Kotlin

Object-Oriented Programming

Introduction and Basic Principles

@kotlin | Developed by JetBrains

Object-Oriented Programming

Object-oriented programming (OOP)

A programming paradigm based on the representation of a program as a set of objects and interactions between them.

Class and Object

Class

A set of **attributes** (fields, properties, data) and related **methods** (functions, procedures) that together represent some abstract entity.

Attributes store **state**,
while **procedures** express **behavior**.

Classes are sometimes called prototypes.

Object

An **instance of a class**,
which has its own specific state.

```
class Person:
  ●  String attribute name
  ●  Boolean attribute married
  ●  Method greet

Person x:
    name = "Olek",
    married = false


x.greet()
```

Abstraction

Objects are data abstractions with internal representations,
along with methods to interact with those internal representations.

There is no need to expose internal implementation details,
so those may stay “inside” and be hidden.

Create Kotlin Classes

```
class UselessClass // Creating a class in Kotlin

fun main() {
   val uselessObject = UselessClass() // () here is constructor invocation
}
```

The “new“ keyword used for object creation is NO longer used.

Constructors

The **primary** constructor, which is used by default.
If it is empty, the brackets can be omitted

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

    init {
        println(“Initializer Block”)
    }

    constructor(name: String) : this(name, 0)
}
```

The **secondary** constructor

The order of initialization: the primary constructor → the `init` block → the secondary constructor

Primary Constructor

A class in Kotlin has a primary constructor and possibly one or more secondary constructors.

The primary constructor is declared in the class header,
and it goes after the class name and optional type parameters.

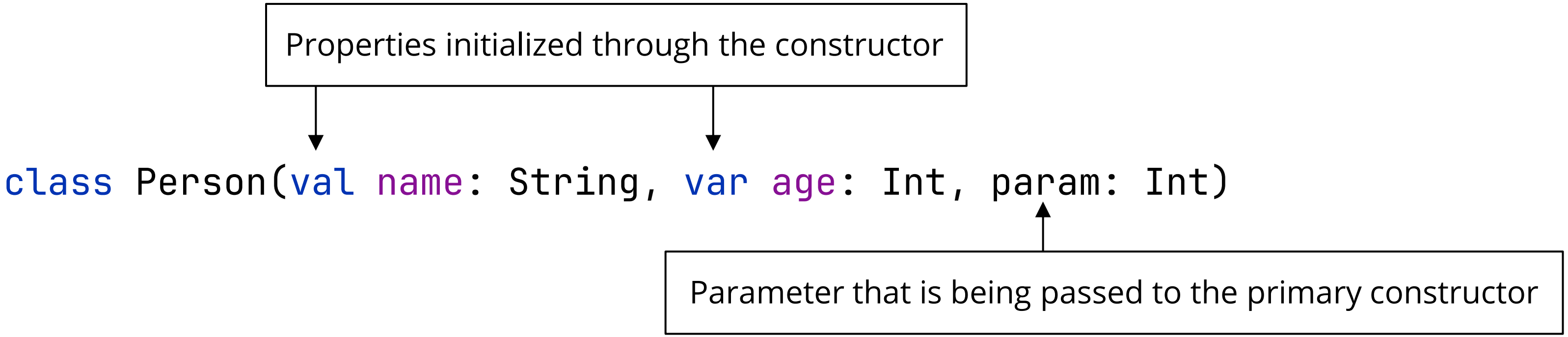
```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers,
the constructor keyword can be omitted.

```
class Person (firstName: String) { /*...*/ }
```

Primary Constructor

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:



Properties initialized through the constructor

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

Parameter that is being passed to the primary constructor

Primary constructor parameters can be used in the initializer blocks.
They can also be used in property initializers declared in the class body

init blocks

```
class Example(val value: Int, info: String) {
    val anotherValue: Int
    var info = "Description: $info"

    init {
        this.info += ", with value $value"
    }

    val thirdValue = computeAnotherValue() * 2

    private fun computeAnotherValue() = value * 10

    init {
        anotherValue = computeAnotherValue()
    }
}
```

There can be several `init` blocks.

Values can be initialized in `init` blocks that are written after them.

