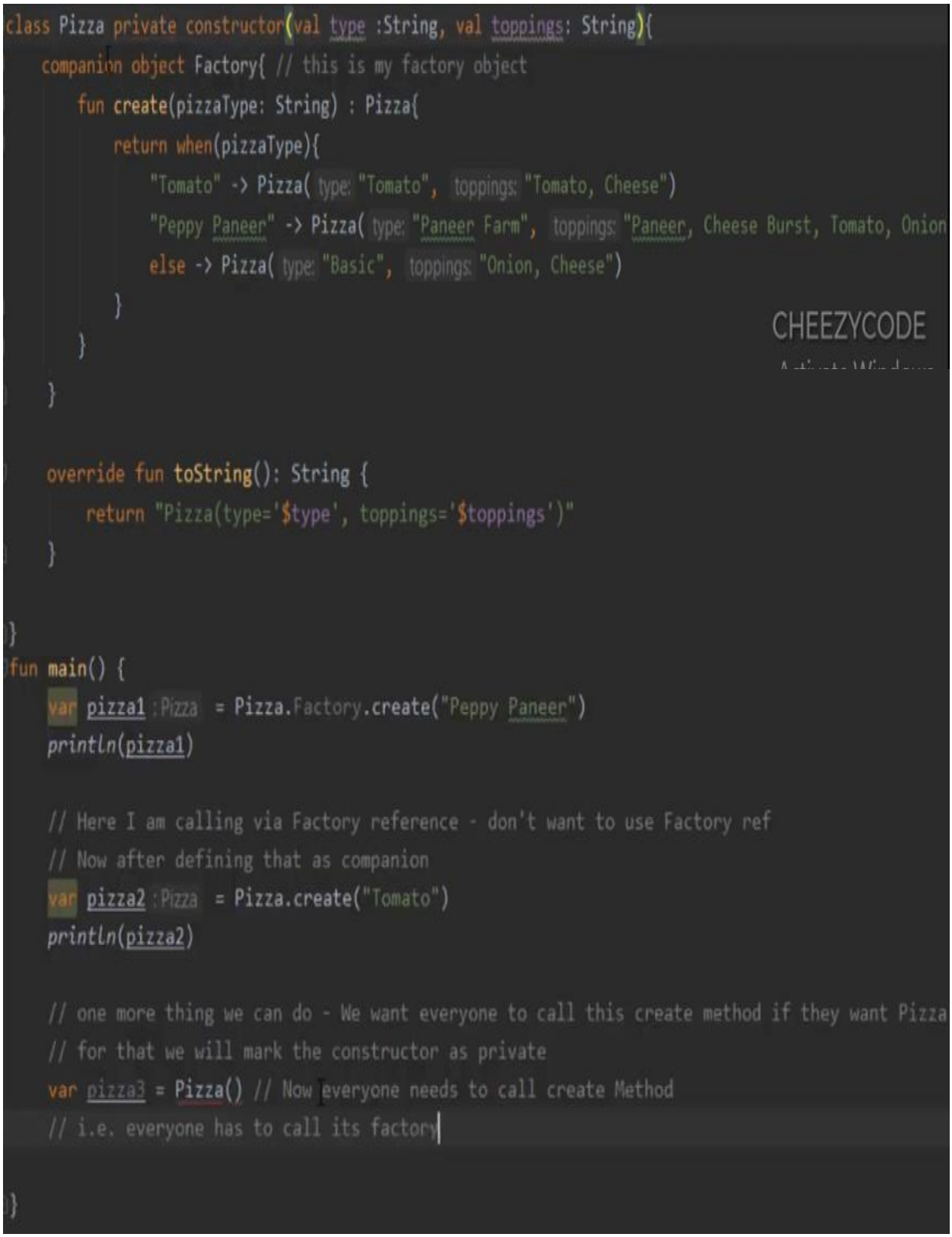


```
fun main()
{
  MyClass.MyObject.f()

  MyClass.f()  // companion object call
}

class MyClass
{
  companion object MyObject {
    fun f() = println("I am F from MyObj")
  }

  object AnotherObject {
    fun g() = println("I am G from Another")
  }
}
```

