JET BRAINS 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()
```

Object (class/type) invariant

Invariants place constraints on the state of an object, maintained by its methods right from construction.
It is the object's own responsibility to ensure that the invariant is being maintained.

Corollaries:

- Public fields are nasty.
- If a field does not participate in the object's invariant, then it is not clear how it belongs to this object at all, which is evidence of poor design choices.

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.

Encapsulation

Encapsulation – The option to bundle data with methods operating on said 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.
- OOP encapsulation differs from encapsulation in abstract data types.

Abstraction vs Encapsulation

Abstraction is about what others see and how they interact with an object.

Encapsulation is about how an object operates internally and how it responds to messages.

Encapsulation

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

In Kotlin:

- **`public`** – Accessible to anyone
- **`private`** – Accessible only inside the class
- **`protected`** – Accessible inside the class and its inheritors
- **`internal`** – Accessible in the module

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.
- Inheritance is often redundant and can be replaced with composition.

Subtyping

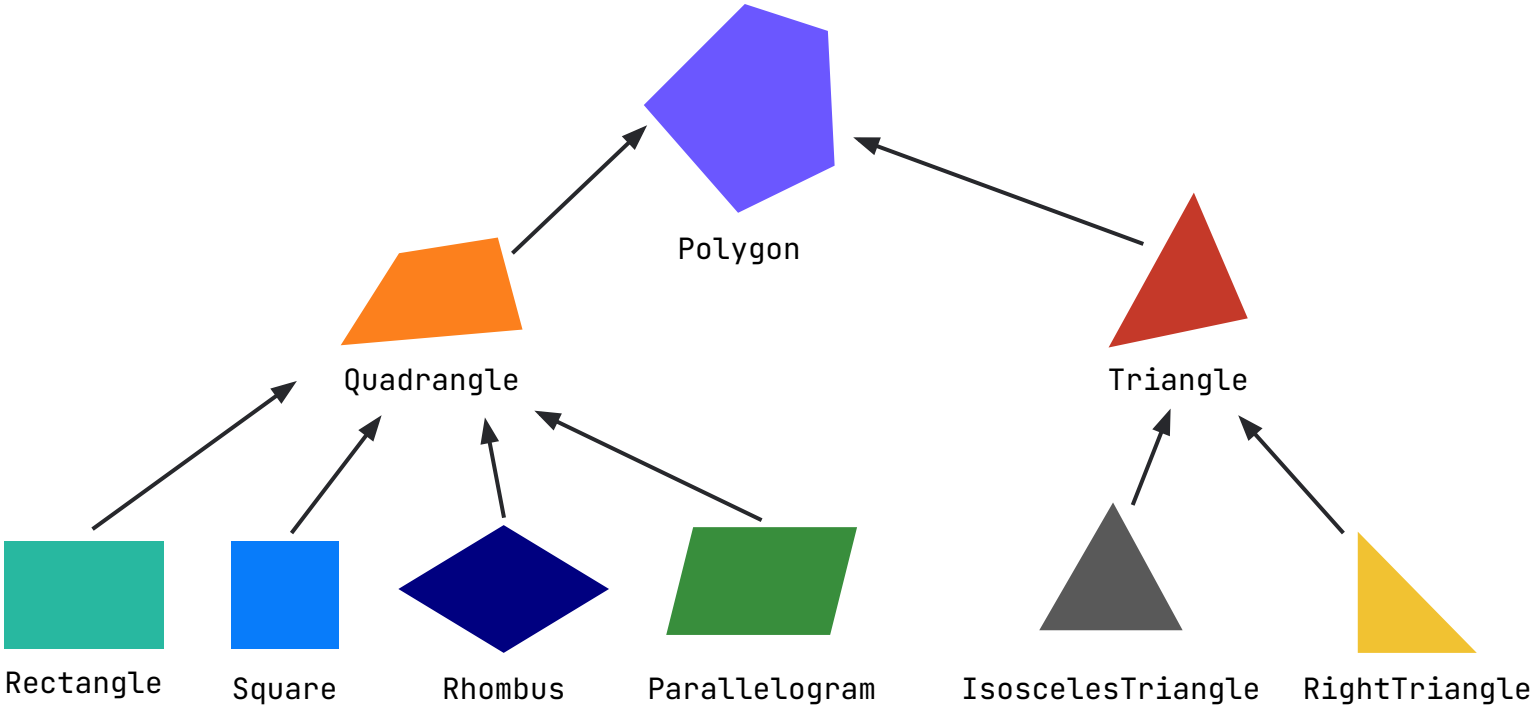
An object can belong to several types (classes) at the same time

- Eleanor – A student, a woman, a beer enthusiast, and the reigning UFC champion.
- Nate – A developer, a man, an anime lover, and a recreational swimmer.

Each type (class) defines an interface and expected behavior.

So, in our example, while Eleanor is a student, she will exhibit a set of expected behaviors (such as turning in homework, studying for tests, etc.). When Eleanor gets her degree, she will stop being a student and she may cease to exhibit the associated behaviors, but her overall identity will not change and the behaviors associated with her other properties will be unaffected.

Subtyping


Polygon
Quadrangle
Triangle
Rectangle
Square
Rhombus
Parallelogram
IsoscelesTriangle
RightTriangle

Polymorphism

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.
    - The client code does not know (or care) if it is working with the base class or some child class, nor does it know what exactly happens "inside".

Liskov substitution principle (LSP) – If for each object o1 of type S, there is an object o2 of type T, such that for all programs P defined in terms of T the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.

OOP in Kotlin