Constructor parameters are accessible in `init` blocks,
so sometimes you have to use `this`.

Secondary Constructors

A class can also declare secondary constructors, which are prefixed with **`constructor`**

```kotlin
class Point(val x: Int, val y: Int) {

    constructor(other: Point) : this(other.x, other.y) { ... }

    constructor(circle: Circle) : this(circle.centre) { ... }
}
```

Constructors can be chained, but they should always call the primary constructor in the end.

A secondary constructor's body will be executed after the object is created with the primary constructor. If it calls other constructors, then it will be executed after the other constructors' bodies are executed.

Defining properties inside the secodary constructors is **NOT** allowed

Constructors

Primary	Secondary
Defined in the class’s header	Defined inside class body
Only one primary constructor is allowed	You can create as many secondary constructors as you wish
Constructor keyword is optional unless you need to add modifier or annotation	Constructor keyword is mandatory
Can’t have a body. Instead, you can use the initializer blocks	Can have a body.
Can have properties defined and initialized.	Can’t define properties inside it.
--	Must call the primary constructor.

Encapsulation

Most programming languages provide special keywords for modifying the accessibility or visibility of attributes and methods.

Modifier	Meaning	Used with top level	Used inside class
`Public`	Accessible to anyone	Yes	Yes
`private`	Accessible only inside the class	Yes	Yes
`protected`	Accessible inside the class and its inheritors	No	Yes
`internal`	Accessible in the module	Yes	Yes

OOP Principles

1. Encapsulation

2. Inheritance

3. Polymorphism
   - Method Overloading
   - Method Overriding
   - Reference Variable Casting

Encapsulation

Encapsulation – The option to bundle data with methods operating on that data,
which also allows you to hide the implementation details from the user.

- An object is a black box. It accepts messages and replies in some way.
- Encapsulation and the interface of a class are intertwined:
  Anything that is not part of the interface is encapsulated.

Properties

In addition to having properties defined inside the constructor,
you can define properties inside the class's body.

The full syntax for declaring a property inside the body of the class is as follows:

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

Properties

```
class User {
    var name: String = "default"
        get() {
            return field
        }
        set(value) {
            field = value.trim()
        }

    var age: Int = 0
        set(value) {
            if (value >= 0) {
                field = value
            }
        }
}
```

Properties can have an initializer, getter, and setter.

By convention,
the name of the setter parameter is **`value`**,
but you can choose a different name if you prefer.

Use the **`field`** keyword to access the values inside the getter or setter,
otherwise you might encounter infinite recursion.

Inheritance

Inheritance – The possibility to define a new class based on an already existing one,
keeping all or some of the base class functionality (state/behavior).

- The class that is being inherited from is called a base or parent class
- The new class is called a derived class, a child, or an inheritor
- The derived class fully satisfies the specification of the base class,
  but it may have some extended features (state/behavior)

Inheritance

- "General concept – specific concept".
  - "Is-a" relationship.

- Motivation
  - Keep shared code separate – in the base class – and reuse it.
  - Type hierarchy, subtyping.
  - Incremental design.

- In Kotlin all classes has a top parent class `Any`

Inheritance

```kotlin
open class Animal(val name: String) {

    fun sound() {
        println("This animal makes a sound.")
    }

}

class Dog(name: String) : Animal(name) { }
```

To allow a class to be inherited by other classes,
the class should be marked with the **`open`** keyword.
(**Abstract** classes are always `open`.)

In Kotlin you can inherit only from **one class**,
and implement as many **interfaces** as you like.

When you’re inheriting from a class,
you have to call its constructor,
just like how secondary constructors have to call the primary.

Inheritance

```
open class Point(val x: Int, val y: Int) {
    constructor(other: Point) : this(other.x, other.y) { ... }

    constructor(circle: Circle) : this(circle.centre) { ... }
}
```

Inheritor class must call parent’s constructor:

```
class ColoredPoint(val color: Color, x: Int, y: Int) : Point(x, y) { ... }
```

In Kotlin **all clases are public & final by default**, to make the inheritable you have to use `open`

Polymorphism

Polymorphism = poly (many) + morphē (form).

