# Classes and Objects

# About me

**Trayan Iliev**

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

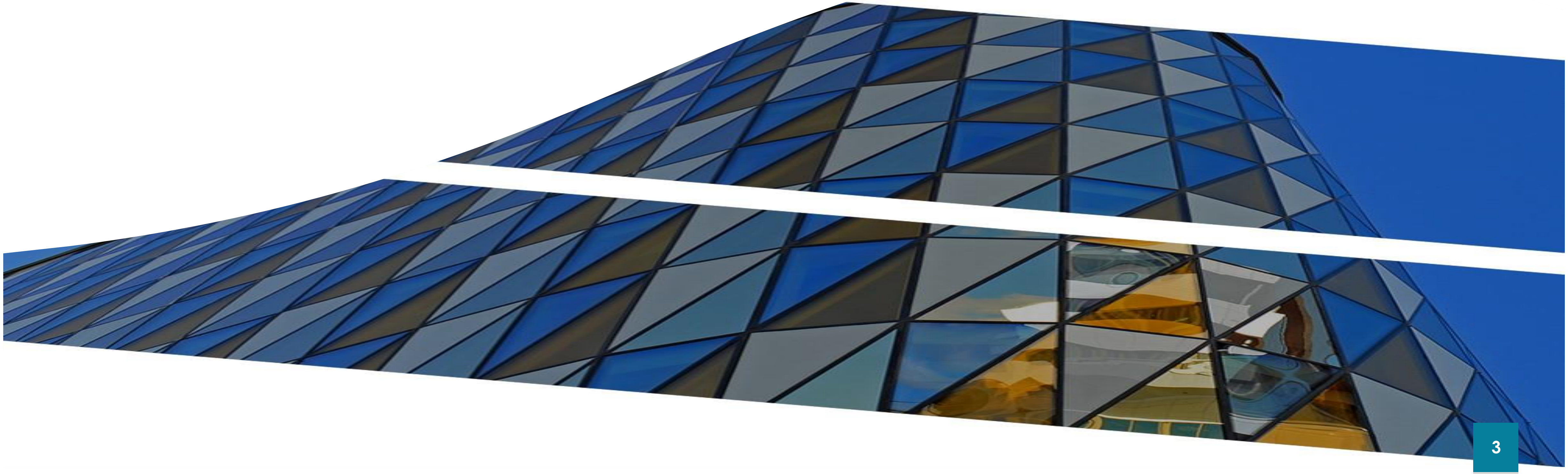
– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer  RoboLearn hackathons and  IoT enthusiast

# Classes



# Classes and Constructors

- Classes in Kotlin are declared using the keyword class:

```kotlin
class Person { /*...*/ }

class Empty
```

- A class in Kotlin can have a primary constructor and one or more secondary constructors. The primary constructor is a part of the class header, and it goes after the class name and optional type parameters:

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

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

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

# Order of Initialization

- The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks prefixed with the **init** keyword:

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println) // 1

    init {
        println("First initializer block that prints ${name}")  // 2
    }

    val secondProperty = "Second property: ${name.length}".also(::println)  // 3

    init {
        println("Second initializer block that prints ${name.length}")  // 4
    }
}
```

# Initializing Class Properties

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

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

- Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor (including default values):

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

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
```

# Constructor with Modifiers / Annotations

- Trailing comas can be added if necessary:

```
class Person6(
    val firstName: String,
    val lastName: String,
    var age: Int, // trailing comma
) { /*...*/ }
```

- If the constructor has annotations or visibility modifiers, the **constructor** keyword is required and the modifiers go before it:

```
class Customer2 public @Inject constructor(name: String) { /*...*/ }
```

# Secondary Constructors

- A class can also declare one or more secondary constructors, which are prefixed with **constructor**:

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

# Secondary Constructors - II

```
class Person(val name: String, val pets: MutableList<Pet> = mutableListOf()) {
    override fun toString() = "$name's pets: $pets"
}
class Pet(val name: String) {
    constructor(name: String, owner: Person) : this(name) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
    override fun toString() = "Pet($name)"
}
fun main() {
    val ivan = Person("Ivan Petrov")
    val Johny = Pet("Johny", ivan)
    val Silvester = Pet("Silvester", ivan)
    val Caty = Pet("Caty", ivan)
    println(ivan) //Ivan Petrov's pets: [Pet(Johny), Pet(Silvester), Pet(Caty)]
}
```

# Constructor Delegation

- Code in initializer blocks effectively becomes part of the primary constructor. Delegation to the primary constructor happens as the first statement of a secondary constructor, so the code in all initializer blocks and property initializers is executed before the body of the secondary constructor.

- Even if the class has no primary constructor, the delegation still happens:

```
class Constructors {
    init {
        println("Init block")
    }
    constructor(i: Int) {
        println("Constructor $i")
    }
}

val c2 = Constructors(42) // Init block, Constructor 42
```

# Private constructors and constructors with default values

- If you don't want your class to have a public constructor, declare an empty primary constructor with non-default visibility:

```
class DontCreateMe private constructor () { /*...*/ }
```

- On the JVM, if all of the primary constructor parameters have default values, the compiler will generate an additional parameterless constructor which will use the default values. This makes it easier to use Kotlin with libraries such as Jackson or JPA that create class instances through parameterless constructors.

```
class Customer(val customerName: String = "")
```

# Creating Class Instances

```kotlin
data class Product(val name: String, val price: Double, var id: Int)
data class Invoice(
    val number:Int,
    val customer: Customer,
    val items: MutableList<Product> = mutableListOf()
)


val customer = Customer("Joe Smith")

val invoice = Invoice(1, customer)

println(invoice) // Invoice(number=1, customer=Customer: JOE SMITH, items=[])
```

- Kotlin does not have a new keyword.

