



# Kotlin

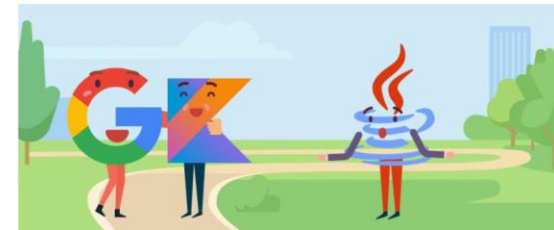
# Table of Contents

1. Declaring Variables
2. Conditional Expressions: If & when
3. Loops
4. Functions
5. OOP

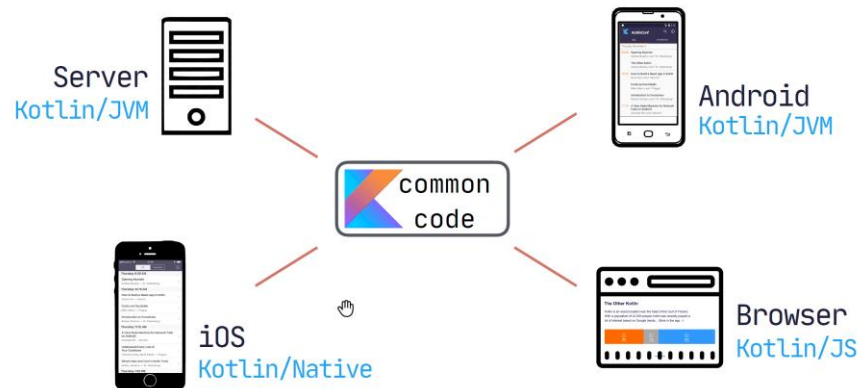
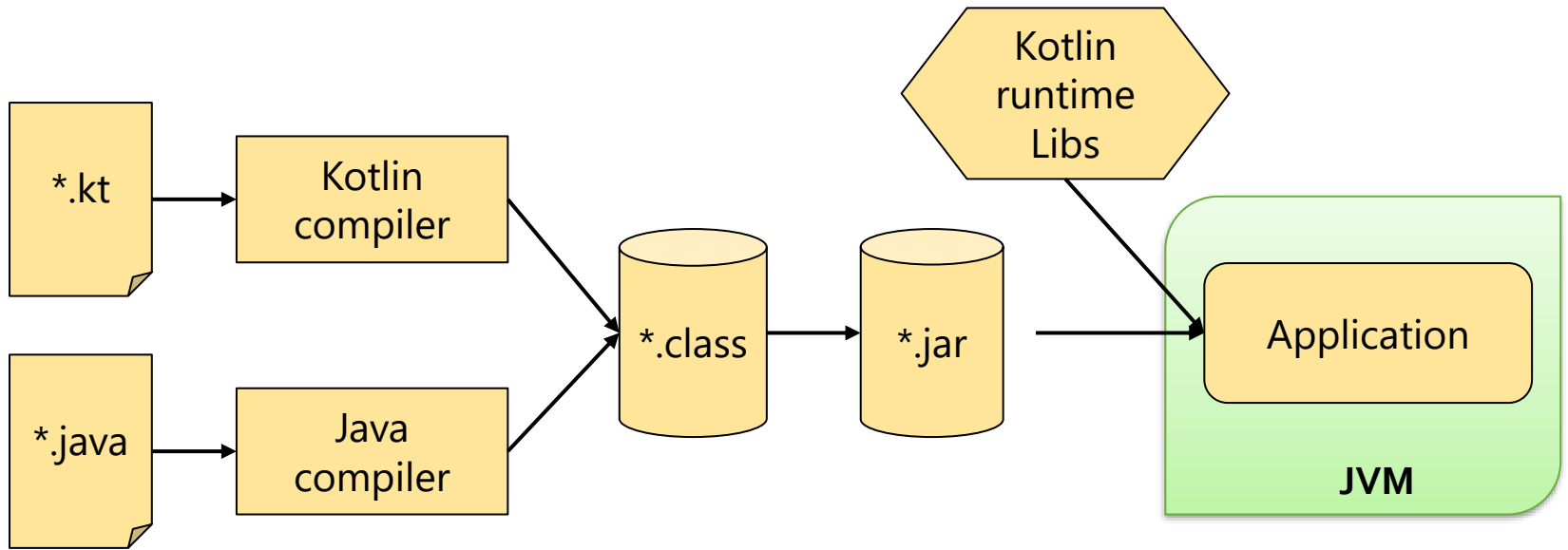
# Declaring Variables