```
class UselessClass

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

Constructors

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


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

    init {
        findJob()
    }

    constructor(name: String, parent: Person) : this(name, parent.surname, 0)
}
```


The **secondary** constructor

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

Constructors

```
open 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.

Inheritor class must call parent's constructor:

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

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`.

Abstraction

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

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

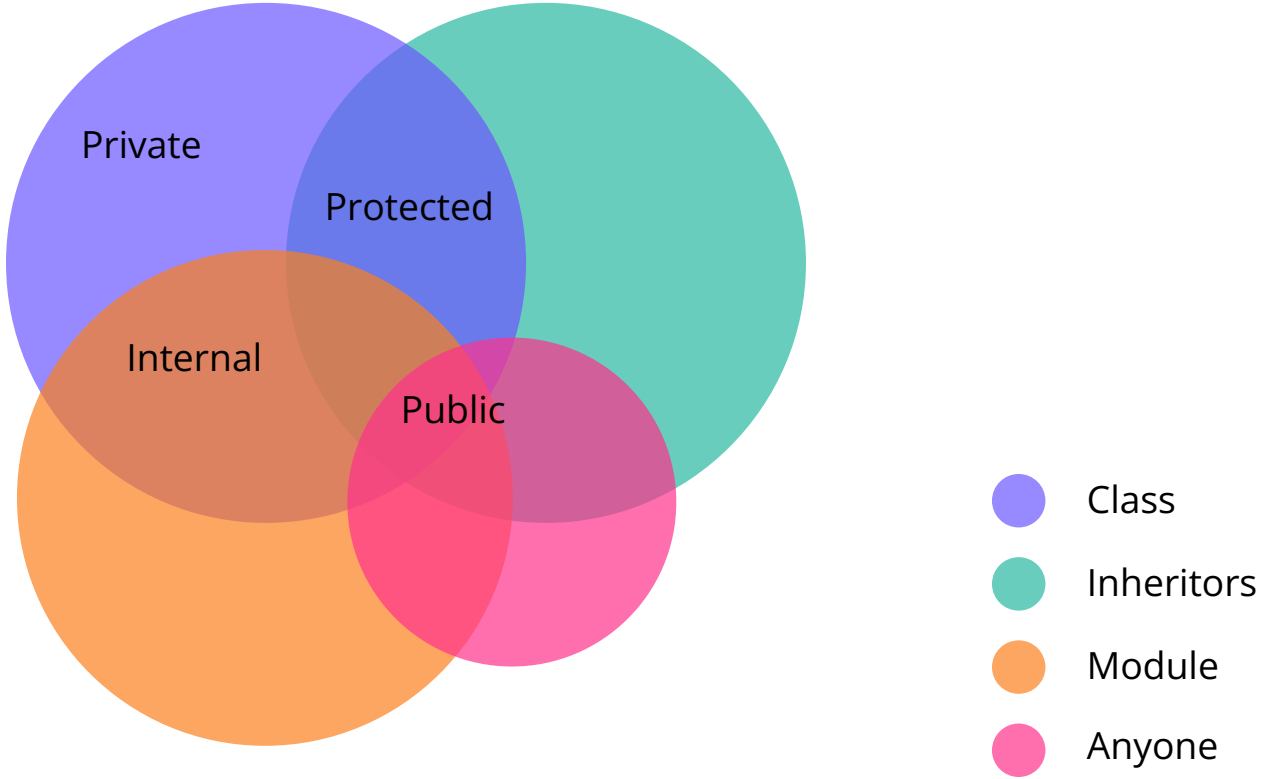
Interfaces **cannot have** a state.
(We'll get back to this a bit later.)

