Classes and Objects

About me

Trayan Iliev

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

– Oracle® certified programmer 15+ Y

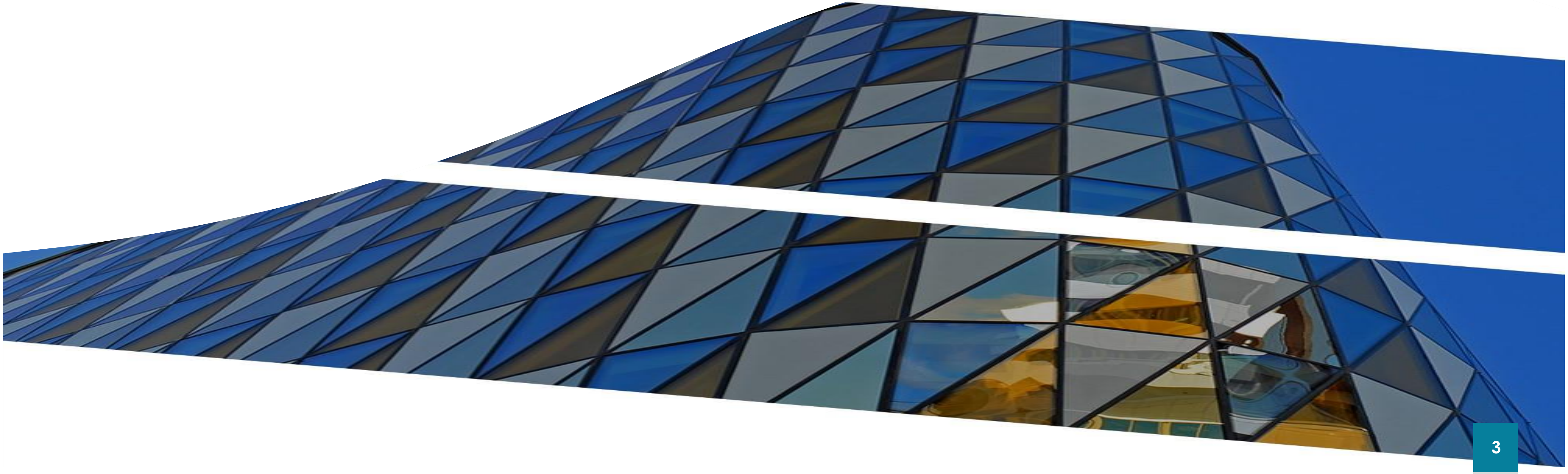
– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 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.