# Class Members

- Constructors and initializer blocks
- Functions
- Properties
- Nested and inner classes
- Object declarations

# Inheritance

- All classes in Kotlin have a common superclass, Any, which is the default superclass for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

- Any class has three methods: equals(), hashCode(), and toString(). Thus, these methods are defined for all Kotlin classes.

- By default, Kotlin classes are final – they can't be inherited. To make a class inheritable, mark it with the open keyword:

```
open class Base(p: Int)


class Derived(p: Int) : Base(p)
```

# Inheritance – base class initialization

- If the derived class has no primary constructor, then each secondary constructor has to initialize the base type using the **super** keyword or it has to delegate to another constructor which does. Different secondary constructors can call different constructors of the base type:

```
class Context
class AttributeSet
open class View(val ctx: Context) {
    private var attributes: AttributeSet = AttributeSet()
    constructor(ctx: Context, attrs: AttributeSet): this(ctx) {
        this.attributes = attrs
    }
}
class MyView : View {
    constructor(ctx: Context) : super(ctx)
    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

# Overriding Methods

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}


class Circle() : Shape() {
    override fun draw() { /*...*/ }
}


open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

# Overriding Properties

```
open class Shape2 {
    open val vertexCount: Int = 0
}

class Rectangle2 : Shape2() {
    override val vertexCount = 4
}
```

```
interface Shape3 {
    val vertexCount: Int
}

class Rectangle3(override val vertexCount: Int = 4) : Shape3 // Always has 4 vertices

class Polygon3 : Shape3 {
    override var vertexCount: Int = 0  // Can be set to any number later
}
```

# Derived class initialization order

```
open class Base(val name: String) {
    init { println("Initializing a base class") }
    open val size: Int =
        name.length.also { println("Initializing size in the base class: $it") }
}
class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println("Argument for the base class: $it") }) {
    init { println("Initializing a derived class") }
    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in the derived class: $it") }
}

fun main() {
    val d = Derived(“ivan", "Petrov")
}
```

→

```
Argument for the base class: Ivan
Initializing a base class
Initializing size in the base class: 4
Initializing a derived class
Initializing size in the derived class: 10
```

# Derived class initialization order

- When the base class constructor is executed, the properties declared or overridden in the derived class have not yet been initialized. Using any of those properties in the base class initialization logic (either directly or indirectly through another overridden open member implementation) may lead to incorrect behavior or a runtime failure.
- When designing a base class, you should avoid using open members in the constructors, property initializers, or init blocks.

# Calling the superclass implementation

- Code in a derived class can call its superclass functions and property accessor implementations using the **super** keyword:

```
open class Rectangle4 {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle4() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

# Calling the superclass implementation in inner class

```
class FilledRectangle2: Rectangle4() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle2.draw() // Calls Rectangle's implementation of draw()
            fill()
            // Uses Rectangle's implementation of borderColor's get()
            println("Drawn a filled rectangle with color ${super@FilledRectangle2.borderColor}")
        }
    }
}
```

# Overriding Rules

- If a class inherits multiple implementations of the same member from its immediate superclasses, it must override this member and provide its own implementation (perhaps, using one of the inherited ones).
- To denote the supertype from which the inherited implementation is taken, use super qualified by the supertype name in angle brackets, such as **super<Base>**:

```
open class Rectangle5 {     open fun draw() { /* ... */ }     }
interface Polygon {    fun draw() { /* ... */ } // interface members are 'open' by default    }

class Square() : Rectangle5(), Polygon { // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle5>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}
```

# Abstract Classes

- 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 Polygon6 {
    abstract fun draw()
}


class Rectangle6 : Polygon6() {
    override fun draw() {
        // draw the rectangle
    }
}
```

```
open class Polygon7{
    open fun draw() {
        // some default polygon drawing method
    }
}
abstract class WildShape : Polygon7() {
    // Classes that inherit WildShape need to provide their own
    // draw method instead of using the default on Polygon
    abstract override fun draw()
}
```

# 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 a state. They can have properties, but these need to be abstract or provide accessor implementations.

```
interface MyInterface {
    fun bar()
    fun foo() {
        // optional body
    }
}

class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

```
interface MyInterface2 {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo" // can not have backing field

    fun foo() {
        print(prop)
    }
}
class Child2 : MyInterface2 {
    override val prop: Int = 29
}
```

# Interfaces Inheritance

- An interface can inherit from other interfaces, meaning it can both provide implementations for their members and declare new functions and properties. Classes implementing an interface are only required to define the missing (abstract member) implementations:

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}
```

# Conflict Resolution

```
interface A {
    fun foo() { print("A") }
    fun bar() // abstract
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}
```

```
class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

# Functional (SAM) Interfaces

- An interface with only one abstract method is called a functional interface, or a Single Abstract Method (SAM) interface. The functional interface can have several non-abstract members but only one abstract member:

```kotlin
fun interface KRunnable {
    fun invoke()
}

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

```kotlin
// Creating an instance of a class
val isEven = object : IntPredicate {
    override fun accept(i: Int): Boolean {
        return i % 2 == 0
    }
}
// Creating an instance using lambda
val isEvenLambda = IntPredicate { it % 2 == 0 }

fun main() {
    println("Is 42 even? - ${isEven.accept(42)}")
    println("Is 42 even? - ${isEvenLambda.accept(42)}")
}
```

# Functional Interfaces vs. Type Aliases - I

```
typealias Predicate<T> = (T) -> Boolean
typealias IntPredicate2 = Predicate<Int>

fun Collection<Int>.areAll(
    predicate: IntPredicate): Boolean {
    for (elem in this) {
        if (!predicate.accept(elem)) return false
    }
    return true
}

fun Collection<Int>.areAll2(
    predicate: IntPredicate2): Boolean {
    for (elem in this) {
        if (!predicate(elem)) return false
    }
    return true
}
```