VS

```
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.

Encapsulation



Encapsulation

```
abstract class RegularCat {
    protected abstract val isHungry: Boolean
    private fun poop(): Poop { /* do the thing */ }
    abstract fun feed(food: Food)
}
class MyCat : RegularCat() {
    override val isHungry: Boolean = false
    override fun feed(food: Food) {
           if (isHungry) { /* do the thing */ }
           else { poop() } // MyCat cannot poop
     }
}
Cannot access 'poop': it is invisible (private in a supertype) in 'MyCat'
```

Inheritance

```
class SickDomesticCat : RegularCat(),
CatAtHospital {
    override var isHungry: Boolean = false
        get() = field
        set(value) {...}

    override fun pet() {...}

    override fun feed(food: Food) {...}

    override fun checkStomach() {...}

    override fun giveMedicine(pill: Pill)
{...}
}
```

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 from 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.

Why do you prohibit a cat from pooping?!

```
abstract class Cat {
    /* final */ fun anotherDay() {
        // various cat activities
        digest(findFood())
        poop(findWhereToPoop())
    }
    private fun poop(where: Place): Poop {...}
    private fun digest(food: Food) {
        // don't know how they work
        poop(findWhereToPoop())
    }
    abstract fun feed(food: Food)
    abstract fun findWhereToPoop(): Place
    abstract fun findFood(): Food
}
```

```
class DomesticCat(
    val tray: Tray,
    val bowl: Bowl
) : Cat() {
    override fun feed(food: Food) {
        // place some food in the bowl
    }

    override fun findWhereToPoop() = tray
    override fun findFood() {
      return bowl.getFood() ?: run {
          // find food somewhere else
      }
    }
}
```

Polymorphism revisited

```
interface DomesticAnimal {
    fun pet()
}

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

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

fun main() {
    val homeZoo = listOf<DomesticAnimal>(Dog(), Cat())
    homeZoo.forEach { it.pet() }
}
```

Properties

```
class PositiveAttitude(startingAttitude: Int) {
    var attitude = max(0, startingAttitude)
        set(value) =
            if (value ⩾ 0) {
                field = value
            } else {
                println("Only positive attitude!")
                field = 0
            }

    var hiddenAttitude: Int = startingAttitude
        private set
        get() {
            if (isSecretelyNegative) {
                println("Don't ask this!")
                field += 10
            }
            return field
        }

    val isSecretelyNegative: Boolean
        get() = hiddenAttitude < 0
}
```

Properties can optionally have an initializer, getter, and setter.

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

Properties may have no (backing) filed at all.

Properties

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

interface AnotherExample {
   /* abstract */ val anotherValue: OpenBase
}

open class OpenChild(value: Int) : OpenBase(value), AnotherExample
{
   override var value: Int = 1000
       get() = field - 7
   override val anotherValue: OpenBase = OpenBase(value)
}

open class AnotherChild(value: Int) : OpenChild(value) {
   final override var value: Int = value
       get() = super.value // default get() is used otherwise
       set(value) { field = value * 2 }
   final override val anotherValue: OpenChild = OpenChild(value)
// Notice that we use OpenChild here, not OpenBase
}
```

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

Interfaces can have properties, but they are always `abstract`.

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

Operator overloading

```
class Example {
   operator fun plus(other: Example): Example { ... }
   operator fun dec() = this // return type has to be a subtype
   operator fun get(i: Int, j: Int): SomeType { ... }
   operator fun get(x: Double?, y: String) = this
   operator fun <T> invoke(l: List<T>): SomeType { ... }
}

operator fun Example.rangeTo(other: Example): Iterator<Example> { ... }

fun main() {
   var ex1 = Example()
   val ex2 = ex1 + --ex1 // -- reassigned ex1, so it has to be var
   for (ex in ex1..ex2) {
       ex[23, 42]
       ex[null, "Wow"](listOf(1,2,3))
   }
}
```

Operator “overloading” is allowed.

