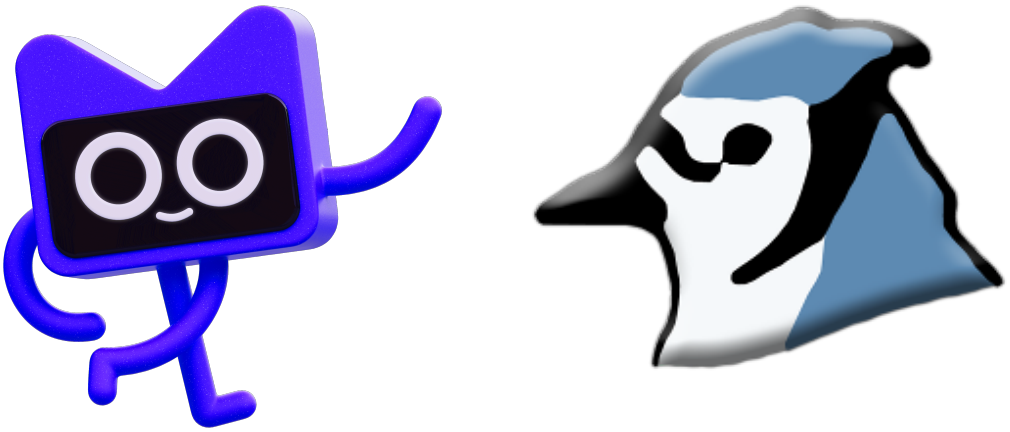


# Kotlin in BlueJ:
# Onboarding Guide

# Table of Contents

# Why Try Kotlin in Your Classroom?

Kotlin is an open-source, statically typed programming language developed by JetBrains. It runs on the JVM, interoperates with Java, and is used in production by major companies[1] for mobile, backend, web, and desktop development. Kotlin is also taught in a wide range of university CS courses, spanning everything from introductory programming to mobile development and compiler modules.[2]

Its value for an OOP classroom, though, is more specific than broad industry adoption or increasing academic adoption. In fact, Kotlin's design choices make object-oriented concepts easier to reach and harder to overlook. Just to name a few examples:

**Conciseness.** In Kotlin, a simple class with properties does not need separate field declarations, a constructor, or a body full of assignments. Students can express the idea in one line and focus on what the class actually does. Educators report that their Kotlin students understand object-oriented concepts better than their Java students do, and that existing coursework shrinks by half when switching from Java to Kotlin.[3]

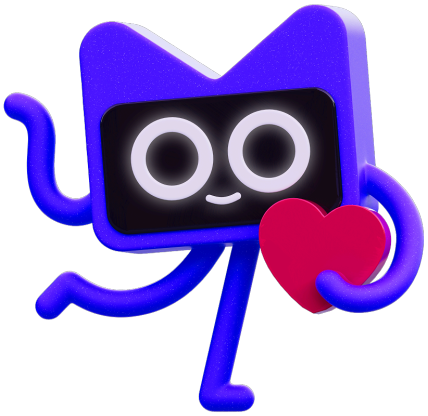
**Explicit mutability.** The distinction between `val` (read-only) and `var` (mutable) encourages students, right from the start, to think about which aspects of an object can change and which ones cannot. This design choice, which is hidden in Java, becomes clear in Kotlin.

**Null safety.** In Kotlin, null safety moves one of the most common beginner runtime crashes – the null pointer exception – to compile time. Null cases must be handled explicitly through safe calls, the Elvis operator, or null checks before accessing properties or methods. This change turns an unexpected failure into a question students must answer before their code will run.

Since BlueJ now supports Kotlin, you can teach introductory OOP in the same IDE while keeping your familiar objects-first approach. This beginner-friendly environment enables students to gain early experience with a modern language that remains relevant across both academic pathways and industry practice.

If you have any feedback or would like to share how you use Kotlin in BlueJ in your classroom, feel free to get in touch with us at education@kotlinlang.org.

Let's teach Kotlin!
Good luck, and have `fun`!



