Object-Oriented Kotlin

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction 3
- Definitions 4
- Classes & Objects 6
- Instantiation 7
- Constructors 8
- Properties 13

Principles 14
- Encapsulation 15
- Modularity 16
- Inheritance 18
- Polymorphism 23

Features 24
- Data Classes 25
- Enum Classes 26
- Sealed Classes 27

Operator Overloading 28
Extensions 29
Infix functions 30
Singleton 31
Bibliography 32

Introduction

Definitions

Kotlin is a class-based object-oriented language [1].

Class

- A set of attributes (fields, properties, data) and related methods (functions, procedures) that together represent some abstract entity.
- Attributes store state, while procedures express behavior.

Object

- An instance of a class.
- Each object has its own state.

Pseudocode

```
class Person:
  String attribute name
  Boolean attribute married
  Method greet

Person x:
  name = "Olek",
  married = false

x.greet()
```

Definitions

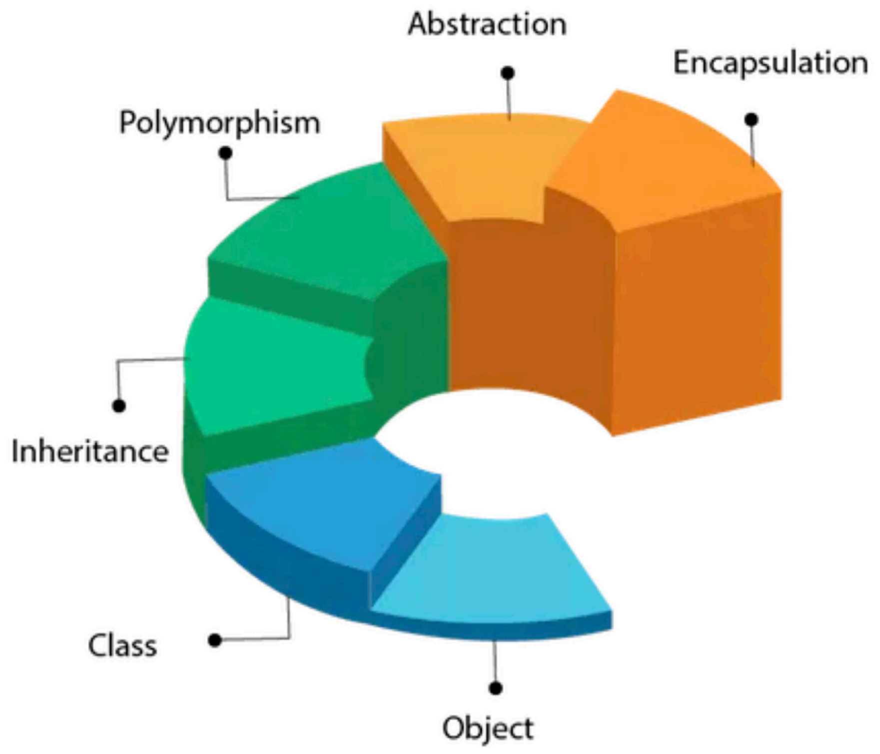


Figure 1: Kotlin is a fully-functional OO language. It resembles Java in some ways, but has significant improvments over that language.

Classes & Objects

```
class Student {
  var studentNumber: Long?
  var first: String?
  var last: String?

  fun toString(): String {
    return ("${studentNumber}: ${last.uppercase()}")
  }
}
```

This is a simple class

- no explicit inheritance,
- properties for `studentNumber`, `first` and `last` names,
- a single method `toString()`.