Almost all operators can be overloaded.

Operators can be overloaded outside of the class.

Extensions

Kotlin provides the ability to extend a class or an interface with new functionality without having to inherit from the class or use *forbidden magic (reflection)*

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' is the given MutableList<T>
    this[index1] = this[index2]
    this[index2] = tmp
}
```

If the extended class **already has** the new method with the same name and signature, the **original** one will be used.

Extensions under the hood

The class that is being extended does not change at all; it is simply a new function that can be called like a method. It cannot access private members, for example.

Extensions have static dispatch, rather than virtual dispatch by receiver type. An extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result from evaluating that expression at runtime.

```
open class Shape
class Rectangle: Shape()

fun Shape.getName() = "Shape"
fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle()) // "Shape", not Rectangle
```

Infix functions

```
data class Person(val name: String, val surname: String)

infix fun String.with(other: String) = Person(this, other)

fun main() {
    val realHero = "Ryan" with "Gosling"
    val (real, bean) = realHero
}
```

ComponentN operator

```
class SomeData(val list: List<Int>) {
   operator fun component1() = list.first()
   operator fun component2() = SomeData(list.subList(1, list.size))
   operator fun component3() = "This is weird"
}

fun main() {
   val sd = SomeData(listOf(1, 2, 3))
   val (head, tail, msg) = sd
   val (h, t) = sd
   val (onlyComponent1) = sd
}
```

Any class can overload any number of `componentN` methods that can be used in destructive declarations.

Data classes have these methods by default.

Data classes

```
data class User(val name: String, val age: Int)
```

The compiler automatically derives:

- `equals()` and `hashCode()`
- `toString()` of the form `User(name=John, age=42)`
- `componentN()` functions corresponding to the properties in their order of declaration.
- `copy()` to copy an object, allowing you to alter some of its properties while keeping the rest unchanged

The standard library provides the `Pair` and `Triple` classes, but named data classes are a much better design choice.

Inline (value) classes

Occasionally you have to wrap a class, but wrapping always causes overhead in both memory and execution time. Inline classes may help you get the desired behavior without paying for it with a drop in performance.

```
interface Greeter {
   fun greet(): Unit
}

class MyGreeter(var myNameToday: String) : Greeter {
   override fun greet() = println("Hello, $myNameToday!")
}

@JvmInline
/* final */ value class BadDayGreeter(val greeter: Greeter) : Greeter {
   override fun greet() {
       greeter.greet()
       println("Having a bad day, huh?")
   }
}
```

```
var greeter: Greeter = MyGreeter("Cyr")
if (today.isBad()) { greeter = BadDayGreeter(greeter)
}
greeter.greet()
```

Inline (value) classes

- An Inline class must have exactly one primary constructor parameter,
- Inline classes can implement interfaces, declare properties (no backing fields), and have `init` blocks.
- Inline classes are not allowed to participate in a class hierarchy, which is to say they are automatically marked with the "final" keyword.
- The compiler tries to use the underlying type to produce the most performant code.

```
@JvmInline
/* final */ value class Name(val name: String) : Greeter {
    init {
        require(name.isNotEmpty()) { "An empty name is absurd!" }
    }

    // val withABackingField: String = "Not allowed"

    var length: Int
        get() = name.length
        set(value) { println("What do you expect to happen?") }

    override fun greet() { println("Hello, $name") }
}
```

Inline (value) classes

Since inline classes are just wrappers and the compiler tries to use the underlying type, name mangling is introduced to solve possible signature clashing problems.

```
fun foo(name: Name) { ... } → public final void foo-<stable-hashcode>(name: String) { ... }

fun foo(name: String) { ... } → public final void foo(name: String) { ... }
```

If you want to call such a function from Java code, then you should use the `@JvmName` annotation.