[1] https://kotlinlang.org/case-studies/
[2] https://kotlinlang.org/education/
[3] https://kotlinlang.org/education/why-teach-kotlin/

# How to Work With This Document

This document is a practical onboarding guide for BlueJ educators who already teach introductory object-oriented programming (OOP) in Java and want to start using Kotlin. It is written for teachers, not students – it shows you how the concepts you already teach map onto Kotlin, and where the language makes different choices.

The material covers the first encounter with OOP: creating and combining objects, branching, looping, encapsulation, inheritance, abstract classes, and interfaces. More advanced Kotlin features, such as generics, higher-order functions, coroutines, and functional-style collections, are not included.

**This document does not go deep into OOP theory.** It does not re-teach what inheritance is or why encapsulation matters; it shows you what the relevant Kotlin syntax looks like and where it differs from Java. The pedagogical decisions – how deeply to explain a concept, how many examples to use, and when to move on – are left up to you.

**Adjust to your classroom's pace.** The units are ordered from simple to advanced, but they can be taught at whatever pace suits your group. Within broad limits, each unit can be approached independently once the concepts from earlier units are in place. You don't have to follow the document linearly from start to finish.

The document covers eight topics, one per unit. Every unit follows the same internal structure:

- **Core concepts summary** is a quick-reference table listing every new syntax element introduced in the unit, with a short description and code example.
- **Body** walks through each concept with explanations and annotated code examples. Concepts only build on what came before or within the unit.
- **Coming from Java,** callouts appear throughout the body wherever Kotlin's design differs meaningfully from Java. For example:

> 💡 **No semicolons required.**
>
> Java requires a semicolon at the end of every statement. Kotlin does not. The compiler infers statement boundaries from line breaks, so semicolons are almost never written in practice. They are still valid syntax – `val x = 1; val y = 2` compiles – but the convention is to omit them entirely.

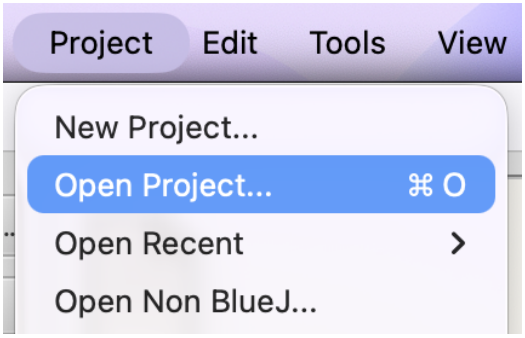
- **Further reading** closes each unit with links to the relevant sections of the official Kotlin documentation, for those who want to explore a topic in more depth.

Each unit is also accompanied by a set of BlueJ projects. The folder layout is as follows:

| Folder | Content |
|---|---|
| `onboardingExamples/` | Complete, working Kotlin files demonstrating every concept introduced in the unit – the same examples as shown in the document body. |
| `demoBasic/`<br>`demoAdvanced/` | Two self-contained BlueJ projects illustrating the unit's core concepts in a simple, focused scenario. They come with a `README.TXT` explaining the example and what to observe. |
| `practiceBasic/`<br>`practiceAdvanced/` | Two skeleton projects for student practice. Key methods are left as `TODO()` stubs; inline comments and a `README.TXT` explain exactly what each piece should do. *Available from unit 2.* |

**Downloading the materials.** The full set of unit folders is available as a `.zip` file located on Google Drive. Open the shared link, download `onboarding_materials.zip`, save it to a convenient location, and unzip it. You will find one folder per unit (`unit1/` through `unit8/`), each containing the subfolders described above.

**Opening a project in BlueJ.** In BlueJ, choose *Project → Open Project* and navigate to the folder you want to open, for example, `unit3/onboardingExamples/`. Click *Compile* to build the project, then right-click any class on the diagram to create an object or open the class in the code editor.




Open a project by selecting *Project → Open Project*.


Compile the project using the *Compile* button (located to the left of the class diagram).

