LEARN KOTLIN EASILY

KOTLIN FOR BEGINNERS

BY CHEEZYCODE

TABLE OF CONTENTS

CHEEZYCODE

KOTLIN INTRODUCTION

CHEEZYCODE

1

WHAT IS KOTLIN?

- New programming language by JetBrains.

- Modern replacement of Java.

- General purpose programming language that targets JVM, Android, Native & JS.

- Powerful language that can be used to target multiple platforms and multiple environments.

HISTORY OF KOTLIN

- 2011, JetBrains announced Kotlin.

- 2012, Kotlin became Open Source.

- 2016, Kotlin 1.0 launched.

- 2017, Google IO announced Kotlin Android Support.

- 2019 - Android became Kotlin First.

FEATURES OF KOTLIN

- Statically Typed Language - Type Checking done at compile time.
- Supports Object Oriented and Functional Style Programming
- 100% Interoperable with Java
- Safe Language, Concise, Powerful

KOTLIN DEVELOPMENT ENVIRONMENT

CHEEZYCODE

2

WAYS TO RUN KOTLIN CODE

If you don't want to install any software,

just go to Kotlin Playground and run your

Kotlin Code Online.

https://play.kotlinlang.org/

CHEEZYCODE

Your Code Editor

Run Your Code By Pressing This

Kotlin

Kotlin Playground is an online sandbox to explore Kotlin programming language. Browse code samples directly in the browser

```
/**
 * You can edit, run, and share this code.
 * play.kotlinlang.org
 */

fun main() {
    println("Hello, world!!!")
}
```

Sample Hello World Program in Kotlin

CHEEZYCODE

SETUP LOCAL DEV ENVIRONMENT

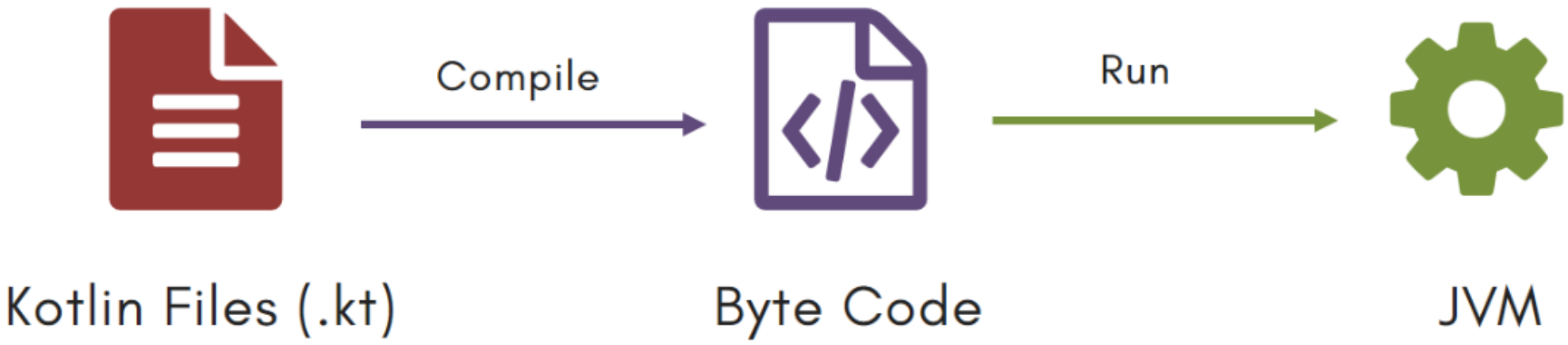
- For development, we need -

    1. Java (atleast JDK 8)
    2. Intelli J Idea (Community Edition)

KOTLIN - JVM LANGUAGE

- Kotlin being a JVM Language - runs on JVM i.e. it is compiled down to Bytecode which then runs on JVM.
- JDK includes JVM and other Java Libraries that are required to run Kotlin if you are targeting JVM

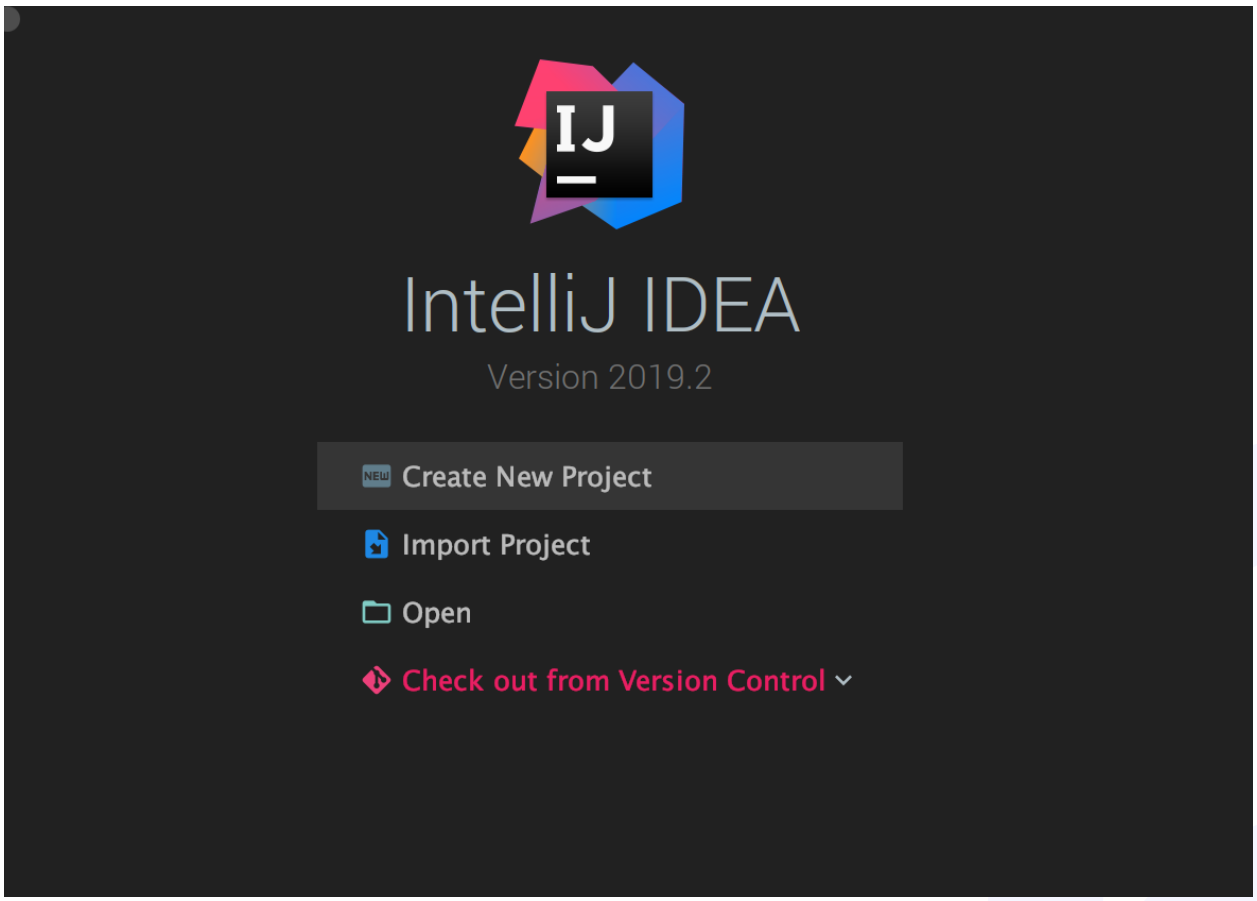
KOTLIN WITH JVM



JDK = Java Development Kit

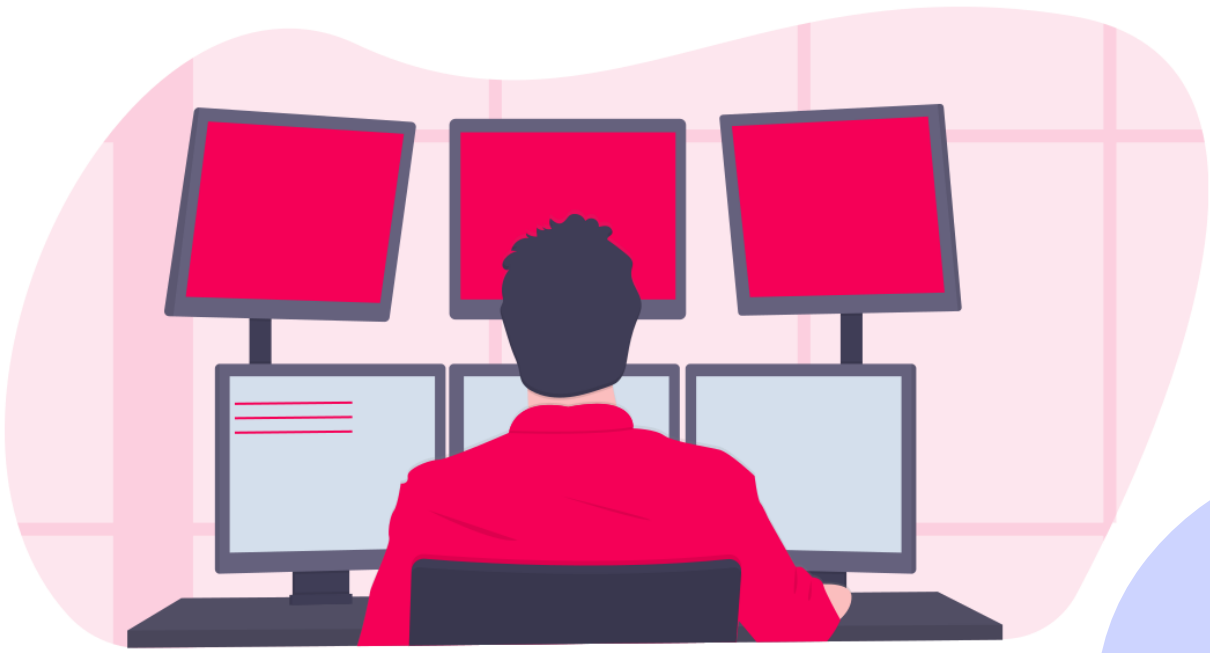
JDK = JVM + Other Libraries and exe

MOST COMMON IDE USED FOR KOTLIN DEVELOPMENT



CHEEZYCODE

KOTLIN HELLO WORLD PROGRAM & COMPILATION

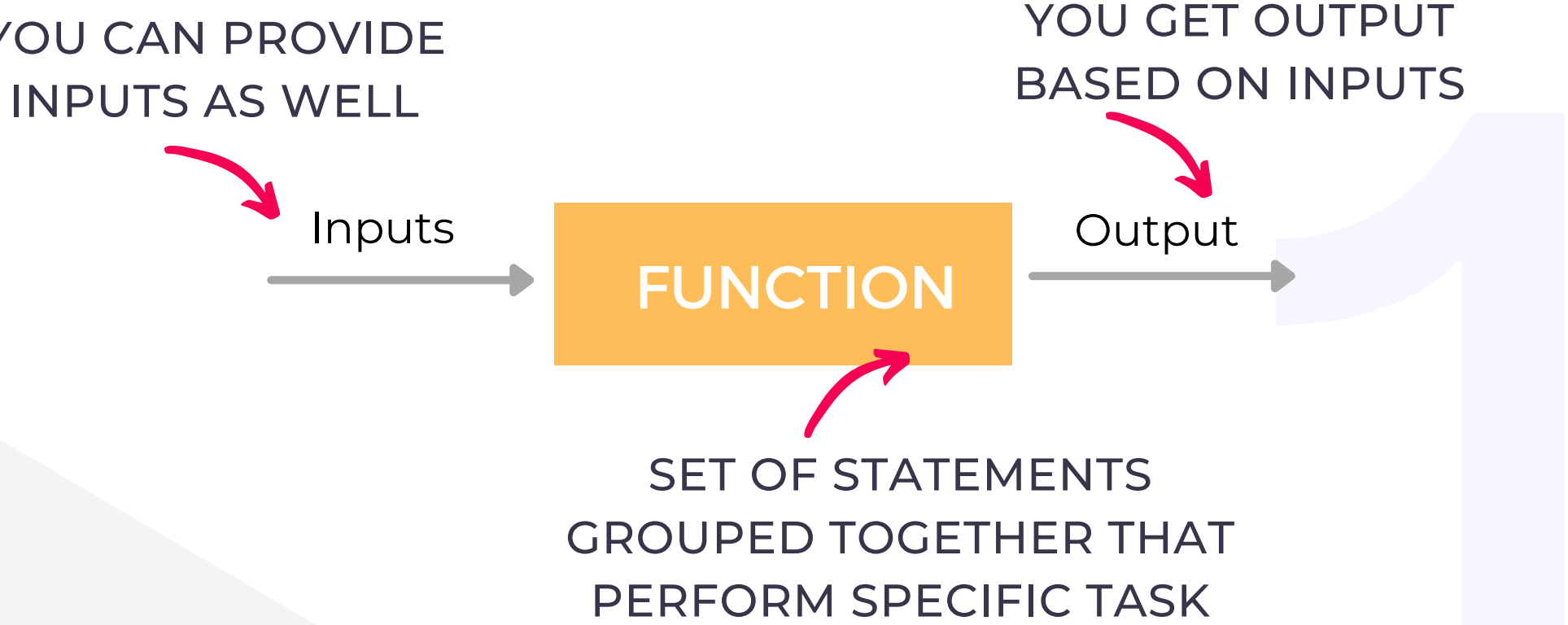


CHEEZYCODE

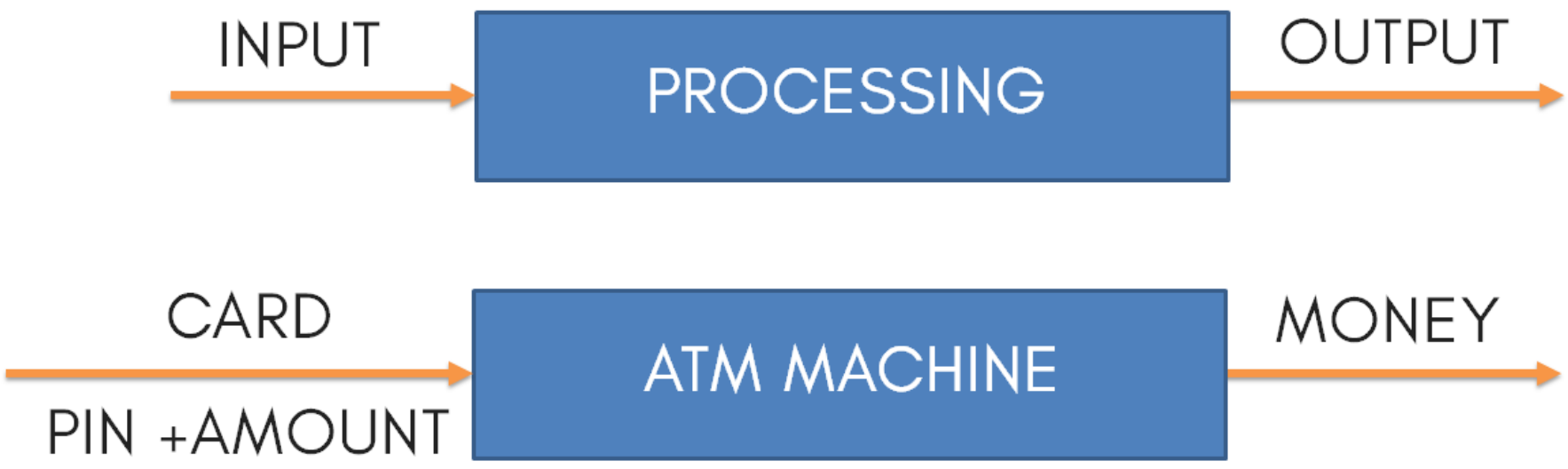
3

Before writing Hello World program, we need to understand - What are methods/functions?

Function is a set of statements (instructions) i.e. block of code that perform some task.



FUNCTIONS/METHODS



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.

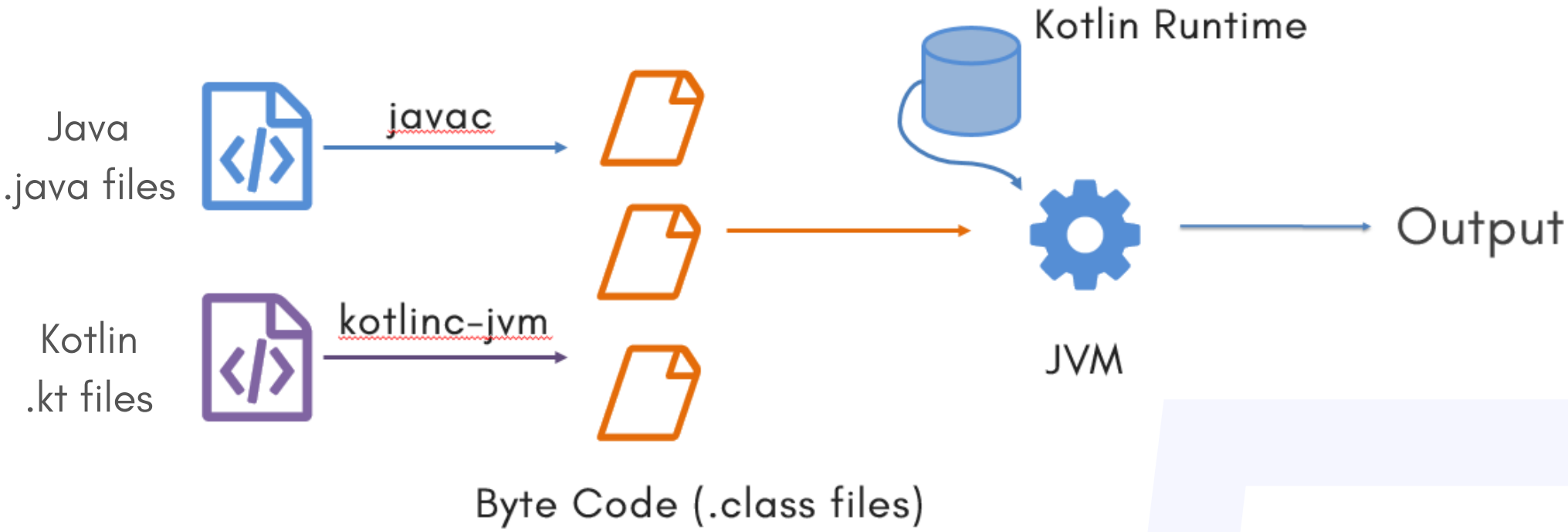
- If you have multiple Kotlin file, JVM looks for a file where Main method is defined and executes that method.

HELLO CHEEZYCODE PROGRAM

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

This is main method. Only instruction defined in this main method is to print Welcome To CheezyCode. You can write multiple statements inside this main method.

COMPILATION



KOTLIN VARIABLES & DATA TYPES

CHEEZYCODE

4

VARIABLES

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

SIMPLE EXAMPLE

- In a game you have a score. When the game progresses your score is updated.

 - You take some power, score is incremented by 5, you kill an enemy score is incremented by 10

 - This score is stored as a variable which keeps on changing as the game progresses.

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

SNOWBROS GAME

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.

Example - Player Name will remain same once you start the game, so it should be val. Score keeps changing so it should be defined as var.

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 IN KOTLIN

CHEEZYCODE

5

OPERATORS

Just like any other programming language, operators are almost similar in Kotlin. Some new operators (?. , !! etc.) have been added, which will be covered as required.