Polymorphism – A core OOP concept that refers to working with objects through their interfaces without knowledge about their specific types and internal structure.

- Inheritors can override and change the ancestral behavior.
- Objects can be used through their parents' interfaces.

Polymorphism (Overriding)

All methods in Kotlin are defined as public final. To make a method overridable you have to:

1. Add `open` keyword in the method’s version inside the parent class.

2. Add `override` keyword as a prefix when overriding it in the Child class

```
open class DomesticAnimal {
    open fun pet() {...}
}

class Dog: DomesticAnimal() {
    override fun pet() {...}
}
```

Polymorphism (Reference Variable Casting)

```
open class DomesticAnimal {
    open fun pet(){}
}

class Dog: DomesticAnimal() {
    override fun pet() {...}
}

class Cat: DomesticAnimal() {
    override fun pet() {...}
}

fun main() {
    var animals : Array<DomesticAnimal> = arrayOf(Dog(), Cat())
    for (currentAnimal in animals)
           currentAnimal.pet()

}
```

Properties

```
open class OpenBase(open val value: Int)


open class OpenChild(value: Int) : OpenBase(value) {
   override var value: Int = 1000
       get() { return field - 7 }
}


open class AnotherChild(value: Int) : OpenChild(value) {
   final override var value: Int = value
       get() { return super.value } // default get() is used otherwise
       set(value) { field = value * 2 }
}
```

Properties may be `open`, which means that their getters and setters might be overridden by inheritors, respectively.

You can prohibit further overriding by marking a property `final`.

Abstract Class

A class may be declared **`abstract`**, along with some or all of its members.

An **abstract member** does not have an implementation in its class.

You don't need to annotate abstract classes or functions with **`open`**.

```
abstract class Shape{
    abstract fun draw()
}

class Rectangle : Shape() {
    override fun draw() {
        // draw the rectangle
    }
}
```

Abstract Class

```kotlin
abstract class Shape{
    abstract fun draw()
    open fun printShapeColor(){
        println("Red")
    }
}

class Rectangle : Shape() {
    override fun draw() {
        // draw the rectangle
    }

    override fun printShapeColor() {
        println("Blue")
    }
}
```

Abstract class can contain both **concrete** and **abstract** members.
For **concrete** members to be overridden...
you should prefix them with the **open** keyword

Interfaces

Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations.

What makes them different from abstract classes is that interfaces cannot store state.
They can have properties, but these need to be abstract or provide accessor implementations.

Interfaces

```
interface MyInterface {
    fun meth1()
    fun meth2(){
        println("Default implementation of method 2")
    }
}

class MyClass : MyInterface {
    override fun meth1() {
        println("My Class Implementation of method 1")
    }

    override fun meth2() {
        println("My Class Implementation of method 2")
    }

}
```

Interfaces & Abstract classes

```kotlin
interface RegularCat {
    fun pet()
    fun feed(food: Food)
}

interface SickCat {
    fun checkStomach()
    fun giveMedicine(pill: Pill)
}
```

Interfaces **cannot have** a state.

vs

```kotlin
abstract class RegularCat {
    abstract val name: String

    abstract fun pet()
    abstract fun feed(food: Food)
}

abstract class SickCat {
    abstract val location: String

    abstract fun checkStomach()
    fun giveMedicine(pill: Pill) {}
}
```

Abstract classes cannot have an instance, but can **have** a state.