---

If you find an issue with the materials, have a suggestion for improvement, or need assistance adapting a unit to your course, don't hesitate to get in touch via education@kotlinlang.org.

# 1. Objects, Classes, and State

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| `class Name(...)` | Defines a class with a primary constructor | `class Dog(val name: String, var age: Int)` |
| `val` | Read-only property – cannot be reassigned after creation | `val name: String` |
| `var` | Mutable property – can be reassigned | `var age: Int` |
| `fun` | Declares a method | `fun bark() { ... }` |
| `println(...)` | Prints a line to the console | `println("Hello!")` |
| `$variable` | Embeds a value in a string | `"Hi, $name!"` |
| `${ expression }` | Embeds an expression in a string | `"Age next year: ${age + 1}"` |
| `ClassName(args)` | Creates a new object – no `new` keyword | `Dog("Rex", 3)` |

---

If you have taught with BlueJ before, you already have the right mental model for this unit. A class is a description – a blueprint. An object is something built from that description: it exists at runtime, it holds data, and it can do things. Nothing about that changes in Kotlin.

What changes is how you write the blueprint. Here is a class in Kotlin:

```kotlin
class Dog(val name: String, var age: Int) {
    fun bark() {
        println("$name says: Woof!")
    }

    fun haveBirthday() {
        age = age + 1
        println("$name is now $age years old.")
    }
}
```

If this looks more compact than you expect from Java, it is – and the reason is mostly one feature: the *primary constructor*, which we will look at closely in a moment. But before the syntax, notice the shape. There is a class, it has data (`name` and `age`), and it has behavior (`bark`, `haveBirthday`). That is still the same object you have been teaching.

💡 **Classes are final by default.**

In Java, any class can be subclassed unless you explicitly mark it `final`. In Kotlin, the default is reversed: every class is effectively final unless you mark it `open`. This does not matter in 1. Objects, Classes, and State, where no inheritance is involved, but it is worth knowing now, so that the `open` keyword in 6. Inheritance makes sense when you encounter it. Kotlin's designers made this choice deliberately: They wanted subclassing to be something you opt *into*, not something that happens unless you remember to prevent it.

---

## Writing a class: primary constructor

In Kotlin, a class can declare its properties directly in the class header, inside parentheses after the class name. This is called the **primary constructor**.

Kotlin
```kotlin
class Dog(val name: String, var age: Int)
```

This one line does several things at once:

- It defines a class called `Dog`.
- It says that every `Dog` must be given a `name` and an `age` at the moment it is created.
- It declares `name` and `age` as **properties** of the object – stored on every `Dog` instance, accessible from outside the class.

💡 **The primary constructor replaces Java's constructor boilerplate.**

In Java, a class with two stored fields typically requires three pieces of code: a field declaration, a constructor parameter, and an assignment in the constructor body:

Java
```java
public class Dog {
    private final String name;
    private int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

The Kotlin equivalent is one line:

```kotlin
class Dog(val name: String, var age: Int)
```

> The `val`/`var` prefix in the constructor header is what makes this work – it tells Kotlin to both receive the value as a constructor argument and store it as a property. Without the prefix, the parameter exists only during construction and is not stored.

---

## Properties: val and var

Notice that `name` is declared with **`val`** and `age` with **`var`**. This distinction is one of the first things Kotlin asks you to think about, and it is worth taking seriously from the start.

- **`val`** declares a property that cannot be changed after the object is created. A dog's name, once given, stays the same.
- **`var`** declares a property that can be changed. A dog's age increases over time.