```
fun main() {
    val numbers = listOf(42, 13, 54, 32, 78)
    println("Are all numbers even in $numbers? –
            ${numbers.areAll({ it % 2 == 0 })}")
    println("Are all numbers even in $numbers? –
            ${numbers.areAll2({ it % 2 == 0 })}")
}
```

# Functional Interfaces vs. Type Aliases - II

Functional interfaces and type aliases serve different purposes:

- Type aliases are just names for existing types – they don't create a new type, while functional interfaces do.
- You can provide extensions that are specific to a particular functional interface to be inapplicable for plain functions or their type aliases.
- Type aliases can have only one member, while functional interfaces can have multiple non-abstract members and one abstract member. Functional interfaces can also implement and extend other interfaces.
- Functional interfaces can be more costly syntactically and at runtime because they can require conversions to a specific interface.

# Objects

Object expressions and declarations, companion objects

# Classes and Objects

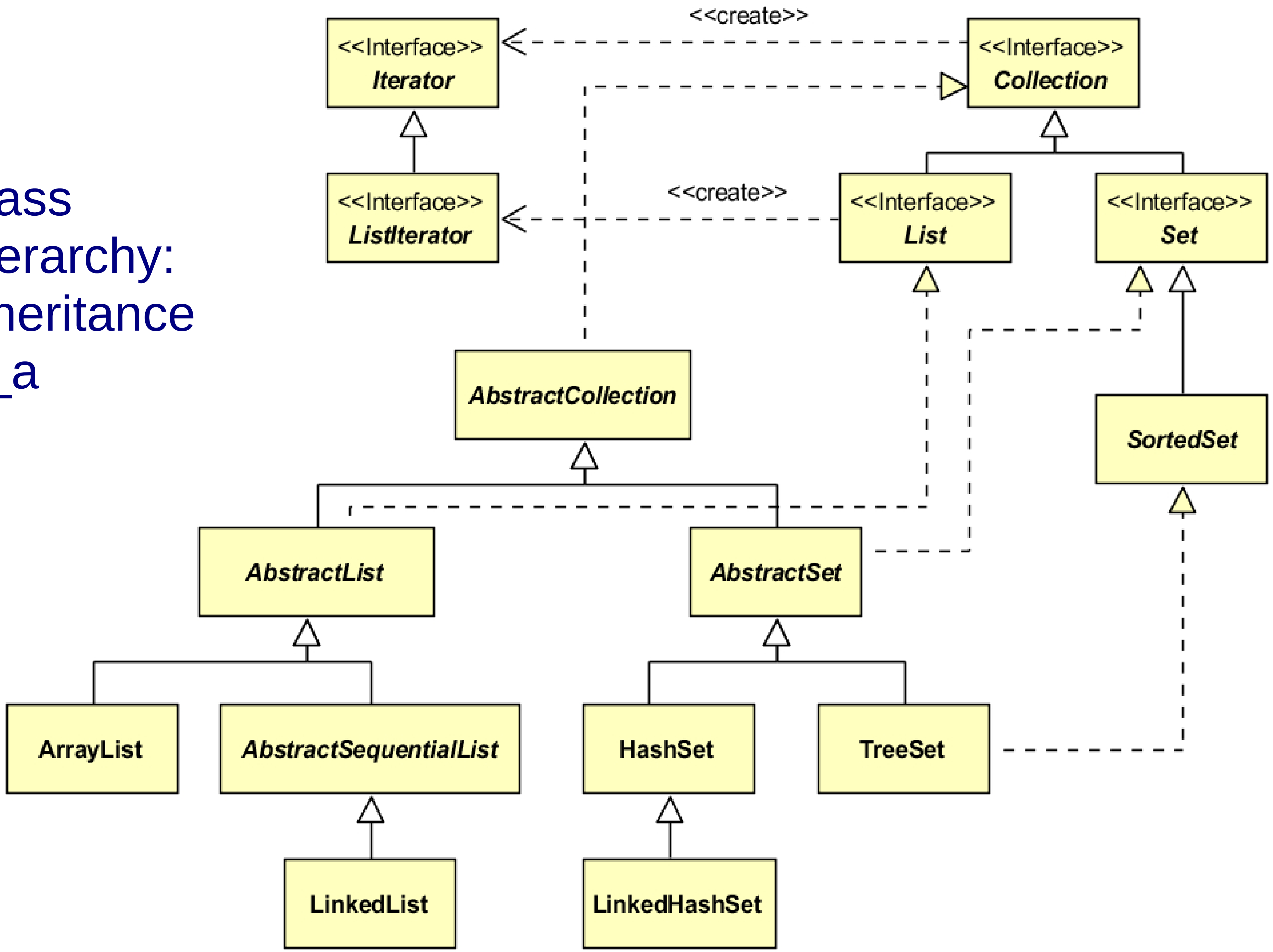
**Class** – describes common features for a set of objects: structure, behavior and possible links to objects of other classes = **objects type**

- structure = attributes, properties, member variables
- behavior = methods, operations, member functions, messages
- relations between classes: association, inheritance, aggregation, composition – modeled as attributes (references to objects from the connected class)

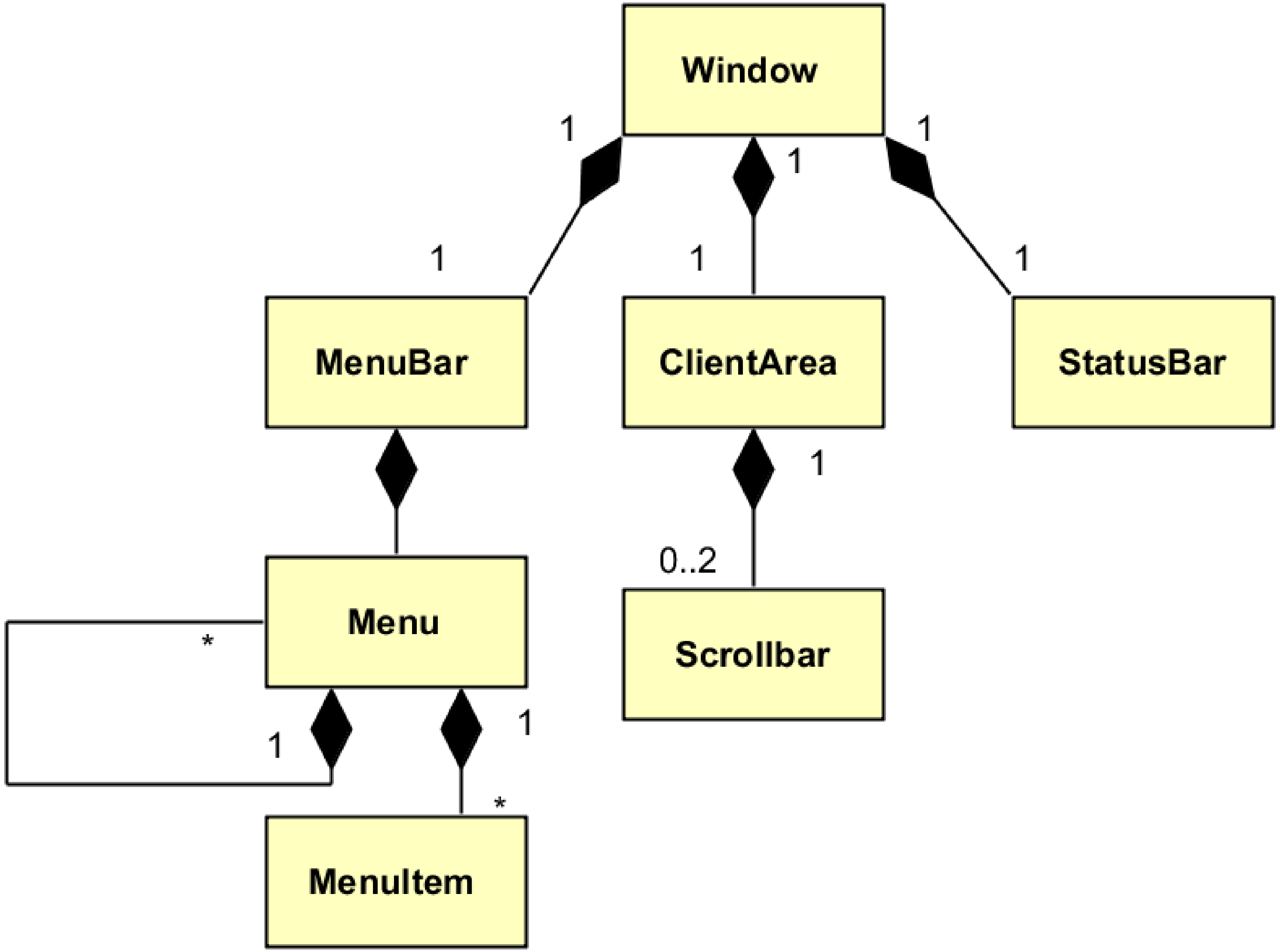
**Objects** are instances of the class, which in addition have:

- own state
- unique identifier = reference pointing towards object

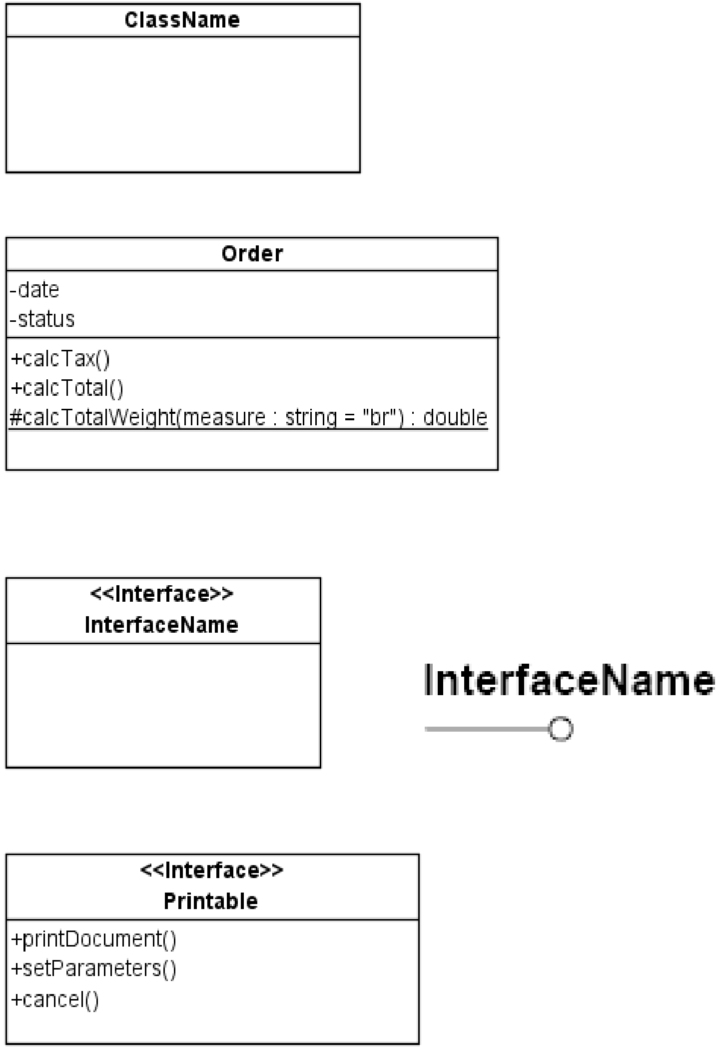
# Class Hierarchy: Inheritance is_a



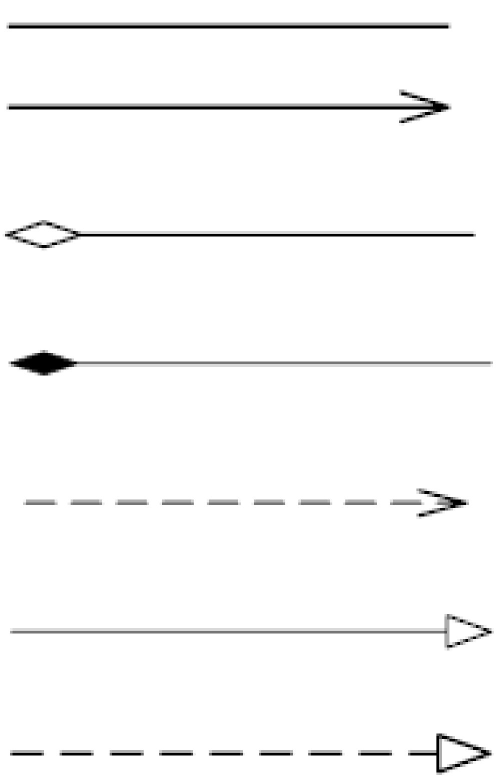
# Object Hierarchy: Composition, has_a



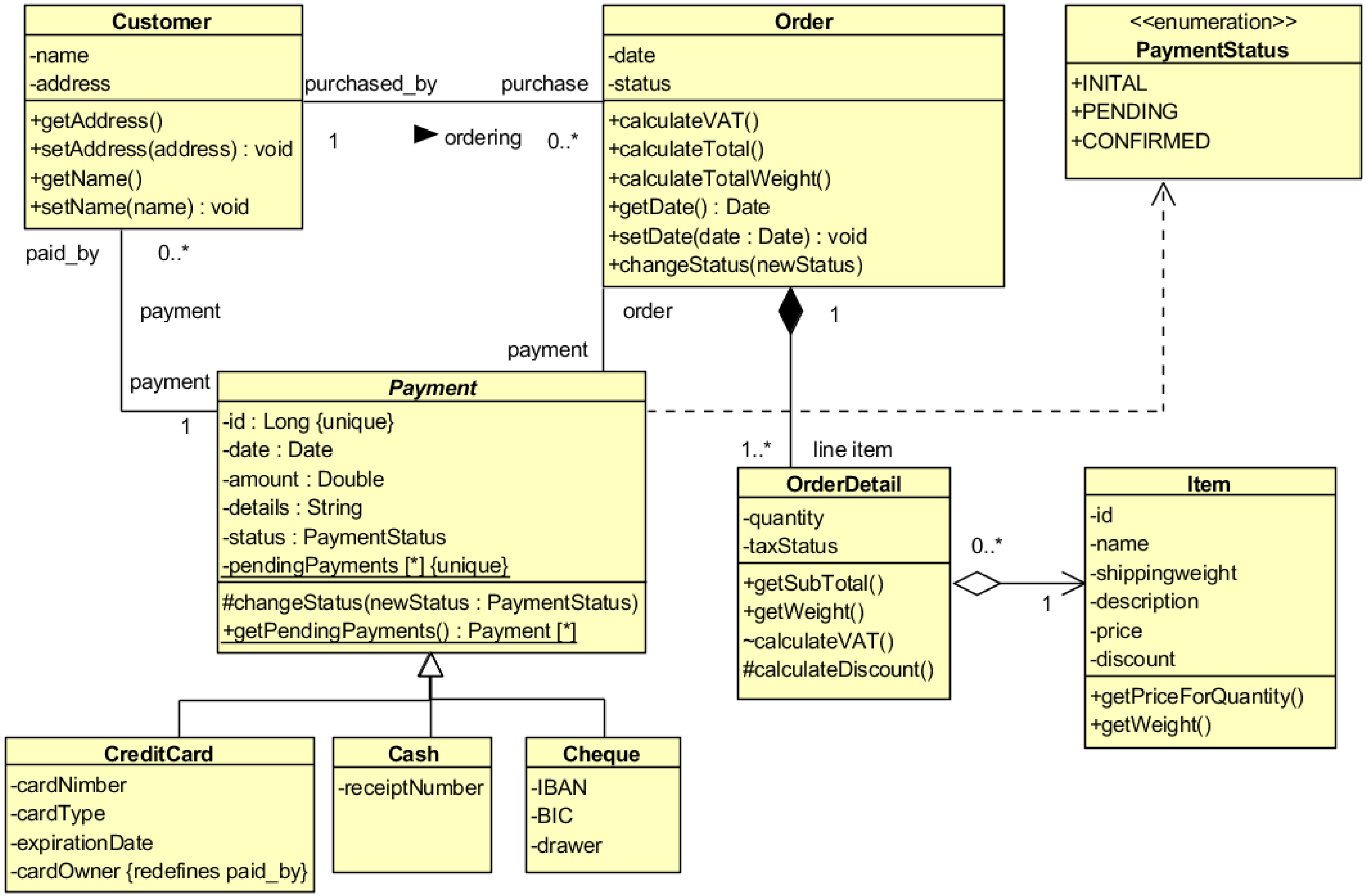