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

These operators allow a program to make a decision based on multiple conditions. Conditions are combined using AND (&&) or OR(||)

```
val condition1 = true
val condition2 = false

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

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.

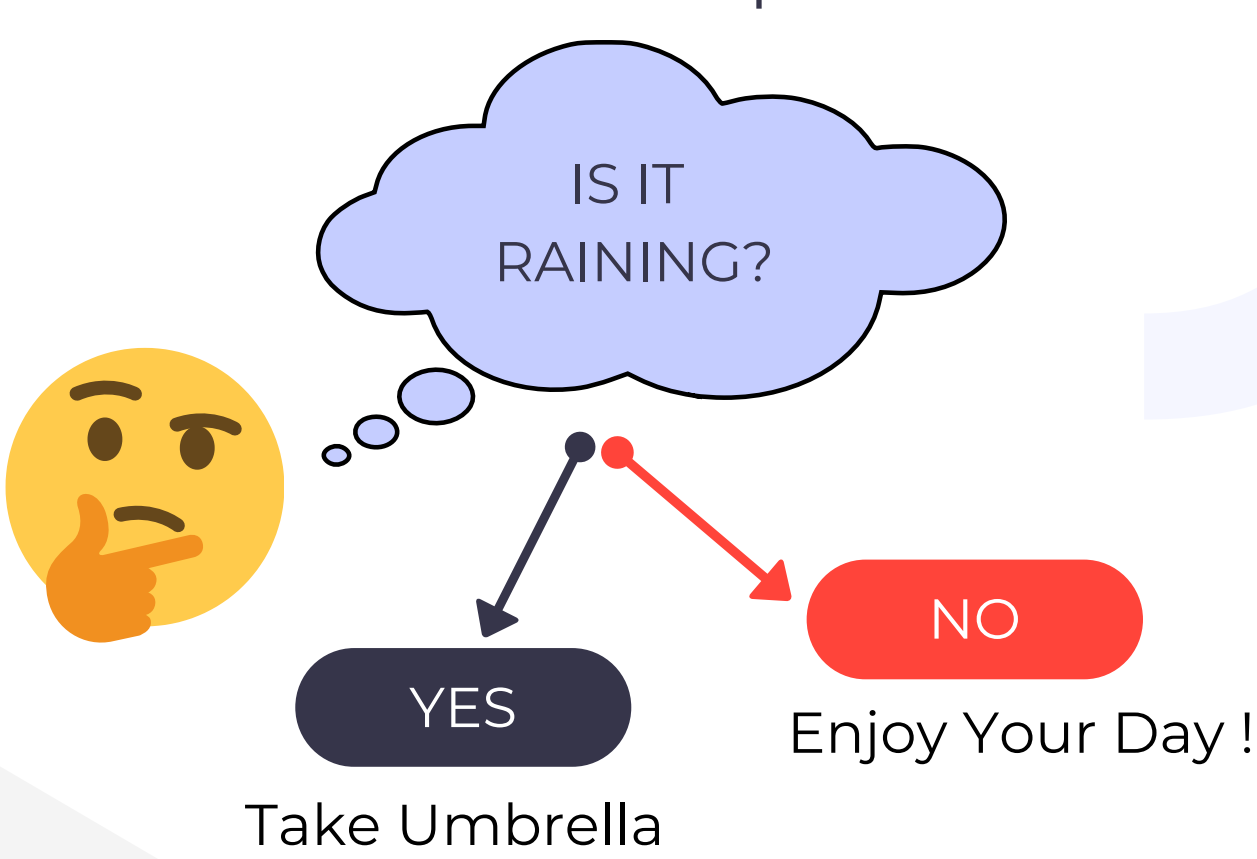
KOTLIN IF-ELSE
STATEMENT & EXPRESSION

CHEEZYCODE

6

IF-ELSE

Humans have the power to take decisions. But we want computer to take the decisions for us. Based on the conditions computer decides, what code to execute - this is where IF-ELSE comes into picture.



IF-ELSE IN KOTLIN

```kotlin
if(yourCondition)
{
    //code executes if condition is true
}
else
{
    //code executes if condition is false
}
```

This is a Boolean condition - it could be either true or false

Kotlin has 2 variations of If-Else. i.e. if-else as statement which is shown above and if-else as an expression

EXAMPLE OF IF-ELSE STATEMENT

Simple program to check
if number is even or odd

```
val num = 3

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

IF-ELSE AS EXPRESSION

Simple program to check
if number is even or odd

```
val num = 3

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

println(result)
```

This is how you write
if-else as an expression

SHORTER VERSION OF IF-ELSE EXPRESSION

```kotlin
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

You can nest multiple if-else statements i.e. if-else inside if-else block. You can have different conditions using else-if ladder.

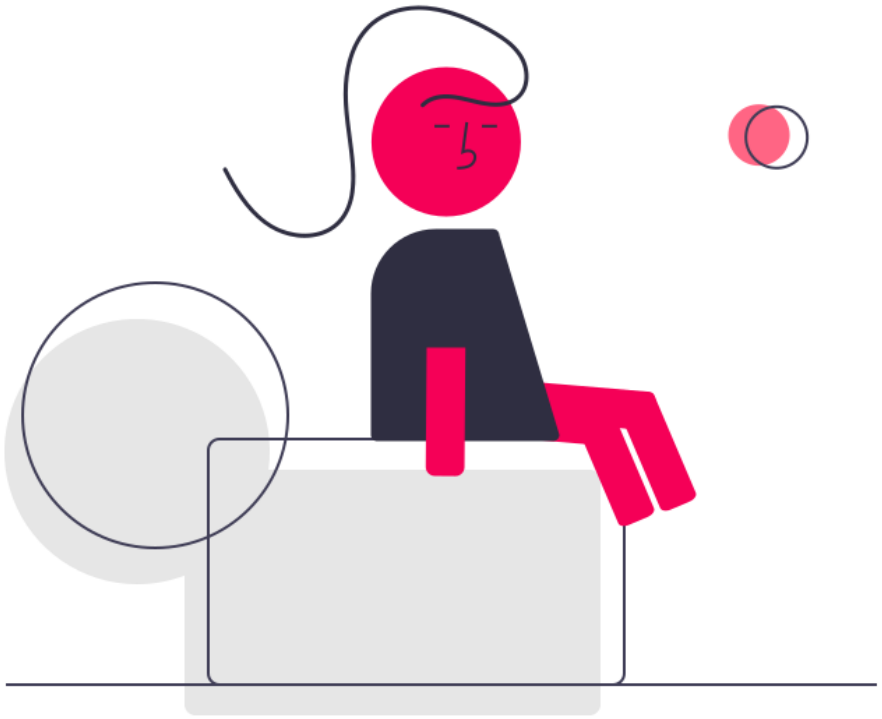
```
val rating = 4
if(rating > 3){}
else if(rating == 3){}
else{}
```

Else-If Ladder

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

Nested If-Else

KOTLIN WHEN
STATEMENT & EXPRESSION

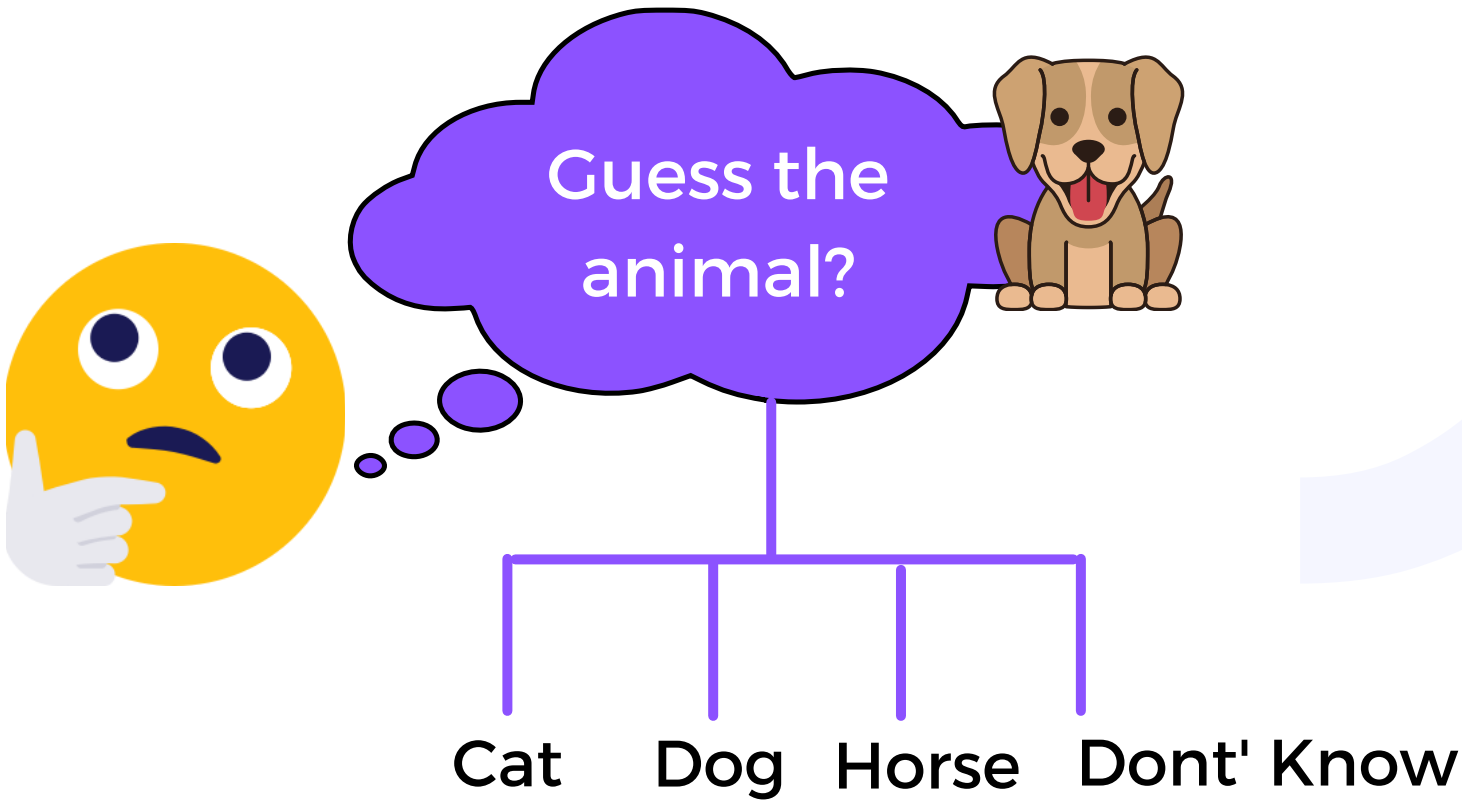


CHEEZYCODE

7

WHEN

When you have to check multiple conditions, if-else conditions becomes less readable. To improve the readability, WHEN is used.



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")
}
```

Although this code works, but readability can be improved

WHEN STATEMENT

Same Program With 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")
}
```

Lot Cleaner !! Kotlin's Statement is more powerful. Feels like switch but in Kotlin there is no switch case.

WHEN EXPRESSION

Same as if-else you can use
when as an expression as well

```
val animal = "Dog"

val result = when(animal) { // When Expression
  "Cat" → "Animal is Cat"
  "Dog" → "Animal is Dog"
  "Horse" → "Animal is Horse"
   else → "Animal Not Found"
}

println(result)
```

Here when is assigned
to a variable

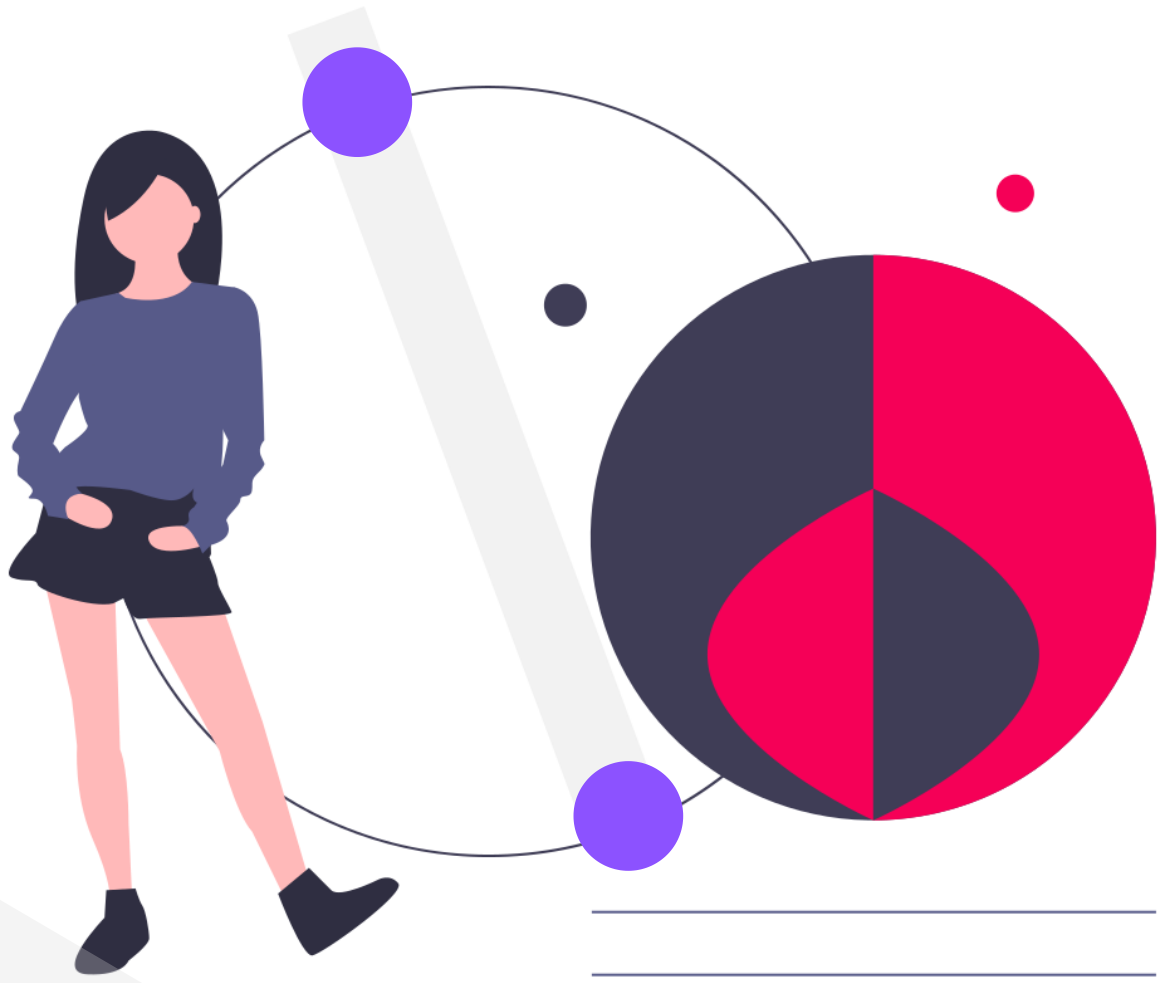
WHEN WITH RANGE

```
val number = 13

val result = when(number) {
  11 → "Eleven"
  12 → "Twelve"
  13..19 → "Teen"
  else → "Not in Range"
}

println(result)
```

This is Range i.e. when will check if the number lies between 13 & 19

Concept of Range - In simple words, if the number lies in this range i.e. between 13 and 19 (Both are inclusive)

KOTLIN LOOPS
WHILE & DO-WHILE LOOP

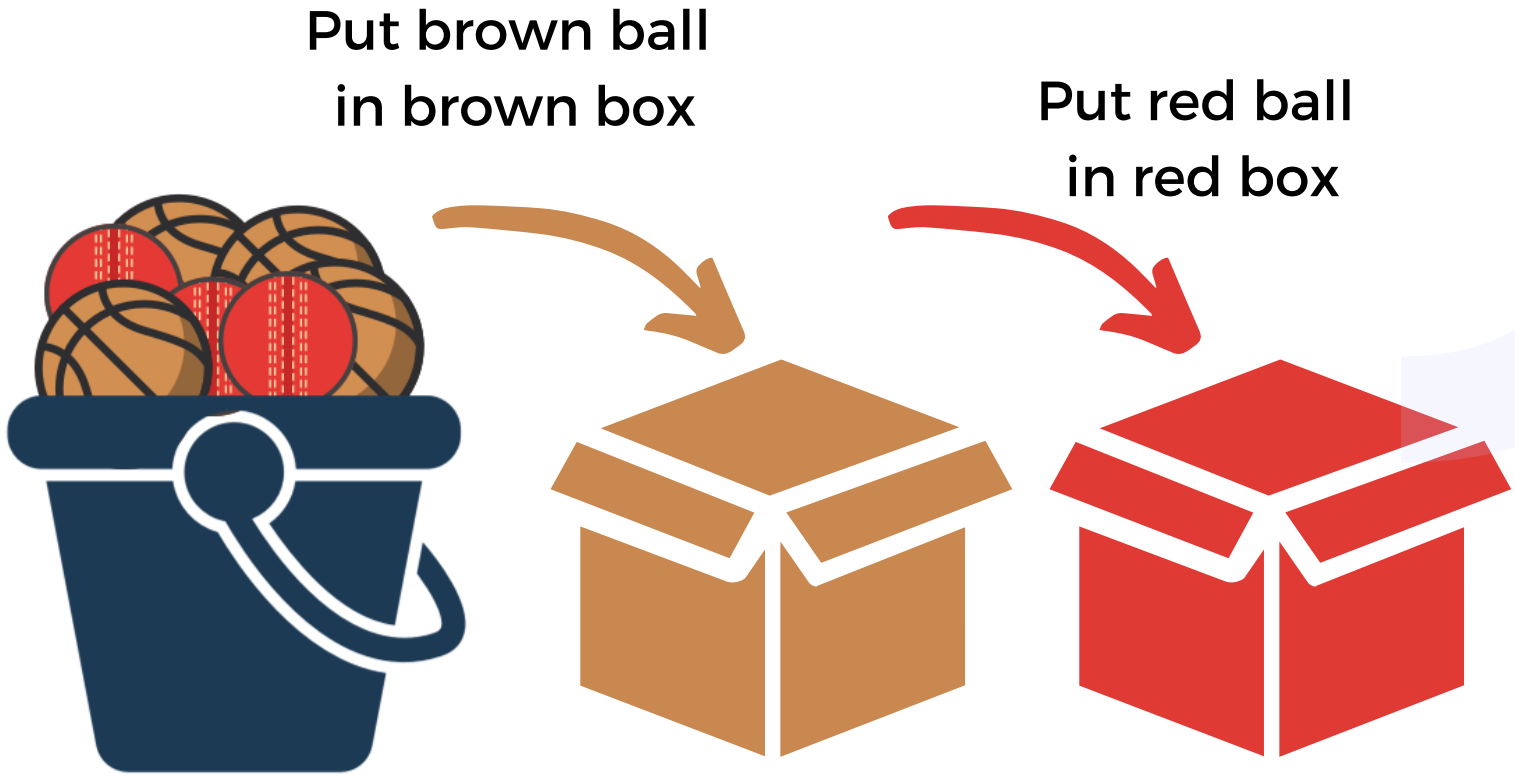


CHEEZYCODE

8

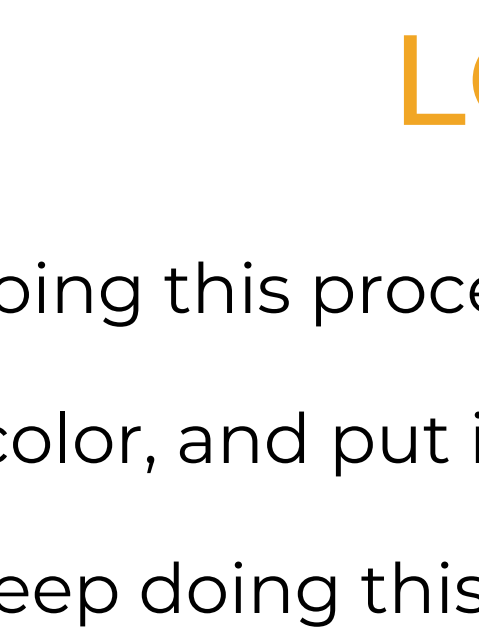
LOOPS

In real life, we use the concept of loop quite often. Let's take an example - we have a bag of balls - we need to distribute these balls into different boxes based on the color.



LOOPS

- We are doing this process using a loop. We pick one ball, check it's color, and put it into the corresponding box.
- We will keep doing this process until all the balls are placed into corresponding boxes.



This is brown ball.
I should put this ball into brown box

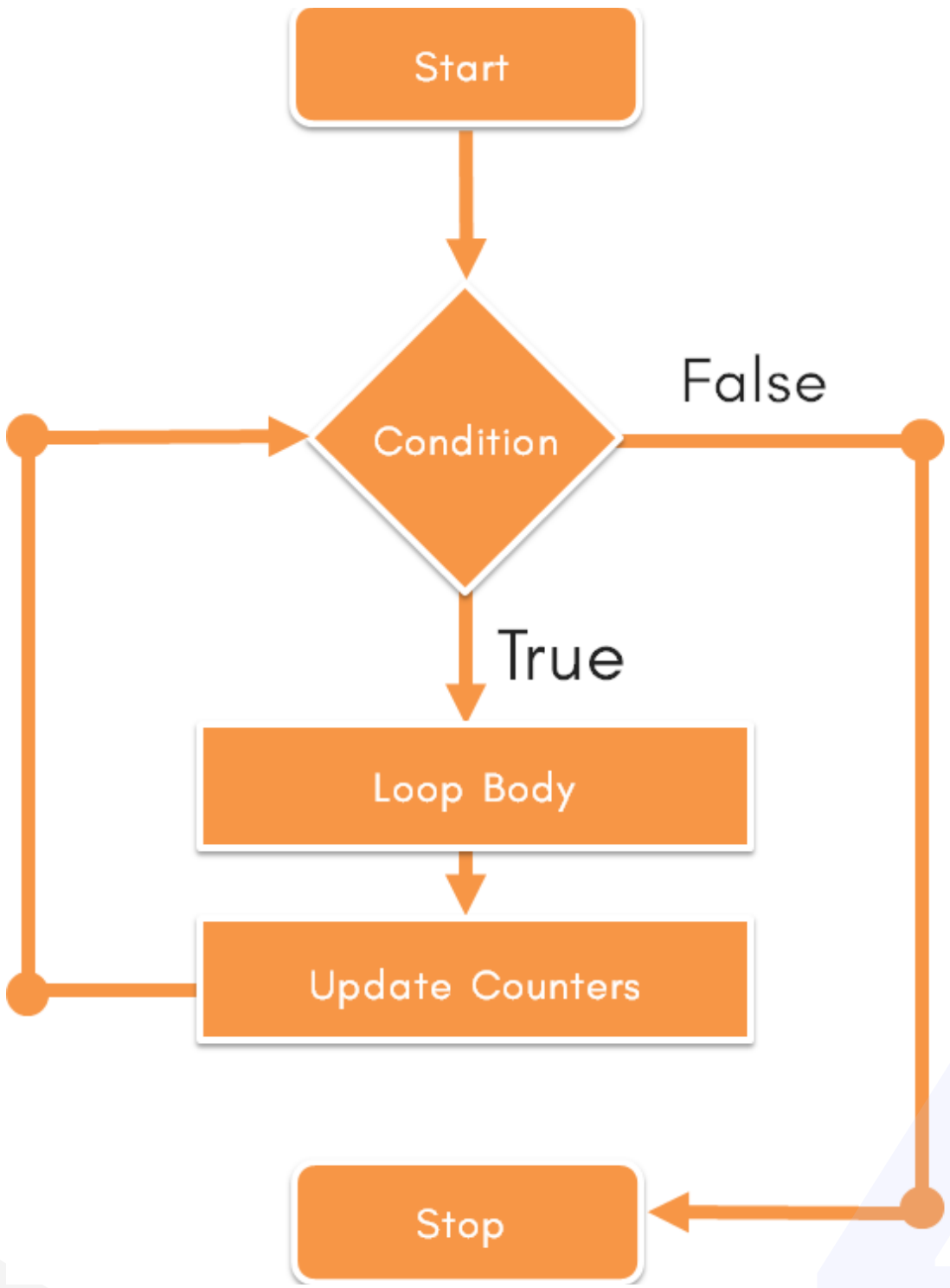
WHILE LOOP

We want computer to do the same - this is where loops come into picture. We keep doing the task until condition evaluates to false. We keep putting the balls until the main box is empty.