```kotlin
val rex = Dog("Rex", 3)

rex.age = 4       // fine – age is a var
rex.name = "Max"  // error – name is a val, it cannot be reassigned
```

Choosing between `val` and `var` is a design decision, not just a syntax choice. When you mark something `val`, you are saying, “this part of the object's identity does not change”. The Kotlin community strongly prefers `val` wherever possible – not because `var` is wrong, but because a property that cannot change is simpler to reason about.

---

## Methods: `fun`

Behavior in Kotlin is written with **`fun`**:

```kotlin
class Dog(val name: String, var age: Int) {
    fun bark() {
        println("$name says: Woof!")
    }

    fun haveBirthday() {
        age = age + 1
        println("$name is now $age years old.")
    }
}
```

`bark` produces output without changing anything. `haveBirthday` changes the object's state and then reports it. Both are ordinary methods: They live inside the class body, and they have direct access to the object's properties by name – no `this.` required (though you can use `this` if you ever need to be explicit).

Methods can also take parameters and return values:

```kotlin
class Dog(val name: String, var age: Int) {
    /* ... */

    fun greet(otherName: String): String {
        return "Hi $otherName, I'm $name!"
    }
}
```

The return type appears after a colon at the end of the parameter list. If a method returns nothing, you can omit the return type entirely – the `Unit` type is used for this role, but you rarely need to write it explicitly.

> 💡 **`void` in Java and `Unit` in Kotlin.**
>
> In Java, `void` is a keyword that marks a method as producing no value. Kotlin has no `void` keyword. Its role is played by `Unit` – a type with exactly one possible value, also named `Unit`. A method that produces no meaningful result implicitly returns `Unit`, and Kotlin infers the return type automatically: `fun bark(): Unit { ... }` and `fun bark() { ... }` are identical. The explicit form is rare in practice, but `Unit` appears in compiler messages and IDE output, so knowing the name helps when you encounter it.

---

## String templates

You have already seen `$name` appearing inside strings. This is Kotlin's **string template** syntax, and it is used throughout Kotlin code.

You can embed any property or variable directly in a string by prefixing it with `$`:

```kotlin
println("$name says: Woof!")
```

Wrap more complex expressions, such as arithmetic operations or method calls, in `${ }`:

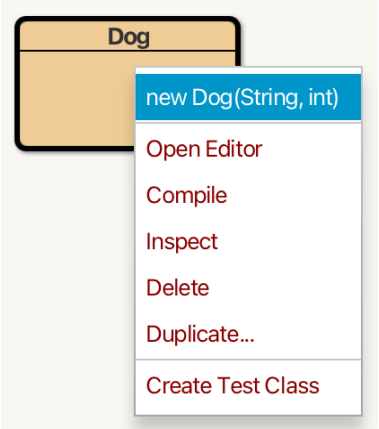
```kotlin
println("In two years, $name will be ${age + 2}.")
```

String templates are an idiomatic replacement for Java's concatenation and `String.format()`. You will find them in virtually every Kotlin program.

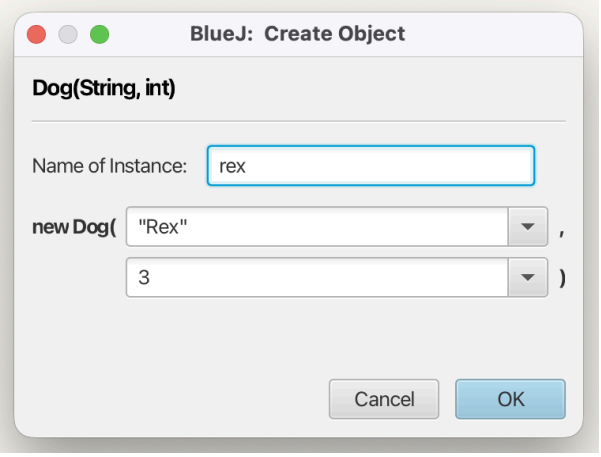
---

## Creating objects in BlueJ

In BlueJ, creating an object from a class works exactly as it always has. Right-click the class on the diagram, choose *new Dog(...)*, and supply the arguments. The arguments BlueJ prompts you for correspond directly to the primary constructor parameters. If the class header is `class Dog(val name: String, var age: Int)`, then BlueJ will ask for a `String` and an `Int`.

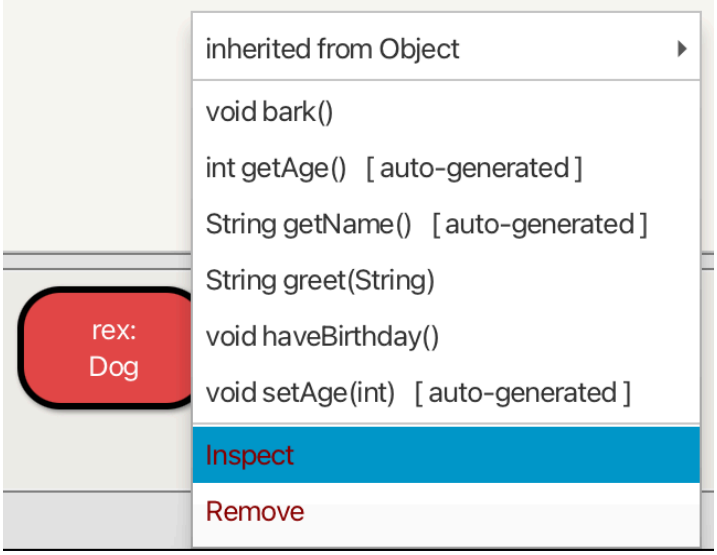



Right-click the `Dog` class and select *new Dog(String, int)* to create a new object.




Enter constructor arguments in the opened *Create Object* dialog to instantiate a `Dog` object.

The resulting object appears on the *Object Bench*, where you can right-click it to call `bark()` or `haveBirthday()` and inspect its current property values.




Kotlin compiles to **JVM bytecode** – the same format used by Java. This is the foundation of Kotlin-Java interoperability: Because both languages share the same bytecode format, Kotlin and Java code can call each other freely. It also explains two things you'll notice in the object's method menu: Java-like signatures and additional methods marked with the `[auto-generated]` tag.

💡 **Method menu.**

**Java-like signatures.** Method signatures look like Java (`void bark()`, `String greet(String)`), although the code is written in Kotlin. That's because the method menu shows the JVM's representation of your code, which uses Java's type system: Kotlin's `Unit` becomes `void`, and so on. Your code is still Kotlin – this is just how it looks from the JVM's perspective.

**Auto-generated getters and setters.** There are getters and setters marked `[auto-generated]` that you never wrote: `String getName()` for the `name` property, and `int getAge()` and `void setAge(int)` for the `age` property. For every Kotlin property, the compiler generates the corresponding get and set methods in the bytecode. This is what allows Java code to access Kotlin properties – Java doesn't have property syntax, so it needs these methods. The `[auto-generated]` tag signals that these were generated by the compiler, not written by you.

In Kotlin itself, the getters and setters generated by the compiler are not accessible, so you should always access properties directly:

```
Kotlin
rex.getName()   // unresolved reference – does not compile
rex.name        // correct in Kotlin
```

In code, the same object instantiation looks like this:

```
Kotlin
val rex = Dog("Rex", 3)
rex.bark()
```

There is no `new` keyword. In Kotlin, `Dog("Rex", 3)` is a constructor call – the class name followed by its arguments. The result is an object. It can be stored in a variable and used immediately.

---

## Each object holds its own state

This is worth dwelling on, because it is the point where the class-versus-object distinction becomes tangible.

```
Kotlin
val rex = Dog("Rex", 3)
val bella = Dog("Bella", 7)

rex.haveBirthday()