JVM STATIC

```
fun main()
{
  MyClass.f() // this is static in Java
}

class MyClass
{
  companion object MyObject {
    @JvmStatic
    fun f() = println("I am F from MyObj")
  }
}
```

FACTORY PATTERN
USING COMPANION



DATA CLASS

```
data class Person(val id: Int, val name: String)
```

Primary constructor must have at least one parameter and must be marked as either val or var.

When you define a data class - compiler automatically generates the following methods for you behind the scenes

- equals() - to check equality of objects
- hashcode() - hash code value for the object
- toString() - String representation of object

Two objects having the same data must be equal for data classes. If two objects are equal, they must return same hashcode

```
data class Person(val id: Int, val name: String)

fun main()
{
    val p1 = Person(1, "John")
    val p2 = Person(1, "John")

    println(p1 == p2)
    println(p1) // calls p1.toString()
    println(p1.hashCode())
    println(p2.hashCode())

    val p3 = p1.copy()
    println(p3)

    println(p1.component1()) // component 1 is id
    println(p1.component2()) // component 2 is name
}
```

Output

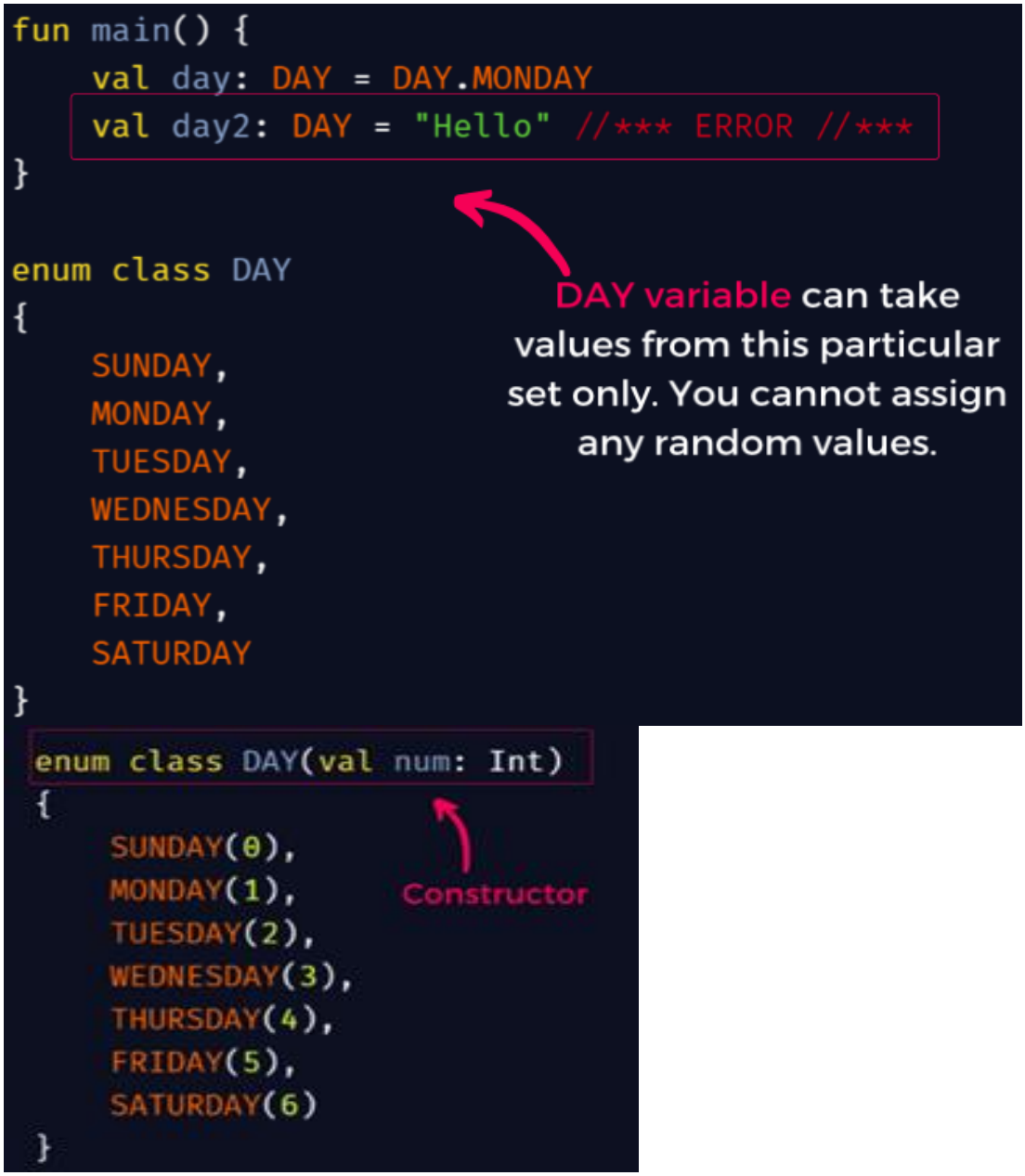
```
true
Person(id=1, name=John)
2314570
2314570
Person(id=1, name=John)
1
John
```