# Highlights of Kotlin

- Statically typed language: Type **validation** at compile time
- Supports **Type Inference**: type automatically determined from the context
- Much more concise and readable code than Java
- Runs on Java JVM
  - Interoperable with Java code + libraries
  - But can also be compiled to JavaScript and iOS Swift
- Both **functional** and **object-oriented**
- Started in 2011 by JetBrains
  - Kotlin v1.0 was released on 15 February 2016
  - On 7 May 2019, Google announced that the Kotlin as its preferred language for Android app development
  - Current version 1.4 (released August 2020)



# How does it work after all?



Can target  
other  
platforms

# val vs. var

- **val** is **immutable** (read-only) and you can only assign a value to them exactly one time
- **var** is **mutable** and can be reassigned

*// val means final - cannot change once initialized*

```
val name = "Ali"
```

```
val c: Int
```

```
c = 1
```

*//Mutable - can be changed*

```
var x = 5
```

```
x += 1
```

*/\* Type is auto-inferred - The variable datatype is derived from the assigned value \*/*

```
val city = "Doha"
```

# Main Data Types

- Int
- Short
- Long
- Byte
- Double
- Float
- Boolean
- Char
- String
- Any (equivalent to Object in Java)

# Strings

*//Strings and String Template*

```
val firstName = "Ali"  
val lastName = "Faleh"
```

- **String Template** allow creating dynamic templated string with placeholders (instead of string concatenation!)
  - Simple reference uses **\$** and an expression uses **\${}**

```
val fullName = "$firstName $lastName"  
val sum = "2 + 2 = ${2 + 2}"
```

*//Multiline Strings*

```
val multiLinesStr = """  
    First name: $firstName  
    Last name: $lastName  
    """
```



# Convert a number to a string

- Use number's *toString* method

```
val num = 10
```

```
val str = num.toString()
```

# Convert a string to a number

- Use string's *toInt* method

```
num = str.toInt()
```

## Smart Cast

```
var myVar: Any = "Ali"  
//Smart auto-cast  
if (myVar is String) {  
    println(myVar.last())  
}
```

# Nullable Types

- By default variables in Kotlin are **non-nullable**
- Nullable variables are declared **explicitly** to accept a null using **?** after the data type
- **Syntax:**
  - `val iCannotBeNull = "Not Null"`
  - `val iCanBeNull: String? = null`
- `val nullableName: String = null`
  - Compilation Error: Can't assign null to a non-null String
- `val nullableName: String? = null`
  - Compiles ok



# Null Safety

- Safe calls:

```
val len = if (name != null) {  
    name.length  
}
```

## Safe accessor ( ? )



```
val len = name?.length
```

- Chaining:

```
student1?.department?.head?.name
```

- Elvis Operator ( ?: )

```
val len = if (name != null) name.length else 0
```

```
// Better syntax is to use the Elvis operator (?:)
```

```
val len = name?.length ?: 0
```

```
// !! means access it as non-null and throw an exception if null
```

```
val len = name!!.length
```