```
var count = 1

while(count <= 5)
{
  println("Hello CheezyCode")
  count--
}
```

Until this condition becomes false - it will keep printing Hello CheezyCode

LOOPS


Start
Condition
False
True
Loop Body
Update Counters
Terminate
Stop

DO - WHILE LOOP

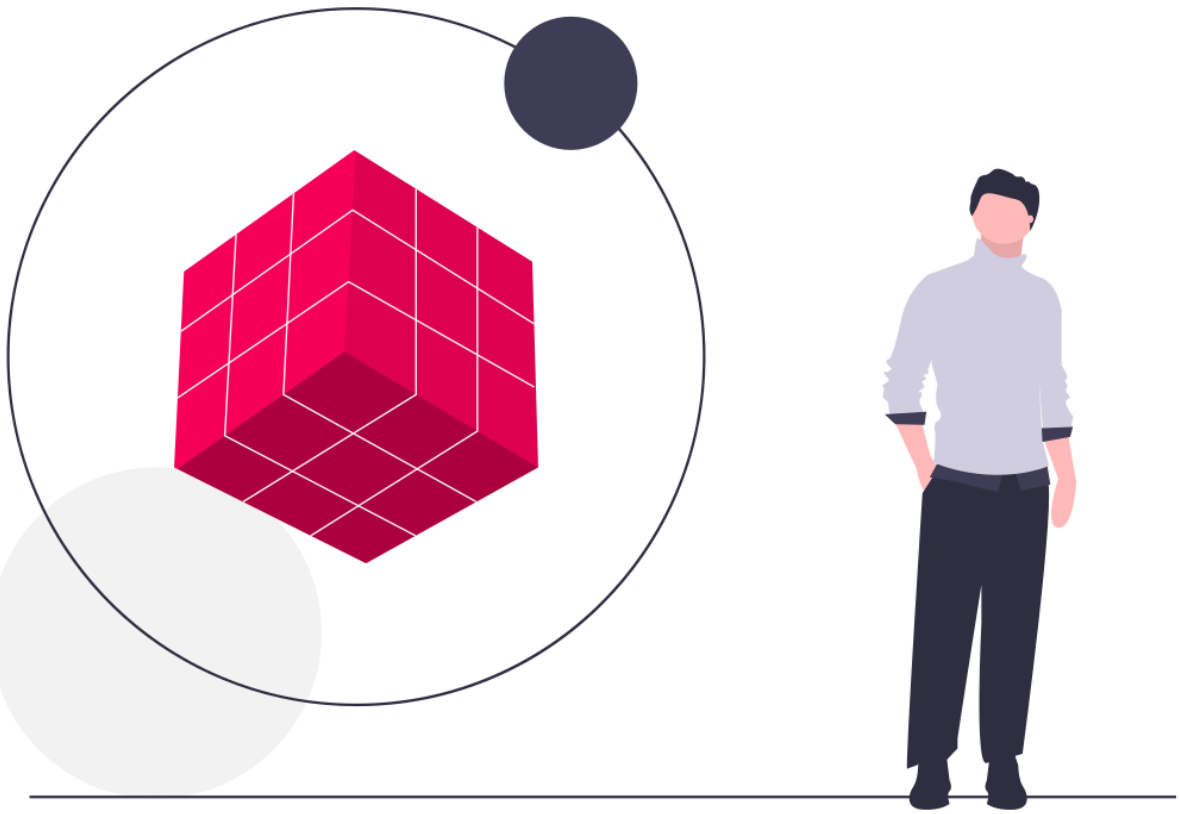
In while loop, condition is check before executing the code whereas in do-while loop condition is checked at the end. So loop executes once, no matter what.

```
var count = 1

do
{
  println("Hello CheezyCode")
}
while(count>5)
```

This will print Hello CheezyCode once.

Condition is checked at the end. So no matter what body of the loop is executed once.

KOTLIN LOOPS
FOR LOOP & VARIATIONS

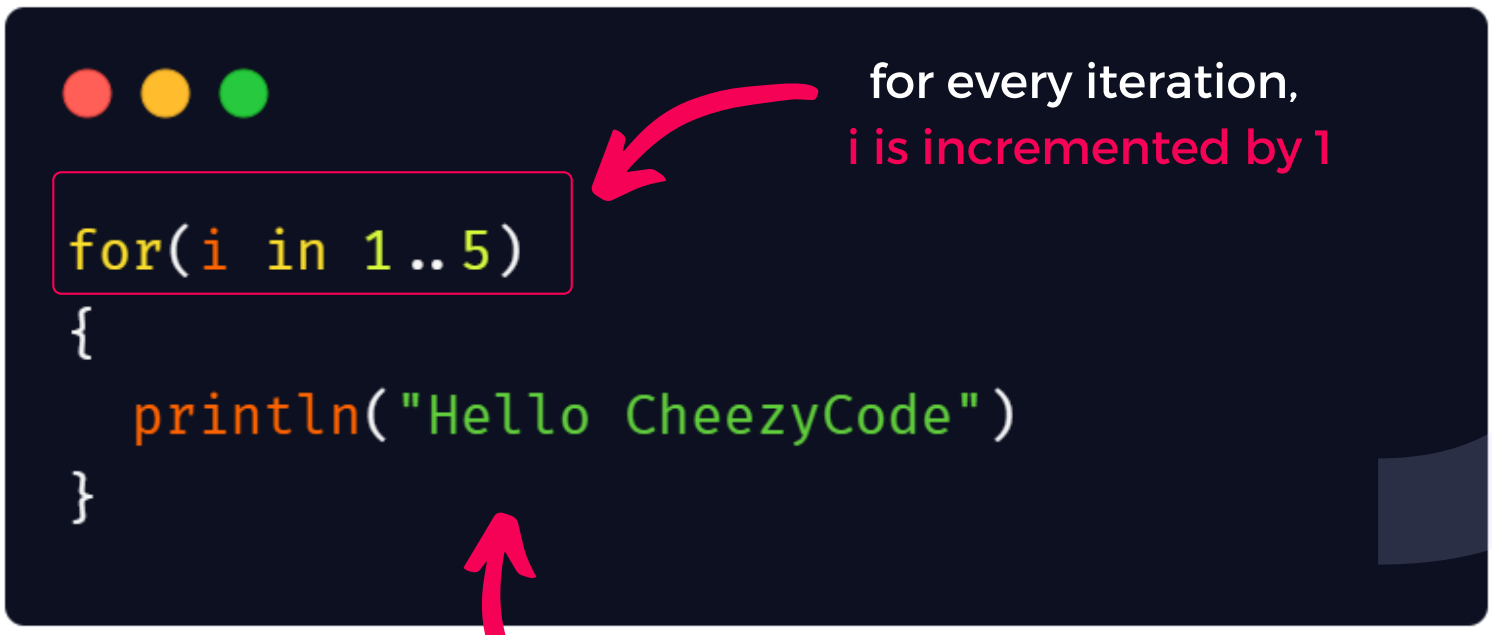


CHEEZYCODE

9

FOR LOOPS

By using a for loop, we can run one or more statements repeatedly for a certain number of times.



FOR LOOP WITH STEP

If you want to increment "i" by certain number then you need to use Step

```
for(i in 1..5 step 2)
{
   println("Hello CheezyCode ${i}")
}

/*
 * Output -
 * Hello CheezyCode 1
 * Hello CheezyCode 3
 * Hello CheezyCode 5
 */
```

Use of step is to let i increment by certain number

You can check the output here

CHEEZYCODE

REVERSE FOR LOOP

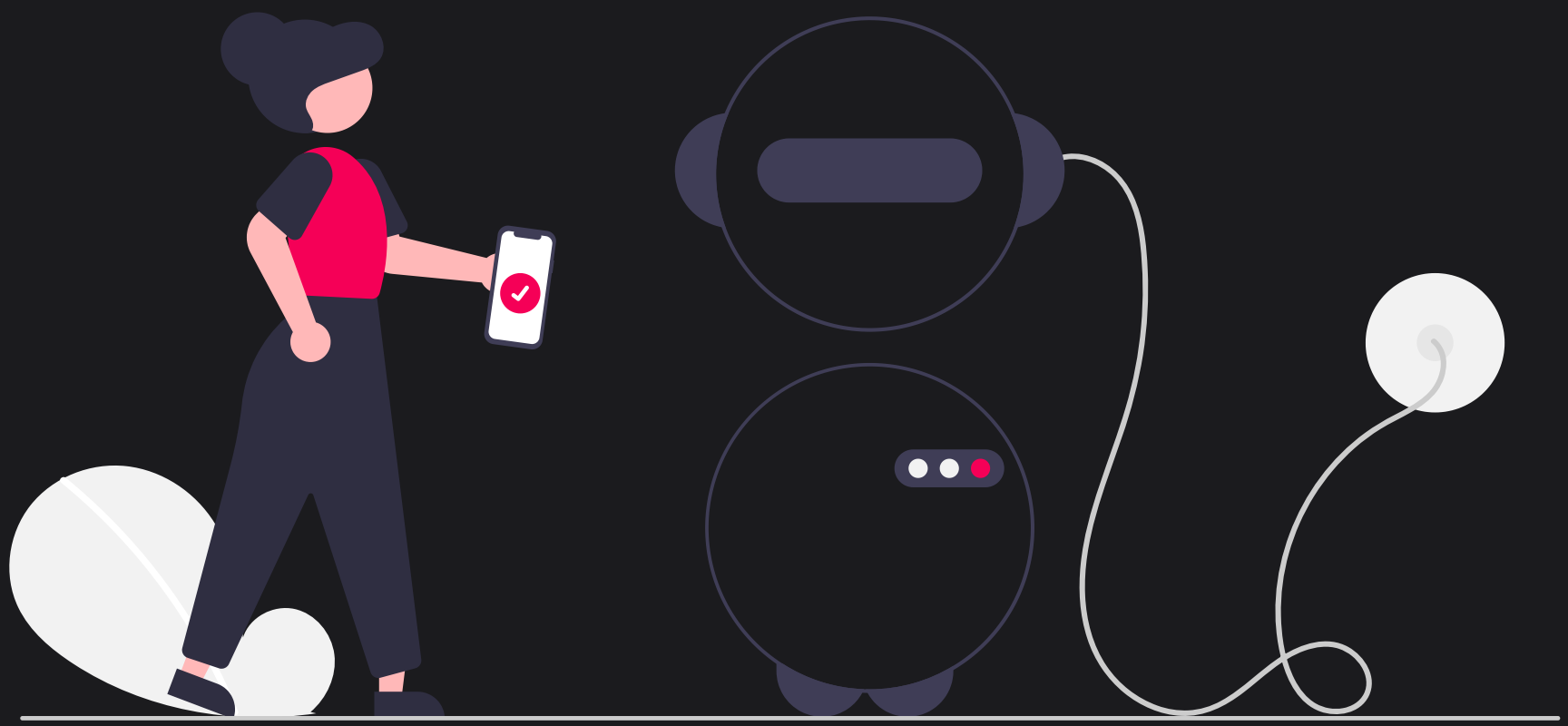
For Loop can be reversed as well i.e. instead of incrementing loop, we can also have decrementing loop. For this - we have downTo

```
for(i in 5 downTo 1)
{
  println("Hello CheezyCode ${i}")
}

/*
 * Output -
 * Hello CheezyCode 5
 * Hello CheezyCode 4
 * Hello CheezyCode 3
 * Hello CheezyCode 2
 * Hello CheezyCode 1
 */
```

downTo is used to reverse the for loop

Step can be used here as well

KOTLIN UNIT TYPE FUNCTION VARIATIONS



CHEEZYCODE

10

FUNCTIONS

Block of code that performs a specific task or functionality. You give input to this function, it runs the block of code based on the input and gives you the output.

KOTLIN FUNCTION

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

This is the data type of output i.e. return type

Since this total variable will be integer (a+b) will be an integer so return type is defined as Int

SINGLE LINE FUNCTIONS

If you have only single expression in your function you can write like this as well.

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

Due to Type Inferencing, we can also remove the return type here - Kotlin infers from the return value that it is going to be an integer.

```
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

CHEEZYCODE

DEFAULT VALUE

You can assign a default value to parameters. For cases where you don't pass a value for this parameter - the default value is used.

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

Default value of count is 1

This function can be called like this - printMessage()

Since count value is not passed here, count is assigned a default value of 1

SOME FACTS

```kotlin
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.

CHEEZYCODE

KOTLIN FUNCTION OVERLOADING & KOTLIN FUNCTION TYPE

CHEEZYCODE

11

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

CHEEZYCODE

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.

CHEEZYCODE

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.

```kotlin
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

FUNCTION TYPE

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

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

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

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

CHEEZYCODE

KOTLIN ARRAY

CHEEZYCODE

12

NEED OF ARRAY

If you want to store multiple items of same type then you need an array. Defining a variable to store each item does not make sense.

Defining variables for each to-do item does not make sense. Writing code to handle every variable is complex.

var note1 : String
var note2 : String
var note3 : String
. . . . .

To solve this - we have Array. Array is a single object that can store a list of notes....

ARRAY

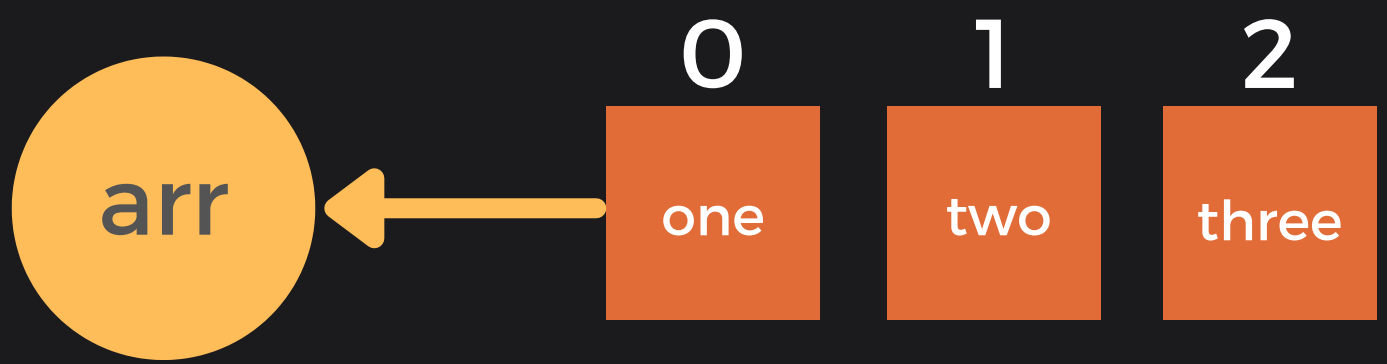
Simple Object that can store multiple values of same type and it has a fixed size.

```
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

}
```

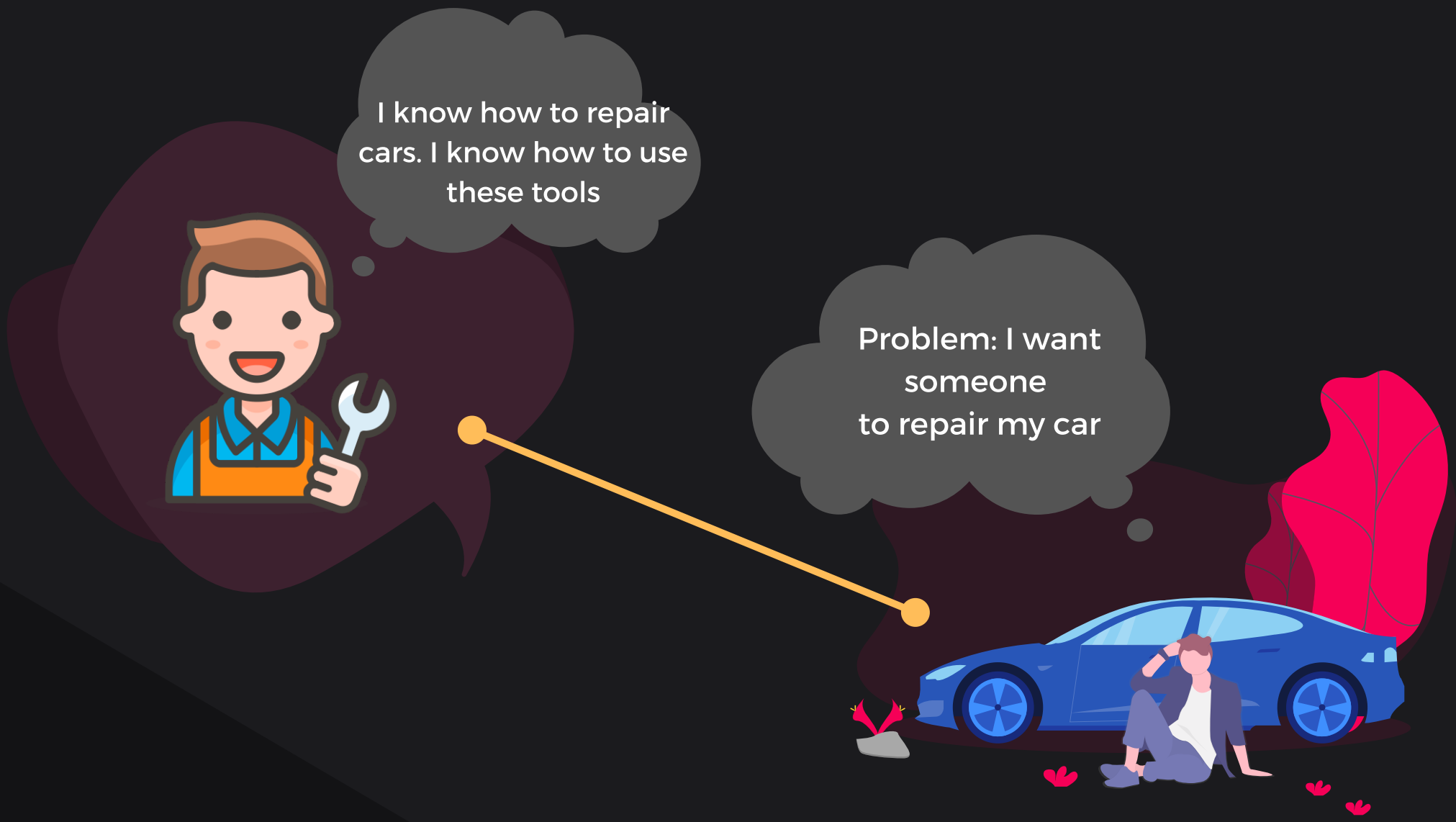
KOTLIN CLASS & OBJECTS - OOP

CHEEZYCODE

13

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.