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

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

```kotlin
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

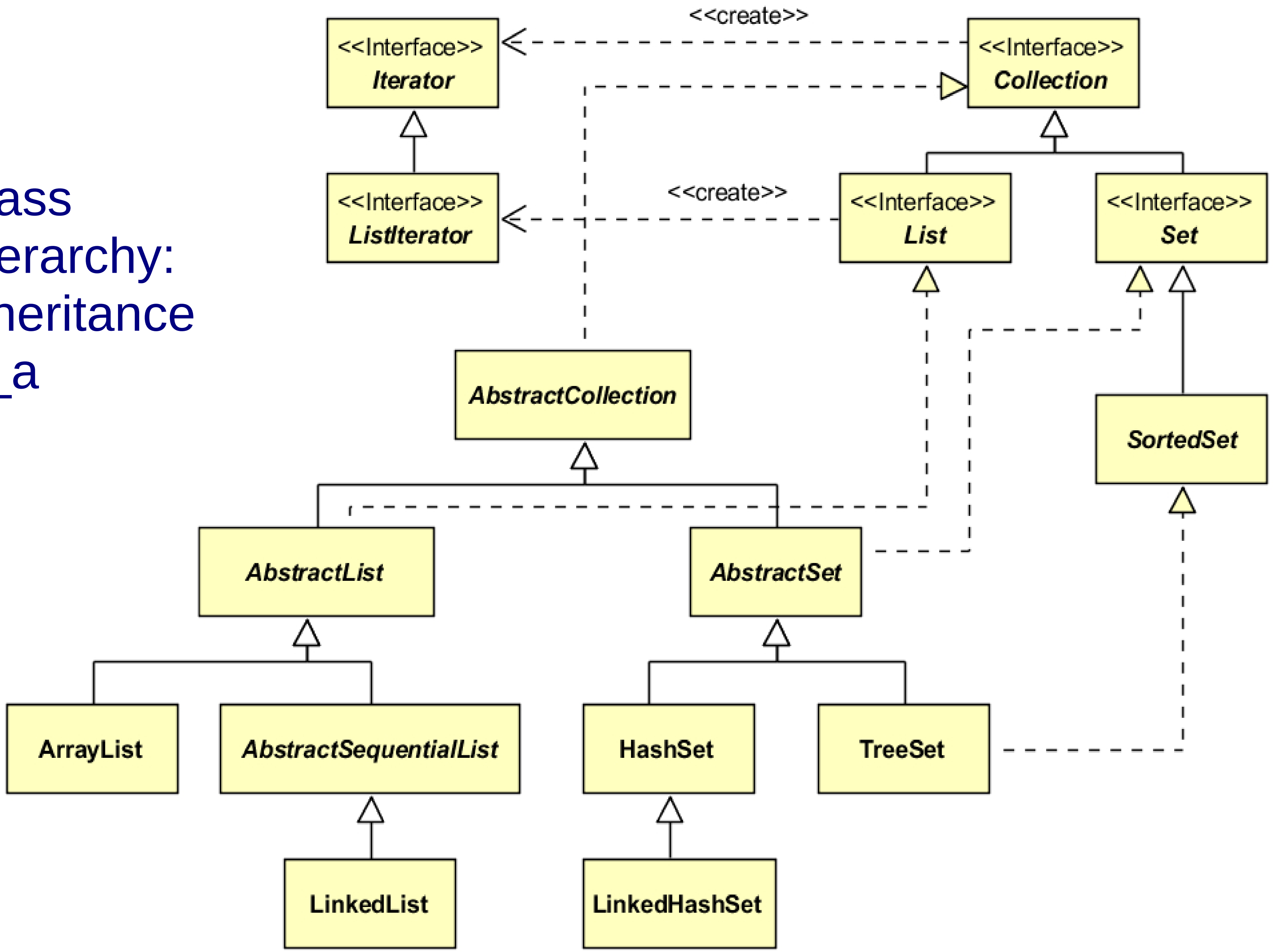
Class – describes common features for a set of objects: structure, behavior and possible links to objects of other classes = **objects type**

- 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)

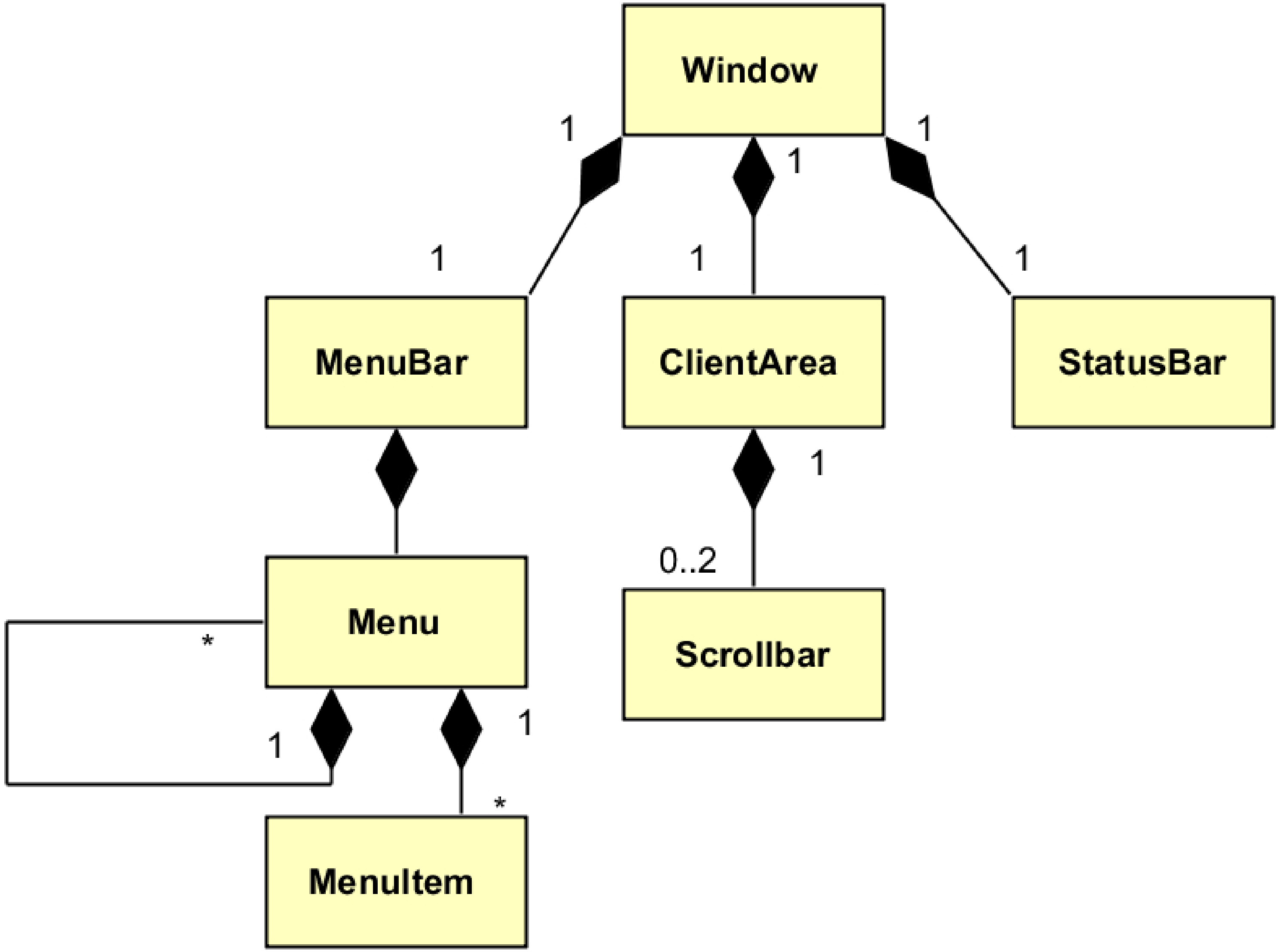
Objects are instances of the class, which in addition have:

- own state
- unique identifier = reference pointing towards object

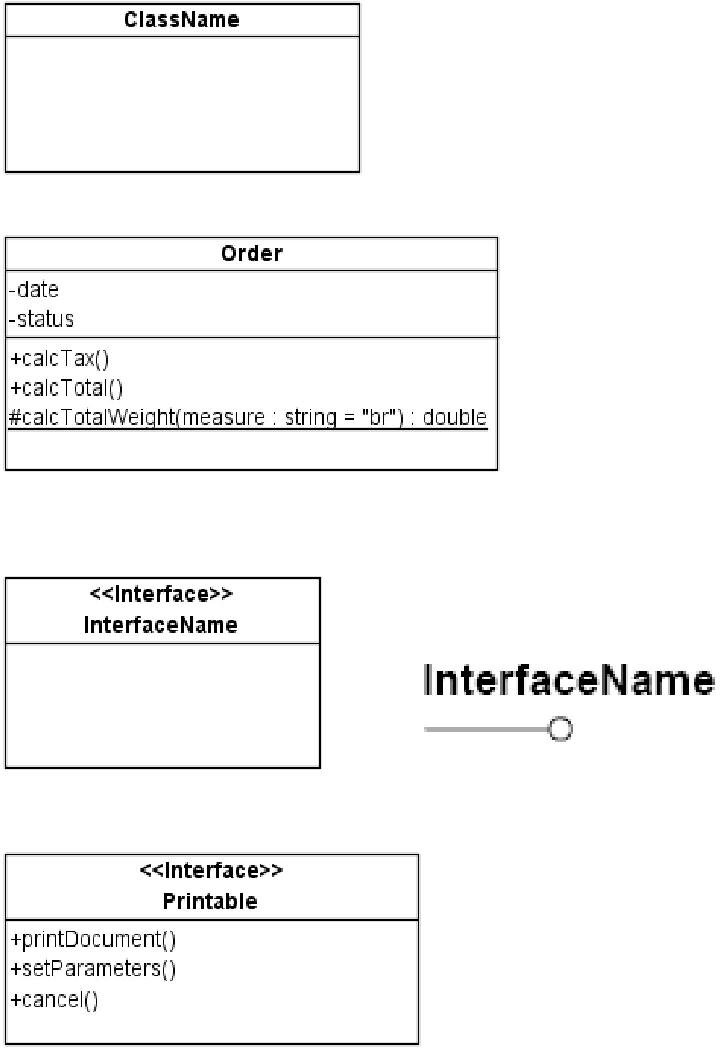
Class Hierarchy: Inheritance is_a



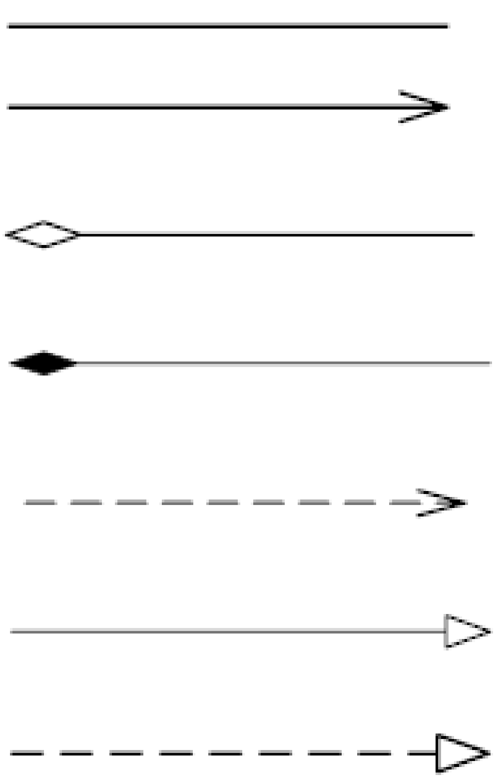
Object Hierarchy: Composition, has_a



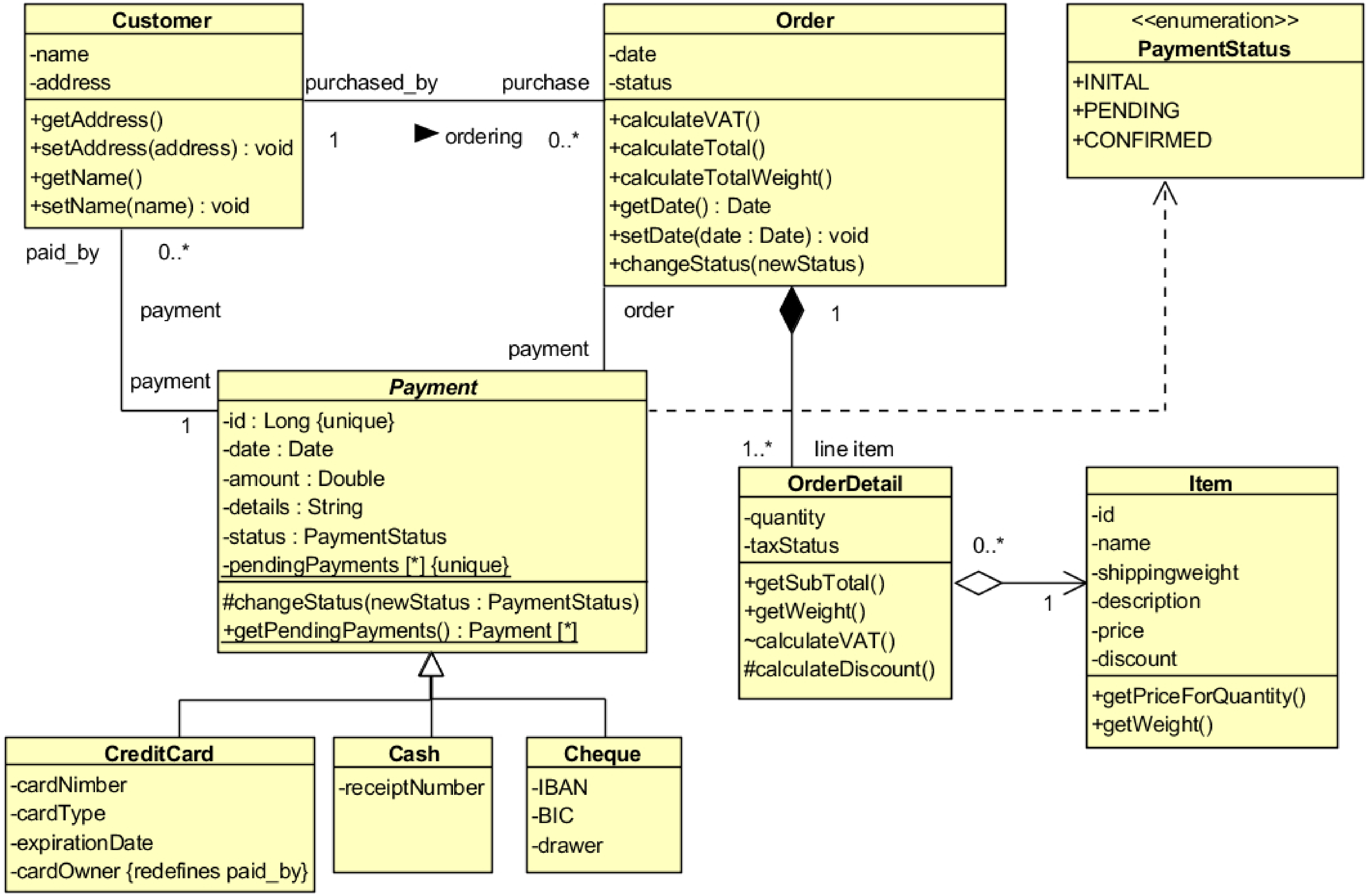
Elements of Class Diagrams



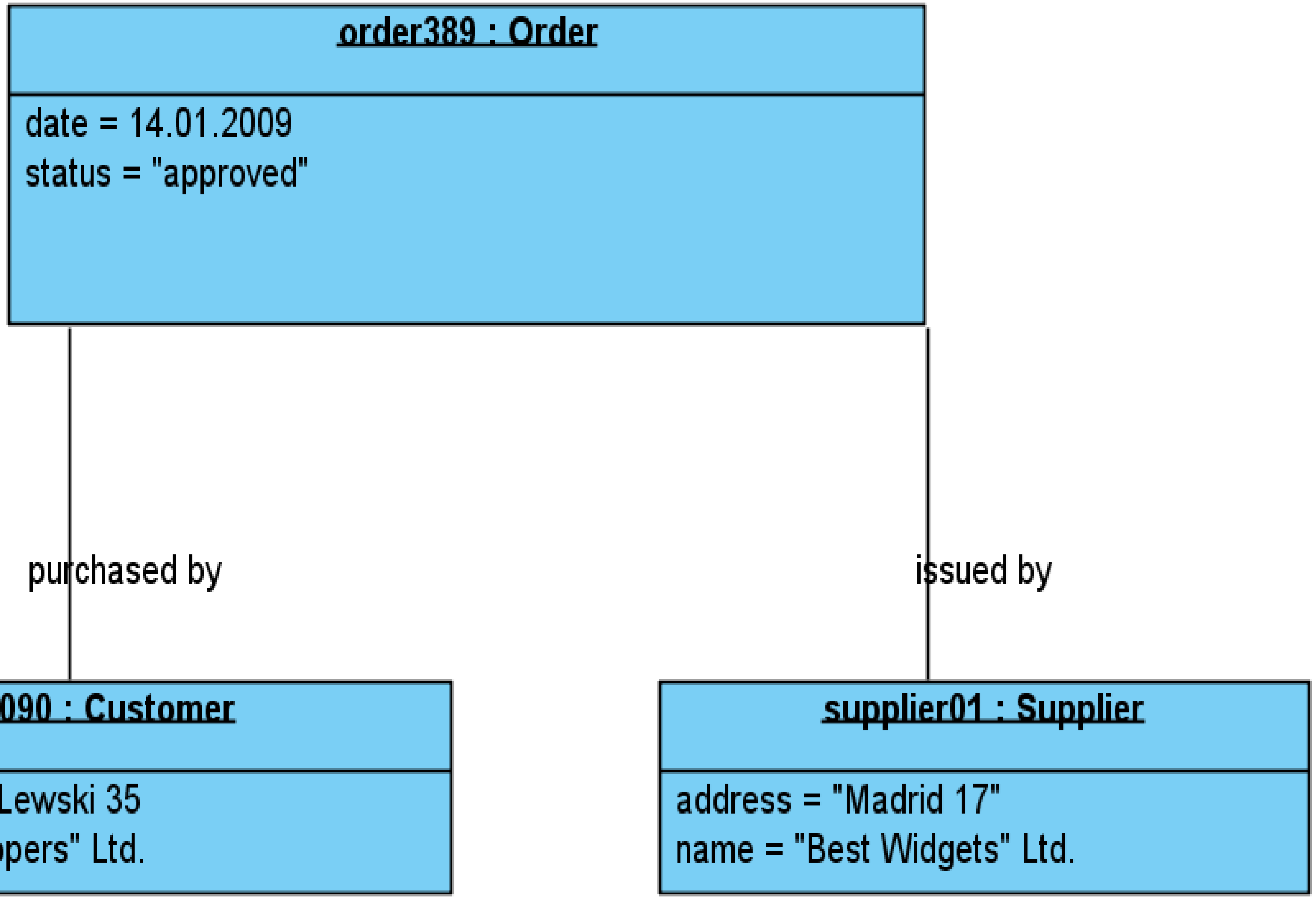
- Types of connections:
- Association
- aggregation
- composition
- dependence
- generalization
- realization



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.

```kotlin
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

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

interface B
```

```
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.

```
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.

Properties

Declaring Properties

```kotlin
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}

fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

Getters and Setters

var <propertyName>[: <PropertyType>] [= <property_initializer>]

[<getter>]

[<setter>]

```
var initialized = 1 // has type Int, default getter and setter
// var allByDefault // ERROR: explicit initializer required, default getter and setter implied
val simple: Int? // has type Int, default getter, must be initialized in constructor
val inferredType = 1 // has type Int and a default getter

val square get() = this.width * this.height

var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string and assigns values to other properties
    }
```

Getters and Setters Visibility

- If you need to annotate an accessor or change its visibility, but you don't need to change the default implementation, you can define the accessor without defining its body:

```
var setterVisibility: String = "abc"
    private set // the setter is private and has the default implementation

var setterWithAnnotation: Any? = null
    @Inject set // annotate the setter with Inject
```

Backing Fields

- In Kotlin, a **field** is only used as a part of a property to hold its value in memory. Fields cannot be declared directly. However, when a property needs a backing field, Kotlin provides it automatically. This backing field can be referenced in the accessors using the **field** identifier:

```
var counter = 0 // the initializer assigns the backing field directly
    set(value) {
        if (value >= 0)
            field = value
        // counter = value // ERROR StackOverflow: Using actual name 'counter' would make setter
recursive
    }

class Square(val size: Int) {
    val isEmpty: Boolean  // no backing field generated
        get() = this.size == 0
}
```

Backing Properties

- If you want to do something that does not fit into this implicit backing field scheme, you can always fall back to having a **backing property**:

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

- On the JVM: Access to private properties with default getters and setters is optimized to avoid function call overhead.

Compile-time Constants

- If the value of a read-only property is known at compile time, mark it as a compile time constant using the **const** modifier. Such a property needs to fulfil the following requirements:

- It must be a top-level property, or a member of an object declaration or a companion object.

- It must be initialized with a value of type String or primitive type

- It cannot have custom getter

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"
@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

Late-initialized Properties and Variables

- Non-null type properties must be initialized in the constructor. However, properties can be initialized through dependency injection, or in the setup method of a unit test. In these cases, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // dereference directly
    }
}
```

Late-initialized Properties and Variables - II

- Non-null type properties must be initialized in the constructor. However, properties can be initialized through dependency injection, or in the setup method of a unit test. In these cases, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class:

```
public class MyTest {
    lateinit var subject: TestSubject
    @SetUp fun setup() {
        subject = TestSubject()
    }
    @Test fun test() {
        if (::subject.isInitialized) {
            subject.method()  // dereference directly
        }
    }
}
```

Delegated Properties

- Lazy values, reading from a map by a given key, accessing a database, notifying a listener on access, etc.:

```
class Delegate {
    var _value: String = ""
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return "$thisRef, thank you for delegating '${property.name}' to me: $_value"
    }
    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
        _value =value
        println("$value has been assigned to '${property.name}' in $thisRef.")
    }
}
class Example {
    var prop: String by Delegate()
}
var topProp: String by Delegate() // top level property
```

Standard Delegates: Lazy properties

- **lazy()** is a function that takes a lambda and returns an instance of **Lazy<T>**, which can serve as a delegate for implementing a lazy property. The first call to **get()** executes the lambda passed to **lazy()** and remembers the result. Subsequent calls to **get()** simply return the remembered result:

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

```
computed!
Hello
Hello
```

Standard Delegates: Lazy properties - II

- **lazy()** is a function that takes a lambda and returns an instance of **Lazy<T>**, which can serve as a delegate for implementing a lazy property. The first call to **get()** executes the lambda passed to **lazy()** and remembers the result. Subsequent calls to **get()** simply return the remembered result:

```
val lazyValue: String by lazy(LazyThreadSafetyMode.PUBLICATION) { // init not synchronized
    println("computed!")
    "Hello"
}

fun main() {
    println(lazyValue)
    println(lazyValue)
}
```

```
computed!
Hello
Hello
```

Standard Delegates: Observable properties

- Delegates.observable() takes two arguments: the initial value and a handler for modifications.
- The handler is called every time you assign to the property (after the assignment has been performed). It has three parameters: the property being assigned to, the old value, and the new value:

```
class User {
    var name: String by Delegates.observable("<no name>") {
            prop, old, new ->
        println("$old -> $new")
    }
}
fun main() {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

Delegating to Another Property

- A property can delegate its getter and setter to another property. Such delegation is available for both top-level and class properties (member and extension). The delegate property can be:
  - A top-level property
  - A member or an extension property of the same class
  - A member or an extension property of another class
- To delegate a property to another property, use the **:: qualifier** in the delegate name, for example, **this::delegate** or **MyClass::delegate**:

```
var topLevelInt: Int = 0
class ClassWithDelegate(val anotherClassInt: Int)
class MyClass7(var memberInt: Int, val anotherClassInstance: ClassWithDelegate) {
    var delegatedToMember: Int by this::memberInt
    var delegatedToTopLevel: Int by ::topLevelInt
    val delegatedToAnotherClass: Int by anotherClassInstance::anotherClassInt
}
var MyClass7.extDelegated: Int by ::topLevelInt
```

Delegating to Another Property – Example @Deprecated

```
class MyClass8 {
    var newName: Int = 0


    @Deprecated("Use 'newName' instead", ReplaceWith("newName"))
    var oldName: Int by this::newName
}


fun main() {
    val myClass8 = MyClass8()
    // Notification: 'oldName: Int' is deprecated.
    // Use 'newName' instead
    myClass8.oldName = 42
    println(myClass8.newName) // 42
}
```

Example: Storing properties in a map

```
class User2(private val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}
val user2 = User2(mapOf(
    "name" to "John Doe",
    "age"  to 25
))

class MutableUser(val map: MutableMap<String, Any?>) {
    var name: String by map
    var age: Int     by map
}
val mutableUser = MutableUser(
    mutableMapOf(
    "name" to "John Doe",
    "age"  to 25
))
```

Local Delegated Properties

```
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

Property Delegate Requirements - val

- For a read-only property (**val**), a delegate should provide an operator function **getValue()** with the following parameters:
  - **thisRef** - must be the same type as, or a supertype of, the property owner (for extension properties, it should be the type being extended).
  - **property** - must be of type KProperty<*> or its supertype.
- getValue() must return the same type as the property (or its subtype):

```
class Resource
class Owner {
    val valResource: Resource by ResourceDelegate()
}
class ResourceDelegate {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return Resource()
    }
}
```

Property Delegate Requirements - var

- For a mutable property (var), a delegate has to additionally provide an operator function **setValue()** with the following parameters:
  - **thisRef** - must be the same type as, or a supertype of, the property owner (for extension properties, it should be the type being extended).
  - **property** - must be of type KProperty<*> or its supertype.
  - **value** - must be of the same type as the property (or its supertype).

```
class Resource2
class Owner2 {   var varResource: Resource2 by ResourceDelegate2() }

class ResourceDelegate2(private var resource: Resource2 = Resource2()) {
    operator fun getValue(thisRef: Owner2, property: KProperty<*>): Resource2 {
        return resource
    }
    operator fun setValue(thisRef: Owner2, property: KProperty<*>, value: Any?) {
        if (value is Resource2) { resource = value }
    }
}
```

Delegation using ReadOnlyProperty and ReadWriteProperty

```
fun resourceDelegate(): ReadWriteProperty<Any?, Int> =
    object : ReadWriteProperty<Any?, Int> {
        var curValue = 0
        override fun getValue(thisRef: Any?, property: KProperty<*>): Int = curValue
        override fun setValue(thisRef: Any?, property: KProperty<*>, value: Int) {
            curValue = value
        }
    }

val readOnly: Int by resourceDelegate()  // ReadWriteProperty as val
var readWrite: Int by resourceDelegate()
```

Translation Rules

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler instead:
class C {
    private val prop$delegate = MyDelegate()
    var prop: Int
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

provideDelegate operator - I

```
class C {
    var prop: Type by MyDelegate()
}

// this code is generated by the compiler instead:
class C {
    private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)
    var prop: Int
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

provideDelegate operator - II

```
class ResourceID<T>(val id: T) {
    companion object { … }
}
class ResourceDelegate4<T> : ReadOnlyProperty<MyUI, T> {
    override fun getValue(thisRef: MyUI, property: KProperty<*>): T = /* … */
}
class ResourceLoader4<T>(id: ResourceID<T>) {
    operator fun provideDelegate(
        thisRef: MyUI,
        prop: KProperty<*>
    ): ReadOnlyProperty<MyUI, T> {
        checkProperty(thisRef, prop.name)
        // create delegate
        return ResourceDelegate4()
    }
    private fun checkProperty(thisRef: MyUI, name: String) { /*...*/
    }
}
```

provideDelegate operator - III

```
class MyUI {
    fun <T> bindResource(id: ResourceID<T>): ResourceLoader4<T> = ResourceLoader4(id)

    val image by bindResource(ResourceID.image_id)
    val text by bindResource(ResourceID.text_id)
}
```

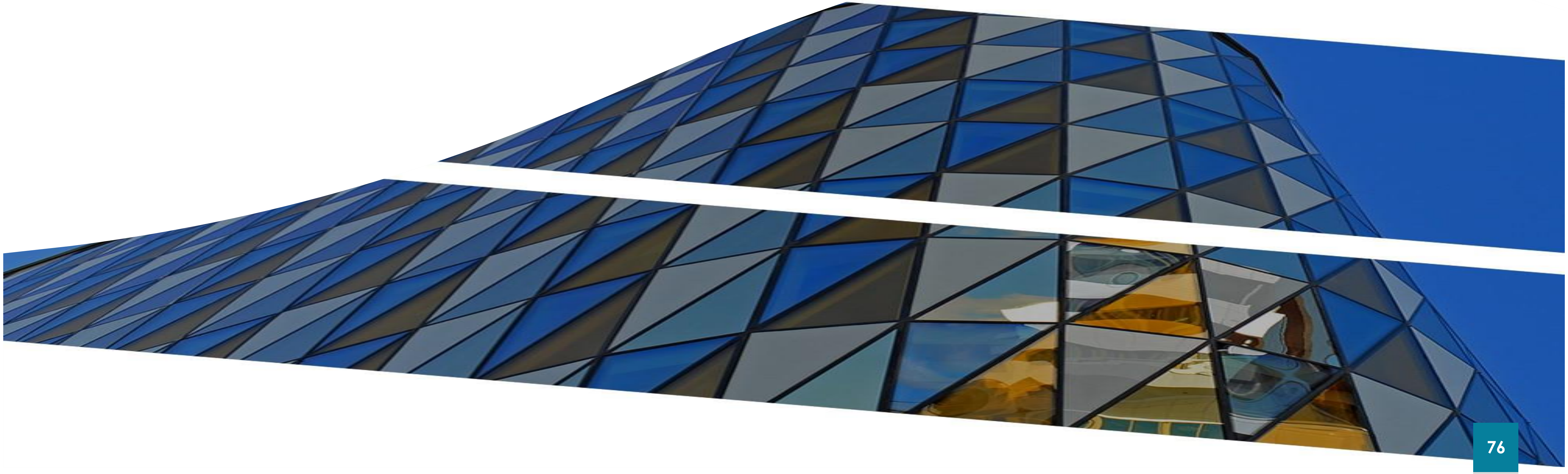
PropertyDelegateProvider

```
val provider = PropertyDelegateProvider { thisRef: Any?, property ->
    ReadOnlyProperty<Any?, Int> {_, property -> 42 }
}
val delegate: Int by provider
```

Local Delegated Properties

```kotlin
fun example(computeFoo: () -> Foo) {
    val memoizedFoo by lazy(computeFoo)

    if (someCondition && memoizedFoo.isValid()) {
        memoizedFoo.doSomething()
    }
}
```

Sealed Classes



Sealed Classes - I

- Sealed classes and interfaces represent restricted class hierarchies that provide more control over inheritance.
- All direct subclasses of a sealed class are known at compile time. No other subclasses may appear after a module with the sealed class is compiled.
- The same works for sealed interfaces and their implementations: once a module with a sealed interface is compiled, no new implementations can appear.
- In some sense, sealed classes are similar to enum classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances, each with its own state.

Sealed Classes - II

- As an example, consider a library's API. It's likely to contain error classes to let the library users handle errors that it can throw. If the hierarchy of such error classes includes interfaces or abstract classes visible in the public API, then nothing prevents implementing or extending them in the client code. However, the library doesn't know about errors declared outside it, so it can't treat them consistently with its own classes. With a sealed hierarchy of error classes, library authors can be sure that they know all possible error types and no other ones can appear later.
- A sealed class is abstract by itself, it cannot be instantiated directly and can have abstract members.
- Constructors of sealed classes can have one of two visibilities: protected (by default) or private.

Example

```
sealed interface Error

sealed class IOError(): Error

class FileReadError(val f: File): IOError()

class DatabaseError(val source: DataSource): IOError()

object RuntimeError : Error

sealed class IOError {
    constructor() { /*...*/ } // protected by default
    private constructor(description: String): this() { /*...*/ } // private is OK
    // public constructor(code: Int): this() {} // Error: public and internal are not allowed
}
```

Location of direct subclasses

- Direct subclasses of sealed classes and interfaces must be declared in the same package. They may be top-level or nested inside any number of other named classes, named interfaces, or named objects. Subclasses can have any visibility as long as they are compatible with normal inheritance rules in Kotlin.
- Subclasses of sealed classes must have a proper qualified name. They can't be local nor anonymous objects.
- enum classes can't extend a sealed class (as well as any other class), but they can implement sealed interfaces.
- These restrictions don't apply to indirect subclasses. If a direct subclass of a sealed class is not marked as sealed, it can be extended in any ways that its modifiers allow.

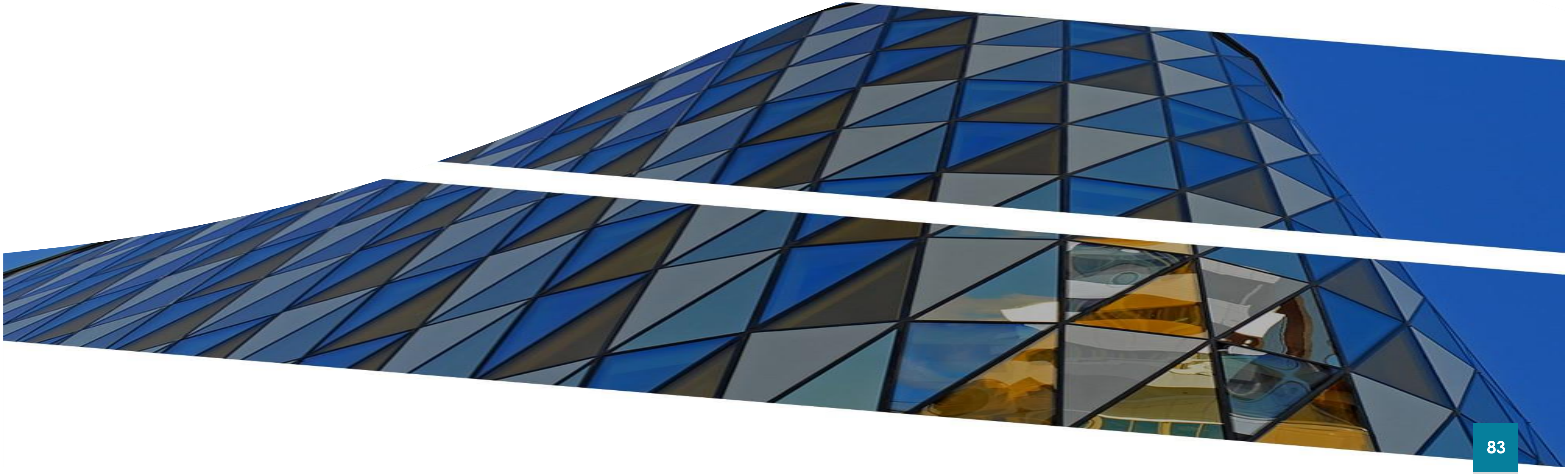
Example

```
sealed interface Error // has implementations only in same package and module

sealed class IOError(): Error // extended only in same package and module
open class CustomError(): Error // can be extended wherever it's visible
```

Sealed classes and when expression

```
fun log(e: Error) = when(e) {
    is FileReadError -> { println("Error while reading file ${e.file}") }
    is DatabaseError -> { println("Error while reading from database ${e.source}") }
    RuntimeError ->  { println("Runtime error") }
    // the `else` clause is not required because all the cases are covered
}
```

Generics: in, out, where



Parameterizied Types: Generics in Java (1)

- Collections and their methods before Java 5 were limited to handle a single type of elements.
- If we want to create typed containers we had to implement different container types for each entity type.
- *Example:* In a e-Bookstore we want to sell **Books** and want the container to contain only **Books** (being strongly typed) --> we should implement separate class **BookList**, as well as for each **Book** we want to keep a list of **Authors** --> we should implement **AuthorList** too, and so on.

Parameterizied Types: Generics in Java (2)

- *Solution:* We can skip writing multiple similar classes (e.g. typed containers for each type of elements) using **Generic types**

- *Generic type invocation:*

```
List<Book> books = new ArrayList<Book>()
List<Author> authors = new ArrayList<Author>()
```

- **<> – *Diamond*** *operator* – new in Java™ 7, allows automatic inference of the generic type:

```
List<Book> books = new ArrayList<>()
List<Author> authors = new ArrayList<>()
```

Parameterizied Types: Generics in Java (3)

- *Generic type declaration:*

```
public class Position<T extends Product> {
        private T product;
        ...
        public Position(T product, double quantity) {
                this.product = product;
                this.quantity = quantity;
                price = product.getPrice();
        }
        public T getProduct() {
                return product;
        }
...
```

Generic data type

Conventions Naming Generic Parameters in Java

- **Generic parameters naming conventions:**

T – type parameter (if there are more – S, U, V, W ...)

E – element of a collection – e.g.: List<E>

K – key in associative pair – e.g.: Map<K,V>

V – value in associative pair – e.g.: Map<K,V>

N – number value

- *Example:*

```
public class Invoice <T extends Product> {
        ...
        private List<Position<T>> positions = new ArrayList<>();
    ...
}
```

Generic Methods in Java (1)

- We can implement generic methods and constructors too:

```
public static <U extends Product> String
getPositionsAsString (List<Position<U>> positions) {
     StringBuilder posStr = new StringBuilder();
     int n = 0;
     for(Position<U> p: positions){
         posStr.append( String.format(
             "\n| %1$3s | %2$30s | %3$6s | %4$4s | %5$6s |%6$8s |",
             ++n, p.getProduct().getName(), p.getQuantity(),
             p.getProduct().getMeasure(),p.getPrice(), p.getTotal()
        ));
     }
     return posStr.toString();
} ...
```

Generic Methods in Java (2)

- Invoking generic method / constructor:

```
result += Invoice.<T> getPositionsAsString(positions);
```

- OR we can let Java to automatically infer the generic type:

```
result += Invoice.getPositionsAsString(positions);
```

Bounded Type Parameters in Java

- We can define upper bound constraint for the possible types that can be allowed as actual generic type parameters of the class / method /constructor:

```
public static <U extends Product> String
getPositionsAsString (List<Position<U>> positions) { ... }
```

- OR

```
public static <U extends Product & Printable> String getPositionsAsString
(List<Position<U>> positions) {
    ...
    p.getProduct().print();
    ...
}
```

Generics Sub-typing

- If the class Product extends class Item, can we say that List<Product> extends List<Item> too? Can we substitute the first with the second?
- The answer is „**NOT**“, because the basic generic type is not designed to reflect the specifics of of the Products.
- Dos and donts when using generics inheritance:

```
interface Service extends Item; Service s = new Service( ...);
Collection<Service> services =  ...; services.add(s); // OK
interface Product extends Item; Product p = new Product( ...);
Collection<Product> products =  ...; products.add(p); // OK
Collection<Item> items = ...; items.add(s); items.add(p); // OK
items = products; // NOT OK
items = services; // NOT OK
```

Using ? as Type Specifier (Wildcards) in Java

- If we want to declare that we expect specific, but not pre-determined type, which for example extends the class **Item**, we could use **?** To designate this:

```
Collection<? extends Item> items; // Upper bound is Item
items = products; // OK
items = services; // OK
Items.add(p); // NOT OK – Can not write into it – it is not safe!
Items.add(s); // NOT OK – Can not write into it – it is not safe!
for(Item i: items) { // OK – Can read it – it is known to be at least Item.
    System.out.println( i.getName() + „:“ + i.getPrice() );
}
List<? super Product> products; // Lower bound is Product
products.add(p); // OK – Can write into it – it is now safe.
Product p = products.get(0); //NOT OK may be superclass of Product
```

- Producer extends and Consumer super (PECS) principle

Generics in Java

- "For maximum flexibility, use wildcard types on input parameters that represent producers or consumers", and proposes the following mnemonic: PECS stands for Producer-Extends, Consumer-Super.
- If you use a producer-object, say, List<? extends Foo>, you are not allowed to call add() or set() on this object, but this does not mean that it is immutable: for example, nothing prevents you from calling clear() to remove all the items from the list, since clear() does not take any parameters at all.The only thing guaranteed by wildcards (or other types of variance) is type safety. Immutability is a completely different story.