# Elements of Class Diagrams



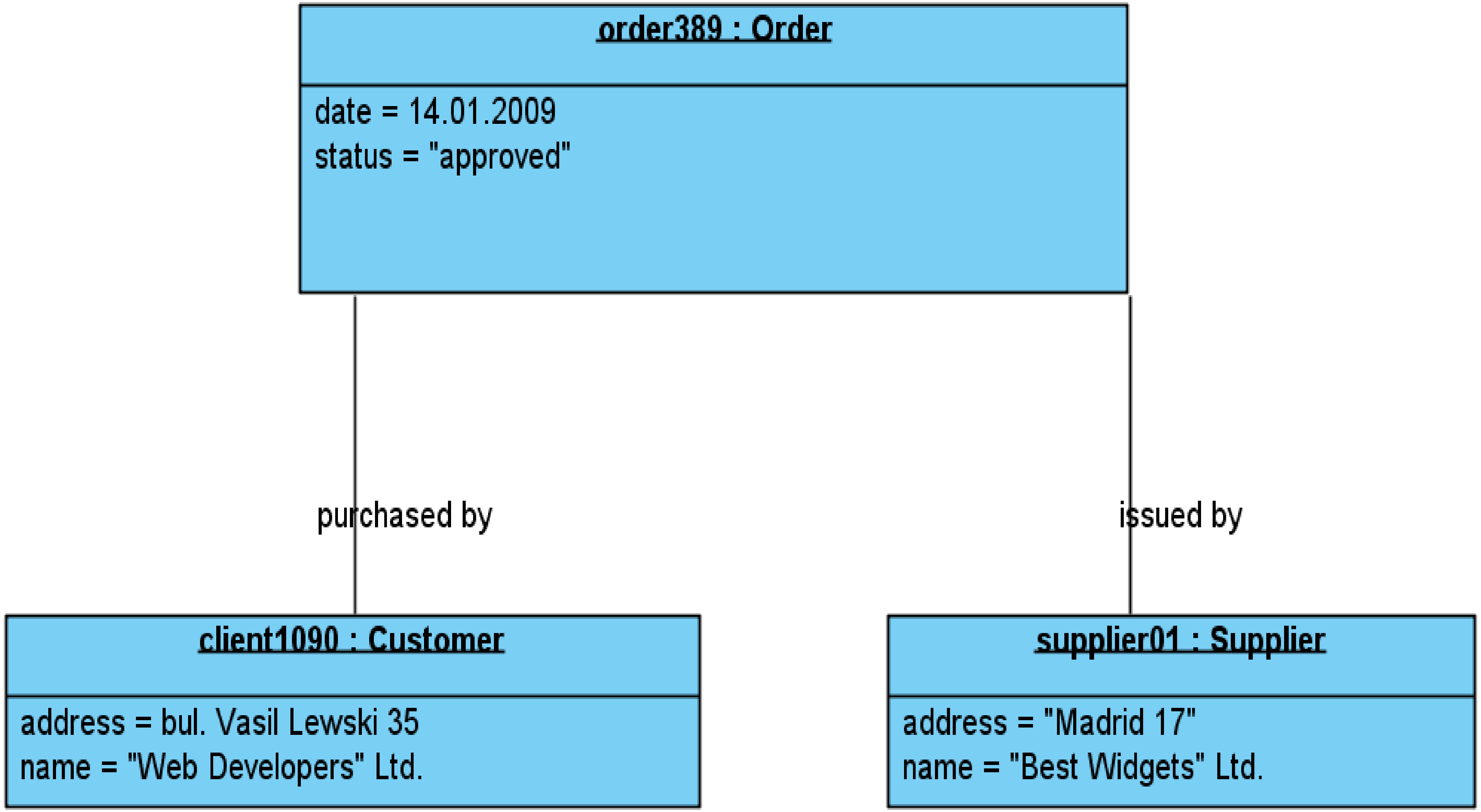

- Types of connections:
- Association
- aggregation
- composition
- dependence
- generalization
- realization



# Class Diagram



# Object Diagram



# Code (Component) Reuse

- Advantages of code reuse
- Ways of implementation:
  - Objects composition – patterns like composite, singleton, decorator, mixin, etc.
  - Inheritance of classes (object types) – features/patterns like dynamic polymorphism, prototype, template method, strategy, etc.

# Object Expressions

- Object expressions create objects of anonymous classes, that is, classes that aren't explicitly declared with the class declaration. Such classes are useful for one-time use. You can define them from scratch, inherit from existing classes, or implement interfaces. Instances of anonymous classes are also called anonymous objects because they are defined by an expression, not a name.