CHEEZYCODE

OBJECT ORIENTED PROGRAMMING

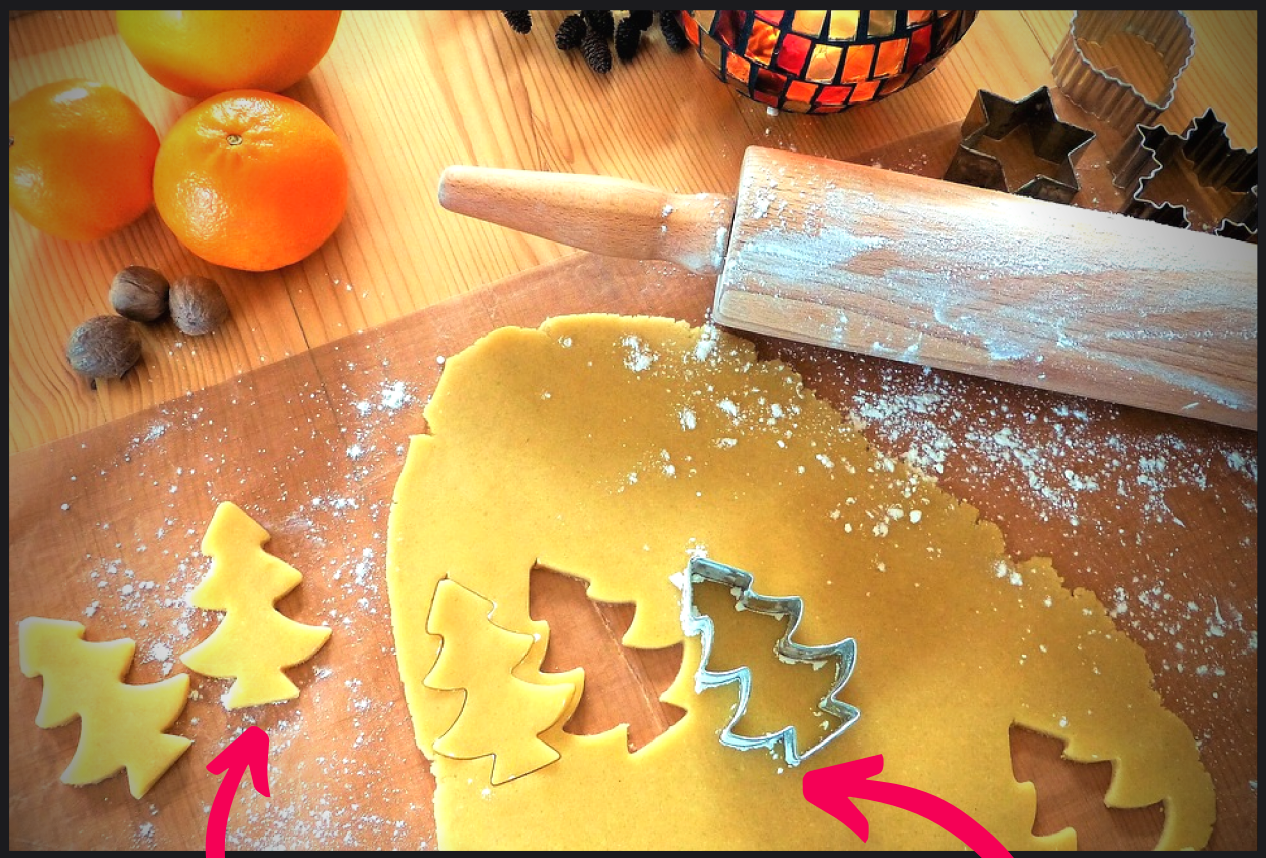
In real life, we do take help from others to solve problems. Here, we have different objects interacting with each other to solve the problem. Objects in this scenario - I am object, Car Mechanic is object, Car is object, Tools are also objects.

```
Problem - Need to repair Car

Objects -

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

Similarly, in programming, we create objects to solve problems. This is a way of writing programs

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.

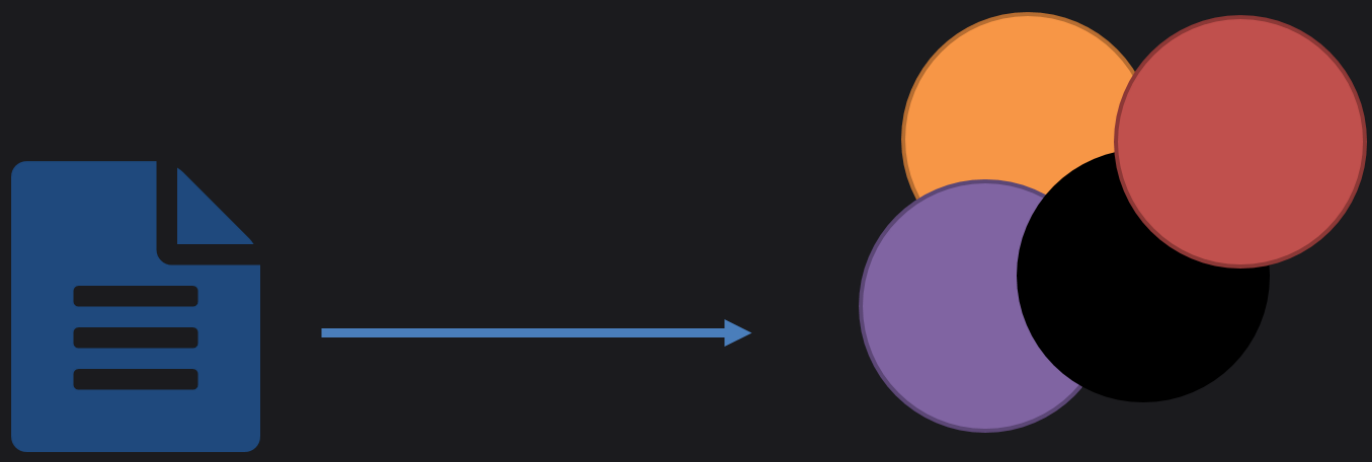


These are the actual objects

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

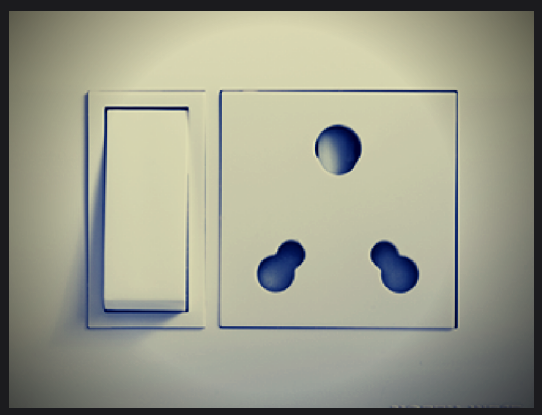
WHAT ARE OBJECTS?

Now we know that we need objects to solve the problems and class defines the structure of the object. But what does it contains?

Object has 2 things - Properties and Methods

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

color : String isOn : Boolean company : String
on() off()



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")
    }
}
```

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

This defines a template or blueprint of a car. With this, we can create multiple car objects. Each object will have its own name and type.

CLASS & OBJECTS IN KOTLIN

```kotlin
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
}
```

This is how you create Car Objects

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

CHEEZYCODE

KOTLIN CLASSES FROM DIFFERENT VIEWPOINT

CHEEZYCODE

14

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
```

USER DEFINED DATA TYPE

```
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.

CHEEZYCODE

EVERY OBJECT HAS A STATE

Once you have a user defined data type,
you can create Objects using that type. Every Object is different - each object has its own set of values

123, Avenue Street

16th F, Casa Homes

- Both are different objects
- Both have different styles
- Both have different no. of rooms
- Both have different address

Both objects have different data for its properties. For instance, House#1 has 3 rooms whereas House#2 has 2 rooms

CHEEZYCODE

METHODS BEHAVIOR IS DEPENDENT ON OBJECT'S STATE

```
class Employee (val name: String, var age: Int, var isHealthy: Boolean)
{
  fun calculateProductivity() : Double
  {
    val factor = if(isHealthy) 0.9 else 0.6
    return 75 * factor          //random formula
  }
}
```

I feel more productive when I am Healthy

I cannot perform well, I am sick

CHEEZYCODE

ANOTHER EXAMPLE

```
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")
    }
  }
}
```

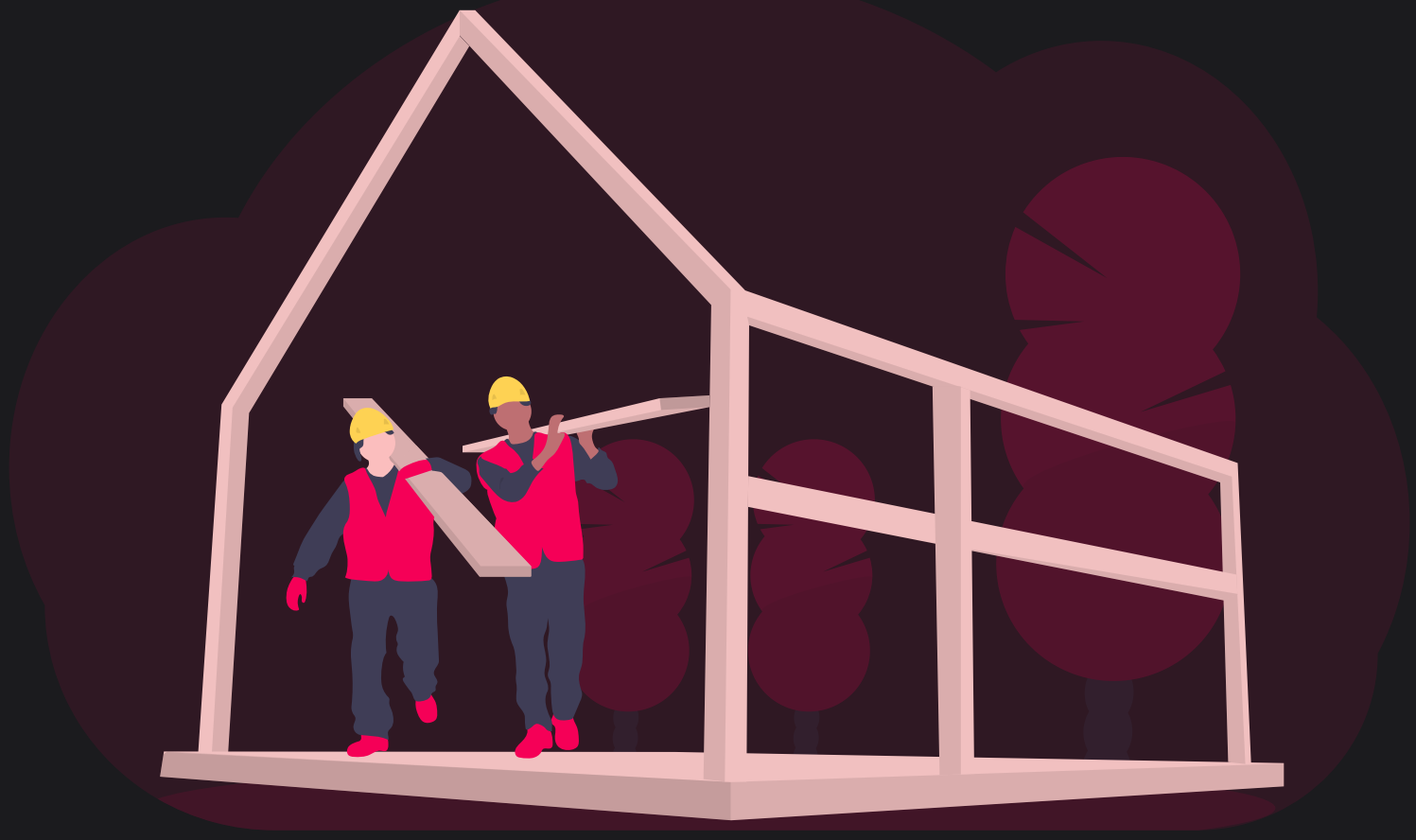
Engine will only start if it has fuel

Output

```
Output:
Mustang - Engine Started
Beetle - Re-fill your fuel tank
```

CHEEZYCODE

KOTLIN CONSTRUCTORS
PRIMARY & SECONDARY



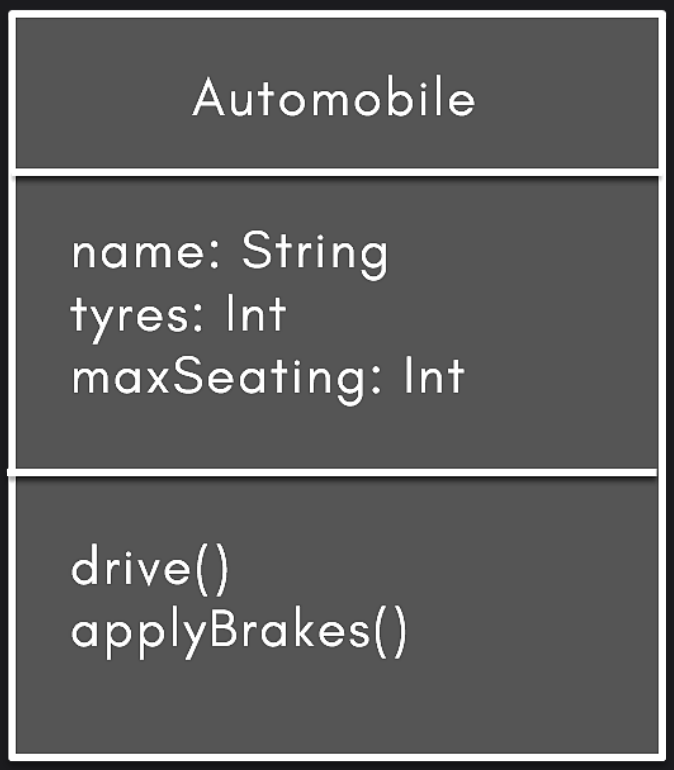
CHEEZYCODE

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)
}
```

These values are passed as Default Values for the object

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

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

ANOTHER WAY

```
/*
    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

WE CAN USE THESE VARIATIONS TOGETHER

```
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.

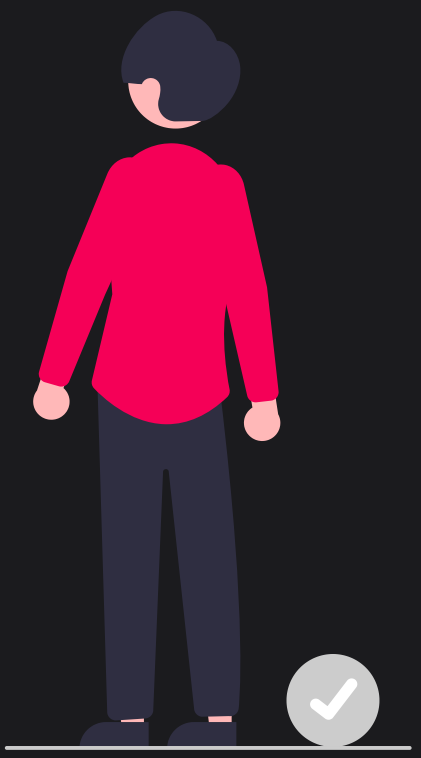
```
fun main()
{
    val car1 = Automobile("Car1", "Petrol")
    val car2 = Automobile("Car2", 2, "Diesel")
}
```

2 ways to create Automobile objects

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

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

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

KOTLIN LATEINIT & GETTERS SETTERS



CHEEZYCODE

16

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.

SETTER EXAMPLE

```
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.

GETTER EXAMPLE

```
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.

KOTLIN INHERITANCE

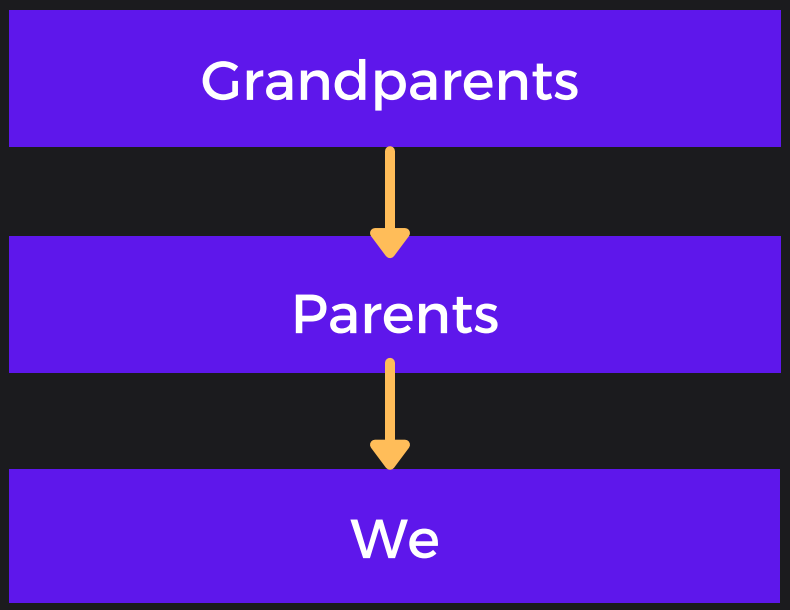
CHEEZYCODE

17

WHAT IS INHERITANCE

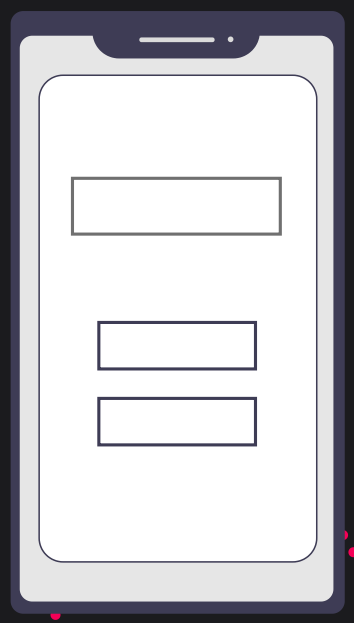
We inherit features and characteristics
from our parents and grandparents

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



EXAMPLE OF INHERITANCE

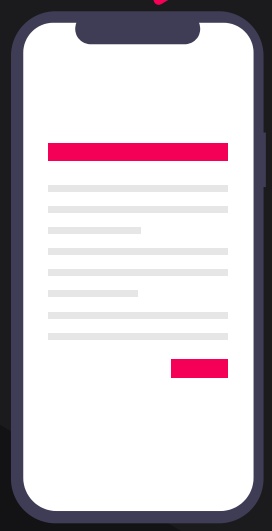
REUSING EXISTING CLASSES TO PREVENT CODE DUPLICATION



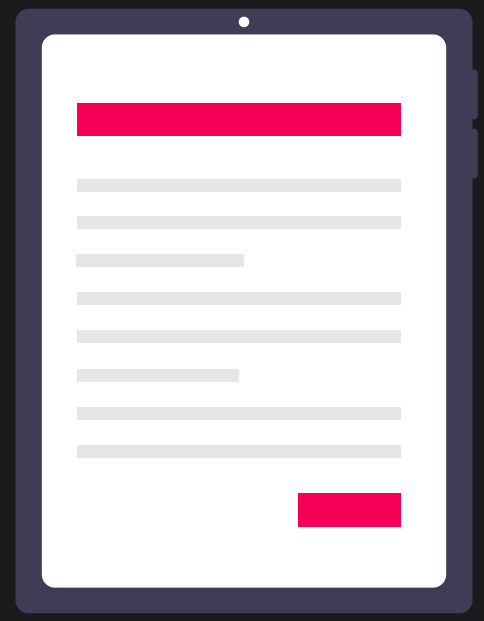
Basic Phone

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.

```
Circle is a Shape, Triangle is a Shape
Cat is an Animal, Dog is an Animal
Full Time Employee, Part-Time Employee
```

Whenever you can argue that you can have IS-A Relationship - You can try to go with Inheritance to structure your classes

INHERITANCE IN KOTLIN

You need to mark class as OPEN for Inheritance

```kotlin
open class Parent
{
    var parentProperty: String = "Parent"
    fun parentFn(){
        println("Parent Function Is Called")
    }
}
```

This is how you inherit from parent class

```kotlin
class Child : Parent()
{
    var childProperty: String = "Child"
    fun childFn(){
        println("Child Function Is Called")
    }
}
```

INHERITANCE IN KOTLIN

```kotlin
fun main()
{
    val child = Child()

    println(child.parentProperty) // ---> Parent Property
    child.parentFn() // ---> Parent Function Is Called

    child.childFn() // ---> Child Function Is Called
}

open class Parent {}

class Child : Parent() {}
```

You can access the properties of parent class - just like your own properties

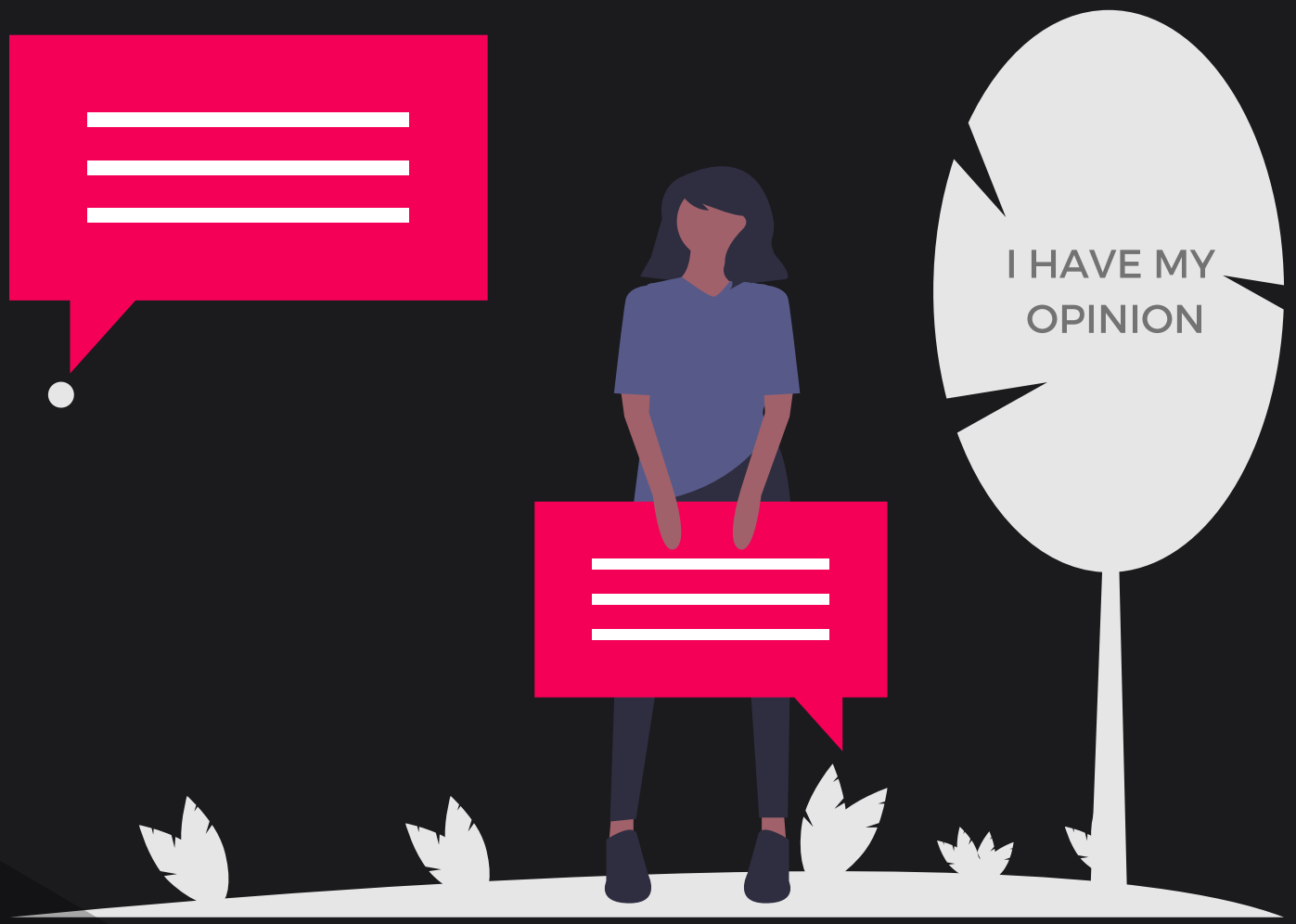
SOME FACTS

- 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.

KOTLIN OVERRIDING & ANY CLASS

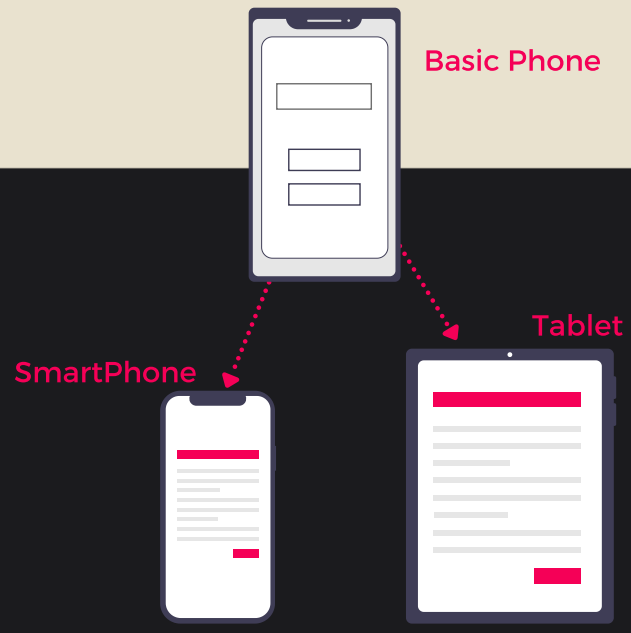


CHEEZYCODE

OVERRIDING

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

- Child class wants to define its own behavior.
- Don't want to use parent's behavior and properties.
- Mobiles have different displays than Tablets.
- Mobile & Tablet should override the display behavior.



OVERRIDING EXAMPLE