```
@JvmName("fooName")
fun foo(name: Name) { ... } → public final void fooName(name: String) { ... }
```

Enum classes

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

Each enum constant is an object.
Each enum is an instance of the enum class, thus it can be initialized as:

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Enum classes can have methods or even implement interfaces.

Kotlin example

```kotlin
val user = User("John", 23)

val (name, age) = user // destructing declaration calls componentN()

val (onlyName) = user

val olderUser = user.copy(age = 42)

val g = Color.valueOf("green".uppercase())

when(g) {
    Color.RED -> println("blood")
    Color.GREEN -> println("grass")
    Color.BLUE -> println("sky")
}
```

Sealed classes

```
sealed class Base {
    open var value: Int = 23
    open fun foo() = value * 2
}
open class Child1 : Base() {
    override fun foo() = value * 3
    final override var value: Int = 10
        set(value) = run { field = super.foo() }
}
class Child2 : Base()

val b: Base = Child1()
when(b) {
    is Child1 -> println(1)
    is Child2 -> println(2)
}
```

All of the inheritors of a `sealed` class must be known at compile time.

Can be used in `when` the same way as enums can be.

Not specific to `sealed` classes:

- Prohibit overriding an `open fun` or property by making it `final`.
- Access parents' methods through `super`.

Functional interfaces (SAM)

Single Abstract Method (SAM) interface

- Interface that has one abstract method.
- Kotlin allows us to use a lambda instead of a class definition to implement a SAM.

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
}
```

```
val isEven = object : IntPredicate {
   override fun accept(i: Int): Boolean {
       return i % 2 == 0
   }
}
```

VS

```
val isEven = IntPredicate { it % 2 == 0 }
```

```
fun main() {
   println("Is 7 even? - ${isEven.accept(7)}")
}
```

Kotlin singleton

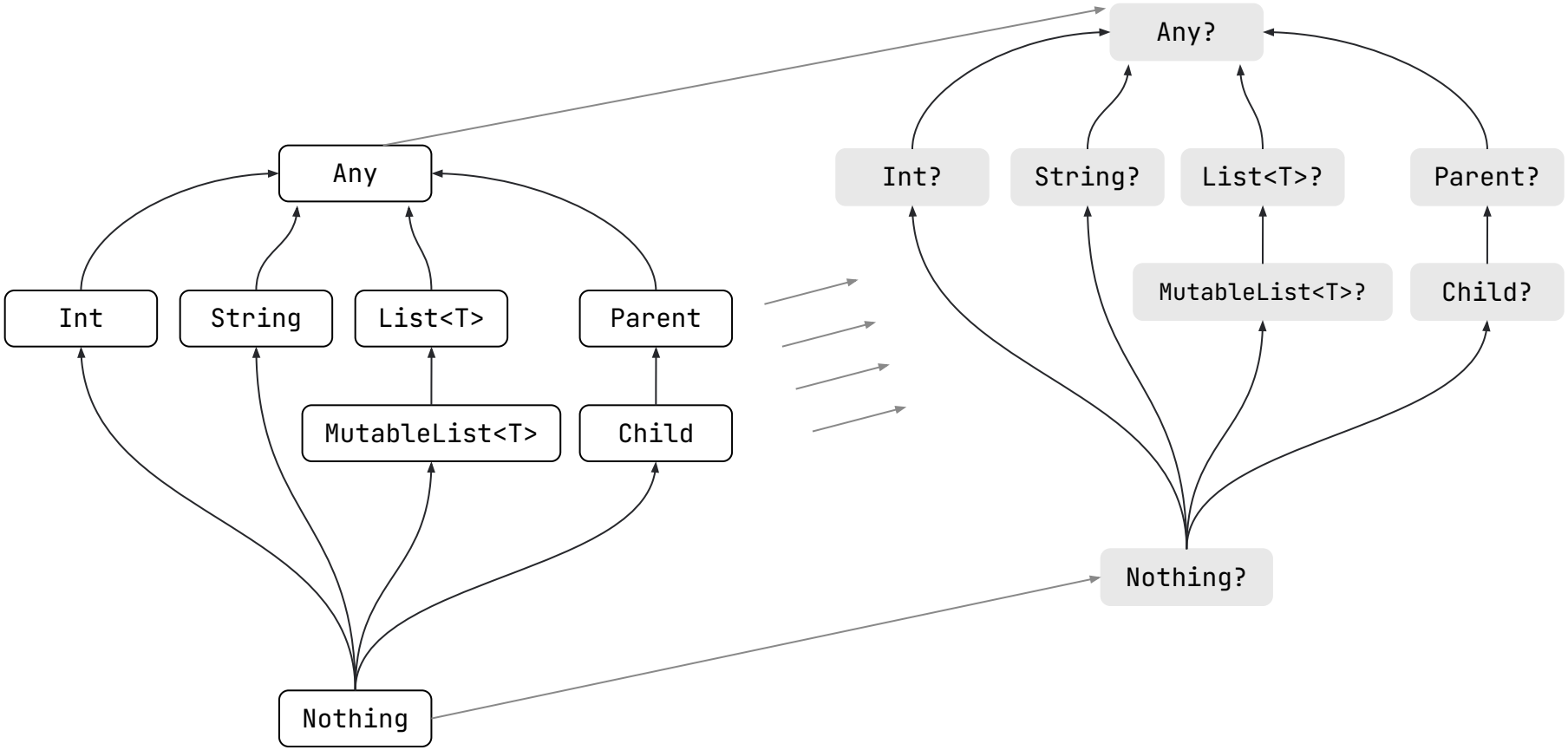
```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}

DataProviderManager.registerDataProvider(...)
```