```
val helloWorld = object {
    val hello = "Hello"
    val world = "World"
    // object expressions extend Any, so `override` is required on `toString()`
    override fun toString() = "$hello $world"
}
fun main() {
    println(helloWorld) // prints: Hello World
}
```

# Inheriting anonymous objects from supertypes

- To create an object of an anonymous class that inherits from some type (or types), specify this type after object and a colon (:). Then implement or override the members of this class as if you were inheriting from it:

```
val window = JFrame("Main Window")

window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { /*...*/ }
    override fun mouseEntered(e: MouseEvent) { /*...*/ }
})

window.size = Dimension(600, 400)
window.defaultCloseOperation = EXIT_ON_CLOSE
window.isVisible = true
```

# Using supertype's constructor

- If a supertype has a constructor, pass appropriate constructor parameters to it. Multiple supertypes can be specified as a comma-delimited list after the colon:

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B { /*...*/ }

val ab: A = object : A(1), B {
    override val y = 15
}
```

# Using anonymous objects as return and value types - I

- When an anonymous object is used as a type of a **local** or **private** but not inline declaration (function or property), all its members are accessible via this function or property:

```
class C {
    private fun getObject() = object {
        val x: String = "x"
    }

    fun printX() {
        println(getObject().x)
    }
}
```

# Using anonymous objects as return and value types - II

If this function or property is **public** or **private inline**, its **actual type** is:

- **Any** if the anonymous object doesn't have a declared supertype
- The **declared supertype** of the anonymous object, if there is exactly one such type
- The **explicitly declared type** if there is more than one declared supertype
- In all these cases, members added in the anonymous object are not accessible. Overridden members are accessible if they are declared in the actual type of the function or property.