ENUM CLASS

When you want to explicitly define the set of values a variable can take - we define Enums

Every value inside an Enum class is an object.

You can even initialize the value of these objects.



SEALED CLASS

Enum class restricts the values,

Sealed class restricts the type of classes.

```
fun main() {
    val tile: Tile = Red("Mushroom", 25)
    val points = when(tile)
    {
        is Red → tile.points * 2
        is Blue → tile.points * 5
    }
    println(points)
}

sealed class Tile
class Red(val type: String, val points: Int) : Tile()
class Blue(val points: Int): Tile()
```

No need to write else block here, tile can be either Red or Blue

NULLABLE TYPES

There are scenarios where you don't the initial value for a variable.

To indicate the absence of value - NULL is used

```
var gender : String = "Male"

var gender2 : String = null // --- ERROR ---

var gender3 : String? = null
```

Having Question Mark (?) at the end indicates that it could accept null values i.e. Nullable Type

NULL SAFETY

```
var gender : String? = null

gender.toUpperCase() // --- ERROR ---
//1st Way
if (gender != null) {
    println(gender.toUpperCase())
}
```

?. is a safe call operator

```
//2nd Way
println(gender?.toUpperCase())
```

ELVIS OPERATOR

```
var gender: String? = null

val selectedValue = gender ?: "Not Defined"
```

If the value of gender is null then Not Defined is assigned otherwise the value of gender is assigned to the variable.