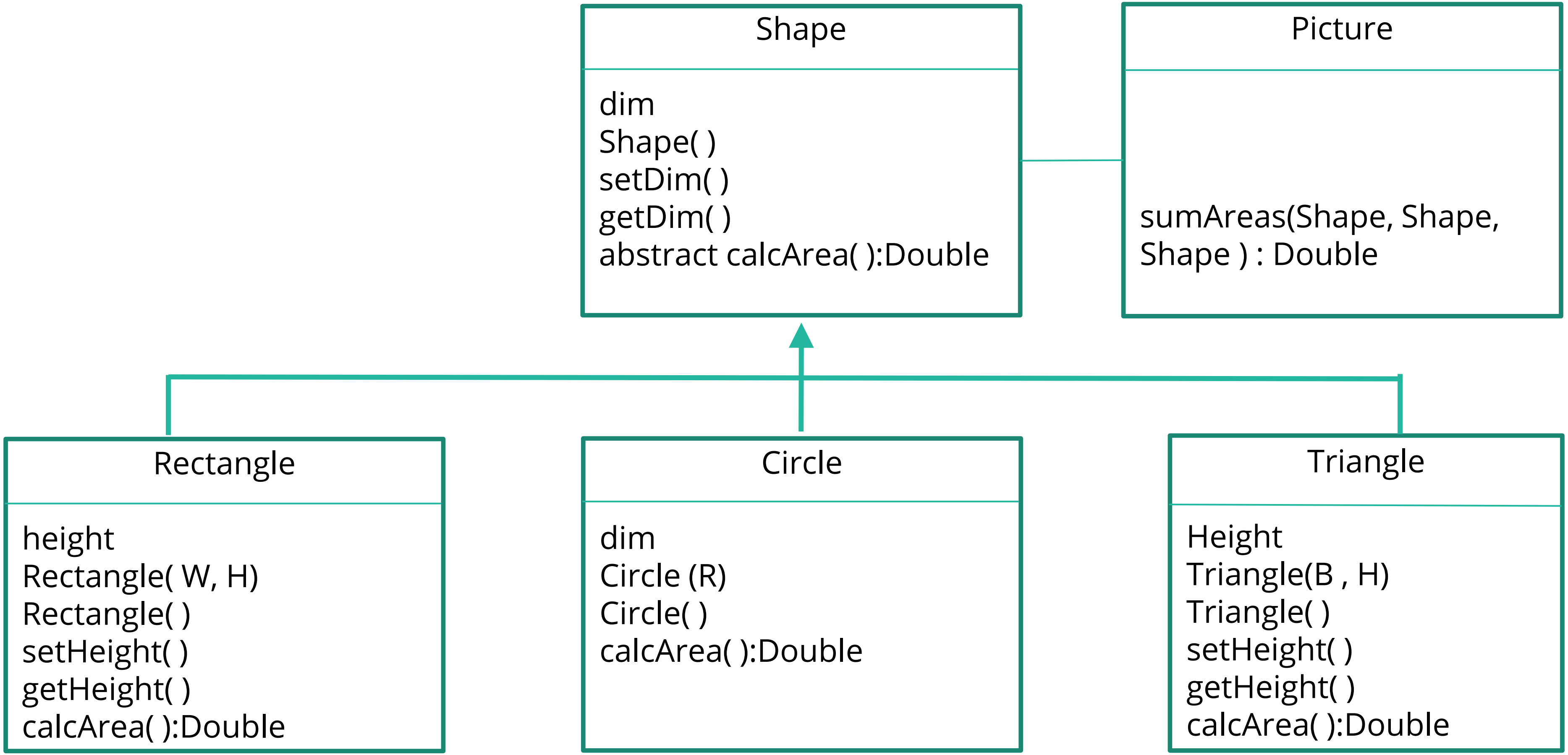
Interfaces & Abstract Classes

	Interfaces	Abstract classes
Class can	Implement more than 1 interface	Extend only 1 abstract class
Can have Constructors	No	Yes
User can create object	No	No
Methods Contained	All methods are open by default	Only abstract methods are open
Can have concrete methods	Yes	Yes
Needs to be prefixed with open?	No	No

Lab

Create a program that has a Picture class that contains 3 Shapes in addition it contains a method `sumAreas( )` that returns the summation of any 3 shapes areas.

Try to use all of what you have learned



Shape

dim
Shape( )
setDim( )
getDim( )
abstract calcArea( ):Double

Picture

sumAreas(Shape, Shape, Shape) : Double

Rectangle

height
Rectangle(W, H)
Rectangle( )
setHeight( )
getHeight( )
calcArea( ):Double

Circle

dim
Circle (R)
Circle( )
calcArea( ):Double

Triangle

Height
Triangle(B , H)
Triangle( )
setHeight( )
getHeight( )
calcArea( ):Double