# Using anonymous objects as return and value types - III

```kotlin
interface A {
    fun funFromA() {}
}

interface B
```

```kotlin
class C {
    // The return type is Any. x is not accessible
    fun getObject() = object {
        val x: String = "x"
    }

    // The return type is A; x is not accessible
    fun getObjectA() = object: A {
        override fun funFromA() {}
        val x: String = "x"
    }

    // The return type is B; funFromA() and x are not accessible
    fun getObjectB(): B = object: A, B { // explicit return type is required
        override fun funFromA() {}
        val x: String = "x"
    }
}
```

# Using anonymous objects as return and value types - II

If this function or property is **public** or **private inline**, its **actual type** is:

- **Any** if the anonymous object doesn't have a declared supertype
- The **declared supertype** of the anonymous object, if there is exactly one such type
- The **explicitly declared type** if there is more than one declared supertype
- In all these cases, members added in the anonymous object are not accessible. Overridden members are accessible if they are declared in the actual type of the function or property.

# Accessing variables from anonymous objects

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

# Object Declarations

- The Singleton pattern can be useful in several cases, and Kotlin makes it easy to declare singletons. The initialization of an object declaration is thread-safe and done on first access.
- This is called an object declaration, and it always has a name following the **object** keyword. Just like a variable declaration, an object declaration is not an expression, and it cannot be used on the right-hand side of an assignment statement.

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

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

# Companion Objects

- If you need to write a function that can be called without having a class instance but that needs access to the internals of a class (such as a factory method), you can write it as a member of an object declaration inside that class.

- Even more specifically, if you declare a companion object inside your class, you can access its members using only the class name as a qualifier.

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
val instance = MyClass.create()
```

```
class MyClass2 {
    companion object { }
}
val x = MyClass2.Companion
```

# Companion Objects - II

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

class MyClass5 {
    companion object : Factory<MyClass5> {
        override fun create(): MyClass5 = MyClass5()
    }
}

val f: Factory<MyClass5> = MyClass5
```

# Difference between object expressions and declarations

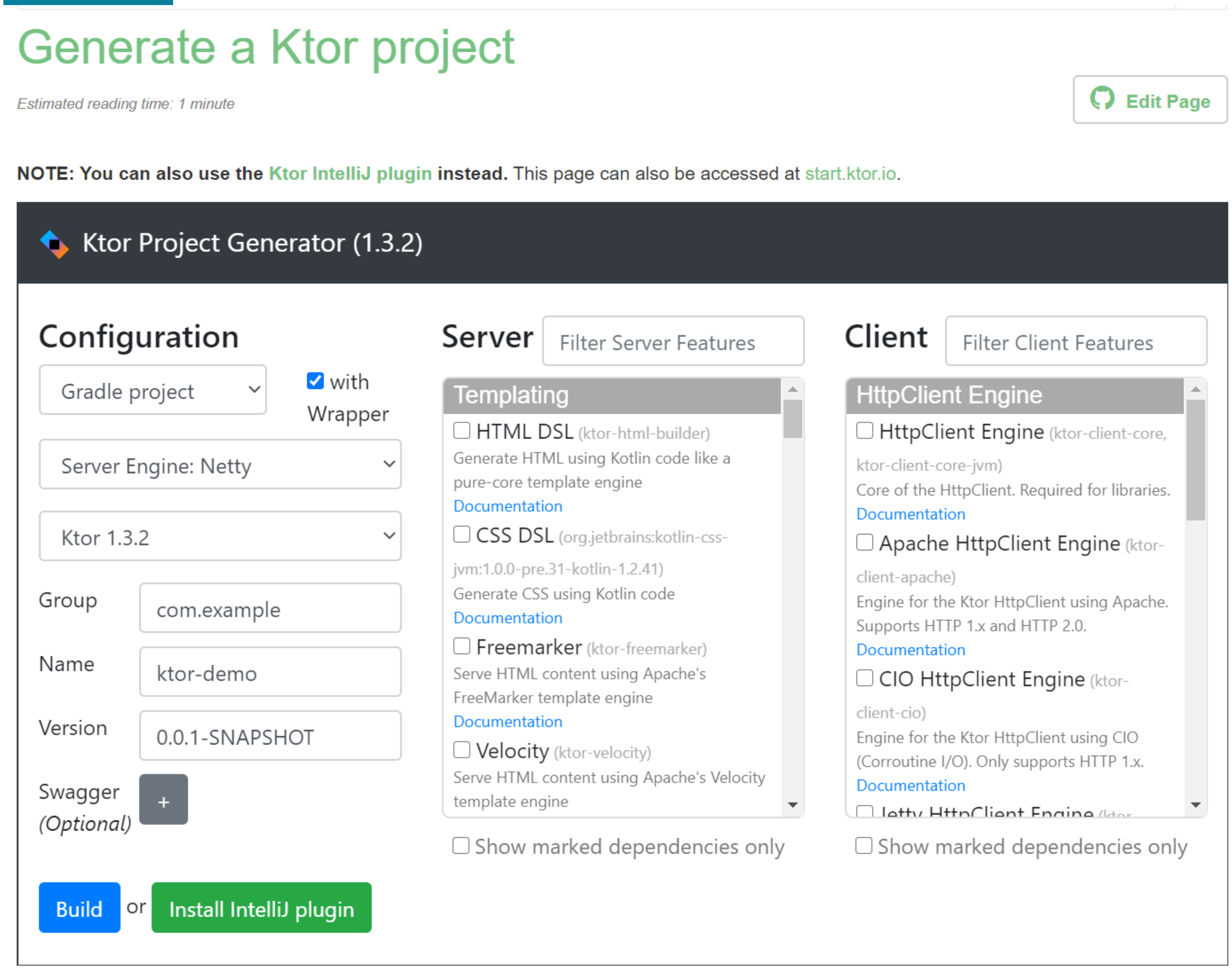
- Object expressions are executed (and initialized) immediately, where they are used.
- Object declarations are initialized lazily, when accessed for the first time.
- A companion object is initialized when the corresponding class is loaded (resolved) that matches the semantics of a Java static initializer.