```
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

CHEEZYCODE

SUPER KEYWORD

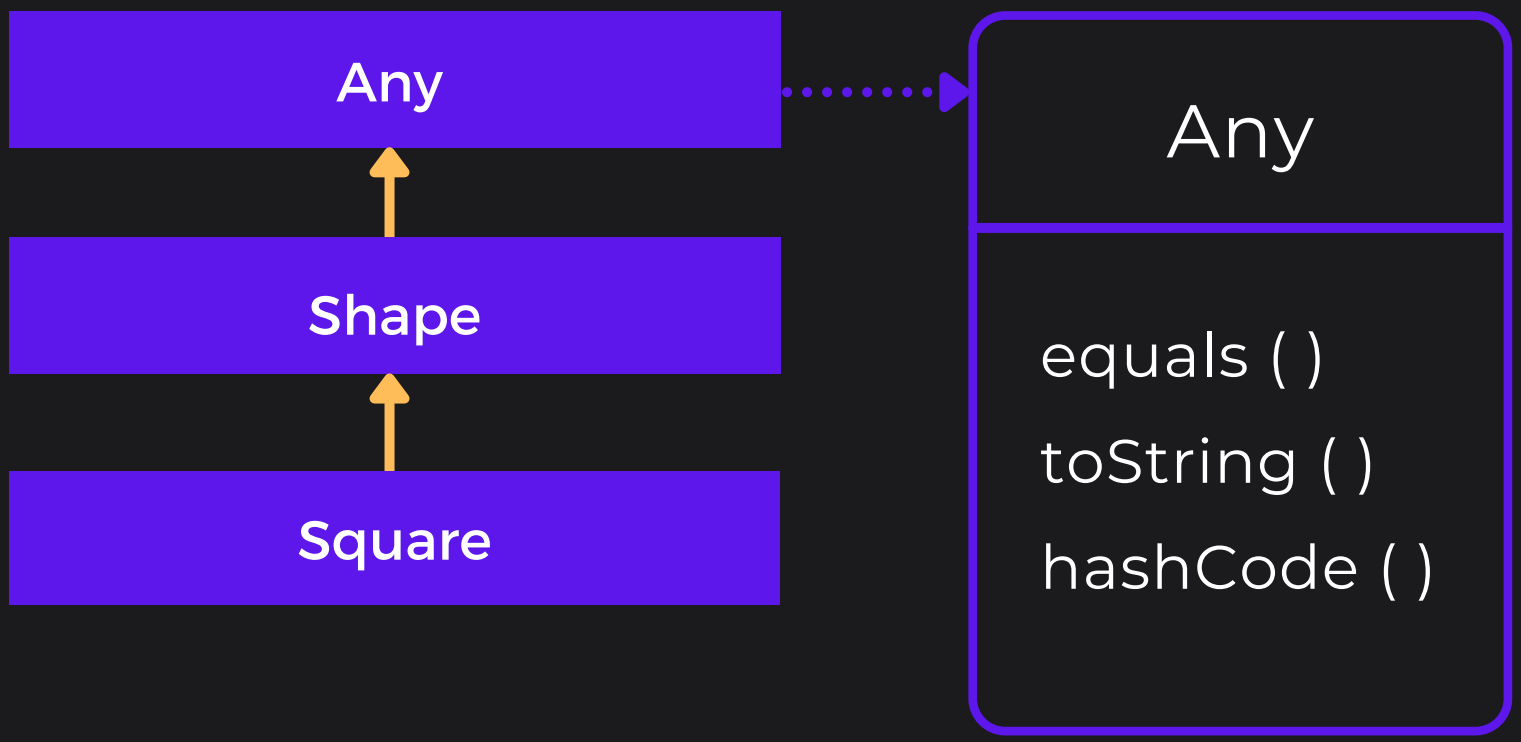
You can still call the parent's method from the child class if you want to - using super

```
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
```

KOTLIN POLYMORPHISM

CHEEZYCODE

19

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.

Same Guy Behaves
Differently At Different Places
Polymorphic Behavior

POLYMORPHISM EXAMPLE

```
open class Shape{
    open fun area() :Double{
        return 0.0
    }
}

class Circle(val radius:Double) : Shape(){
    override fun area(): Double {
        return Math.PI * radius * radius
    }
}

class Square(val side:Double) : Shape(){
    override fun area(): Double {
        return side * side
    }
}

fun calculateAreas(shapes: Array<Shape>){
    for(shape in shapes){
        println(shape.area())
    }
}
```

Area Method here behaves Polymorphically. Based on the shape object, area will be calculated

INHERITANCE & POLYMORPHISM

- Inheritance & Polymorphism go hand in hand. Here Shape is a parent class and we have different child classes - Circle, Square.
- Later if a new shape comes into the picture - say triangle - calculate Area method needs no change.

```
fun calculateAreas(shapes: Array<Shape>){
    for(shape in shapes){
        println(shape.area())
    }
}
```

SIMPLY PUT

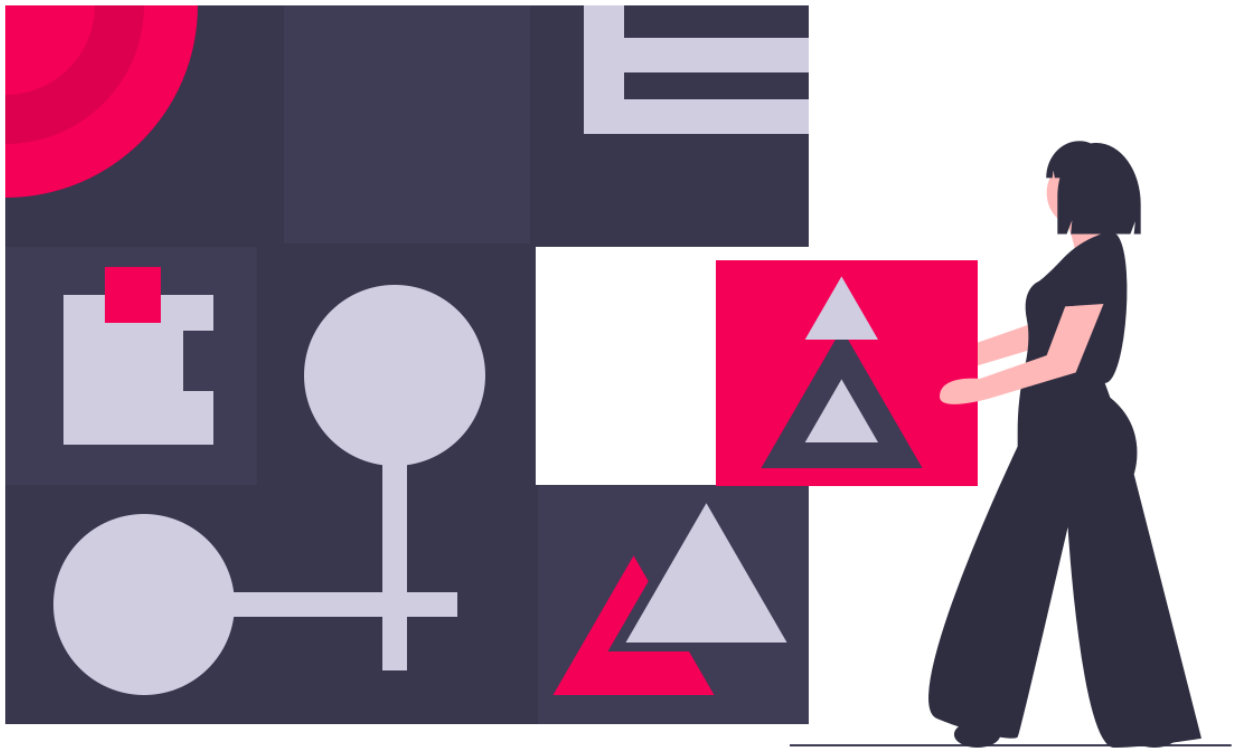
A parent can hold a reference to its child and can call methods which are provided by the Parent Class.

Shape being a parent can hold objects of any of its child classes

```
fun main()
{
    val circle : Shape = Circle(4.0)
    val square : Shape = Square(4.0)

    circle.area()
    square.area()
}

open class Shape{
    open fun area() :Double{
        return 0.0
    }
}
```

Being a reference of a Parent Class - Shape - it can only call area method

KOTLIN ABSTRACT CLASS
ABSTRACT METHODS



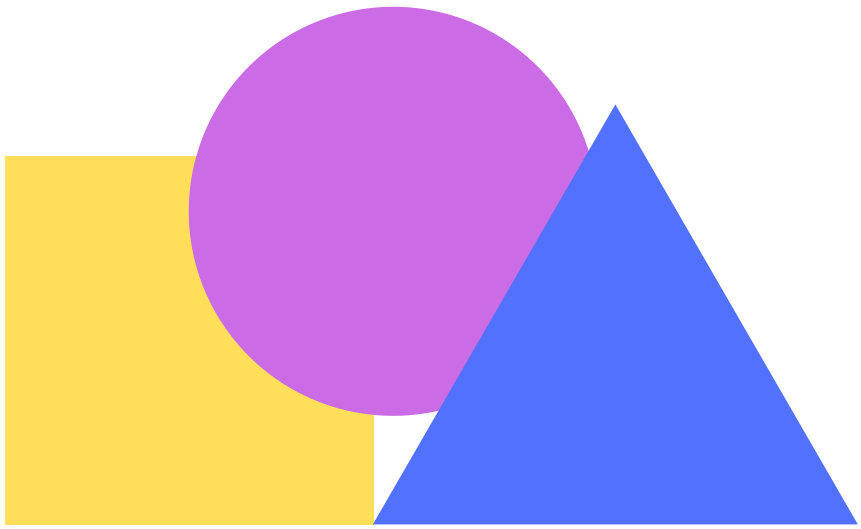
CHEEZYCODE

20

ABSTRACT CLASS

What comes to your mind when I say Shape?

What comes to your mind when I say Person?



Shapes

We either think of a particular shape
i.e. either Square or Circle.

Men or Women

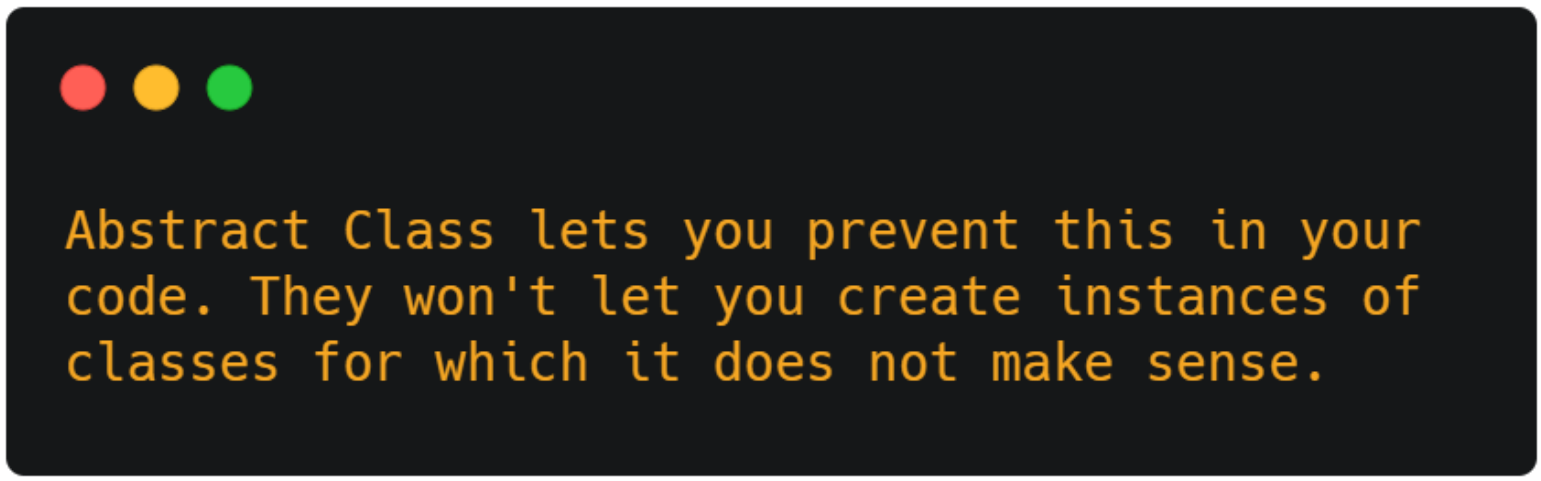
Similarly, for a person - we think
about a particular person.

ABSTRACT CLASS

Shape or Person is just a concept, we always think about the actual objects. So is it a good idea to implement a Shape instance or Person Instance?

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.



ABSTRACT CLASS

But when you can't create an instance of a class,

What's the point of creating a class then?

Child classes should follow a protocol. When you say Circle is a Shape - there is a picture that comes to your mind. That Hey! It's a Shape - there must be a way to calculate its area. There must be some way to display it as well.

This protocol is defined using Abstract Class. These classes are good candidates for being a parent class or base class.

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.**

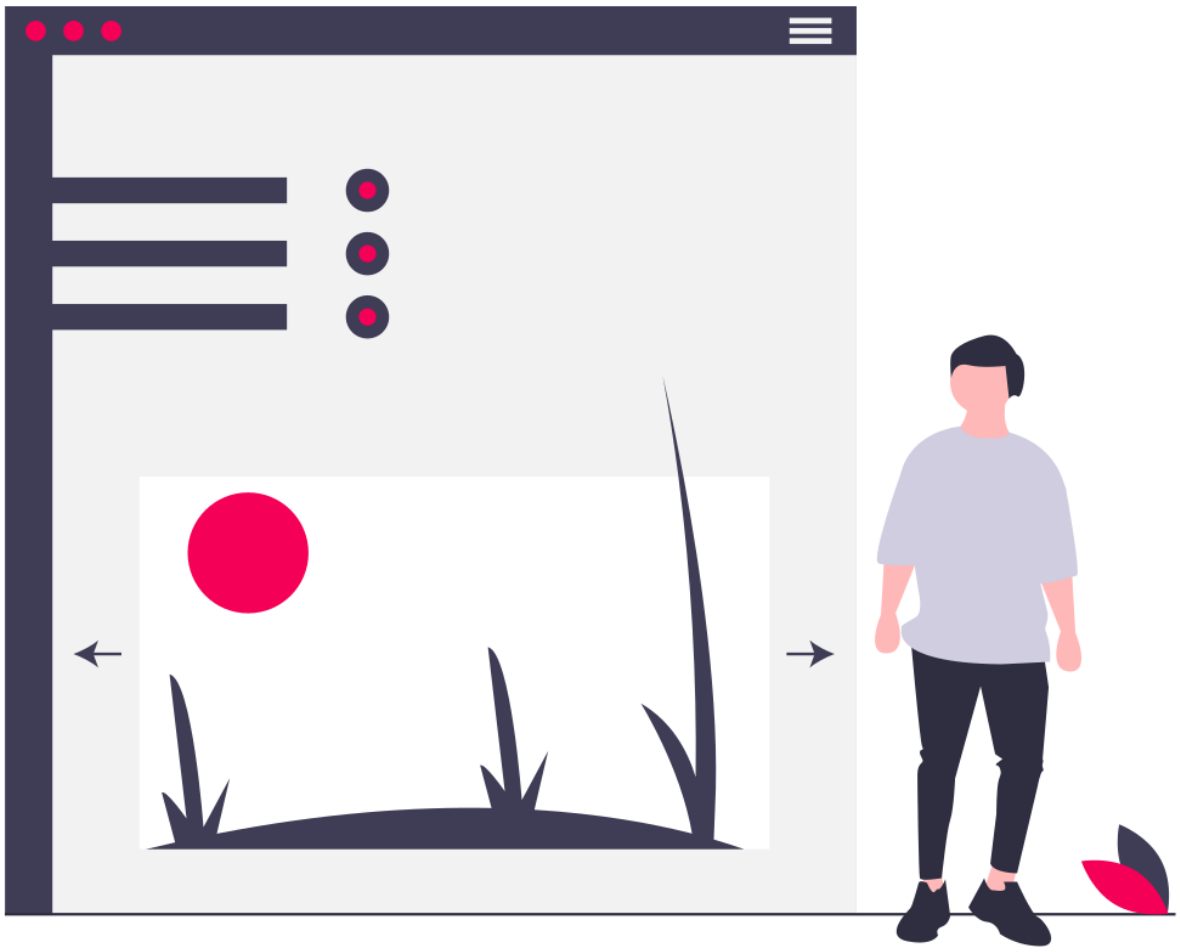
ENOUGH THEORY

```
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.



KOTLIN INTERFACES

CHEEZYCODE

21

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.
```

INTERFACE

When you want your class to play a certain role or exhibit certain behavior irrespective of the class hierarchy - Use Interface

I am Interface. I just define a set of abstract methods. I make sure that if somebody implements me, it should provide the behavior that I commit.

INTERFACE

Interface just declares what should be implemented but how it should be implemented is left to its child classes

```
interface Draggable
{
    fun drag()
}
```

Whoever implements this interface, provides the body for this drag method because different objects are dragged differently. So it is up to the child class how they want to implement this behavior

POLYMORPHISM ❤ INTERFACES

Polymorphism can be achieved using Abstract classes as well but the Abstract class belongs to a particular inheritance tree. In the case of Interfaces, anyone can implement that interface and helps achieve polymorphism.

```
fun dragObjects(objects: Array<Draggable>)
{
  for(obj in objects)
  {
     obj.drag()
  }
}
```

Any object which implements Draggable can be passed in this array.

```kotlin
fun main() {
    dragObjects(arrayOf(Circle(4.0),Player("Smiley")))
}

fun dragObjects(objects: Array<Draggable>){
    for(obj in objects){
        obj.drag()
    }
}

interface Draggable{
    fun drag()
}

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

class Circle(val radius:Double) : Shape(){
    override fun area(): Double =  Math.PI * radius * radius
    override fun drag() = println("Circle is dragging")
}

class Player(val name: String) : Draggable{
    override fun drag() = println("$name is dragging")
}
```

Polymorphic behavior is achieved using Interface. No need to change this code for future classes as well.

