

Kotlin Types

References

Kotlin In Action

- Chapter 4: Classes, Objects and Interfaces
- Chapter 6: Kotlin Type System
- Chapter 9: Generics

Kotlin Documentation

- <https://kotlinlang.org/docs/null-safety.html>
- <https://kotlinlang.org/docs/generics.html>

Kotlin Type System Overview

Java is not pure OOP.

Pure object oriented programming (OOP)

1. All data are objects with methods.

```
int x = 1;  
x.toString(); ❌
```

2. All objects are instances of concrete classes, created by constructors.

```
int[] x = int[]{1,2,3};  
x.getClassName(); ❌
```


3. Everything is invocation of objects' methods.

```
// Not method invocation  
for(int i=0; i < 10; i++) {  
    ...  
}
```

Kotlin Type System Overview

Pure object oriented programming (OOP)

1. All data are objects with methods.

```
val x = 1  
x.toString() 
```

2. All objects are instances of concrete classes, created by constructors.

3. Everything is invocation of objects' methods.

Kotlin Type System Overview

Pure object oriented programming (OOP)

1. All data are objects with methods.

2. All objects are instances of concrete classes, created by constructors.

3. Everything is invocation of objects' methods.

```
val x = arrayOf(1,2,3)
x.javaClass 
```



Kotlin Type System Overview

Pure object oriented programming (OOP)

1. All data are objects with methods.
2. All objects are instances of concrete classes, created by constructors.

3. Everything is invocation of objects' methods.


```
for(i in 1..3) {  
    println(i)  
}
```

`(1..3).javaClass`
`=> kotlin.ranges.IntRange`

`{i:Int -> i in 1..3}`
`=> Int -> Boolean`

`{i:Any -> println(i)}`
`=> Any -> Unit`

Kotlin Type System

- Concrete classes 
- Abstract classes
- Interfaces
- Functions
- Generics

Abstract Classes

```
abstract class Polygon {  
    abstract val sides: Int  
    abstract fun draw()  
}  
  
class Rectangle : Polygon() {  
    override val sides = 4  
    override fun draw() {  
        // draw the rectangle  
    }  
}
```

An abstract class is a class with *missing* implementation details.

A concrete class can be obtained from an abstract class by inheritance.

Abstract Classes

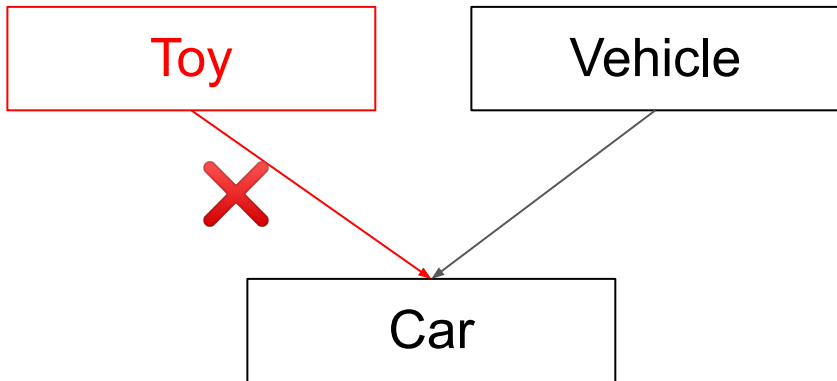
```
class Polygon {  
    var sides: Int = 0  
    fun draw() {  
        // some default polygon drawing method  
    }  
}  
  
abstract class WildShape : Polygon() {  
    abstract override fun draw()  
}
```

An abstract class can be constructed by inheriting certain members from a concrete class.

Here WildShape is an abstract class. Any subclasses of WildShape will have this.sides member, but must provide their own draw() implementation.

Inheritance From Abstract Classes

Super is unique.



```
abstract class Vehicle {  
    abstract wheels: Int  
    abstract fun drive(miles Int)  
}
```

Interfaces

- Specifications on what classes should have.
- Provides default implementations.
- Multiple interfaces can be inherited from.
- Conflict resolution.

Interface

```
interface WithWheels {  
    val numWheels: Int  
  
    fun travel(miles: Int) {  
        println("Traveled $miles on road.")  
    }  
}
```

This interface states that all WithWheels instances must have

- member numWheels
- method travel(Int)

Interface

```
interface WithWings {  
    val wingSpan: Float  
  
    fun travel(miles: Int) {  
        println("Traveled $miles in air.")  
    }  
}
```

This interface states that all WithWings instances must have

- member wingSpan
- method travel(Int)

Interface

```
class Propellor(  
    override val wingSpan: Float,  
    override val numWheels: Int,  
) : WithWheels, WithWings {  
    override fun travel(miles: Int) {  
        println("A propellor can:")  
        super<WithWheels>.travel(miles)  
        super<WithWings>.travel(miles)  
    }  
}
```

A propellor class can inherit from both interfaces.

So, it must have:

- val wingSpan
- val numWheels

It can have its own travel method, but it can also make use of the default implementations that were provided by the interfaces.

Open vs closed classes

Open class is a concrete class that can be extended by a sub-class.

Kotlin *requires* an **open** keyword to make a concrete class open.

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

```
class Terrier(...) : Dog(...)
```

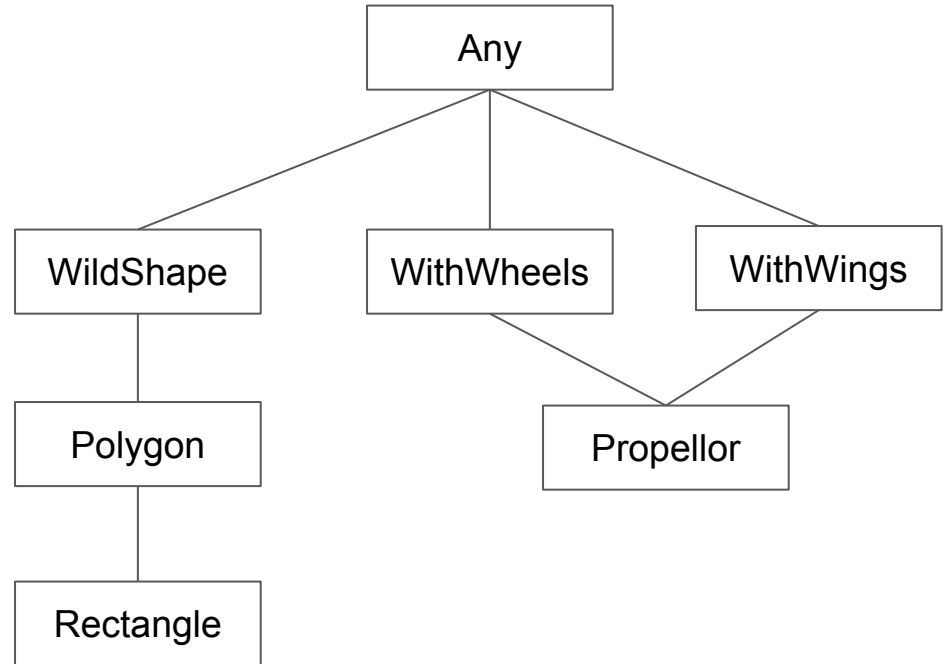
Default concrete classes are *closed*, which means that they can no longer be extended further.

Type Hierarchy: a first look

We can see that types can be organized by their IS-A relation.

Example:

- A Rectangle is a Polygon.
- A Polygon is a WildShape.
- A Propellor is a WithWheels and WithWings.
- Everything is an Any



Functions

Kotlin supports functional programming.

We can declare functions as top-level names using **fun**.

Function invocations are type checked with the requirements that:

- Arguments are instances of the parameter types
- The return value is an instance of the return type.

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

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

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

Type signature of add:

$(\text{Int}, \text{Int}) \rightarrow \text{Int}$

Function invocation:

`add(10, 20)`

Functions as values

Kotlin provides syntax to create function **values** which can be bound to symbols.

Function literals involves:

- Typed parameters
- Body as multi-line expressions; there is no **return** keyword
- The last expression is returned
- Return type is *always* inferred

```
val add: Type = function literal
```

Lvalue type

Functional literal:

```
{ x: Type, y: Type -> ... }
```

Parameter type

```
val add: (Int, Int) -> Int = {  
    x: Int, y: Int ->  
    println(x)  
    x + y  
}
```

Functions as values with type inference

Type inference allows Kotlin to *figure out* the types from the context.

- L-value type is optional if the parameter types are given.
- Parameter types are optional if L-value type is given.

```
val add: (Int, Int) -> Int = {  
    x: Int, y: Int ->  
    println(x)  
    x + y  
}
```

```
val add = {  
    x: Int, y: Int ->  
    println(x)  
    x + y  
}
```

```
val add: (Int, Int) -> Int = {  
    x, y ->  
    println(x)  
    x + y  
}
```

Nullability

Kotlin really cares about null-pointer safety.

Namely:

- Type assertion that a symbol will always bind to an object. This is called non-nullable types.
- Method invocation of non-nullable data will *never* result in `NullPointerException`.

```
// a non-nullable string
```

```
var x: String = ...  
x.length // always safe
```

```
// a nullable string
```

```
var y: String? = ...  
y.length // Compilation error ❌
```

```
// safe access of nullable value
```

```
var z: String? = ...  
z?.length // safe. Type is Int?
```

Unit

Data: `// What is the type signature of println(...)?`

- The result of a computation `(String) -> Unit`

Type:

- Description of the result of a computation

Summary

- Kotlin is a pure OOP language
 - All data are objects
 - All types are classes
- Beyond concrete classes
 - Abstract class
 - Interface
 - Nullable vs non-nullable
 - Generics (to be discussed later)
- Functions

Next lecture:

- Type parameters
- Constraints
- Covariant and contravariant generics
- Examples