println(rex.age)    // 4
```

```
println(bella.age)  // 7 – unchanged
```

`rex` and `bella` are two separate objects. Each has its own copy of `name` and `age`. Calling `haveBirthday()` on `rex` does nothing to `bella`. The class is shared – there is only one definition of `Dog` – but every object created from it gets its own independent data.

BlueJ makes this visible in a way that is genuinely useful for teaching: You can create several `Dog` objects on the *Object Bench* and inspect them side by side. Students can watch `rex.age` increment while `bella.age` stays put.

Until students are confident that changing one object cannot affect another, they haven't yet grasped the object model. It's worth testing this understanding explicitly – ask them to predict what `bella.age` will be *before* calling `println` – because the prediction exposes whether the blueprint-versus-instance distinction has actually landed.

---

## Further reading

**Classes** – The reference for class syntax, covering primary and secondary constructors, as well as initializer blocks. The first few sections are directly relevant, while the rest apply to later units.

**Properties** – Covers `val` and `var` in depth, including custom getters and setters. For this unit, the basic declaration syntax at the top of the page is what matters. The getter and setter material becomes relevant in 5. Encapsulation.

**Functions** – The full reference for `fun`, including parameters, return types, and default argument values. It provides more details than 1. Objects, Classes, and State requires, but it will prove useful as you write more and more methods.

**String templates** – A short section explaining `$` and `${ }` syntax. The page also covers multiline strings and other string features that might come up in examples.

# 2. Objects Working Together

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| Property typed as another class | Stores a reference to another object | `val owner: Owner` |
| `owner.method()` | Calls a method on a held reference | `owner.contactDetails()` |
| `Type?` | Declares a nullable type – may hold null | `var owner: Owner?` |
| `?.` | Calls a method only if the reference is not null | `dog.owner?.name` |
| `?:` | Provides a fallback when the left side is null | `dog.owner?.name ?: "Unknown"` |

---

In 1. Objects, Classes, and State, every object stood on its own. A `Dog` had a name and an age – simple data, primitive values. Real programs are rarely so self-contained. A dog has an owner. A book has an author. An order has a customer. Objects, in practice, know about other objects.

This unit is about what that looks like in code: how an object stores a reference to another object, how it delegates work to that object, and what happens when the object on the other end of a reference is not there at all.

---

## Object that knows about another object

Suppose we want to model a dog and the person who owns it. Alongside the `Dog` class from 1. Objects, Classes, and State, we write an `Owner` class:

```kotlin
class Owner(val name: String, val phone: String) {
    fun contactDetails(): String = "$name ($phone)"
}
```

Now `Dog` can hold a reference to an `Owner`:

```kotlin
class Dog(val name: String, var age: Int, val owner: Owner) {
    fun describe() {
        println("$name's owner: ${owner.contactDetails()}")
    }
}
```

The `owner` property has the type `Owner` – not a `String` or an `Int`, but a reference to another object. Inside `describe()`, `owner.contactDetails()` is a method call on that object. When `rex.describe()` runs, execution moves into `Dog`, reaches the call to `owner.contactDetails()`, jumps into `Owner`, runs the method, returns the result, and only then does `println` fire. Tracing this chain – which object is active at each step – is a skill worth practicing with students explicitly.

Creating the two objects and connecting them:

```kotlin
val alice = Owner("Alice", "07700 900123")
val rex   = Dog("Rex", 3, alice)

rex.describe()
// Rex's owner: Alice (07700 900123)
```

`alice` is created first. Only then can it be passed into `Dog`. This is the typical pattern – construct the parts before you construct the whole.

Objects can also be passed to methods rather than stored at construction time:

```kotlin
class Dog(val name: String, var age: Int, var owner: Owner?) {
    /* ... */

    fun transferTo(newOwner: Owner) {
        owner = newOwner
        println("$name is now owned by ${newOwner.name}.")
    }
}
```

Here, `newOwner` exists only for the duration of the call and is not stored. The difference is in the declaration: A `val` or `var` in the constructor header becomes a persistent property, whereas a plain parameter in a method signature does not.

In BlueJ, this relationship is visible. Create an `Owner` on the *Object Bench*, then create a `Dog` and supply the `Owner` as the constructor argument. Inspect the resulting `Dog` object – it shows a reference to the `Owner`, not a copy of its data.

---

## Variables hold references, not objects

Before going further, there is a conceptual point worth making explicit, because it underpins everything that follows.

Suppose you write:

```kotlin
val rex = Dog("Rex", 3, alice)
```

Here, `rex` is not the dog. `rex` is a reference – an arrow that points to the `Dog` object living elsewhere in memory. The variable holds the arrow, not the thing.

This becomes visible when two variables point to the same object:

```kotlin
val rex   = Dog("Rex", 3, alice)
val myDog = rex

myDog.age = 5

println(rex.age)   // 5
```

`myDog` and `rex` are two different variables pointing to the *same* object. Changing `myDog.age` is the same as changing `rex.age` – there is only one dog.

---

## Absent object: `null`

Now that the reference model is clear, a natural question arises: What if there is no object at the end of the reference?

Consider a rescue dog in a shelter – it has a name and an age, but it has not been assigned an owner yet. Kotlin represents this with **`null`**: a reference that does not point to any object. To allow a property to hold `null`, add **`?`** after the type:

```kotlin
class Dog(val name: String, var age: Int, var owner: Owner?) {
    // ...
}

val stray = Dog("Buddy", 2, null)
```

`Owner?` means "an `Owner`, or null". Kotlin treats `Owner` and `Owner?` as distinct types – a property declared as `Owner` can never be `null`, and the compiler enforces this. If you try to call a method on an `Owner?` directly, Kotlin refuses to compile it, because there might be no object there to receive the call. This turns null errors from runtime surprises into compile-time errors.

When the stray dog finds a home, the same property can hold `Owner` instead of `null`:

```kotlin
stray.owner = Owner("Alice", "07700 900123")
```

💡 **Nullability is part of the type system.**

In Java, any object reference can be `null` at any time, and nothing in the type system marks which ones might be. `NullPointerException` is common precisely because the compiler cannot warn you in advance.

In Kotlin, nullability is encoded in the type: `Owner` can never be `null`, but `Owner?` might be. The compiler enforces this at every call site. It is one of Kotlin's most practically significant design decisions.

---

## Safe-call operator: `?.`

The safe-call operator `?.` calls a method or accesses a property only if the reference is not `null`. If it is `null`, the expression evaluates to `null` instead of failing:

```kotlin
println(stray.owner?.name)   // null
println(rex.owner?.name)     // Alice
```

---

## Elvis operator: `?:`

Safe calls produce `null` when the reference is absent. The **Elvis operator** `?:` provides a fallback value instead:

```kotlin
println(stray.owner?.name ?: "No owner")   // No owner
println(rex.owner?.name   ?: "No owner")   // Alice
```

Read `?:` as: If the left side is null, use the right side. Together, `?.` and `?:` form a pattern that handles the entire nullable case in one line:

```kotlin
println(dog.owner?.name ?: "Unknown owner")
```

---

## Further reading

**Null safety** – The official reference for nullable types, safe calls, and the Elvis operator. Covers the full picture, including the `!!` operator and `let`, which go beyond what this unit requires.

# 3. Conditions and Branching

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| `if` / `else if` / `else` | Branches on a boolean condition | `if (dog.age < 1) ... else ...` |
| `if` as an expression | Produces a value from a condition | `val s = if (dog.owner != null) "Adopted" else "Available"` |
| `when (value) { ... }` | Branches on a value, matches cases | `when (species) { "dog" -> ... }` |
| `when { ... }` | Branches on independent conditions | `when { dog.age < 1 -> ... }` |
| `when` as an expression | Produces a value from a matched branch | `val stage = when (age) { ... }` |
| `a, b -> ...` | Multiple values sharing one branch | `"dog", "puppy" -> "Kennel block"` |
| `in range ->` | Matches a value within a range | `in 3..8 -> "Adult"` |
| `!in range` | Checks if a value falls outside a range | `dog.age !in 0..20` |

---

The `Dog` class knows a dog's name and age. But knowing a value and acting on it are different things. A shelter receptionist does not just record a dog's age – they use it, for example, by treating the dog as a puppy with extra care or as a senior that needs special placement. The objects we have built so far can store information and expose it. This unit gives them the ability to make decisions based on it.

---

## `if` / `else if` / `else`

Kotlin's `if` tests a condition and runs one of two blocks depending on whether it is true or false:

Kotlin