# Comments

*// slash slash line comment*

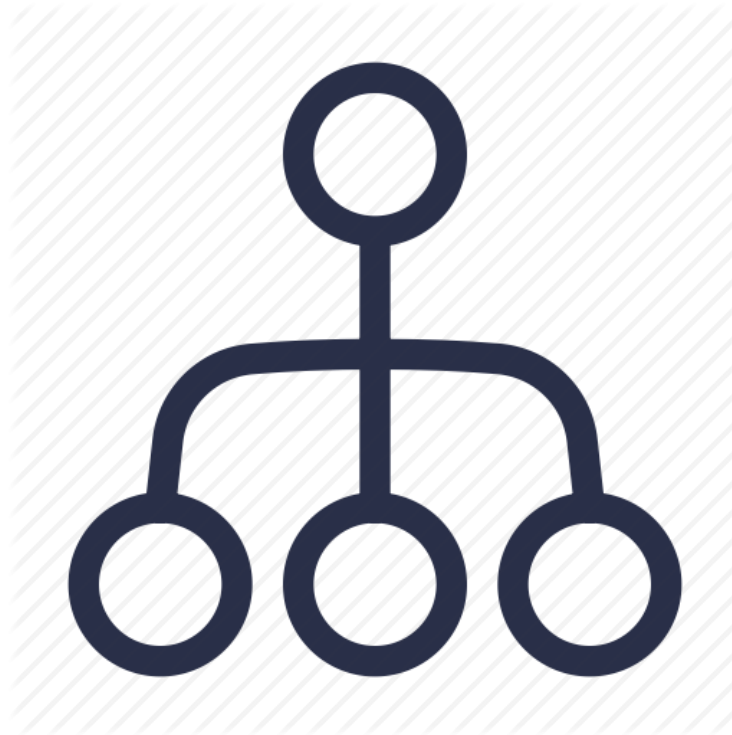
*/\**

*slash star*

*block comment*

*\*/*

# Control Flow: if, when expressions



# if-else expression

```
val age = 20
```

```
// Using 'if' as an expression
```

```
val ageCategory = if (age < 18) {  
    "Teenager"  
} else {  
    "Young Adult"  
}
```

# When expression

- Assign a value based on matching condition

```
val month = 8
val season = when (month) {
    12, 1, 2 -> "Winter"
    in 3..5 -> "Spring"
    in 6..8 -> "Summer"
    in 9..11 -> "Autumn"
    else -> "Invalid Month"
}
```

# Equals & reference equality

```
val set1 = setOf(1, 2, 3)  
val set2 = setOf(1, 2, 3)
```

calls equals

```
set1 == set2
```

**true**

checks reference equality

```
set1 === set2
```

**false**

- `===` return true only if both variables hold a reference to the same object



**while (...)**  
**do { ... }**  
**for { ... }**  
**Loops**

Execute Blocks of Code Multiple Times



# While Loop

- While Loop:

```
while (condition) {  
    statements;  
}
```



- Do-While Loop:

```
do {  
    statements;  
}  
while (condition);
```

# for Loop Example

```
val names = listOf("Sara", "Fatima", "Ali")
```

```
for (name in names) {  
    println(name)  
}
```

*// Loop with index and value*

```
for ( (index, value) in names.withIndex()) {  
    println("$index -> $value")  
}
```

# Ranges

- Usually defined by: `1..100`
- `1 until 100` // Range excludes 100
- Negative step: `100 downTo 40`
- Decrement by 3  
`100 downTo 40 step 3`

## Caution!

```
val notARange = 100 to 40  
// => Pair(100, 40)
```

- To check if a value belongs to the range:  
`val is5inRange = 5 in range`

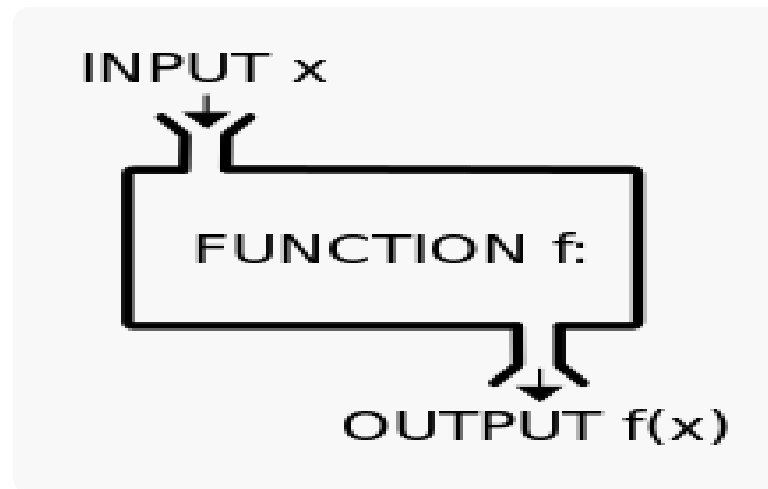
# Ranges

```
if (i in 1..10) { // 1 <= i && i <= 10    for (i in 1..4 step 2) print(i) // "13"
    println(i)
}

for (i in 1..4) print(i) // "1234"        for (i in 4 downTo 1 step 2)
                                           print(i) // "42"

for (i in 4..1) print(i) // No Output    // i in [1, 10), 10 is excluded
for (i in 4 downTo 1)                    for (i in 1 until 10) {
    print(i) // "4321"                    println(i)
}
```