Companion objects

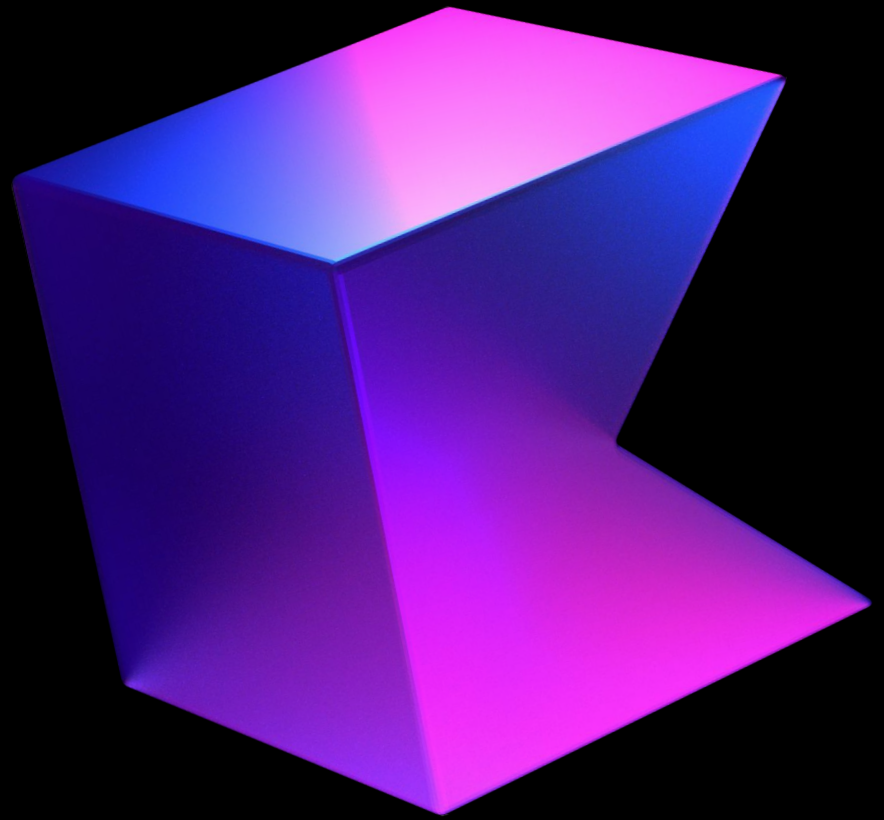
- An object declaration inside a class can be marked with the companion keyword.
- Companion objects are like static members:
  - The Factory Method
  - Constants
  - Etc.
- Visibility modifiers are applicable.
- Use @JvmStatic to go full static.

```
interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        private var counter: Int = 0
        override fun create(): MyClass =
                MyClass().also { counter += 1}
    }
    // … some code …
}

val f: Factory<MyClass> = MyClass.Companion
val instance1 = f.create()
val instance2 = f.create()
```

Kotlin Type Hierarchy



Thanks!



@kotlin | Developed by JetBrains