# Examples

```kotlin
data class Product(val name: String, val price: Double)
class Order(val number: Int, val products: List<Product>, val date: LocalDateTime = LocalDateTime.now() ){
    fun calculateTotal(): Double {
        return products
            .map { it.price }
            .reduce { acc, prod  ->  acc + prod }
        }
    }

// And if you check a type is right, the compiler will auto-cast it for you
fun calculateTotal(obj: Any): Double? {
    if (obj is Order)  return obj.calculateTotal()
    return 0.0
}

fun main() {
    val products = listOf(Product("Keyboard", 27.5),  Product("Mouse", 17.1))
    val order = Order(1, products)
    println(calculateTotal((order))); // => 44.6
}
```




Source: https://kotlinlang.org/

# Simple Web Service with Ktor

```kotlin
fun main() {
    val server = embeddedServer(Netty, 8080) {
        routing {
            get("/hello") {
                call.respondText("<h2>Hello from Ktor and Kotlin!</h2>", ContentType.Text.Html)
            }
        }
    }
    server.start(true)
}
```

**… And that’s all :)**

```kotlin
data class Product(val name: String, val price: Double, var id: Int)

object Repo: ConcurrentHashMap<Int, Product>() {
    private idCounter = AtomicInteger()
    fun addProduct(product: Product) {
        product.id = idCounter.incrementAndGet()
        put(product.id, product)
    }
}
fun main() {
    embeddedServer(Netty, 8080, watchPaths = listOf("build/classes"), module= Application::mymodule).start(true)
}
fun Application.mymodule() {
    install(DefaultHeaders)
    install(CORS) {  maxAgeInSeconds = Duration.ofDays(1).toSeconds() }
    install(Compression)
    install(CallLogging)
    install(ContentNegotiation) {
        gson {
            setDateFormat(DateFormat.LONG)
            setPrettyPrinting()
        }
    }
```

```kotlin
routing {
    get("/products") {
        call.respond(Repo.values)
    }
    get("/products/{id}") {
        try {
            val item = Repo.get(call.parameters["id"]?.toInt())
            if (item == null) {
                call.respond(
                    HttpStatusCode.NotFound,
                    """{"error":"Product not found with id = ${call.parameters["id"]}"}"""
                )
            } else {
                call.respond(item)
            }
        } catch(ex :NumberFormatException) {
            call.respond(HttpStatusCode.BadRequest,
                """{"error":"Invalid product id: ${call.parameters["id"]}"}""")
        }
    }
```

```
post("/products") {
            errorAware {
                val product: Product = call.receive<Product>(Product::class)
                println("Received Post Request: $product")
                Repo.addProduct(product)
                call.respond(HttpStatusCode.Created, product)
            }
        }
    }
}

private suspend fun <R> PipelineContext<*, ApplicationCall>.errorAware(block: suspend () -> R): R? {
    return try {
        block()
    } catch (e: Exception) {
        call.respondText(
            """{"error":"$e"}""",
            ContentType.parse("application/json"),
            HttpStatusCode.InternalServerError
        )
        null
    }
}
```

# Ktor Applications

- **Ktor Server Application** is a custom program **listening to one or more ports** using a **configured server engine**, **composed by modules** with the application logic, that install **features, like routing, sessions, compression**, etc. to handle **HTTP/S 1.x/2.x** and **WebSocket** requests.
- **ApplicationCall** – the context for handling routes, or directly intercepting the pipeline – provides access to two main properties **ApplicationRequest** and **ApplicationResponse** ,as well as **request parameters, attributes, authentication, session, typesafe locations**, and the **application** itself. Example:

```
intercept(ApplicationCallPipeline.Call) {
    if (call.request.uri == "/")
        call.respondHtml {
            body {
                a(href = "/products") { + "Go to /products" }
} } }
```

# Routing DSL Using Higher Order Functions

- **routing**, **get**, and **post** are all **higher-order functions** (functions that take other functions as parameters or return functions).
- Kotlin has a convention that **if the last parameter to a function is another function**, we can place this **outside of the brackets**
- **routing** is a **lambda with receiver** == higher-order function taking as parameter an **extension function** => anything enclosed within **routing** has access to members of the type **Routing**.
- **get** and **post** are functions of the **Routing** type => also **lambdas with receivers**, with own members, such as **call**.
- This combination of **conventions** and **functions** allows to create elegant DSLs, such as Ktor's **routing DSL**.

# Features

- A **feature** is a **singleton** (usually a **companion object**) that you can **install and configure for a pipeline**.
- Ktor includes some **standard features**, but you **can add your own** or other features from the community.
- You can install features in **any pipeline**, like the **application** itself, or specific **routes**.
- Features are **injected into the request and response pipeline**. Usually, an application would have a series of features such as **DefaultHeaders** which add headers to every outgoing response, **Routing** which allows us to define routes to handle requests, etc.

# Installing Features

Using *install*:

```
fun Application.main() {
    install(DefaultHeaders)
    install(CallLogging)
    install(Routing) {
        get("/") {
            call.respondText("Hello, World!")
        }
    }
}
```

Using routing DSL:

```
fun Application.main() {
    install(DefaultHeaders)
    install(CallLogging)
    routing {
        get("/") {
            call.respondText("Hello, World!")
        }
    }
}
```

# Learn Kotlin by Example & Kotlin idioms

https://play.kotlinlang.org/byExample/

https://kotlinlang.org/docs/idioms.html

# Thank’s for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD