# Hello, World!



Basics · Intermediate · Advanced

- Program entry point
  - `main()` function
  - `println("Hello, World!")`
- Packages and imports
  - Import syntax
  - Default imports
  - Visibility of top-level declarations
- Control flow
  - Conditions and loops
    - `if` expression
    - `for` loops and ranges
    - `while` loops
    - `when` expression
    - `break` and `continue`
  - Returns and jumps
    - `break` and `continue` labels
    - return to labels
  - Exceptions
    - `try` expression
    - `Nothing` type
    - Java interoperability
- Types
  - Variables
    - `val`
    - `var`
  - Basic types
    - Numbers
    - Booleans
    - Characters
    - Strings
    - Arrays
  - Special types
    - `Any`
    - `Unit`
    - `Nothing`
  - Type checks and casts
    - Type check operators `is` and `!is`
    - Smart casts
    - The unsafe cast operator, `as`
    - Safe "nullable" casts
    - Type erasure and generic type checks

## Code examples

### The main function

```
Hello World!
```

```
fun main() {
    val name = "World"
    println("Hello, $name!")
}
```

### Looping over a range of numbers. Using the when expression for matching conditions

```
for (n in 1≤..≤100) {
    when {
        n % (3 * 5) == 0 -> println("FizzBuzz")
        n % 3 == 0 -> println("Fizz")
        n % 5 == 0 -> println("Buzz")
        else -> println(n)
    }
}
```

### Smart casts

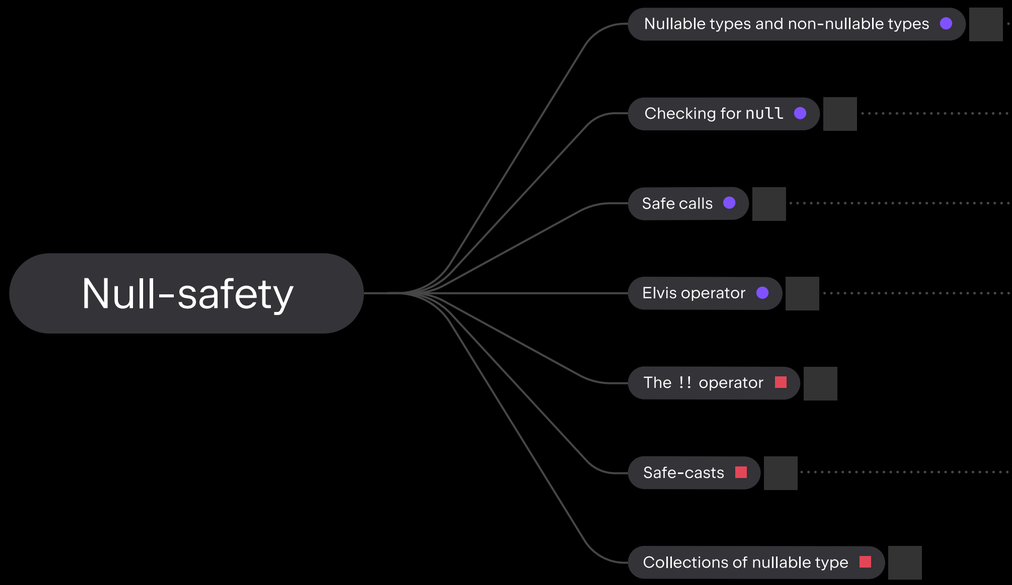
```
Variable x is automatically cast to String

fun demo(x: Any) {
    if (x is String) {
        println(x.length)
    }
}
```

Kotlin

# Null safety

Basics Intermediate Advanced






## Code examples

Nullable types annotated with '?' can be assigned to null value

```
var abc: String = "abc"
abc = null // cannot assign null
val str: String? = null // ok
```

You have to check the value of nullable type before accessing its properties

```
println(str.length) // compilation error
if (str != null) {
    println(str.length)
}

// function can return null
fun findUser(): User? {...}
val user = findUser()
val city = user?.address?.city ?: IllegalArgumentException("City is missing")
```

Safe cast with as? operator returns null on class cast failure

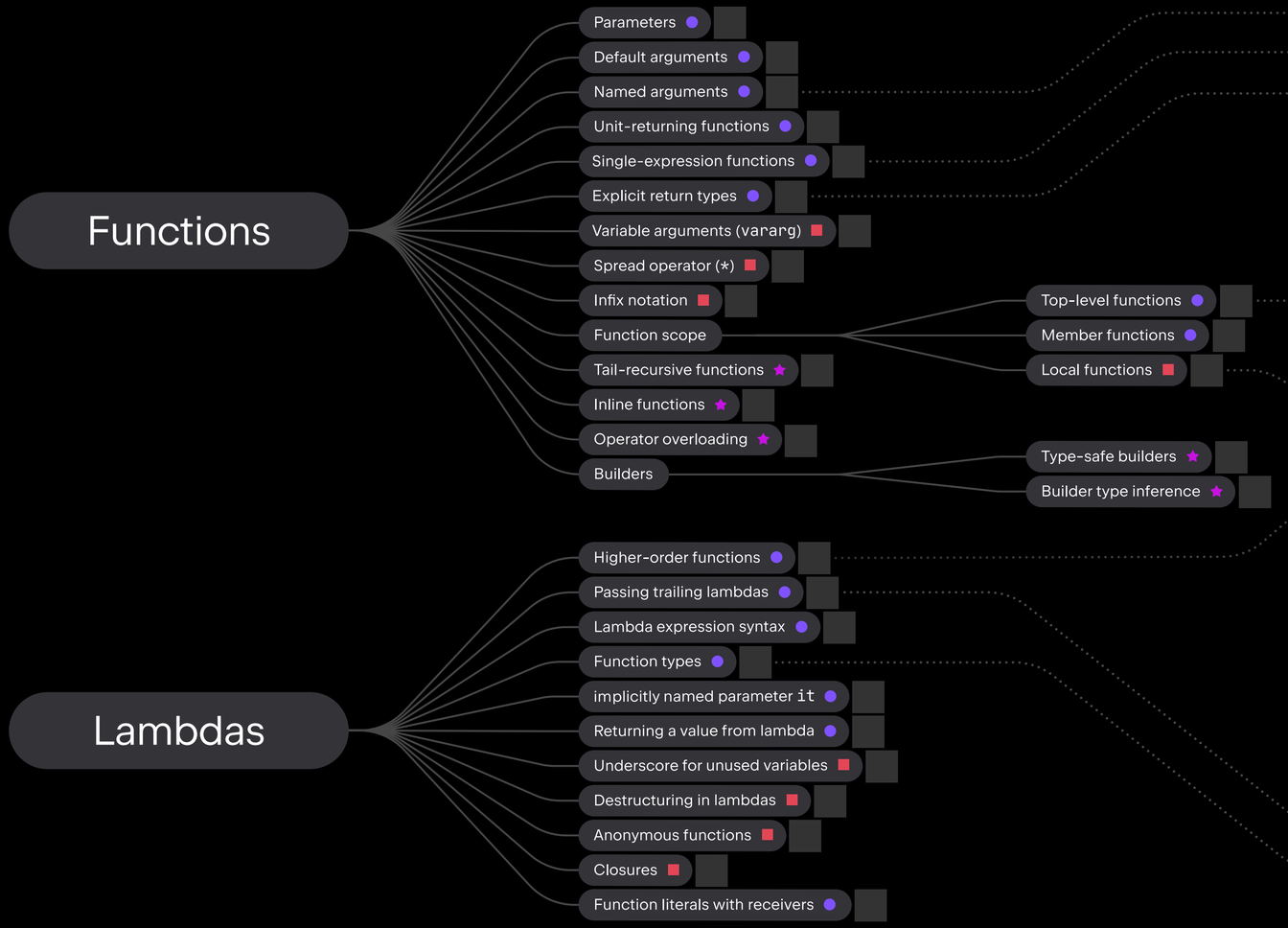
```
fun printAnyUserName(o: Any) {
    // prints null on cast failure
    println((o as? User)?.name)

    // throws exception on cast failure
    println((o as User).name)

}
```

Kotlin

# Functions & Lambdas

Basics Intermediate Advanced






## Code examples

### Lambdas and functions

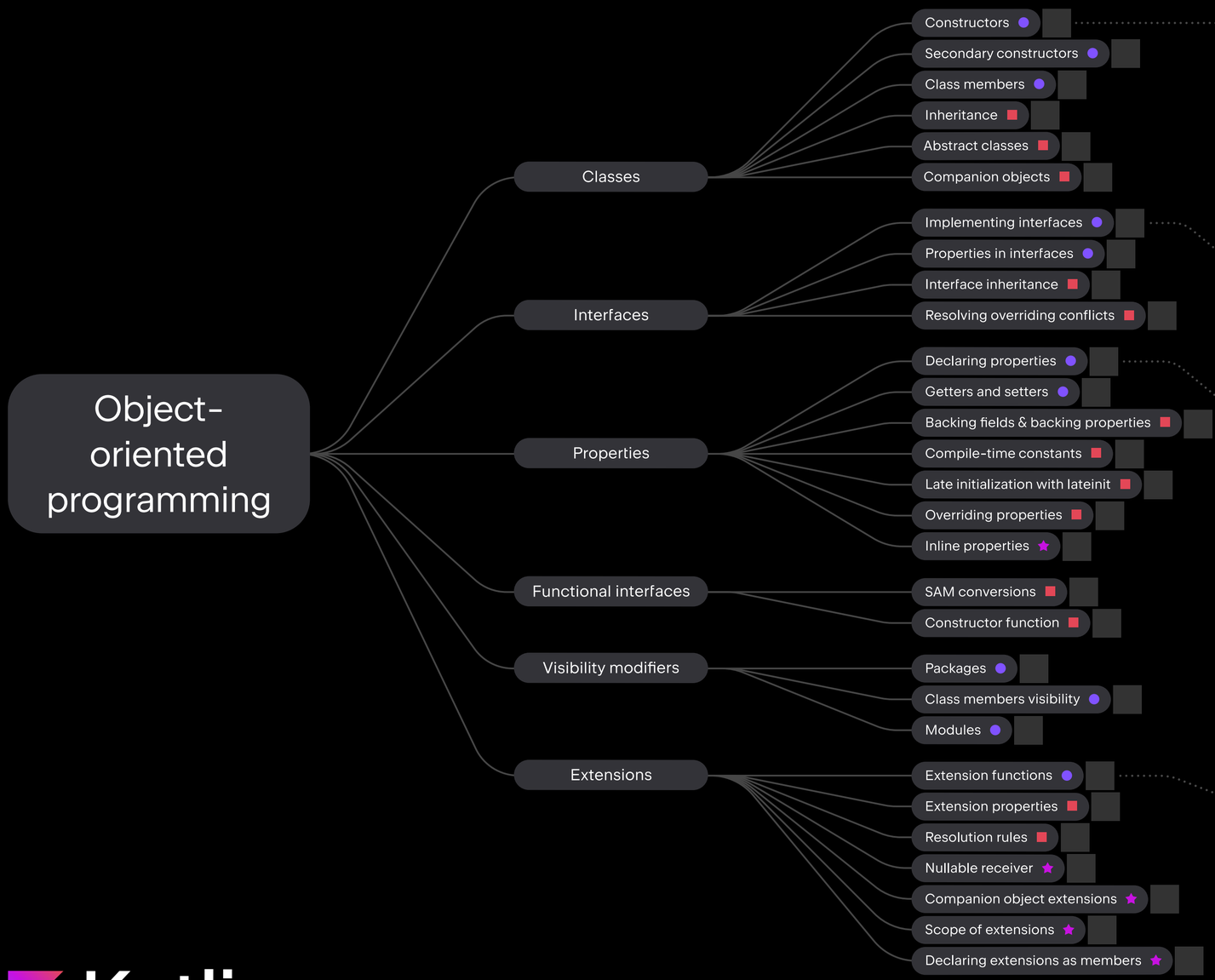
```
fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int): Int {
    return operation(x, y)
}

fun main() {
    fun add(x: Int, y: Int) = x + y

    val sum = calculate(x:4, y:5, operation = ::add)
    val mul = calculate(x:4, y:5,) { a, b -> a * b }

    println("sum=$sum, mul=$mul")
}
```

Kotlin

# Object-oriented programming. Part 1

Basics ● Intermediate ■ Advanced ★






## Code examples

### Implementing interfaces

```
interface Shape {
    fun draw()
    fun area(): Double
}

class Circle(diameter: Double) : Shape {
    val radius = diameter / 2

    override fun area(): Double = Math.PI * radius * radius
    override fun draw() {
        ...
    }
}
```

The extension function randomly changes caracters to lowercase and uppercase

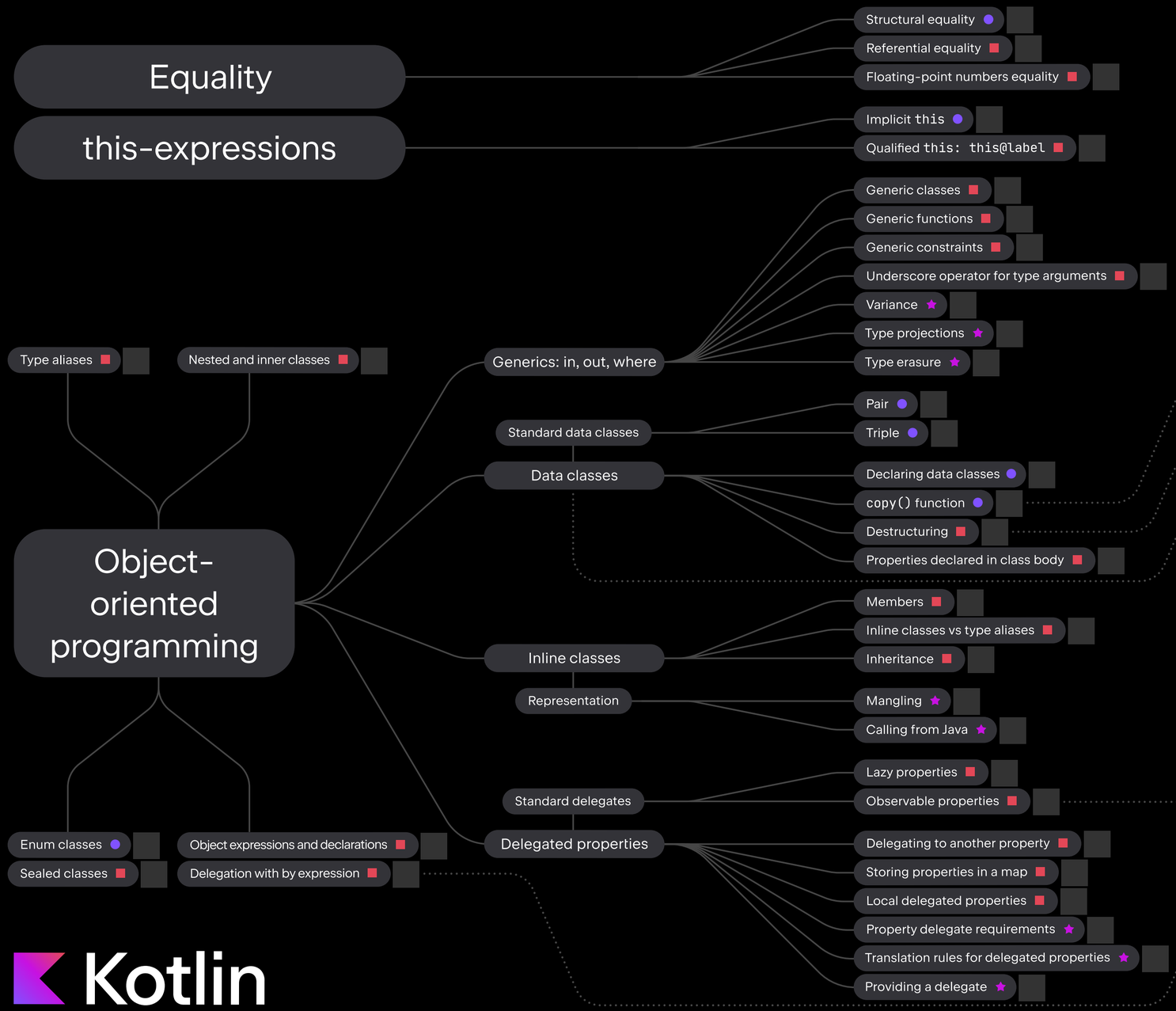
```
// "Hello, World!".sarcastic() -> HeLlO, wOrld!
fun String.sarcastic() = asIterable().joinToString(separator:"") { it: Char
    if (Random.nextBoolean()) it.uppercase() else it.lowercase()
}
```

Kotlin

# Object-oriented programming. Part 2

Basics · Intermediate · Advanced






## Code examples

The main purpose for data classes is to hold data

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

val jack = User(name="Jack", age=24)
val jill = jack.copy(name = "Jill")

val mapOfUsers = mapOf(
    1 to jack,
    2 to jill
)

val (name, age) = jack
println("$name is $age")
```

Delegated properties. Using the observable delegate to react on property value change

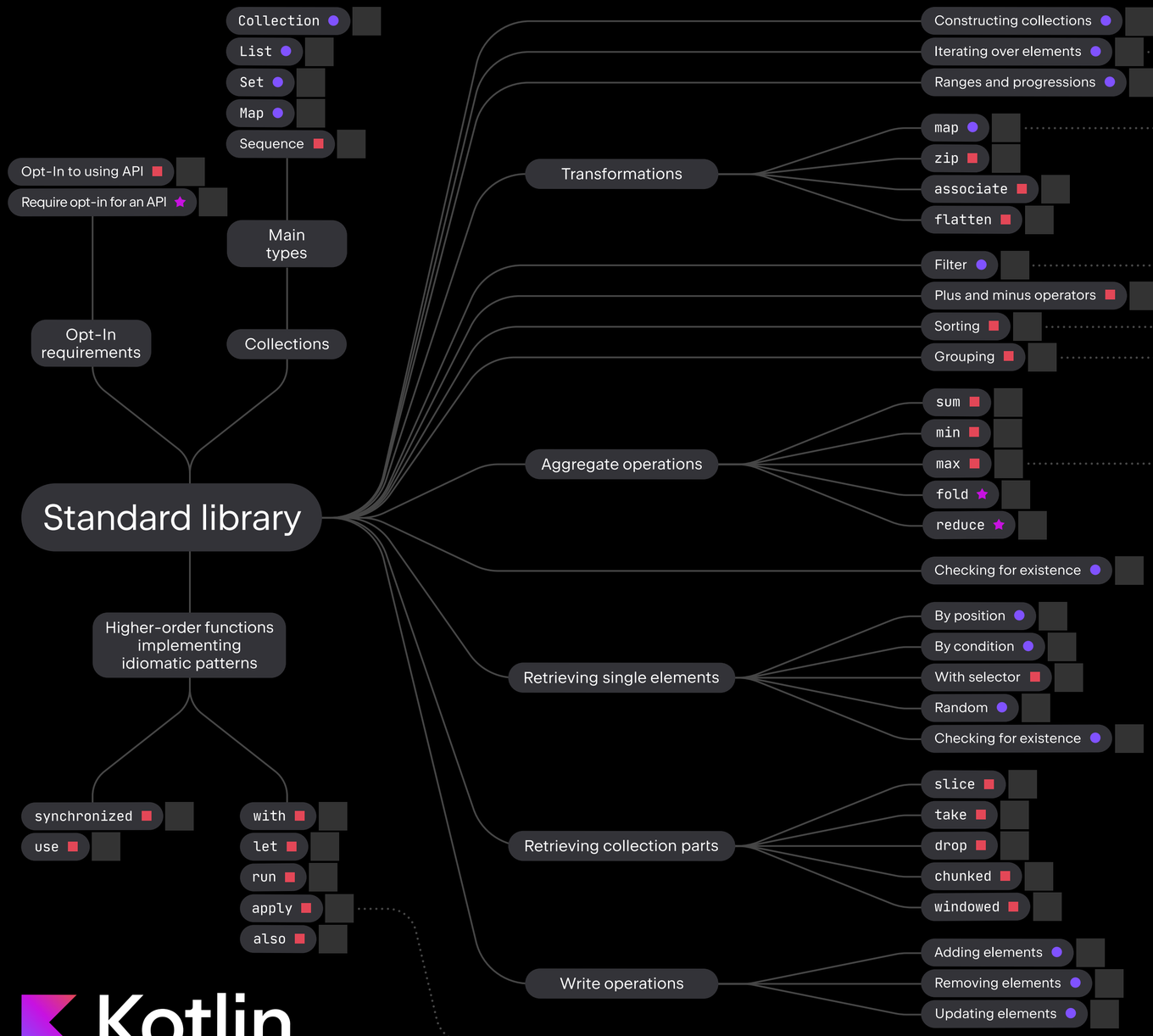
```
class User {
    var name: String by Delegates.observable(initialValue="N/A") {
        property, old, new -> println("$old -> $new")
    }
}

val user = User()
user.name = "Joe"  // N/A -> Joe
user.name = "John" // Joe -> John
```

Kotlin

# Standard library

Basics Intermediate Advanced






## Code examples

The collections library provides a number of useful functions

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

val users = listOf(
    User(name:"Jack", age:21),
    User(name:"Jill", age:22),
    User(name:"Jane", age:27),
    User(name:"Anton", age:41),
    User(name:"Leo", age:25),
)

for (user in users) {
    println(user)
}

users.filter { it.name.startsWith(prefix"J") }
users.map { it.name }

users.sortedBy { it.name.last() }
users.sortedByDescending { it.age }

users.maxBy { it.age }
users.groupBy { it.name.first() }
```

Use ‘apply’ function for grouping object initialization

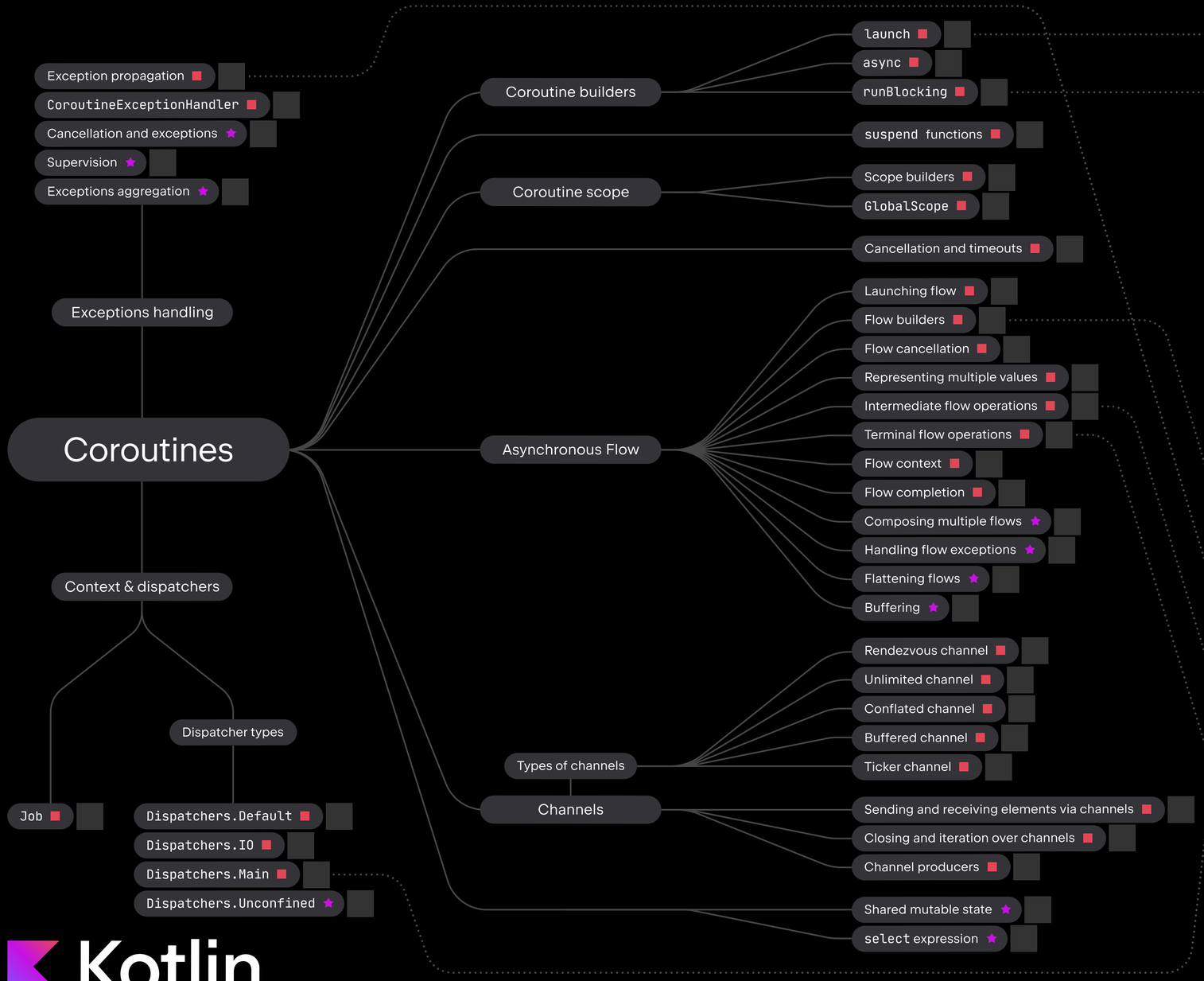
```
val dataSource = BasicDataSource().apply { this: BasicDataSource
    driverClassName = "com.mysql.jdbc.Driver"
    url = "jdbc:mysql://domain:3306/db"
    username = "username"
    password = "password"
}
```

Kotlin

# Coroutines

Basics Intermediate Advanced






Kotlin

## Code examples

### Launching a coroutine in the main thread

```
fun main() = runBlocking { this: CoroutineScope
    launch(Dispatchers.Main) { this: CoroutineScope
        doWorld()
    }
    println("Hello")
}

suspend fun doWorld(): String {
    delay(1000L)
    return "World!"
}
```

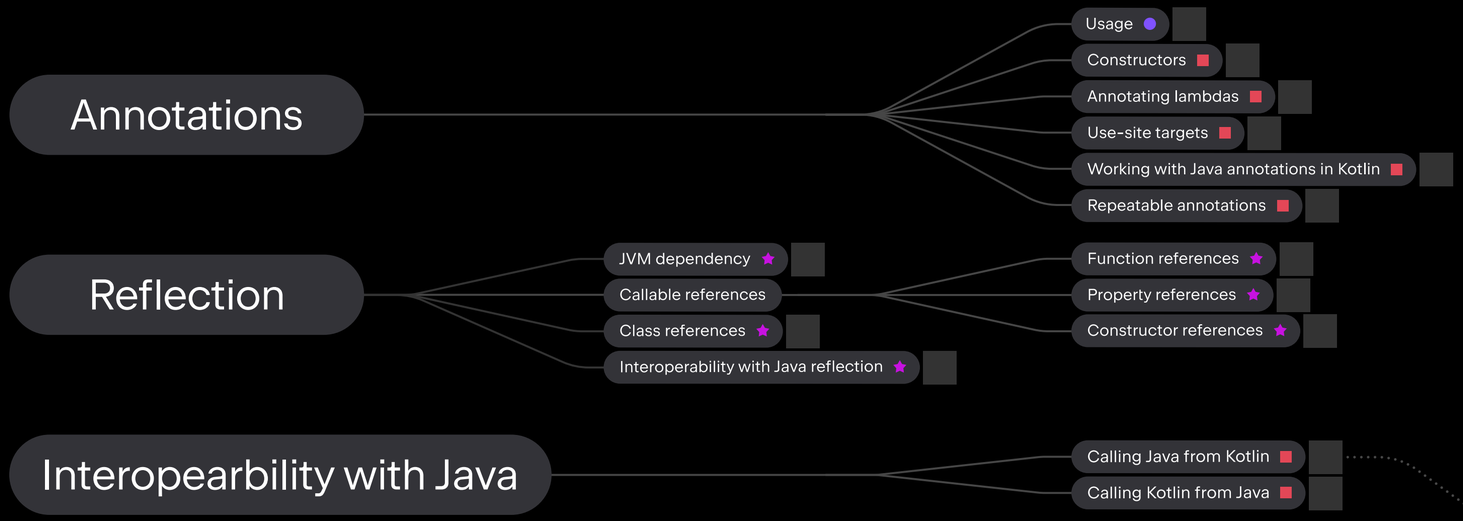
### Flow is for asynchronous processing of value streams

```
fun simple(): Flow<Int> = flow { this: FlowCollector<Int>
    for (i in 1≤..≤10) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
    throw Exception("Catch me!")
}

fun main() = runBlocking { this: CoroutineScope
    simple()
        .catch { e -> println("Caught an exception!") }
        .transform { number ->
            if (number % 2 == 0) emit(value: number * number)
            else emit(number)
        }
        .collect { value -> println(value) }
}
```

# Miscellaneous topics

Basics Intermediate Advanced






## Code examples

```java
public class JavaClass {

    String id;
    String desc;

    public JavaClass(String id, String desc) {...}

    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
    public String getDesc() { return desc; }
    public void setDesc(String desc) { this.desc = desc; }

    public void someMethod() throws IOException {...}
}
```

Calling Java from Kotlin is smooth and easy with a rules to follow

```kotlin
val jc = JavaClass(id:"1", desc:"")

// property access syntax
jc.desc = "This is Java class"
println(jc.desc)

//no checked exceptions in Kotlin
jc.someMethod()
```

Kotlin