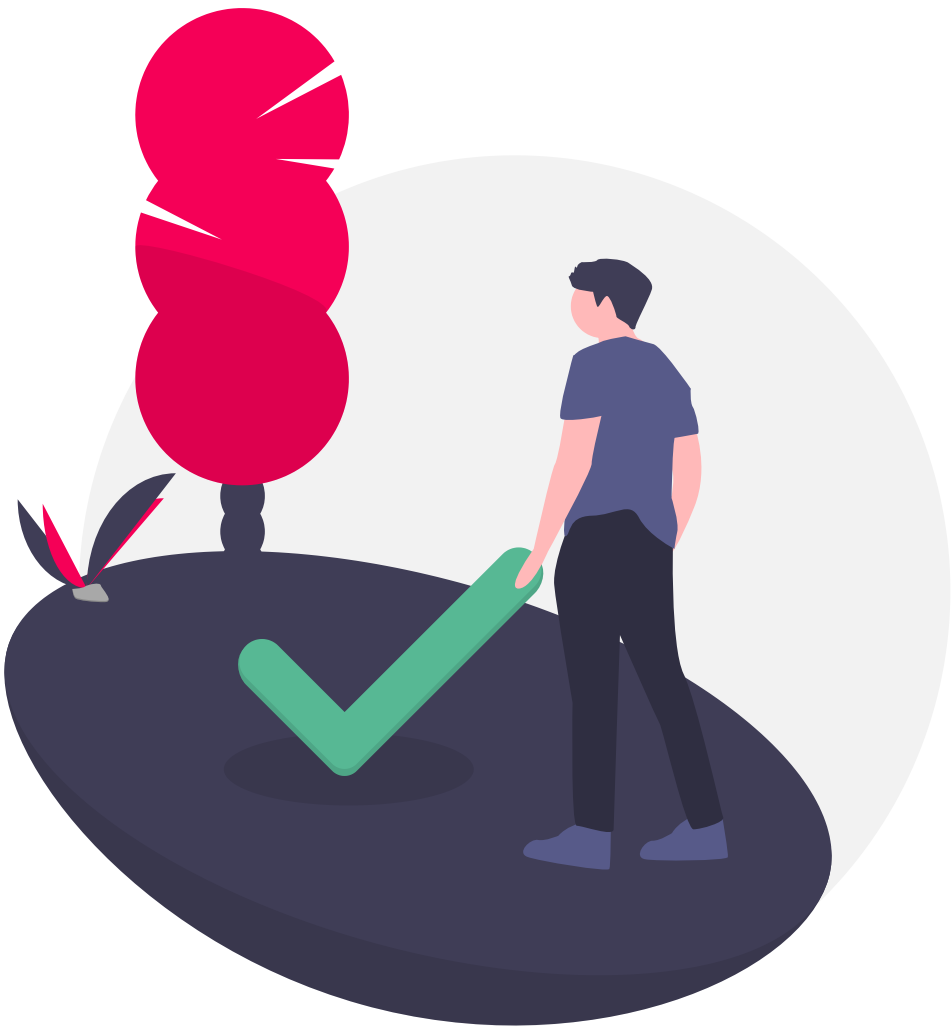
This is how you implement Interface

Person is from different hierarchy

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.

KOTLIN TYPE CHECKING & SMART CASTING



CHEEZYCODE

22

TYPE CHECKING

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

```
val shapesArr: Array<Shape> = arrayOf(Circle(5), Square(4))

for(obj in shapesArr)
{
  if(obj is Circle)
  {
     println(obj.area())
  }
}
```

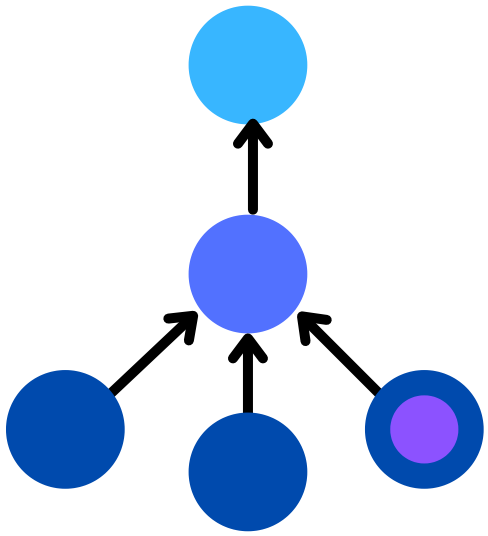
We are checking if the type of the object is Circle.

is operator

TYPE CHECKING

To take full advantage of Polymorphism - we generally use Parent Class's Reference and objects could be of any subtype.

Program to an interface, not an implementation



What is the type?

We might check if the object is of a particular type, then only we will call its method.

SMART CASTING

Kotlin is very smart - we don't have to do explicitly casting. is operator does casting for us automatically.

```
if(obj is Circle)
{
    println(obj.area())
}

//Another Example

if (x !is String) return
println(x.length)
```

In this if block - obj acts like it is a Circle object. We don't need to do explicit casting in this case. All the properties of the Circle object are accessible - This is smart casting.

Here compiler knows that it is a string object because if it is not a string - return would have been executed

MAKE SURE YOU KNOW WHAT YOU ARE DOING

```
if(obj is Circle)
{
    println(obj.area())
}
else
{
    (obj as Player).sayName()
}
```

This is explicit casting. Using as operator.

Since we are not checking the type before and

if obj is not the object of Player class

- You will get a ClassCastException.

KOTLIN VISIBILITY MODIFIERS & ENCAPSULATION

CHEEZYCODE

VISIBILITY MODIFIERS

Let's understand this by example

```
fun main()
{
    val p1 = Person("John", 22)
    p1.age = -10
}

class Person(val name: String, var age: Int)
```

Here you can assign a negative value to the age property. There is no way to prevent this because the age field is publicly accessible i.e. anyone can access this property and change its value.

VISIBILITY MODIFIERS

There are certain things that need to be public and certain things that need to be private.

Visibility Modifiers To Rescue

You can define the visibility of the fields and methods so that they are only accessible where you want them to be accessible. Kotlin has 4 types of visibility modifiers -

Public, Internal, Private, Protected

TOP LEVEL DECLARATIONS

These are top-level declarations defined directly inside a package.

Default visibility is Public (if you don't use any modifier)

```
fun main() { }

class A

var b = 20

fun gn() { }

interface Drag { }
```

VISIBILITY MODIFIERS

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

```
class A {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    public val d = 4
}

private var b = 20

internal fun gn() { }

interface Drag { }
```

Protected can't be used with top-level declarations.

VISIBILITY 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

KOTLIN OBJECT KEYWORD & SINGLETONS

CHEEZYCODE

24

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)
  {

  }
}
```

Single Logger Object is created. It acts like a normal object but here only a single instance of this type is created in the application.

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")
    }
  }
}
```

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

KOTLIN COMPANION OBJECT & JVM STATIC

CHEEZYCODE

SINGLETON INSIDE CLASS

- You can create objects using object keyword inside a class as well. This object will be a singleton.
- You can create multiple instances of this class but this singleton will be associated with your class i.e. only one object will be created.

Here MyObject is associated with the class not with a particular instance of a class

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

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

SINGLETONS INSIDE CLASS

You can create multiple objects inside a class. When you want to associate some functionality with the class - you can use these objects.

```
fun main()
{
  MyClass.MyObject.f()
  MyClass.AnotherObject.g()
}

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

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

COMPANION OBJECT

Companion means - a friend with whom you share everything. S/he is a friend - you use each other stuff without asking for permissions.

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

MyClass.MyObject.f()

MyClass.f()

MyObject is marked as Companion - no need to use its reference - MyClass and MyObject are friends

COMPANION OBJECT

```
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")
  }
}
```

CHEEZYCODE

JVM STATIC

- If you know Java, we can call only static methods with the class name. In Kotlin, companion object methods give the same feel but behind the scenes implementation is different.
- If you want static methods like Java - i.e. if you want to consume this code in Java & treat these companion object methods as static - you need to annotate them as @JvmStatic

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

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

CHEEZYCODE

FACTORY PATTERN USING COMPANION

```
//Example of Factory Pattern Using Companion
class Pizza private constructor(val type: String, val toppings: String)
{
    companion object Factory
    {
        fun create(pizzaType: String): Pizza
        {
            return when (pizzaType) {
                "Tomato" → // Pizza( ... )
                "Peppy Paneer" → // Pizza( ... )
                else → // Pizza( ... )
            }
        }
    }
}
```

KOTLIN
DATA CLASSES

CHEEZYCODE

DATA CLASS

In any application, we create classes that are used to store data. Classes for UI Layer, DB Layer, and so on - sole purpose is to store data.

In other programming languages - we create these classes manually - by defining getters/setters, different methods for comparing the objects, and so on.

TOO MUCH BOILERPLATE CODE

DATA CLASS

Kotlin loves Developers.

No need to write boilerplate code.

> **Kotlin Has Data Classes**

```kotlin
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.

DATA CLASS

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 PROVIDES MORE

Along with overriding methods for you, Data Class provides

copy method and **ComponentN** functions as well.

```kotlin
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
```

KOTLIN ENUM CLASSES & SEALED CLASSES

CHEEZYCODE

27

ENUM CLASS

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

```
fun main() {
    val day: DAY = DAY.MONDAY
    val day2: DAY = "Hello" //*** ERROR //***
}

enum class DAY
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

DAY variable can take values from this particular set only. You cannot assign any random values.

ENUM CLASS

Every value inside an Enum class is an object.

You can even initialize the value of these objects.

```
enum class DAY
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

7 objects of
Day class.

```
enum class DAY(val num: Int)
{
    SUNDAY(0),
    MONDAY(1),
    TUESDAY(2),
    WEDNESDAY(3),
    THURSDAY(4),
    FRIDAY(5),
    SATURDAY(6)
}
```

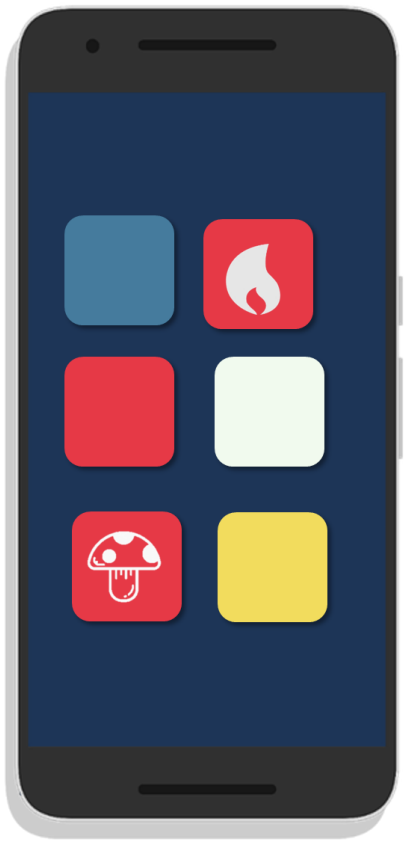
Constructor

Objects can be initialized
with a value as well

SEALED CLASS

Enum class restricts the values,

Sealed class restricts the type of classes.



- To implement this game, we can see that there are 4 types of Tiles - Red, Blue, White, and Yellow.
- Red Tiles might have a different state as well. (Red Mushroom or Fire).

Enum class can't be used here because objects defined in Enum are restricted to only a single instance and those instances have a fixed state.

SEALED CLASS

In simple words, we are restricting the value to be one of the types from a limited set, but cannot have any other type

```
sealed class Tile

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

You can create 'n' number of objects of any subclass and can have different states of the objects as well. Not like Enum where you define constant objects.

Type is restricted to these subclasses

SEALED CLASS & WHEN

Since the types are restricted

we don't need else block inside when

```
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

KOTLIN NULLABLE TYPES
SAFE CALL & ELVIS

CHEEZYCODE

28

NULLABLE TYPES

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

Suppose in a registration application, gender is not marked as a required field. User submits the application - What should be the value of Gender in this case?

You need something that will indicate the absence of value

NULLABLE TYPES

To indicate the absence of value - NULL is used

Kotlin being a Safe Language does not allow you to store Null Values inside your normal data types. You need to explicitly define that - Hey Kotlin! I am aware that this variable can have null values - allow me to store nulls as well

```
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

CHEEZYCODE

NULL SAFETY

If you mark the data type as Nullable, Kotlin gives you an error if you call any method of that type

```
var gender : String? = null

gender.toUpperCase() // --- ERROR ---
```

Since Gender could be null, to prevent this - Kotlin gives you the error.

Hey! You need to check for null value otherwise your program will crash

CHEEZYCODE

NULL SAFETY

Ways to Handle Nulls

```
var gender2: String? = null

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

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

//3rd Way
gender?.let {
    println("Line 1")
    println("Line 2 $gender")
    println("Line 3 $it")
}
```

?. is a safe call operator

You can use safe call with let to execute multiple statements & you can access gender as it inside this let block.

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.

CHEEZYCODE

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

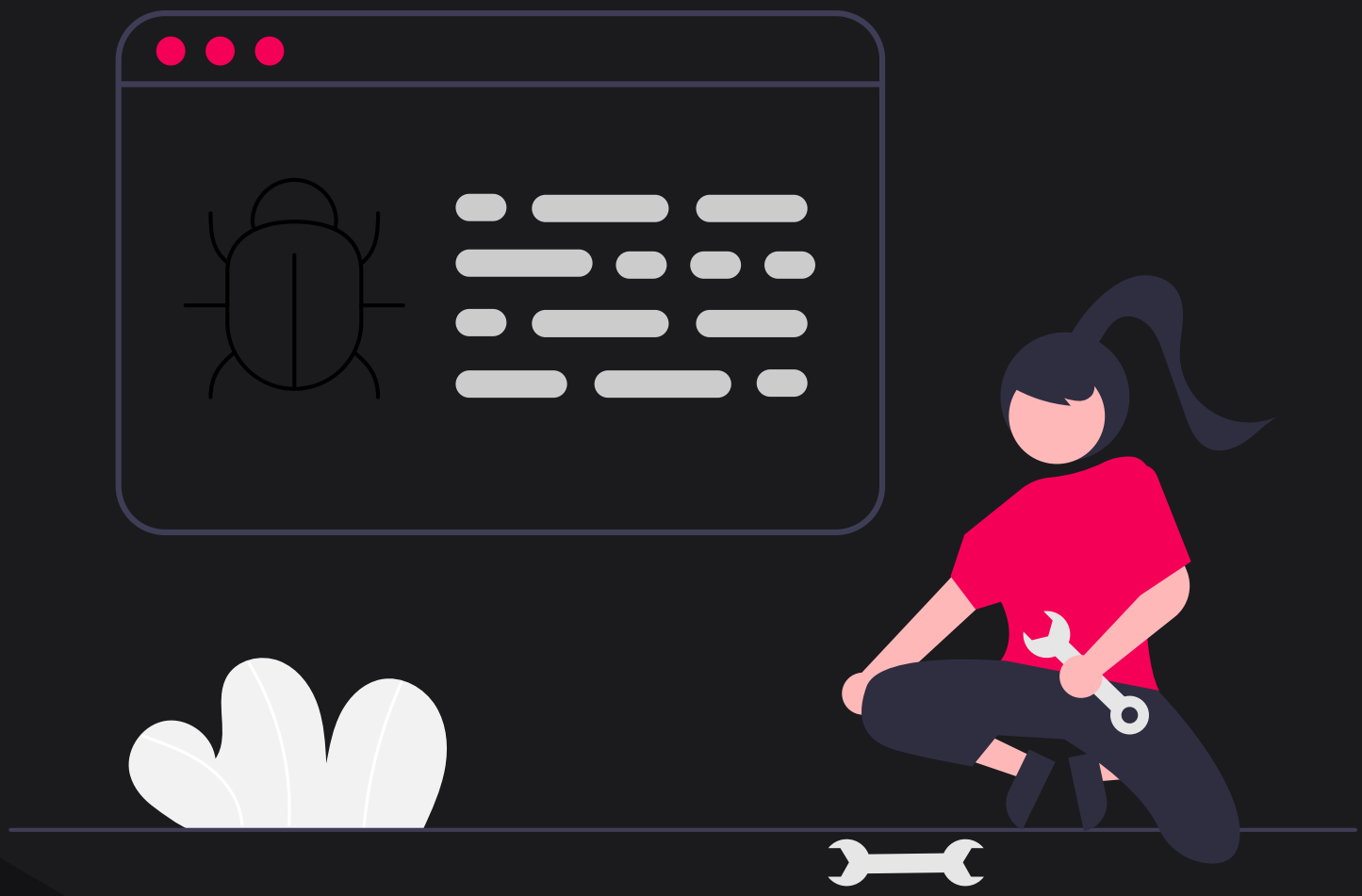
```
fun main() {
    var gender : String? = "male"
    println(gender!!.toUpperCase())
    gender = null
    println(gender!!.toUpperCase())
}
```

Output

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

Non-null operator converts any value to a non-null type and throws an exception if the value is null.

KOTLIN
EXCEPTION HANDLING



CHEEZYCODE

29

EXCEPTION HANDLING

During the execution of the program, when something erroneous happens - we say that an Exception has occurred.

- Expecting a number input but the user entered a string
- While printing a document, the printer runs out of paper
- Download interrupted, not enough space in memory
- Tried accessing the item that does not exist & so on ...

Common scenarios that causes exceptions

TRY-CATCH

Try Catch is a way to gracefully handle exceptions that can occur in a program.

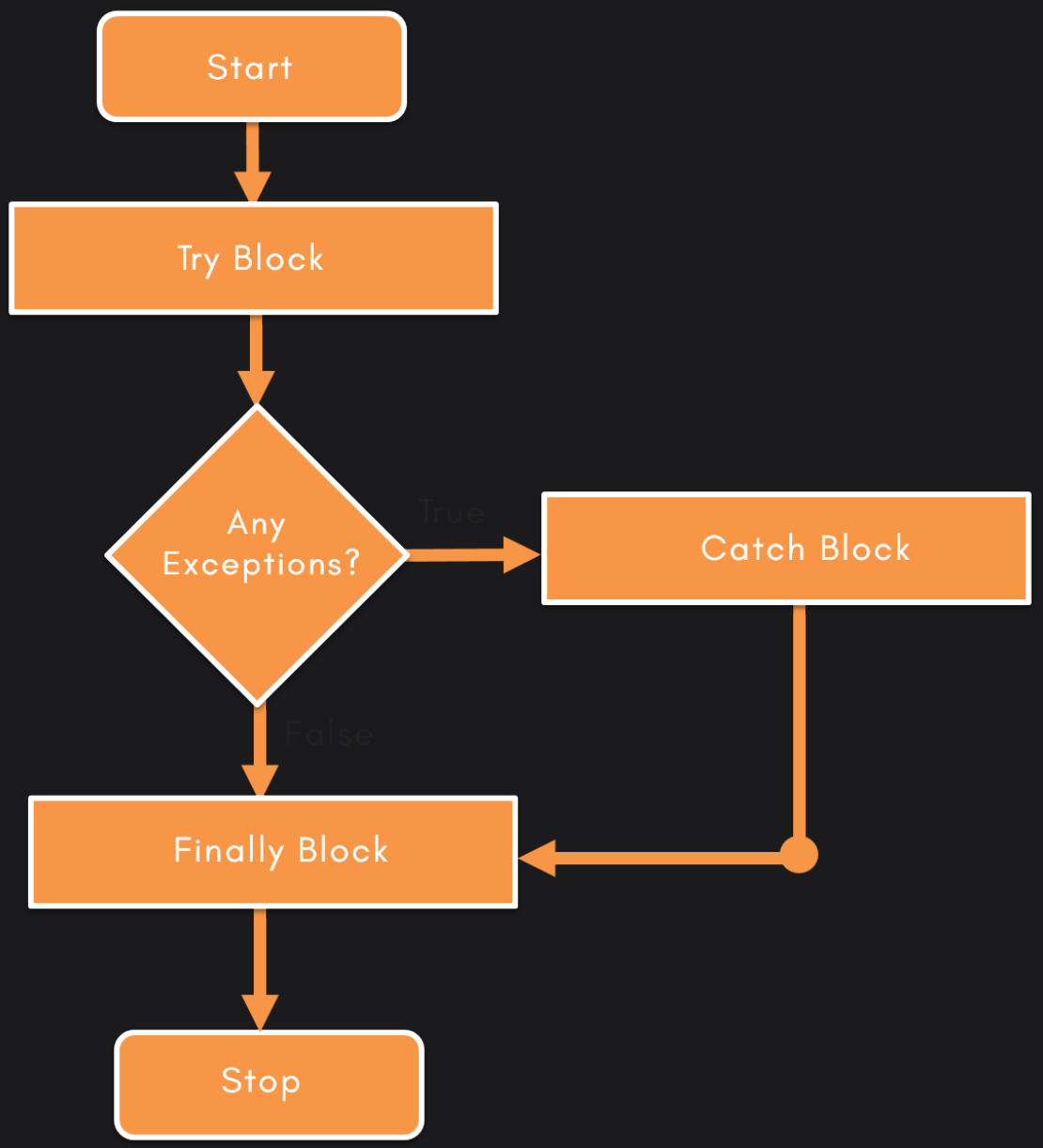
Try running this block of code - if any exception comes up in executing this code, catch is there to help you.

I am here to help you. Trust me I got the ability to handle any exception.

CHEEZYCODE

FINALLY

This block of code is executed - whether you have any exceptions in your code or not.



CHEEZYCODE

TRY CATCH FINALLY