Example: Generics in Java

```
void copyAll(Collection<Object> to, Collection<String> from) {
        to.addAll(from);
        // !!! Would not compile with the naive declaration of addAll:
        // Collection<String> is not a subtype of Collection<Object>
}
public static void main(String[] args) {
        // Java
        List<String> strs = new ArrayList<String>();
        List<Object> objs = strs; // !!! A compile-time error here saves us from a runtime exception later.
        objs.add(1); // Put an Integer into a list of Strings
        String s = strs.get(0); // !!! ClassCastException: Cannot cast Integer to String

}
interface Collection<E> {
    void addAll(Collection<E> items);
}
interface Collection<E> {
    void addAll(Collection<? extends E> items);
}
```

Type Erasure & Reification in Java

- **Type Erasure** – chosen in java as backward-compatibility alternative – information about generic type parameters is erased during compilation, and is NOT available in runtime – the generic type becomes compiled to its basic raw type:

  Collection<Product> products; --(runtime)--> Collection products;

  This design decision creates problems if we want to create generic type instance with **new**, or to convert to the generic type, or to check the generic type using **instanceof**.

- **Reification** – better alternative strategy, implemented in languages such as C++, Ada и Eiffel, using which the generic type information is accessible in runtime.

Generics in Kotlin

```kotlin
class Box<T>(t: T) {
    var value = t
}


val box: Box<Int> = Box<Int>(1)


val box2 = Box(1) // 1 has type Int, so the compiler figures out that it is Box<Int>
```

Declaration-site variance

```
interface Source<out T> {           // Covariant
    fun nextT(): T
}
fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // This is OK, since T is an out-parameter
    // ...
}

interface Comparable<in T> {      // Contravariant
    operator fun compareTo(other: T): Int
}
fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double, which is a subtype of Number
    // Thus, you can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

The Existential Transformation: Consumer in, Producer out! :-)

Type projections

```
class Array<T>(val size: Int) {
    operator fun get(index: Int): T { /*...*/ return 1 as T }
    operator fun set(index: Int, value: T) { /*...*/ }
}

fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices) {
        to[i] = from[i]
    }
}
fun main() {
    val ints: Array<Int> = arrayOf(1, 2, 3)
    val any = Array<Any>(3) { "" }
    copy(ints, any) // Type mismatch. Required: Array<Any> Found: Array<Int>
    //   ^ type is Array<Int> but Array<Any> was expected
}
```

Type projections

```
class Array<T>(val size: Int) {
    operator fun get(index: Int): T { /*...*/ return 1 as T }
    operator fun set(index: Int, value: T) { /*...*/ }
}

fun copy(from: Array<out Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices) {
        to[i] = from[i]
    }
}
fun main() {
    val ints: Array<Int> = arrayOf(1, 2, 3)
    val any = Array<Any>(3) { "" }
    copy(ints, any) // Type mismatch. Required: Array<Any> Found: Array<Int>
    //   ^ type is Array<Int> but Array<Any> was expected
}
```

Type projections

- Array<T> is invariant in T, and so neither Array<Int> nor Array<Any> is a subtype of the other. Why not? Again, this is because copy could have an unexpected behavior, for example, it may attempt to write a String to from, and if you actually pass an array of Int there, a ClassCastException will be thrown later. To prohibit the copy function from writing to from, you can:

```
fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

- This is type projection, which means that from is not a simple array, but is rather a restricted (projected) one. You can only call methods that return the type parameter T, which in this case means that you can only call get(). This is Kotlin's approach to use-site variance, and it corresponds to Java's Array<? extends Object> while being simpler.
- You can project with **in** too:

```
fun fill(dest: Array<in String>, value: String) { ... }
```

Star-projections

- For Foo<out T : TUpper>, where T is a covariant type parameter with the upper bound TUpper, Foo<*> is equivalent to Foo<out TUpper>. This means that when the T is unknown you can safely read values of TUpper from Foo<*>.
- For Foo<in T>, where T is a contravariant type parameter, Foo<*> is equivalent to Foo<in Nothing>. This means there is nothing you can write to Foo<*> in a safe way when T is unknown.
- For Foo<T : TUpper>, where T is an invariant type parameter with the upper bound TUpper, Foo<*> is equivalent to Foo<out TUpper> for reading values and to Foo<in Nothing> for writing values.
- interface Function<in T, out U> :
- Function<*, String> means Function<in Nothing, String>
- Function<Int, *> means Function<Int, out Any?>
- Function<*, *> means Function<in Nothing, out Any?>

Generic Functions

```
fun <T> singletonList(item: T): List<T> {
    return emptyList()
}
fun <T> T.basicToString(): String { // extension function
    return ""
}
```

- Calling site (generic type can be inferred):

```
val l1 = singletonList<Int>(1)
val l2 = singletonList(1)
```

Generic constraints

```kotlin
fun <T : Comparable<T>> sort(list: List<T>) {  /*...*/ }
```

- Calling site (generic type can be inferred):

```kotlin
sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a subtype of Comparable<HashMap<Int, String>>
```

- The default upper bound (if there was none specified) is Any?. Only one upper bound can be specified inside the angle brackets. If the same type parameter needs more than one upper bound, you need a separate where -clause:

```kotlin
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>
        where T : CharSequence,
              T : Comparable<T> {
    return list.filter { it > threshold }.map { it.toString() }
}
```

Nested, Inner, Inline Classes

Nested Classes

```kotlin
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2

interface OuterInterface {
    class InnerClass
    interface InnerInterface
}

class OuterClass {
    class InnerClass
    interface InnerInterface
}
```

Inner Classes

- Can access members of the outer class, because is created with outer class as closure:

```kotlin
class Outer2 {
    private val bar: Int = 1
    inner class Inner2 {
        fun foo() = bar
    }
}

val demo2 = Outer2().Inner2().foo() // == 1
```

Qualified this

```
class A { // implicit label @A
    inner class B { // implicit label @B
        fun Int.foo() { // implicit label @foo
            val a = this@A // A's this
            val b = this@B // B's this
            val c = this // foo()'s receiver, an Int
            val c1 = this@foo // foo()'s receiver, an Int
            val funLit = lambda@ fun String.() {
                val d = this // funLit's receiver
            }
            val funLit2 = { s: String ->
                // foo()'s receiver, since enclosing lambda expression
                // doesn't have any receiver
                val d1 = this
            }
        }
    }
}
```