# Functions



# Functions

- Can be declared at the **top level** of a file (without belonging to a class)
- Can have a **block or expression body**
- Can have default parameter values to avoid method overloading
- Can use **named** arguments in a function call

```
fun max(a: Int, b: Int): Int { //name - parameters - return type  
|   return if(a>b) a else b //function block body  
}
```

```
fun max(a: Int, b: Int) = if(a>b) a else b //expression body
```

```
max(a = 1, b = 2) //call with named arguments  
max( a: 1, b: 2)
```

# Functions

*// Function with block body*

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

*// Function with expression body*

*// Omit return type*

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

*//Arrow function - called Lambda expression*

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



# Unit return type

- When defining a function that doesn't return a value, we can use **Unit** as the return type (Unit is equivalent to void in Java)
  - Specifying Unit as a return type is NOT mandatory can omit it

```
fun display(value : Any) : Unit {  
    println(value)  
}
```

# Use default parameters instead of overloads

```
fun print() {  
    print(",")  
}
```

```
fun print(separator: String) {  
}
```

```
fun main() {  
    print("|")  
}
```

```
fun print(separator: String = ",") {  
}
```

```
fun main() {  
    print(separator = "|")  
}
```

# Extension Function

- Enable adding functions and properties to existing classes

*// Extension method extending Int class*

```
fun Int.isEven() = this % 2 == 0
```

```
fun main() {  
    val num = 10  
    println("Is $num even: ${num.isEven()}")  
}
```

# Extension Function Example

```
fun String.lastChar() = this.get(this.length - 1)
```



this can be omitted

```
fun String.lastChar() = get(length - 1)
```

```
val c: Char = "abc".l
```

- λ lastChar() for String in com
- λ last {...} (predicate: (Char
- λ last() for String in kotlin
- λ lastOrNull {...} (predicate:
- λ lastOrNull() for String in k
- ✓ length

# Infix function calls

- Functions marked with the **infix** keyword can be called using the infix notation (omitting the dot and the parentheses for the call)
- Infix function must satisfy 3 requirements:
  - Must be member function or extension function.
  - Must have a single parameter.
  - The parameter must not accept a variable number of arguments

```
infix fun Int.add(b : Int) : Int = this + b
```

```
fun main() {  
    val x = 10.add(20)  
    val y = 10 add 20    // infix call  
}
```

# Exceptions

- Throw:

```
throw Exception("Invalid input")
```

- Handling

```
try {  
}  
catch (e: Exception) {  
}  
finally {  
}
```

- Expression

```
val num = try { input.toInt() }  
           catch (e: NumberFormatException){ null }
```

# OOP

# Java Class vs. Kotlin Class


```
public class Person {  
    private final String name;  
    private final int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

person.getName() 



Concise primary constructor


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

person.name 



# Class

```
class Person(val firstName: String,  
             val lastName: String,  
             val age: Int) {  
    val fullName: String  
        get() = "$firstName $lastName"  
    fun isUnderAge() = age < 18  
}
```



Properties

- **Instantiate:**

```
val student = Person ("Fatima", "Ali", 18)
```

- **Named arguments:**

```
val student = Person (firstName = "Fatima",  
                      lastName = "Ali", 18)
```

# Properties are directly accessible without getters / setters

- `val` – read only properties
- `var` – read/write properties
- The primary constructor **cannot** contain any code.

```
class Person(val firstName: String,  
             val lastName: String,  
             var age: Int) {  
    val fullName: String  
        get() = "$firstName $lastName"  
    fun isUnderAge() = age < 18  
}
```

```
val student = Person ("Fatima", "Ali", 18)  
student.age = 20
```

# Class with a computed property

```
class Rectangle(val height: Int, val width: Int) {  
    val isSquare: Boolean  
        get() {  
            return height == width  
        }  
}
```

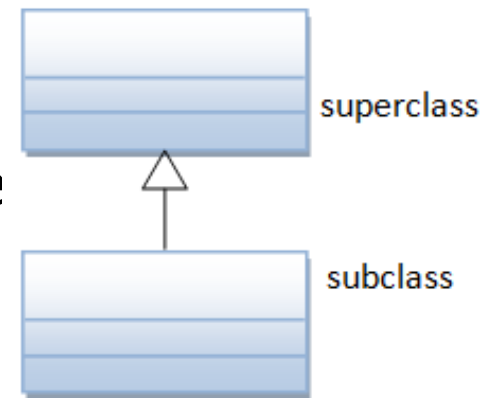
# Secondary Constructor

```
class Conference(val name: String,  
                 val city: String,  
                 val isFree: Boolean = true) {  
    var fee : Double = 0.0  
  
    // Secondary Constructor  
    constructor(name: String,  
                city: String,  
                fee: Double) : this(name, city, false) {  
        this.fee = fee  
    }  
}  
  
fun main() {  
    al conference = Conference("Kotlin Conf.", "Doha", 200.0)  
}
```

# Inheritance

- **Ideas**

- Common properties and methods are placed in a **superclass** (also called *parent class* or *base class*)
- You can create a subclass that **inherits** the properties and methods of the super class
  - Subclass also called *child class* or *derived class*
- Subclass can extend the superclass by **adding new properties/methods** and/or **overriding the superclass methods**



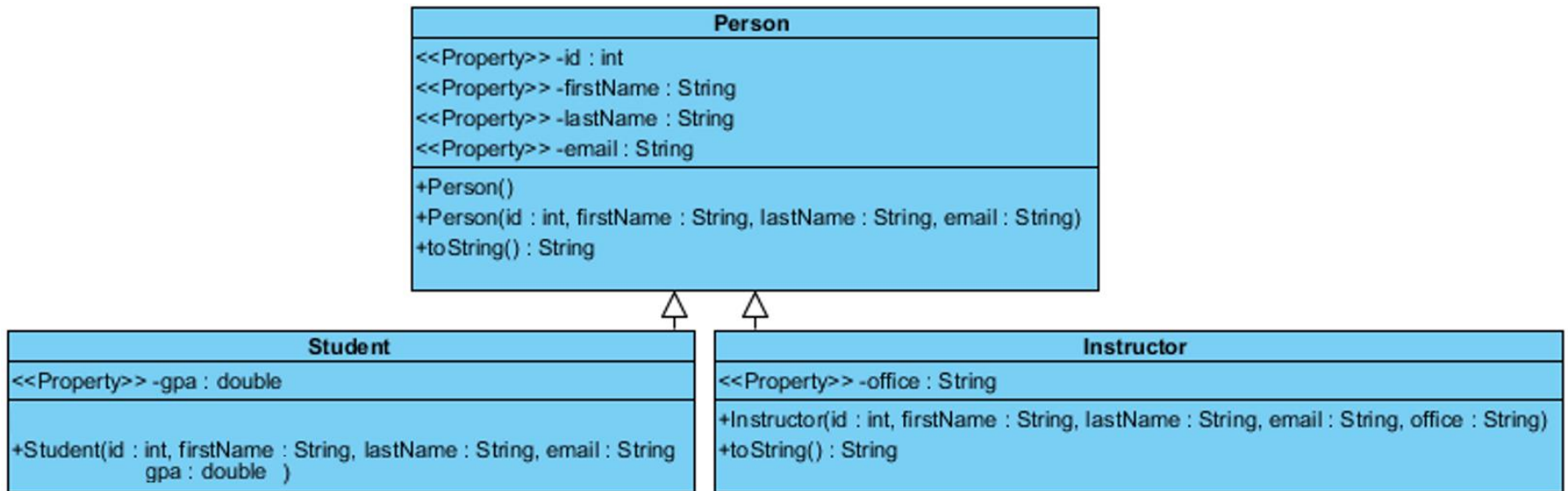
- **Syntax**

```
class SubClass( ... ) : SuperClass( ... ) { ... }
```

- **Motivation**

- Allow **code reuse**. **Common properties and methods are placed in a super class** then inherited by subclasses (i.e., avoids writing the same code twice to ease maintenance)

# Inheritance – Person Example



- The Person class has the common properties and methods
- Each subclass can add its own specific properties and methods (e.g., **office** for Instructor and **gpa** for Student)
- Each subclass can **override** (redefine) the parent method (e.g., Instructor class overrode the `toString()` method).

# Inheritance – Person Example

```
open class Person( ... ) { ... }
```

```
class Student(firstName: String,  
              lastName: String,  
              age: Int,  
              val gpa: Double  
              ) : Person(firstName, lastName, age) {  
  
    /*  
    - Override a base class method  
    - super keyword to call the implementation of the base class  
    */  
    override fun toString() = "${super.toString()}. GPA: ${gpa}"  
}
```

- Add **open** keyword to the base class and to properties and methods to be overridden

# Data Classes

- Data classes provide autogenerated implementations of **equals()**, **hashCode()**, **copy()** and **toString()** methods

```
data class User(val name: String, val age: Int)
val ali = User(name = "Ali", age = 18)
```

*//Copy:*

```
val olderAli = ali.copy(age = 19)
```

*//Destructuring*

```
val (name, age) = ali
```

*// prints "Ali, 18 years of age"*

```
println("$name, $age years of age")
```



# Use 'copy' method for data classes

```
class Person(val name: String,  
             var age: Int)  
  
fun happyBirthday(person: Person) {  
    person.age++  
}
```

```
data class Person(val name: String,  
                  val age: Int)  
  
fun happyBirthday(person: Person) =  
    person.copy(  
        age = person.age + 1)
```

# No static keyword -> alternatives

- **Top-level** functions and properties  
(e.g. placed outside the class)
- **Companion objects:** special object inside a class to place static properties and methods
- **object** declaration used to create a **Singleton** (i.e., a single instance for the whole app):
  - Used for declaring the class
  - And providing a single instance of it

```
class Foo {  
    companion object {  
        fun bar() {  
            //...  
        }  
    }  
}
```

```
object Singleton {  
    fun doSomething() {  
        //..  
    }  
}
```

Foo.bar()

Singleton.doSomething()

# object = singleton

- Once instance for the whole app

```
object Util {  
    fun getNumberOfCores() = Runtime.getRuntime().availableProcessors()  
  
    val randomInt: Int  
        get() = Random().nextInt()  
}  
  
fun main() {  
    println(Util.getNumberOfCores())  
    println(Util.randomInt)  
}
```

# Enum class

- Represents an enumeration

```
enum class Gender {  
    FEMALE, MALE  
}
```

```
enum class Direction {  
    LEFT, RIGHT, UP, DOWN  
}
```

# Abstract Classes

- Idea
  - Use an abstract class when you want to define a **template** to guarantee that all **subclasses** in a hierarchy will have certain common methods
  - Abstract classes can contain implemented methods and **abstract methods** that are NOT implemented
- Syntax

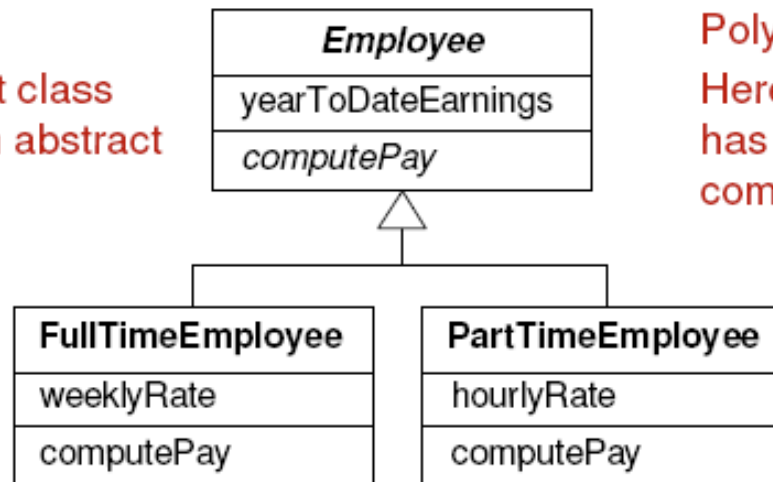
```
abstract class SomeClass() {  
    fun abstract method1(...): SomeType // No body  
    fun method2(...): SomeType { ... } // Not abstract  
}
```
- Motivation
  - Guarantees that all subclasses will have certain methods => **enforce a common design.**
  - Lets you make collections of mixed type objects that can be processed polymorphically

# Abstract Classes

- An abstract class has one or more abstract properties/methods that subclasses **MUST** override
  - Abstract properties/methods do not provide implementations because they **cannot be implemented in a general way**
- An abstract class cannot be instantiated

Abstraction:

Employee is an abstract class and *computePay()* is an abstract operation (italicized)



Polymorphism:

Here, each type of Employee has its own version of *computePay()*

# Abstract Class Example

## Shape.kt

```
abstract class Shape {  
    abstract val area: Double  
    open val name: String  
        get() = "Shape"  
}
```

## Rectangle.kt

```
class Rectangle(val width: Double,  
                val height: Double) : Shape() {  
    override val area: Double  
        get() = width * height  
  
    override val name: String  
        get() = "Rectangle"  
}
```

## Circle.kt

```
class Circle(val radius: Double) : Shape() {  
    override val area: Double  
        get() = Math.PI * radius.pow(2)  
  
    override val name: String  
        get() = "Circle"  
}
```

# Interfaces

- Idea
  - **Interfaces** are used to define a set of common properties and methods that must be implemented by **classes not related by inheritance**
  - The interface specifies **what** methods a class must perform but does not specify **how** they are performed
- Syntax

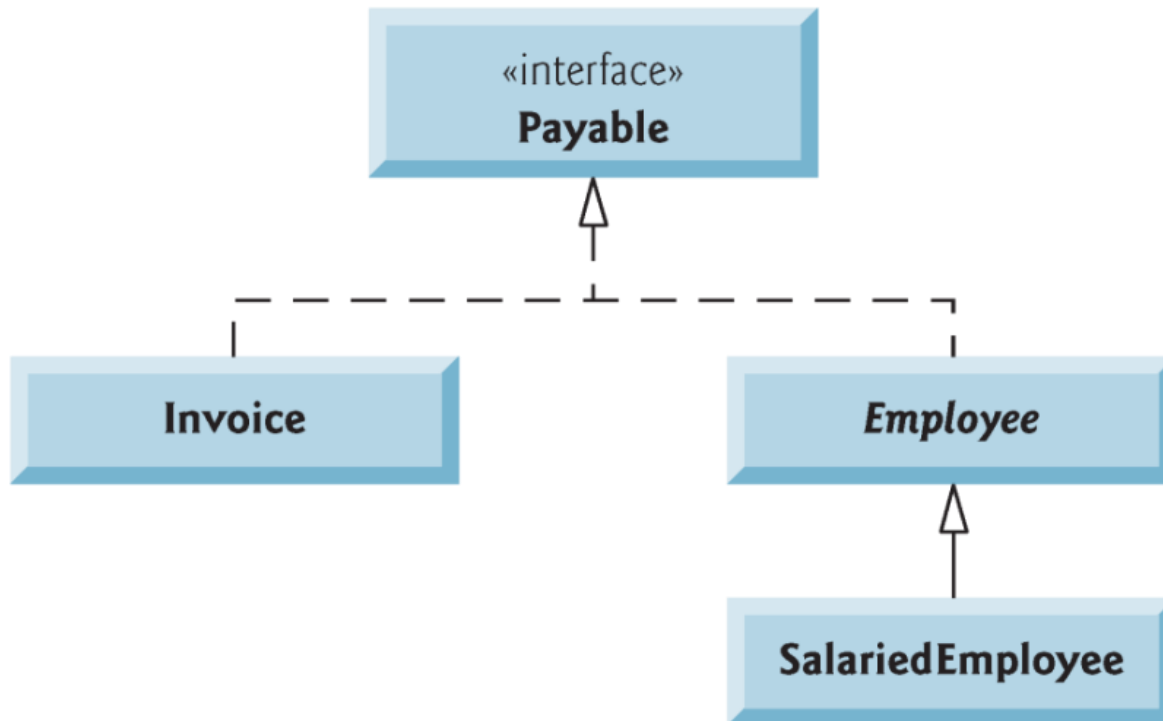
```
interface SomeInterface {  
    fun method1(...): SomeType // No body  
    fun method2(...): SomeType // No body  
}  
class SomeClass() : SomeInterface {  
    // Real definitions of method1 and method 2  
}
```

- Motivation
  - Interfaces enables requiring that **unrelated classes implement a set of common methods**
  - **Ensure consistency** and guarantee that classes has certain methods
  - Lets us make collections of mixed type objects that can processed polymorphically



# Interface Example

- A finance system has Employees and Invoices
- Employee and Invoice are not related by inheritance
- But to the company, they are both *Payable*



# Interface Example

## Payable.kt

```
interface Payable {  
    fun getPayAmount(): Double  
}
```

## Employee.kt

```
class Employee ( ... ) : Payable {  
    ...  
    override fun getPayAmount() = salary  
    ...  
}
```

## Invoice.kt

```
class Invoice ( ... ) : Payable {  
    ...  
    override fun getPayAmount() = totalBill  
    ...  
}
```

# Polymorphism Using interfaces

- A way of coding **generically**
  - way of referencing many related objects as one generic type
    - Cars and Bikes can both `move()` → refer to them as **Transporter** objects
    - Phones and Teslas can both `getCharged()` → refer to them as *Chargeable* objects, i.e., objects that implement **Chargeable** interface
    - Employees and invoices can both `getPayAmount()` → refer to them as *Payable* objects

```
for (payable : payables ) {  
    println ( payable.getPayAmount() )  
}
```

# Abstract Class vs. Interface

- Abstract classes and interfaces cannot be instantiated
- Abstract classes and interfaces may have abstract methods that must be implemented by the subclasses
- Classes that implement an interface **can be from different inheritance hierarchies**
  - An interface is often used when unrelated classes need to provide **common properties and methods**
  - When a class implements an interface, it establishes an **IS-A** relationship with the interface type. Therefore, interface references can be used to invoke polymorphic methods just as an abstract superclass reference can.
- Concrete subclasses that extend an abstract superclass are **all related to one other by inheriting from a shared superclass**
- Classes can extend only ONE abstract class but they may implement more than one interface

# Summary

- Inheritance = “factor out” the common properties and methods and place them in a single superclass
  - => Removing code redundancy will result in a smaller, more flexible program that is easier to maintain.
- Interfaces are contracts, can’t be instantiated
  - force classes that implement them to define specified methods
- Polymorphism allows for generic code by using superclass/interface type variables to manipulate objects of subclass type
  - make the client code more **generic** and ease extensibility

# Kotlin Resources

- Kotlin online courses
  - <https://www.coursera.org/learn/kotlin-for-java-developers>
  - <https://www.udacity.com/course/kotlin-bootcamp-for-programmers--ud9011>
- Kotlin learning resources
  - <https://developer.android.com/courses/kotlin-bootcamp/overview>
  - <https://codelabs.developers.google.com/kotlin-bootcamp/>
  - <https://kotlinlang.org/docs/reference/>