```
try
{
   //code
}
catch(e: Exception)
{
   //showing proper message to the user
}
finally
{
   //code to free any resource
}
```

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

CHEEZYCODE

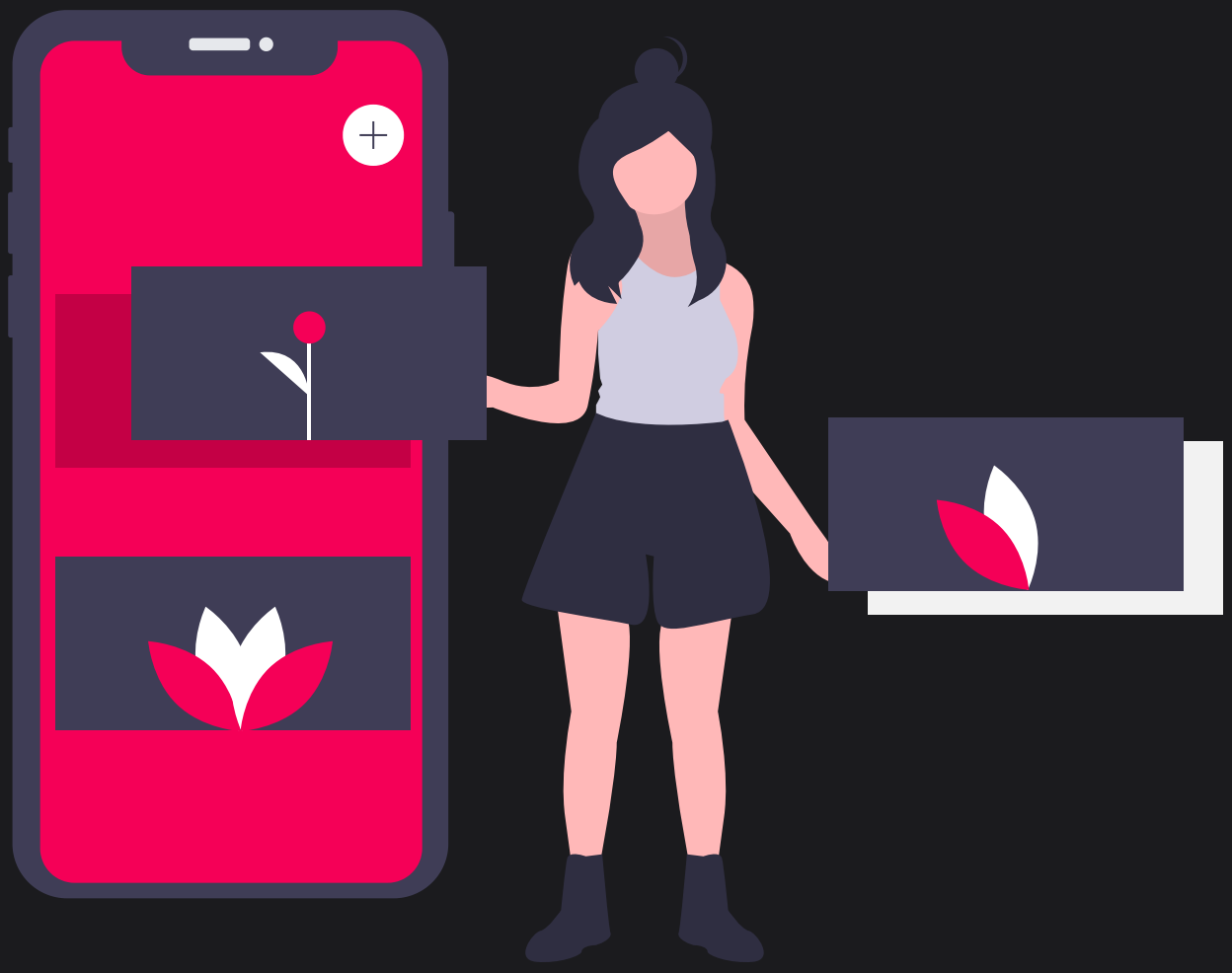
COMMON FACTS

- 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.

CHEEZYCODE

KOTLIN COLLECTIONS
LIST AND MAP



CHEEZYCODE

30

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

CHEEZYCODE

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

```kotlin
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
```

CHEEZYCODE

KOTLIN HIGHER ORDER FUNCTIONS & LAMBDAS

CHEEZYCODE

31

FIRST CLASS FUNCTIONS

Functions in Kotlin are treated as values just like Integers or Strings i.e. functions are treated as First-Class Citizens.

You can store functions in a variable

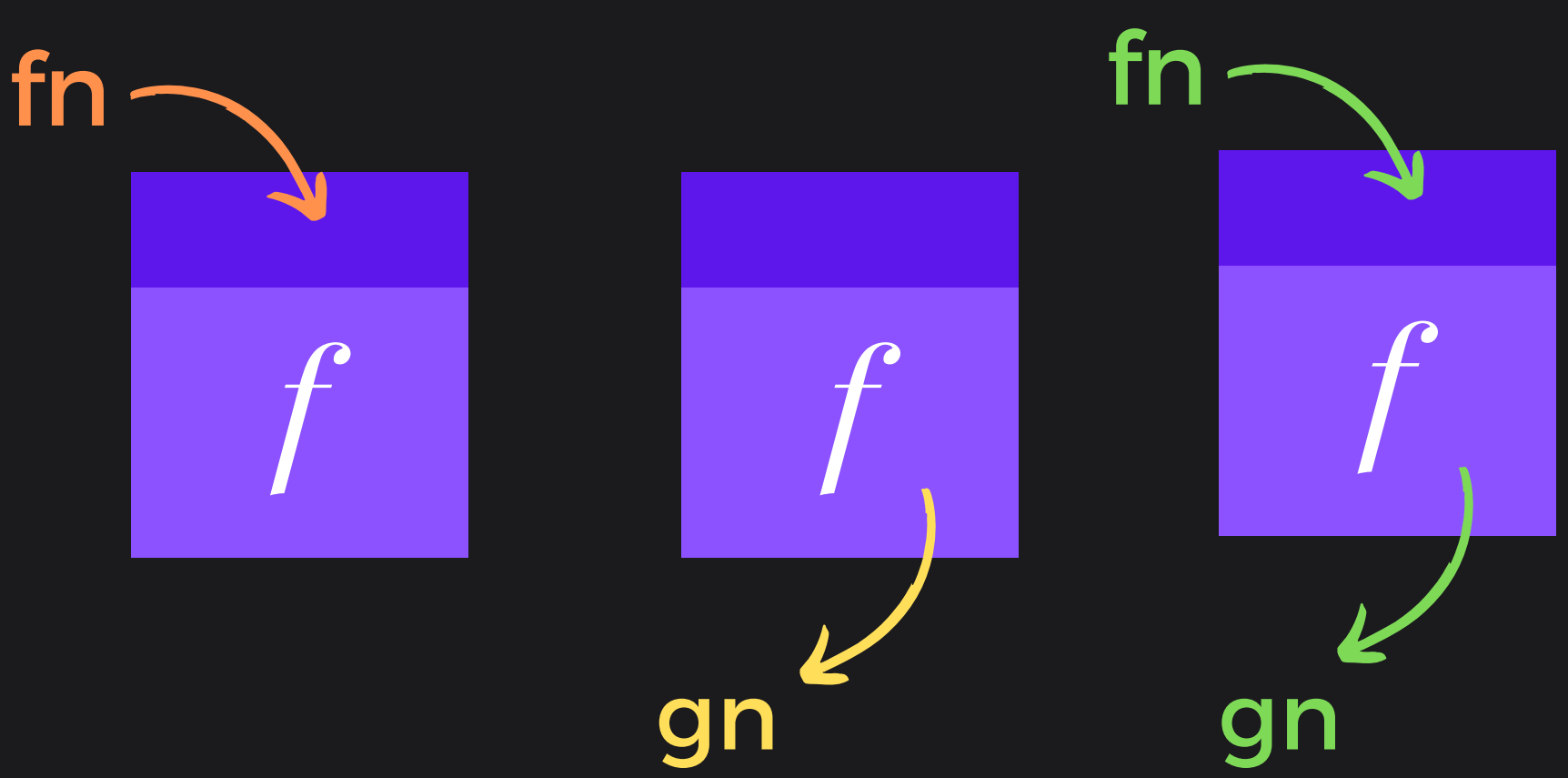
You can pass functions as arguments

You can return functions from functions

CHEEZYCODE

HIGHER ORDER FUNCTIONS

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



CHEEZYCODE

EXAMPLE

```
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)

CHEEZYCODE

LAMBDA EXPRESSION

Lambdas are functions without any name

i.e. Anonymous Function.

Function Literals that do not need any declaration - these are used as an expression that can be assigned to a variable or pass as arguments.

CHEEZYCODE

LAMBDA EXPRESSION

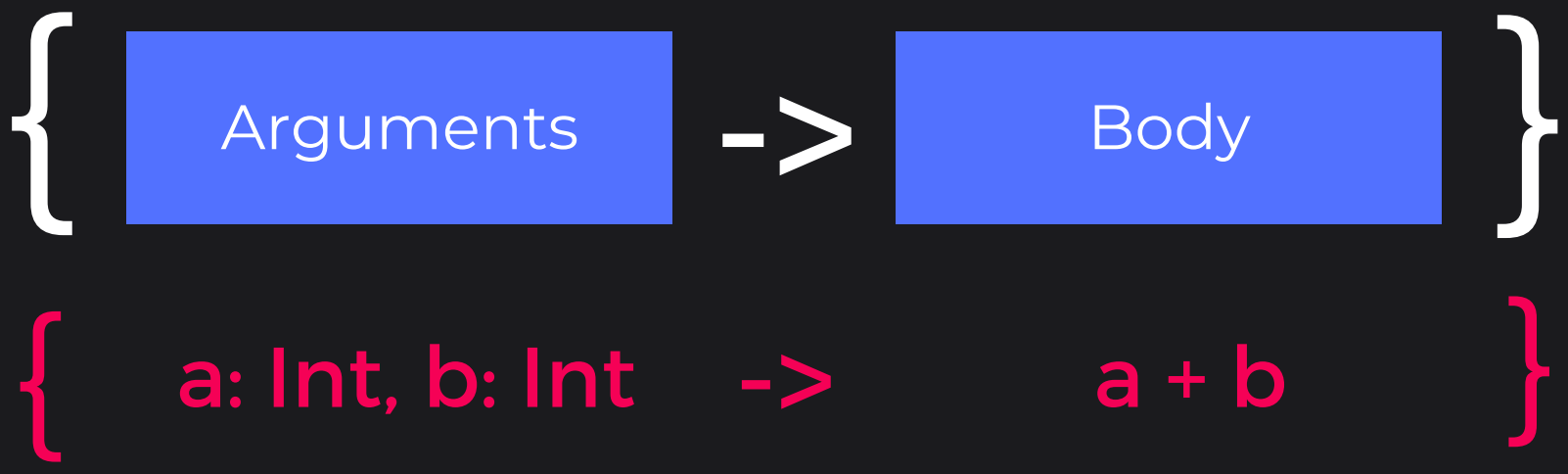
```
fun add (a : Int, b : Int) : Int = a + b
```

Converted
to Lambda

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

As you can see, before the arrow (->) you have your arguments i.e. a & b and after arrow you have your function body.

LAMBDA EXPRESSION



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") }
```

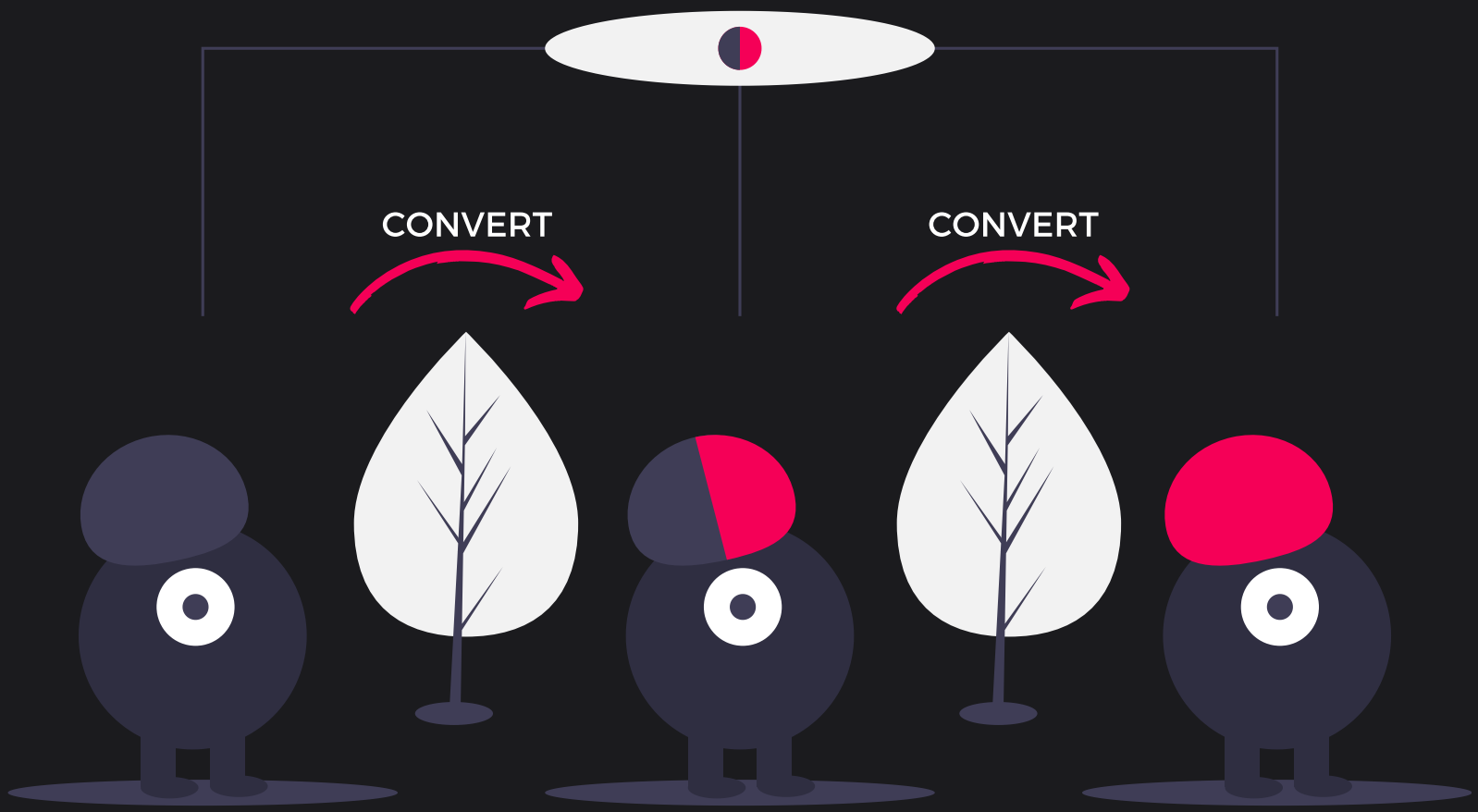
LAMBDAS VARIATIONS

```kotlin
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.

KOTLIN COLLECTIONS - FILTER, MAP & FOREACH

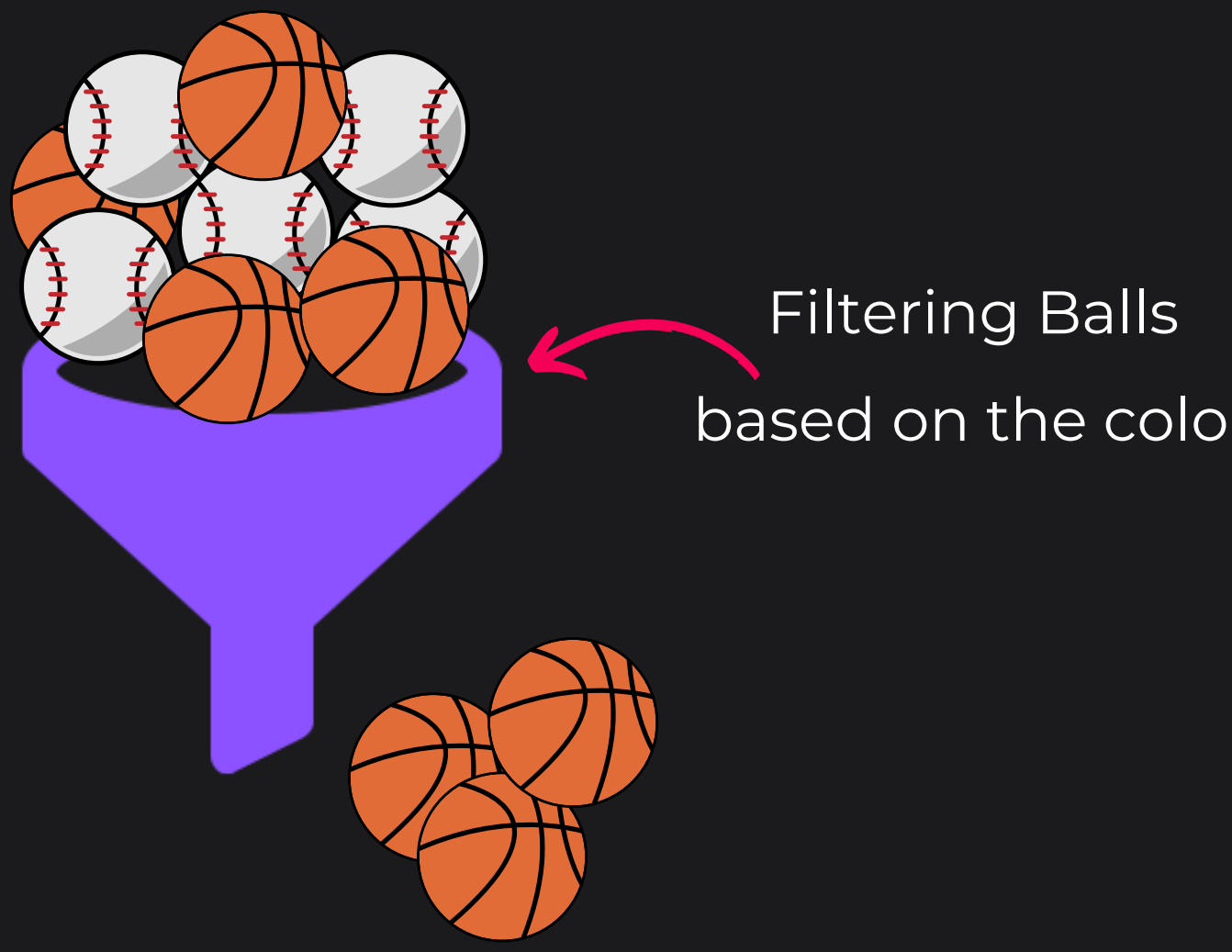


CHEEZYCODE

32

FILTER FUNCTION

Filtering a list based on some condition is a common task. You provide the criteria by which you want to filter a list.



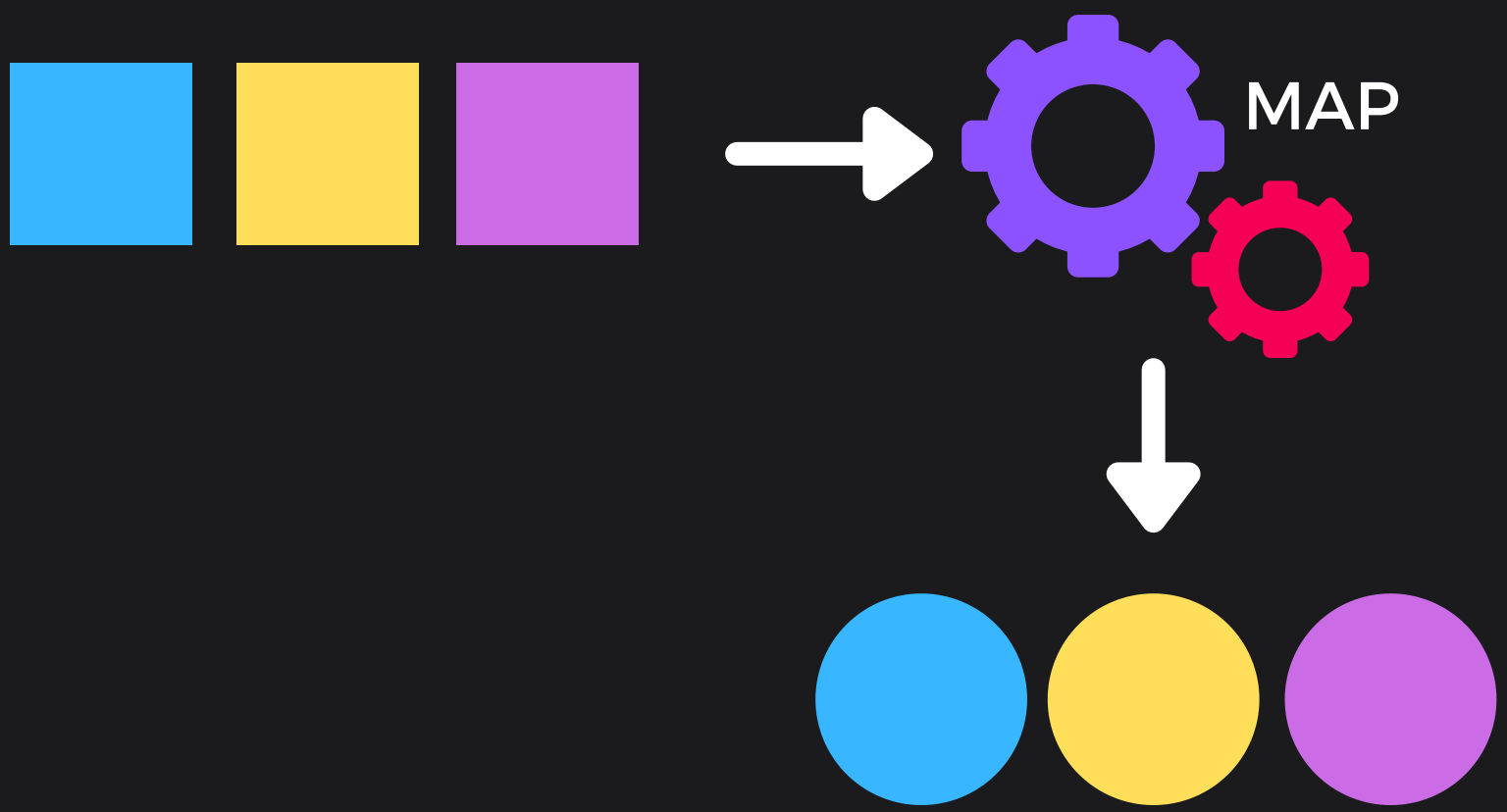
FILTER FUNCTION

```
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



MAP FUNCTION

```kotlin
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)
```

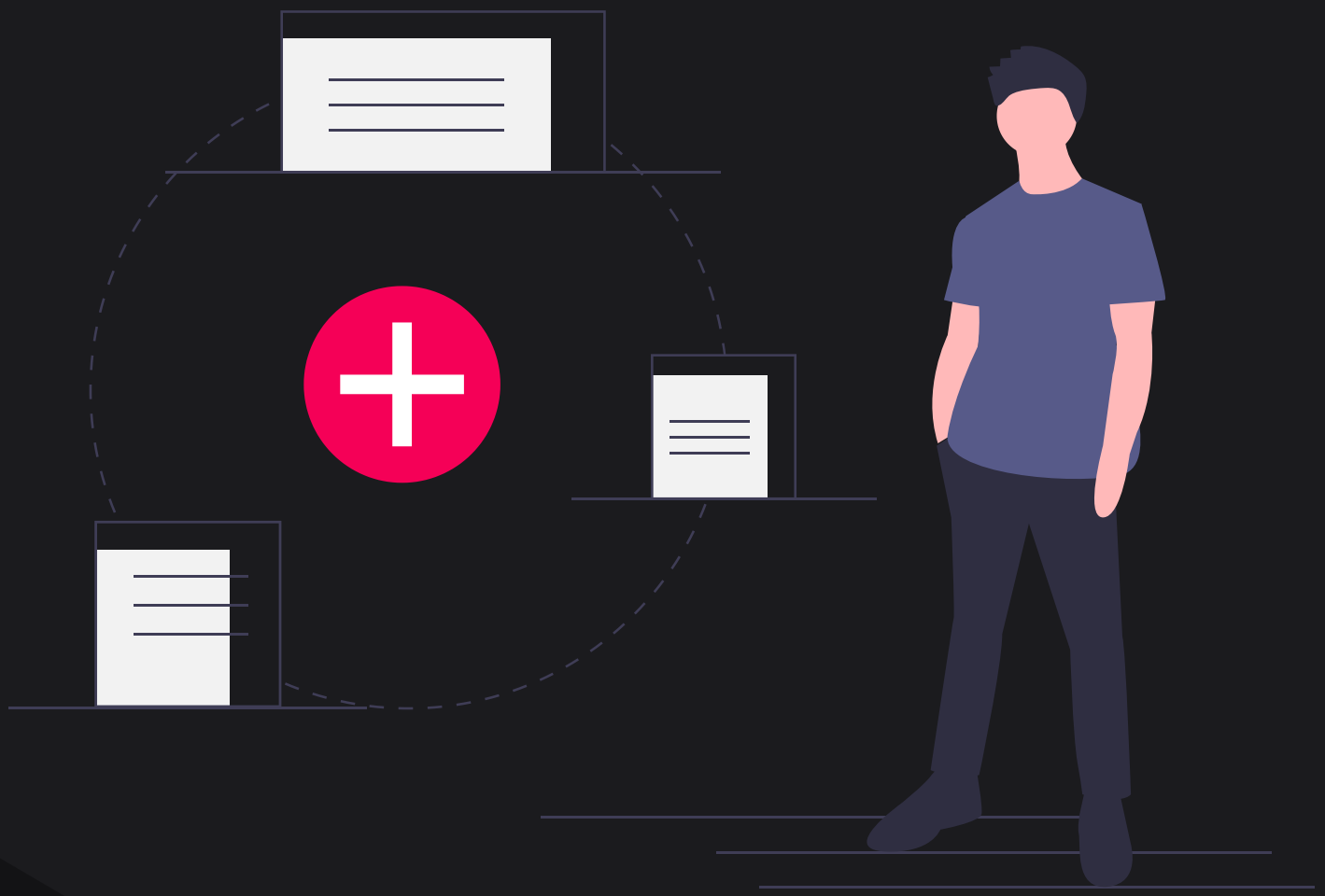
FOR EACH

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

KOTLIN EXTENSION & INLINE FUNCTIONS



CHEEZYCODE

33

EXTENSION FUNCTION

If you want to add additional functionality to a class, you need to first derive that class and then you can add new functionality.

In Kotlin, you can add functionality to a class without defining a new class. You can just extend that class using Extension functions.

EXTENSION FUNCTION

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

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

We call this extension function just like other string methods.

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.

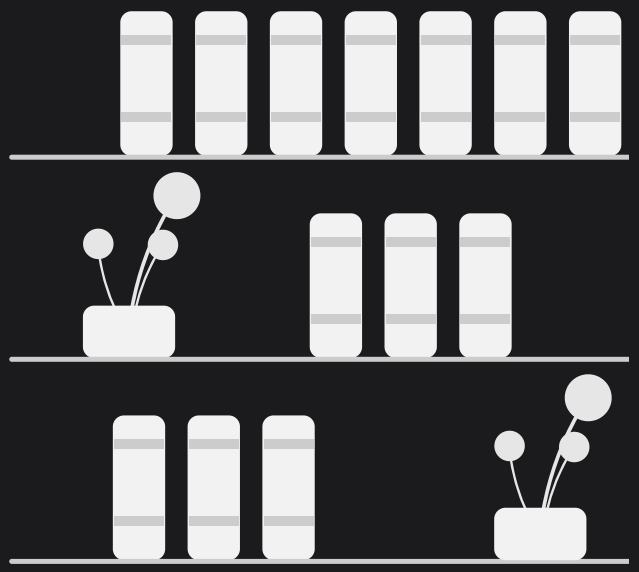
```kotlin
fun main()
{
    lambda(1,2)
    lambda(2,4)
    lambda(3,6)
}

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

```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)

}
```

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

CHEEZYCODE

KOTLIN SCOPE FUNCTIONS
APPLY, LET, WITH, RUN

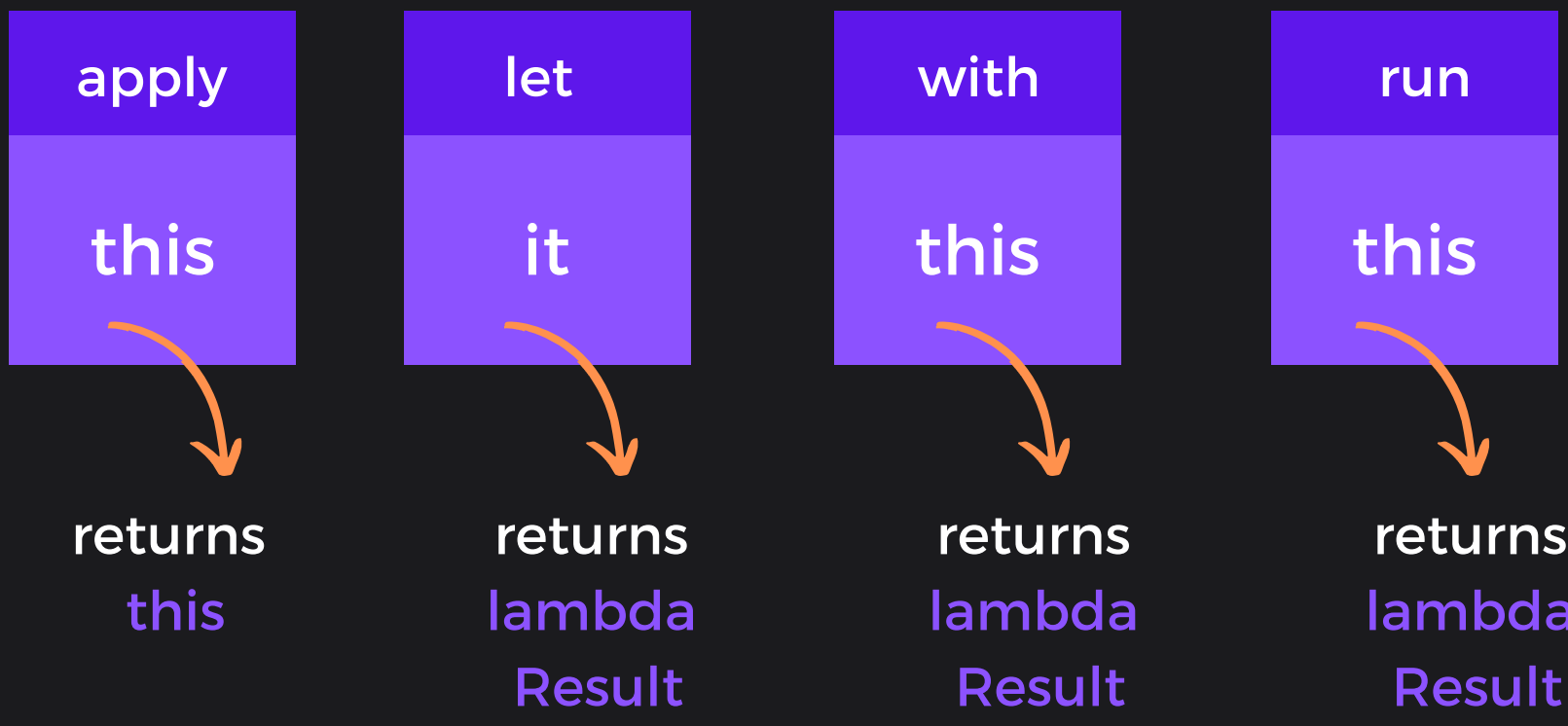


CHEEZYCODE

34

UTILITY FUNCTIONS

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



APPLY FUNCTION

When you have to set multiple properties, you can use apply function. No need to type object.property every time.

```
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

CHEEZYCODE

LET FUNCTION

Another way to set the properties of an object using the let function. Properties are accessed using it. Commonly used with Safe call operator ?.

```
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

Not an extension function, context is passed i.e. object is passed as an argument

```
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

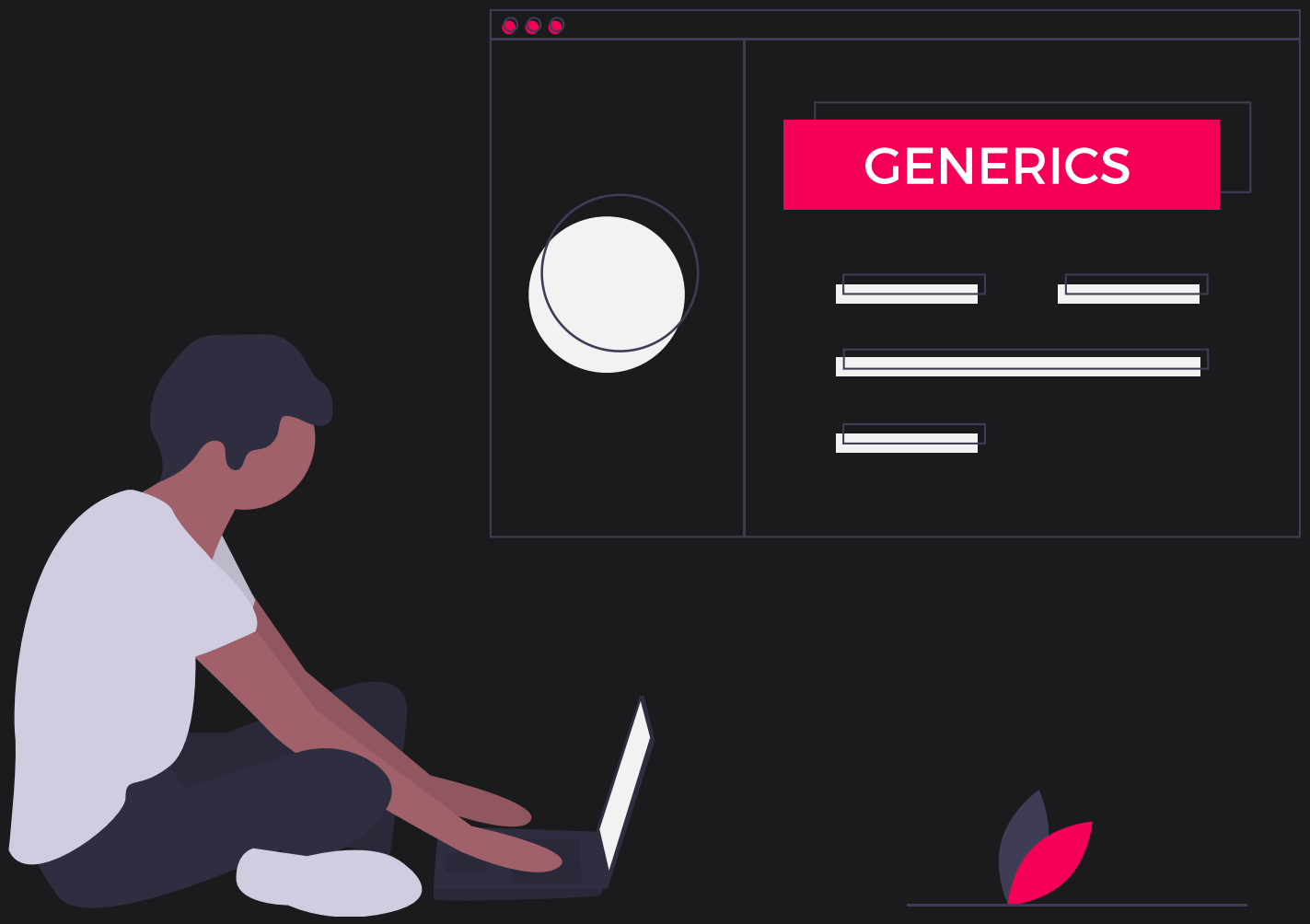
RUN FUNCTION

Extension function on the context object. Similar to with but called as let 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

KOTLIN GENERICS

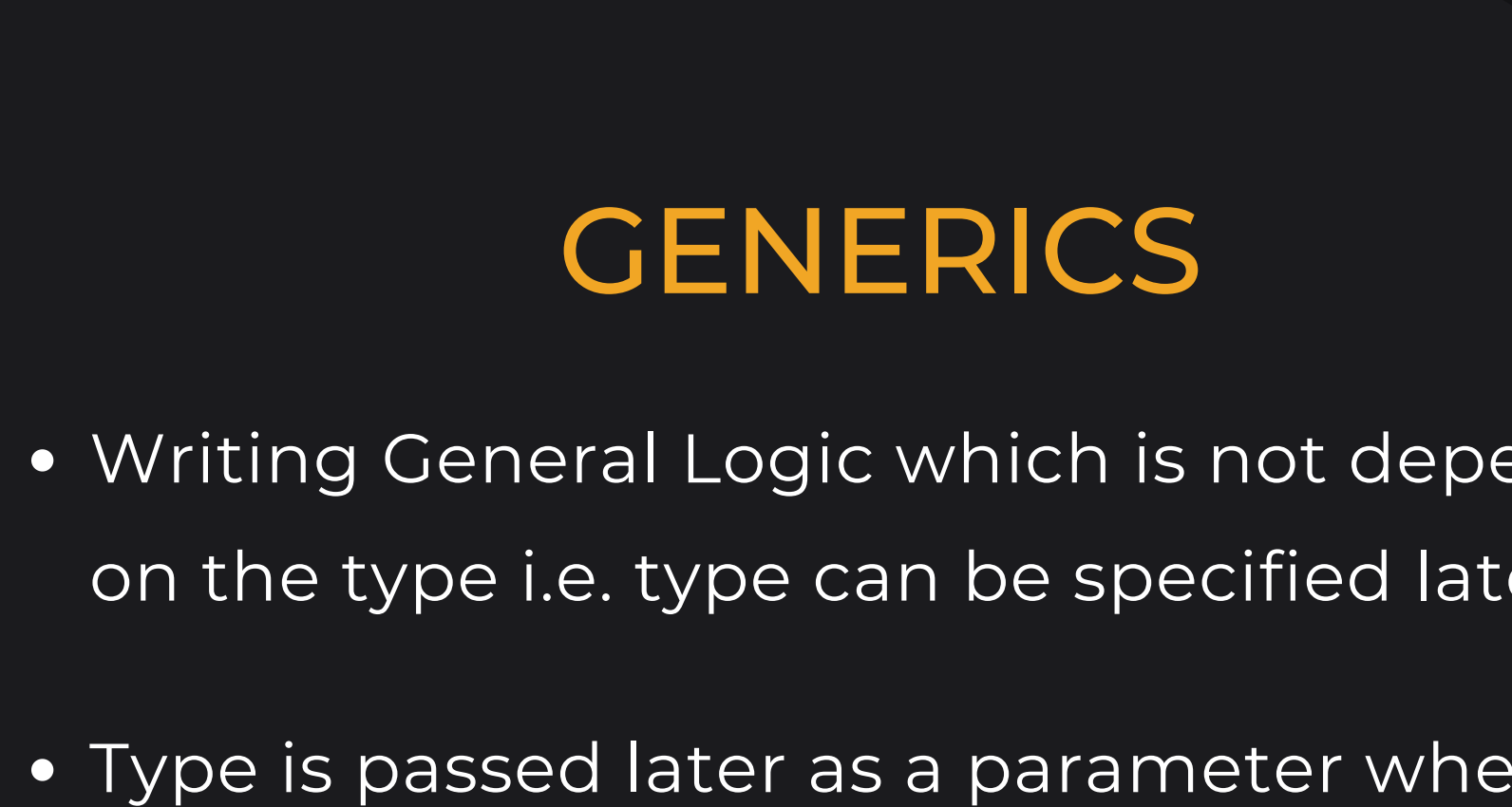


CHEEZYCODE

35

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.



CHEEZYCODE

EXAMPLE

Suppose, you need this same logic for different types. Duplicating the same logic is not a good idea. We can use Generics here.

```
class IntContainer(var data: Int)
{
    fun setValue(value : Int) {
        data = value
    }
    fun getValue(): Int
    {
        return data
    }
}
```

```
class StringContainer(var data: String)
{
    fun setValue(value : String) {
        data = value
    }
    fun getValue(): String
    {
        return data
    }
}
```

EXAMPLE

```
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

GENERICS

- You can use this for methods, classes, & properties.
- You don't need to explicitly typecast Objects.
- Compile Time Checking and Type Safety adds up to the advantage while using Generics

```
public interface MutableMap<K, V> : Map<K, V>

public fun <T> emptyList(): List<T> = EmptyList
```

KOTLIN NESTED INNER CLASS

CHEEZYCODE

36

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")
        }
    }
}
```

WHAT'S THE USE

- When two classes are closely tied to each other - it makes sense to nest them.

- Encapsulation - Nested class defines the implementation details of the main class. The outer class wants to keep that functionality as a separate type and private

- Naming Collision - Not defining it as a nested class will pollute the namespace.

EXAMPLE OF NESTED CLASS

```
class LinkedList
{
    class Node { //// }
}
```

Node for Linked List is different from Node for a Graph or Tree. It makes sense to have LinkedList's Node be a nested class

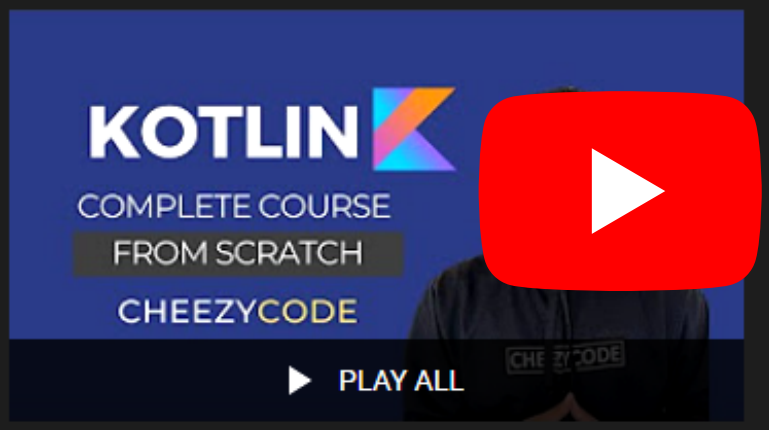
INNER CLASS

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



Kotlin Beginner Tutorials Hindi | Complete Series

41 videos • 72,411 views • Last up[illegible]ed on Aug 6, 2020

Kotlin Tutorials From Scratch. Learn Kotlin Step By Step

If you want to learn Kotlin - you can have a look at this playlist

a look and let us know what you think about this series. We love feedback. Cheers from CheezyCode

Cheezy Code

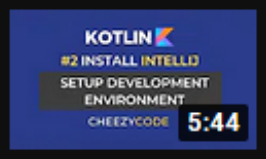
SUBSCRIBE

1 

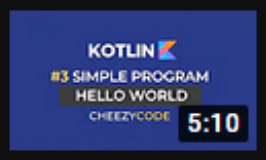
Kotlin - Complete Course From Scratch | CheezyC
Cheezy Code

2 

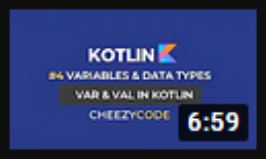
What is Kotlin? Introduction To Kotlin Tutorial | Ch
Cheezy Code

3 

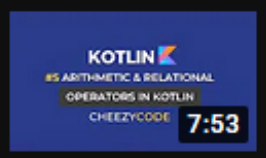
Kotlin - Development Environment Setup Tutorial |
Cheezy Code

4 

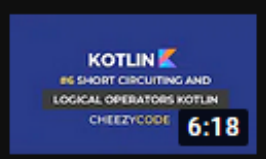
Kotlin - Hello World Program | Compilation & Byte
Cheezy Code

5 

Variables & Data Types in Kotlin Tutorial | var & va
Cheezy Code

6 

Kotlin Operators Tutorial | Arithmetic & Relational
Cheezy Code

7 

Kotlin Logical Operators & Short Circuiting Tutoria
Cheezy Code

CHEEZYCODE

STAY TUNED AND FOLLOW US FOR KOTLIN TUTORIALS

CHEEZYCODE

01

UNDRAW.CO
FOR BEAUTLIFUL
ILLUSTRATIONS

02

CARBON.NOW.SH
FOR BEAUTIFUL
SOURCE CODE IMG

03

KOTLINLANG.ORG
FOR THIS POWEFUL
LANGUAGE

CHEEZYCODE

cheezycode.com