Instantiation

```
class Student {
  var studentNumber: Long? = null
  var first: String? = null
  var last: String? = null

  override fun toString(): String {
    return ("${studentNumber}: ${last?.uppercase()}")
  }
}

fun main() {
    val s1 = Student() // create a Student instance, 'new' not required
    s1.studentNumber = 123456
    s1.first = "Sally"
    s1.last = "Sitwell"
    println(s1) // 123456: SITWELL
}
```

Every class has a `primary constructor`, which is part of the class declaration.

source

Constructors

You can change your primary constructor to accept parameters.

```
class Student(
  studentNumber: Long?,
  first: String?,
  last: String?) {

  val _studentNumber = studentNumber
  val _first = first
  val _last = last

  override fun toString(): String {
    return ("${_studentNumber}: ${_last?.uppercase()}") // ok
  }
}

fun main() {
  val s1 = Student(123456, "Sally", "Sitwell")
    println(s1)
}
```

source

Constructors

Using `val` in front of arguments to create properties automatically!

```
class Student(
  val studentNumber: Long?,
  val first: String?,
  val last: String?) {

  override fun toString(): String {
    return ("${studentNumber}: ${last?.uppercase()}") // ok
  }
}

fun main() {
  val s1 = Student(123456, "Sally", "Sitwell")
    println(s1)
}
```

source

Constructors

Secondary Constructors

```
class Student(
  val studentNumber: Long?,
  val first: String?,
  val last: String?) {

  constructor(): this(null, null, null) // secondary has to delegate

  override fun toString(): String {
    return ("${studentNumber}: ${last?.uppercase()}") // ok
  }
}

fun main() {
    val s1 = Student(123456, "Sally", "Sitwell")
    println(s1)
}
```

source

This feels clunky. Swap the constructors around.

Constructors

```
class Student() {
  var sn: Long? = null
  var first: String? = null
  var last: String? = null

  constructor(sn: Long?, first: String?, last: String?): this() {
    this.sn = sn
    this.first = first
    this.last = last
  }

  override fun toString(): String {
    return ("${sn}: ${last?.uppercase()}") // ok
  }
}

fun main() {
    val s1 = Student(123456, "Sally", "Sitwell")
    println(s1)
}
```

source

Constructors

Init Blocks

Initialization blocks run setup code after the primary constructor. Invocation order is: 1. primary constructor, 2. init blocks in order, and 3. secondary constructor.

```
class Student(var sn: Long?, var first: String?, var last: String?) {

  init {
    first = first?.uppercase()
    last = last?.uppercase()
  }
  // ...
}

fun main() {
    val s1 = Student(123456, "sally", "sitwell")
    println(s1) // 123456: SITWELL
}
```

source

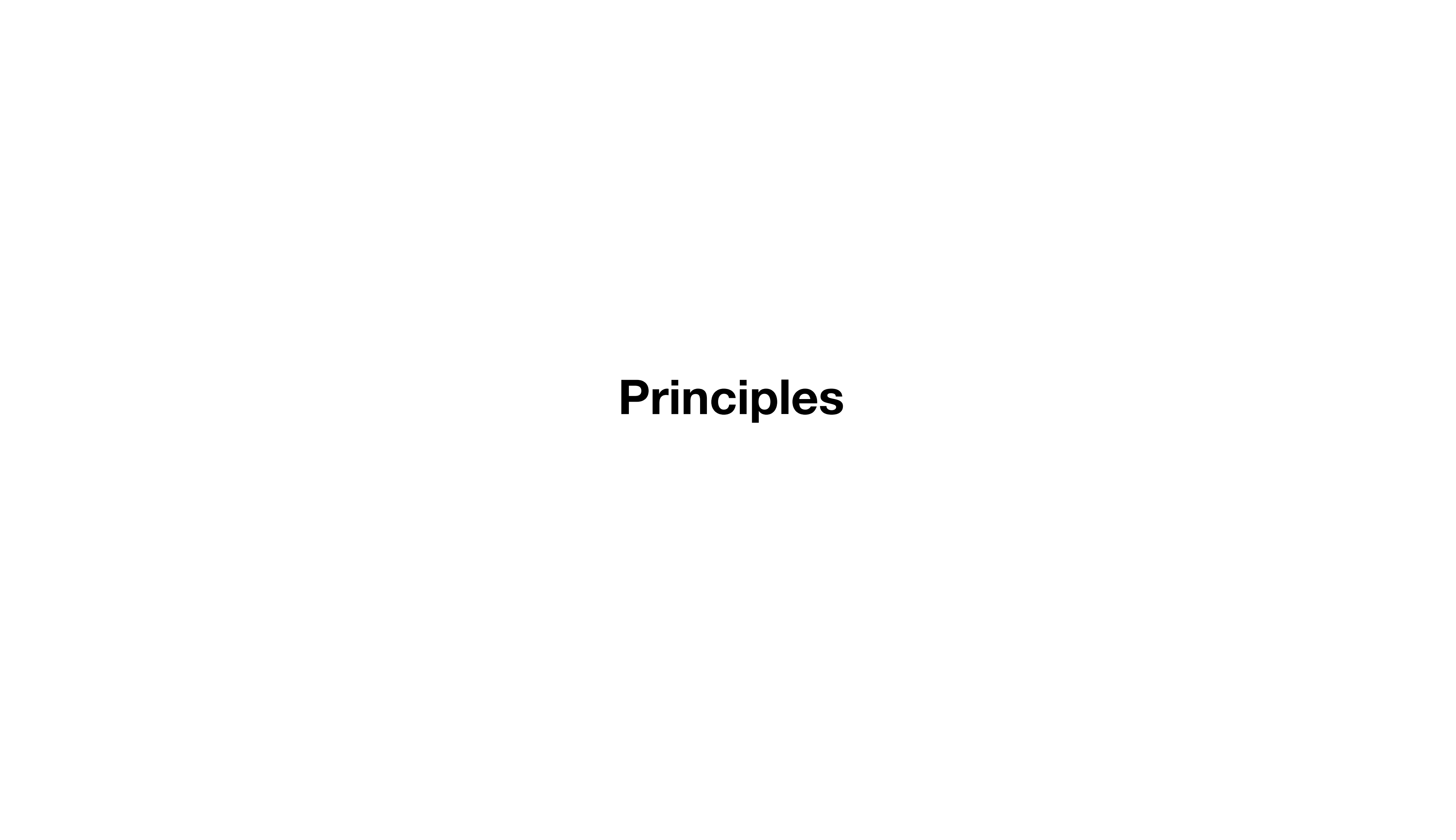
Properties

Properties are managed by each class instance.

- You *can* override `getters` and `setters` if you wish.

```kotlin
class Person {
  var age: Int = 0
    get() = field     // Custom getter (same as default)
    set(value) {      // Custom setter with validation
      if (value >= 0) {
        field = value
      } else {
        println("Age cannot be negative")
      }
    }
}

fun main() {
  val p1 = Person()
  p1.age = -5 // Age cannot be negative
}
```

[source]


Principles

Encapsulation

Abstraction: Objects are data abstractions with internal representations, along with methods to interact with those internal representations. There is no need to expose internal implementation details, so those may stay “inside” and be hidden.

Encapsulation: The option to bundle data with methods operating on said data, which also allows you to hide the implementation details from the user.

- An object is a black box. It accepts messages and replies in some way.
- Encapsulation and the interface of a class are intertwined: anything that is not part of the interface is encapsulated.
- OOP encapsulation differs from encapsulation in abstract data types.

> **Note**
>
> Encapsulation is how we achieve and enforce abstractions.

Modularity

Packages

Kotlin uses the `package` keyword to group related code into namespaces.

- Each source file should have a top-level `package` statement.

```
package ca.uwaterloo.cs346 // default package
```

If my source code was further subdivided into this folder structure, each folder would contain classes or files in their own related namespace.

```
source
├── database (ca.uwaterloo.cs346.database)
├── model (ca.uwaterloo.cs346.model)
└── user-interface (ca.uwaterloo.cs346.user-interface)
```

Use the `import` command to include classes from a different package.

```
#include ca.uwaterloo.cs346.database.DBService
#include ca.uwaterloo.cs346.model.*
```

Modularity

Access Levels

Most programming languages provide special keywords for modifying the access level of attributes and methods.

In Kotlin:

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

Kotlin defaults to `public` if not specified.

> Why use `public` by default? Kotlin design leans towards using `sensible defaults`, and public is seen as more common than other levels. However you should always use the most suitable access level.

Inheritance

Inheritance is the ability to define a new class based on an already existing one, keeping all or some of the base class functionality (state/behavior).

- The class that is being inherited from is called a base or parent class
- The new class is called a derived class, a child, or an inheritor
- The derived class fully satisfies the specification of the base class, but it may have some extended features (state/behavior)

Concept
- Specialization. Use a “general concept” and derive a “specific concept”.
- Describes an “is-a” relationship e.g., a triangle is-a shape, a car is-a-vehicle.

Motivation
- Keep shared code separate – in the base class – and reuse it.
- Type hierarchy, subtyping; incremental design.

Inheritance can reduce code duplication but often increases complexity.
- Inheritance is often redundant and can be replaced with composition.

Inheritance

Subtyping

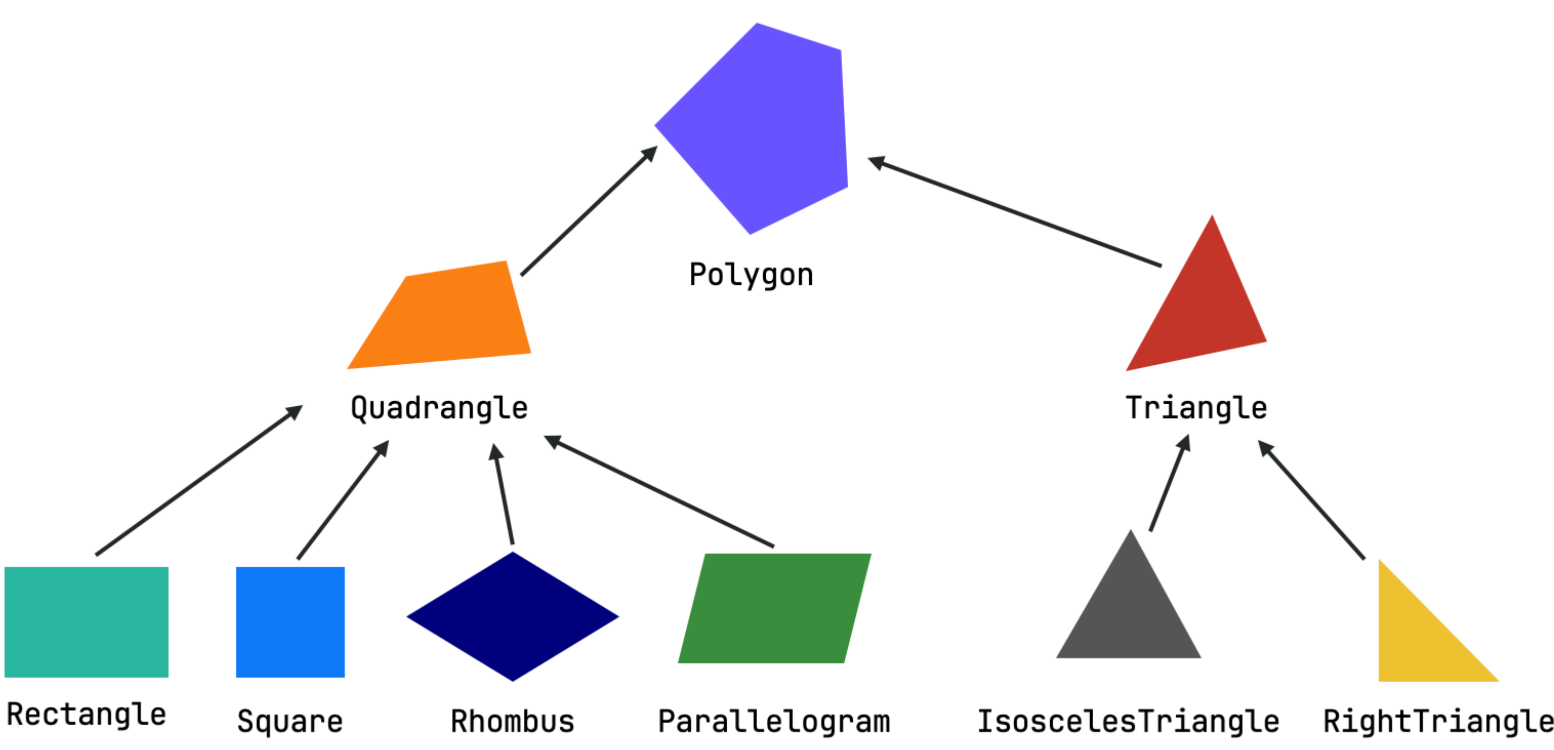


Figure 2: A single-inheritance type hierarchy. Every type can have more than one inheritor, but no more than a single base class.

Inheritance

We inherit from a base class like this:

```
abstract class Shape {
    abstract fun calculateArea(): Double
}

class Circle(val radius: Double) : Shape() {
    override fun calculateArea() = Math.PI * radius * radius
}

class Rectangle(val width: Double, val height: Double) : Shape() {
    override fun calculateArea() = width * height
}
```

We derive a subclass using a colon and list of classes/interfaces.

Inheritance

Abstract Classes

An `abstract` class is a class that cannot be instantiated directly.

- `abstract` keyword indicates that something needs to be overridden.

In our previous example, we used an abstract class as our base class. The abstract method *must* be overriden in the derived class.

```
abstract class Shape {
    abstract fun calculateArea(): Double
}
```

What if we want to use an implementation class instead?

- `open` indicates that something may be overridden (class, function, property)
- i.e. a class as `open` for inheritance; a property or function may be overridden.

```
open class Rectangle {
  open fun calculateArea(): Double {
    return width * height;
  }
}
```

Inheritance

Interfaces

Kotlin has a single inheritance model, meaning that you cannot have more than a single base class. You *can* derive from multple interfaces.

[What's an interface](#)? It's a file that contain declarations of methods.

- `interface` keyword.
- Consists of methods declarations and optional implementation code.
- Cannot store state and cannot be instantiated.

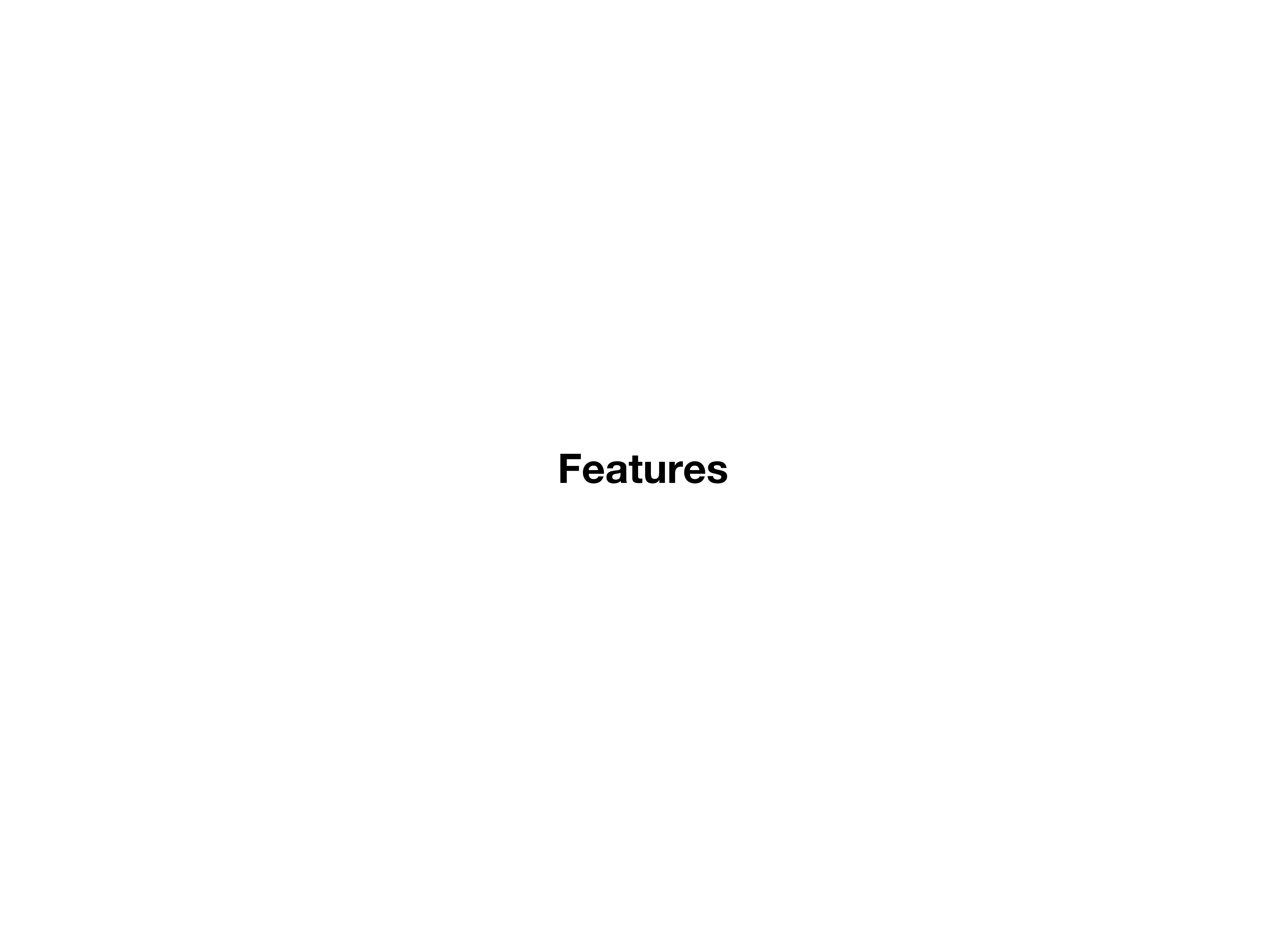
```kotlin
interface MyInterface {
    fun bar()
    fun foo() { /* optional body */ }
}

class Child : MyInterface {
    override fun bar() { /* implement interface function here */ }
}
```

Polymorphism

Polymorphism – A core OOP concept that refers to working with objects through their interfaces without knowledge about their specific types and internal structure.

- Inheritors can override and change the ancestral behavior.
- Objects can be used through their parents' interfaces.
- The client code does not know (or care) if it is working with the base class or some child class, nor does it know what exactly happens "inside".

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


Features

Data Classes

A data class is a specialized class that is meant to primarily store data. It's meant as a convenience class.

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

The compiler automatically derives:

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

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

> It's really common to have classes that just store data!

Enum Classes

`Enums` in Kotlin are managed through specialized classes.

- Each enum constant is an object.
- Each enum is an instance of the enum class.

Enum classes can have methods or even implement interfaces.

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

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

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

Sealed Classes

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

- This allows control over how and when a class can be subtyped.

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

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

Operator Overloading

You can overload named operators i.e., `+`, `-`, `*`, `/`, `++` and so on.

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus() = Point(-x, -y)

fun main() {
  val point = Point(10, 20)
  println(-point)  // prints "Point(x=-10, y=-20)"
}
```

> **Tip**
>
> Don't overuse operators and infix notation! Most of the time, they don't make a lot of sense e.g., what does it mean to increment a `book` or multiply a `repository'?

Extensions

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

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]
    this[index1] = this[index2]
    this[index2] = tmp
}
```

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

Infix functions

Use the `infix` prefix to declare an infix function. This examples extends the built-in String class.

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

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

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

Singleton

```
object DataProviderManager {
    fun registerData(provider: DataProvider) { /* code */ }
    val allDataProviders: List<DataProvider> {
        get() = // ...
    }
}

// automatically created and managed as a singleton
DataProvider.registerDataProvider(...)

for (provider in DataProvider.allDataProviders) {
  /* code */
}
```

Bibliography

[1] S. Aigner, R. Elizarov, S. Isakova, and D. Jemerov, *Kotlin in Action*, Second. New York, USA: Manning Publications, 2024. [Online]. Available: https://www.manning.com/books/kotlin-in-action-second-edition