NON NULL ASSERTED OPERATOR

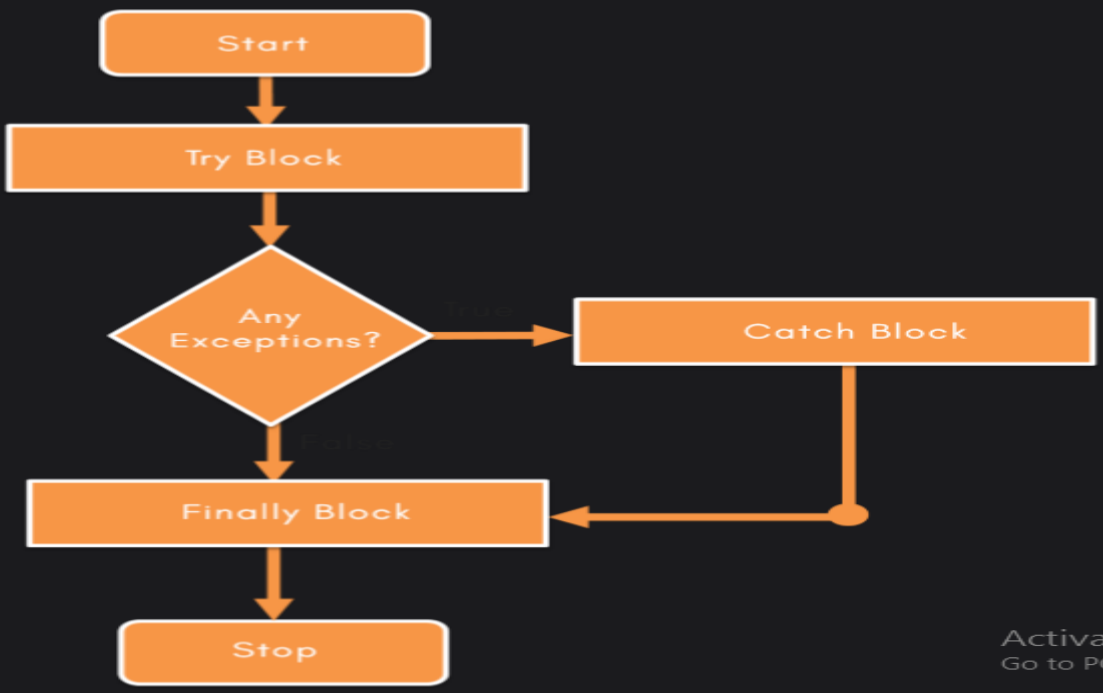
When you are sure that the variable is not going to be null - you assert that it is not null by using !! operator

```
fun main() {
    var gender : String? = "male"
    println(gender!!.toUpperCase())
    gender = null
    println(gender!!.toUpperCase())
}
```

Output

```
MALE
Exception in thread "main" kotlin.KotlinNullP
```

EXCEPTION HANDLING




```
fun main()
{
    val arr = arrayOf(1,2,3)

    try {
        println(arr[5])
    }
    catch(e: Exception) {
        println("Element does not exists")
    }
    finally {
        println("No matter what! I will be executed")

    }
}
```

Index is out of the bounds of array

Output

```
Element does not exists
No matter what! I will be executed
```

THROW KEYWORD

When you do not want to handle the error,
you throw an Exception.

```
fun main() {
    createUserList(5)
    createUserList(-2)
}

fun createUserList(count: Int)
{
    if(count<0)
    {
        // Raise Exception
        throw IllegalArgumentException("Count must be greater than 0")
    }
    else
    {
        println("User list created containing $count users")
    }
}
```

You don't want to handle this scenario, you throw an IllegalArgument Exception

COMMON FACTS

- You can have multiple catch blocks.
- Order of the catch blocks matter, the most specific one should come first, and then your general Exception class.
- You cannot have a try block without a catch or finally. Either define catch or finally or both.

ARRAYS REVISITED

Array is a collection of objects but there are few downsides with it -

- Fixed Size - You cannot change its size
- Mutable - You can change the values inside it.

We need collections that are dynamic in nature & based on the scenario - we want collections to be mutable and immutable.

KOTLIN COLLECTIONS

To overcome these problems, Kotlin has Collections API that provides collections for different needs.

List - Collection of objects (Dynamic array)

Map - Collection of key-value pairs

Set - Collection with no duplicates

All these collections are dynamic and come in both flavors - mutable and immutable

BUILT-IN METHODS

Just like the arrayOf method, we have different methods in Kotlin to create collections.

LIST	MAP	SET
listOf mutableListOf	mapOf mutableMapOf	setOf mutableSetOf

LIST COLLECTION

```
fun main()
{
    val list = listOf(1,2,3)
    list[0] = 2 // ⟶ Error Immutable List ⟵


    val list2 = mutableListOf(1,2,3) // MutableList
    //Element can be added
    list2.add(4) // -- [1, 2, 3, 4]

    //Element can be added at particular index
    list2.add(0, 0) // -- [0, 1, 2, 3, 4]

    //Particular element to be removed
    list2.remove(2) // -- [0, 1, 3, 4]

    //Element at particular index to be removed
    list2.removeAt(0) // -- [1, 3, 4]

}
```

MAP COLLECTION

```
fun main()
{
    val map = mapOf( 1 to "One", 2 to "Two", 3 to "Three")
    map[1] = "CheezyCode" // ---> Error - Immutable Map<---

    val map2 = mutableMapOf( 1 to "One", 2 to "Two", 3 to "Three")

    //Element can be added
    map2.put(4, "Four")
    map2[5] = "Five"
    println(map2)

    //Element can be removed
    map2.remove(5)
    println(map2)

    //Looping Elements
    for (entry in map2)
    {
        println(entry.key.toString() + " " + entry.value)
    }
}
```

Output

```
{1=One, 2=Two, 3=Three, 4=Four, 5=Five}
{1=One, 2=Two, 3=Three, 4=Four}
1 One
2 Two
3 Three
4 Four
```

HIGHER ORDER FUNCTIONS

Functions that accept functions as input
or return a function as output or both are
known as Higher-Order Functions


```
fun main() {
    calculator(2,4, ::sum)
    calculator(2,4, ::mul)
}

fun sum(a: Int, b: Int): Int = a + b
fun mul(a: Int, b: Int): Int = a * b

fun calculator(a: Int, b: Int, op: (Int, Int) → Int) {
    val result =  op(a, b)
    println(result)
}
```

Function Type
must match

Calculator is a Higher Order Function -
it takes a function as an input (op)

LAMBDA EXPRESSION

Lambdas are functions without any name
i.e. Anonymous Function.

```
fun add (a : Int, b : Int) : Int = a + b
```

Converted to Lambda

```
val add = { a: Int, b: Int -> a + b }
```

{ Arguments -> Body }

{ a: Int, b: Int -> a + b }

Data type of the last expression is the return type of the Lambda. So in this case it will be an Integer

Variations

```
val singleParamLambda = { x: Int -> x * x }

val noParamLambda = { println("Hello CheezyCode") }
```

```
fun main()
{
    //Single Parameter Lambda
    val singleParamLambda = { x: Int → x * x }

    // Above lambda can be simplified - Using it
    val simplified: (Int) → Int = { it * it }

    // When you have Lambda as last parameter, you can write it like this
    calculator(1, 2) { a, b → a + b }
}

fun calculator(a: Int, b: Int, op: (Int, Int) → Int) = op(a, b)
```

No need to name the argument here, you can use it

When you have Lambda as the last parameter of the function, you can write the lambda outside the brackets. This is a concept of DSL.

FILTER FUNCTION

```
fun main()
{
  val numList = listOf(1,2,3,4,5)

  // Function Reference Operator
  numList.filter( ::isEven )

  // Anonymous Function
  numList.filter( fun(a: Int) : Boolean { return a % 2 == 0 } )

  // Lambda Expression
  numList.filter( { a: Int → a % 2 == 0 } )

  // Lambda Expression Using It
  numList.filter( { it % 2 == 0 } )

  // Lambda Expression - DSL
  numList.filter { it % 2 == 0 }

}

fun isEven(a: Int) = a % 2 == 0
```

Function passed is evaluated for every item in the list. Item is excluded if function evaluates to false.

MAP FUNCTION

Mapping of objects from one form to another form. You pass a transformation function that transforms every object in the list

```
fun main()
{
  val numList = listOf(1,2,3,4,5)
  val squareList = numList.map { it * 2 }
  println(squareList)


  val personList = listOf ( Person("John", 22), Person("Jason", 17) )
  val modifiedList = personList.map {
    ModifiedPerson(it.name, it.age, it.age > 18)
  }
  println(modifiedList)
}

data class Person(val name: String, val age: Int)
data class ModifiedPerson(val name: String, val age: Int, val canVote: Boolean)
```

FOR EACH

To loop through each item in the list

```
fun main()
{
  val numList = listOf(1,2,3,4,5)

  numList.forEach{
      println("Number is $it")
      println("Square of the number is ${it * it}")
  }
}
```

No need to write separate for loop - you can use List's collection forEach function

EXTENSION FUNCTION

We call this extension function just like other string methods.

```
fun main() {
    println("Hello CheezyCode".formattedString())
}

fun String.formattedString(): String{
    return "*****\n$this\n*****"
}
```

Here we have added a new function to the String class where we just do custom formatting of the message.

INLINE FUNCTION

- For each Lambda function - Kotlin generates a class for you behind the scenes.
- If you define multiple lambda functions, this will increase memory usage and impact performance.
- To solve this, you can instruct the compiler to just replace the calling of a lambda with the lines of code written inside the lambda.

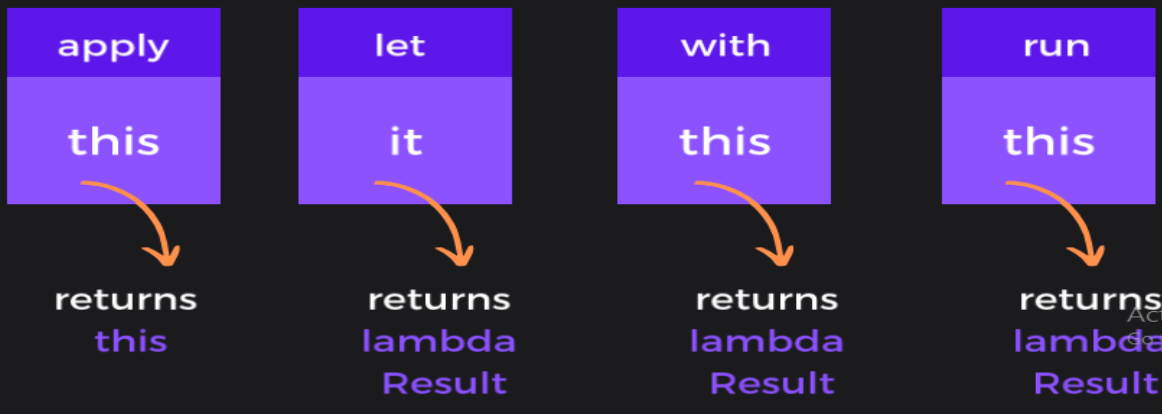

```kotlin
fun main()
{
    lambda(1,2)
    lambda(2,4)
    lambda(3,6)
}

inline fun lambda(a: Int, b: Int) = println(a + b)
```

This is just for demonstration.
Using inline with a function just replaces the call with the body of the inline function.

```kotlin
fun main()
{
    val a1: Int = 1
    val b1: Int = 2
    println(a + b)

    val a2: Int = 2
    val b2: Int = 4
    println(a + b)

    val a3: Int = 3
    val b3: Int = 6
    println(a + b)

}
```

UTILITY FUNCTIONS

Kotlin has different utility functions that are useful in writing concise code.



APPLY FUNCTION

```
fun main() {
    val emp: Employee = Employee()
    emp.age = 20
    emp.name = "John"

    //Apply Function
    emp.apply {
        age = 20
        name = "John"
    }
}

data class Employee (
  var name: String = "",
  var age: Int = 18
)
```

this Reference is available inside this lambda

LET FUNCTION

```
fun main() {
    val emp: Employee = Employee()
    emp.age = 20
    emp.name = "John"

    //let Function
    emp.let {
        it.age = 20
        it.name = "John"
    }
}

data class Employee (
  var name: String = "",
  var age: Int = 18
)
```

it is used inside this lambda to access the properties.

WITH FUNCTION

```
fun main() {
    val emp: Employee = Employee()
    emp.age = 20
    emp.name = "John"

    //with Function
    with(emp){
        age = 30
        name = "XYZ"
    }
}

data class Employee (
  var name: String = "",
  var age: Int = 18
)
```

this Reference is available inside this function

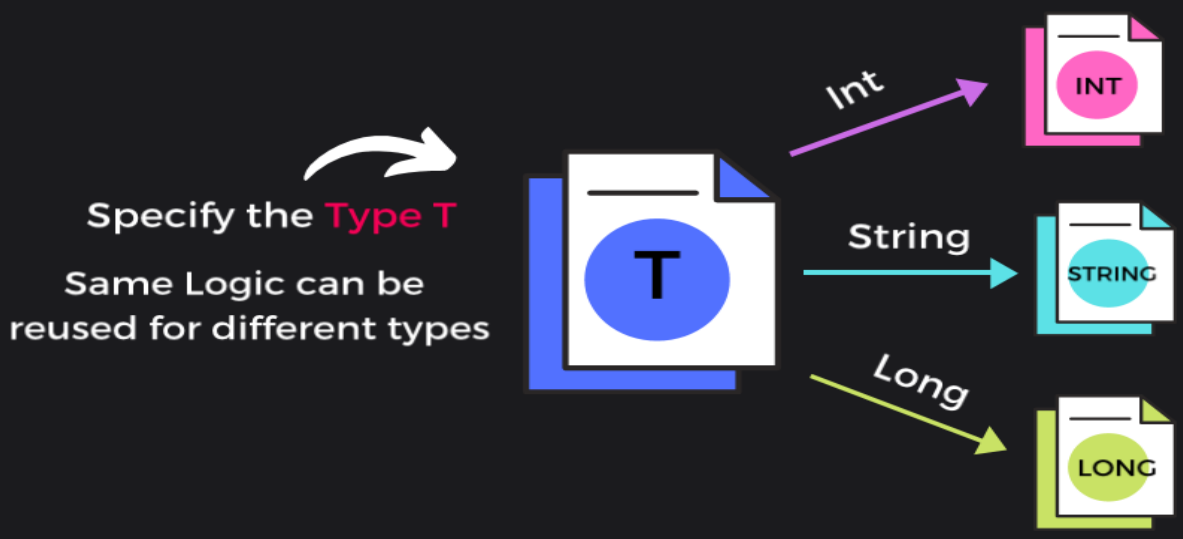
RUN FUNCTION

```
fun main() {
    val emp: Employee = Employee()
    emp.age = 20
    emp.name = "John"

    //run Function
    emp.run{
        age = 35
        name = "PQR"
    }
}

data class Employee (
  var name: String = "",
  var age: Int = 18
)
```

this Reference is available inside this function

GENERICS

- Writing General Logic which is not dependent on the type i.e. type can be specified later.
- Type is passed later as a parameter when that logic is needed for the specific type.




```
class Conatiner<T>(var data: T)
{
    fun setValue(value : T)
    {
        data = value
    }
    fun getValue(): T
    {
        return data
    }
}
```

All the logic is written using a placeholder Type T, you pass this type T when you use this class

NESTED CLASS

You can define classes inside another class
i.e. classes can be nested in other classes

```
class Outer
{
    var i = 0

    class Nested
    {
        fun test()
        {
            println("I am in nested class")
        }
    }
}
```

INNER CLASS

A nested class marked as inner can access the
members of its outer class.

```
fun main()
{
    val inner = Outer().Nested()
    inner.test()
}

class Outer
{
    var i = 0
    inner class Nested
    {
        fun test()
        {
            println("I am in nested class $i")
        }
    }
}
```

Inner class holds a reference
of Outer Class Object.

Note - You cannot access
Inner Class directly