Anonymous Inner Classes

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


val listener = ActionListener { println("clicked") }
```

Inline Classes

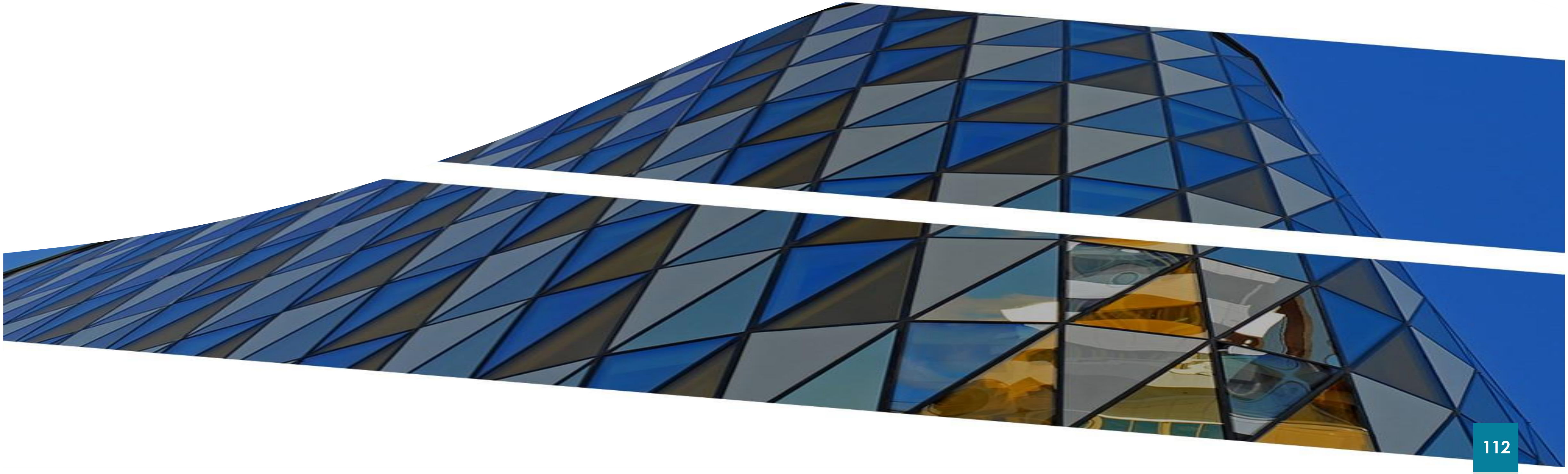
- An inline class must have a single property initialized in the primary constructor. At runtime, instances of the inline class will be represented using this single property
- Inline class properties cannot have backing fields. They can only have simple computable properties (no lateinit /delegated properties).
-

```
// For JVM backends
@JvmInline
value class Password(private val s: String)

// No actual instantiation of class 'Password' happens
// At runtime 'securePassword' contains just 'String'
val securePassword = Password("Don't try this in production")
```

Inline Classes

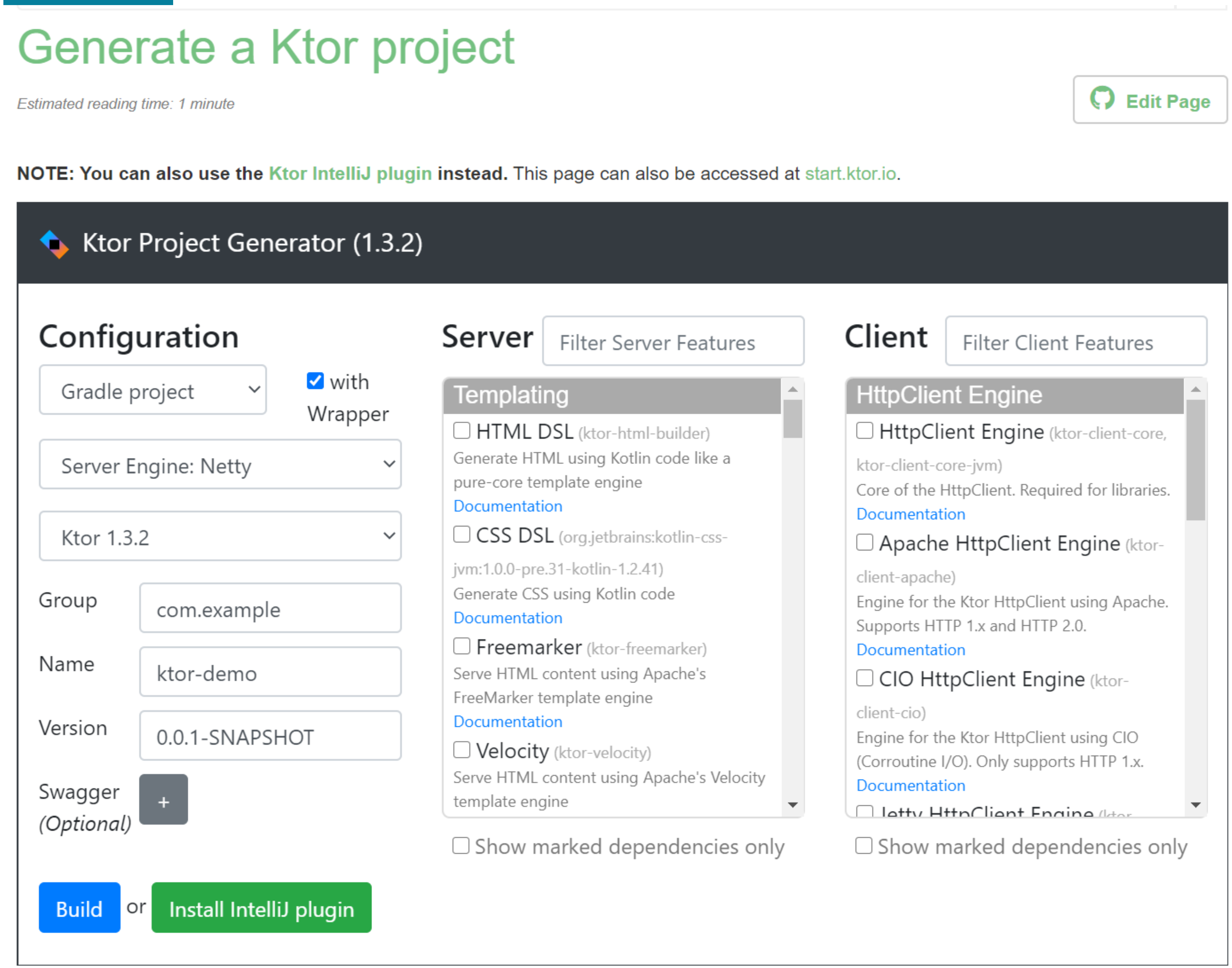
```
@JvmInline
value class Name(val s: String) {
    init {
        require(s.length > 0) { }
    }
    val length: Int
        get() = s.length

    fun greet() {
        println("Hello, $s")
    }
}
fun main() {
    val name = Name("Kotlin")
    name.greet() // method `greet` is called as a static method
    println(name.length) // property getter is called as a static method
}
```

Examples



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

```
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