```kotlin
if (dog.age < 1) {
    println("${dog.name} is a puppy.")
} else if (dog.age < 3) {
    println("${dog.name} is a young dog.")
} else if (dog.age < 8) {
    println("${dog.name} is an adult.")
} else {
    println("${dog.name} is a senior.")
}
```

Each `else if` adds an additional branch. Multiple branches are evaluated in sequence, and only the first matching one runs. The optional `else` branch acts as a fallback when nothing else matched.

What makes Kotlin's `if` worth a closer look is what happens when you want not just an action, but a value – the result of choosing between two things, rather than the side effect of doing one of them.

---

## `if` as an expression

In Kotlin, `if` is not only a statement – it is also an **expression**. An expression is code that produces a value. This means you can write:

```kotlin
val status = if (dog.owner != null) "Adopted" else "Available"
```

The `if` expression evaluates the condition and produces one of the two strings, which is then stored in `status`. The `else` branch is required when using `if` as an expression: Kotlin needs to know what value to produce in every case, and it cannot guarantee that without both branches:

```kotlin
val label = if (dog.age >= 2) "Adult" else "Puppy"
```

The `if` expression can also be used inside string templates and method calls:

```kotlin
println("${dog.name}: ${if (dog.owner != null) "Adopted" else "Available"}")
```

Use `if` as an expression when you are choosing between two or more values. Use it as a statement when you are choosing between two or more actions (printing, modifying state, calling a method, etc.).

---

## when: branching across multiple cases

When you have more than two or three alternatives, a chain of `if` / `else if` quickly becomes difficult to read. Kotlin's **`when`** handles this cleanly – and like `if`, it can act as either a statement or an expression.

Each `when` branch is written as `value -> body`. Multiple values can share a branch, separated by commas.

As a **statement**, each branch runs an action:

```kotlin
fun printHousingArea(species: String) {
    when (species) {
        "dog", "puppy"         -> println("Kennel block")
        "cat", "kitten"        -> println("Cat room")
        "rabbit", "guinea pig" -> println("Small animals")
        else                   -> println("Ask staff")
    }
}
```

As an **expression**, each branch produces a value, which can be returned or assigned:

```kotlin
val area = when (species) {
    "dog", "puppy"         -> "Kennel block"
    "cat", "kitten"        -> "Cat room"
    "rabbit", "guinea pig" -> "Small animals"
    else                   -> "Other area"
}
```

When `when` is used as an expression, `else` is required unless the compiler can verify that every possible case is covered. When used as a statement, `else` is optional, though including it makes the fallback case explicit.

---

## Matching ranges

`when` can match against ranges using the **`in`** keyword:

```kotlin
fun lifeStage(age: Int): String {
    return when (age) {
        0        -> "Newborn"
        in 1..2  -> "Young"
        in 3..8  -> "Adult"
        else     -> "Senior"
    }
}
```

`1..2` is a **range** – all integers from 1 to 2, inclusive. `in 1..2` checks whether the value falls within it. Ranges also work outside `when`:

- `dog.age in 1..8` is a valid boolean expression anywhere a condition is expected.
- `dog.age !in 0..20` checks whether a value falls outside a range.

---

## when without an argument

`when` can also be used without a value in the parentheses, in which case each branch provides its own full condition. This is the idiomatic way to replace a long `if` / `else if` chain when the conditions do not all test the same variable:

```kotlin
fun classify(dog: Dog): String {
    return when {
        dog.owner == null -> "Stray"
        dog.age < 1       -> "Puppy"
        dog.age < 3       -> "Young"
        else              -> "Adult with owner"
    }
}
```

Each branch is evaluated in sequence; the first one whose condition is true is selected. This form is particularly readable when the branches test different properties or combine conditions, because each branch is self-contained and states its own meaning.

---

## Further reading

**Conditions and loops** – The official reference for `if`, `when`, and loops in Kotlin. The `if` and `when` sections are directly relevant here; the `for` and `while` sections are covered in 4. Collections and Loops.

**Ranges and progressions** – Covers `..`, `until`, `downTo`, and step – the full range API. For this unit, `in a..b` is all that is needed; the rest becomes relevant if students want to iterate over ranges in specific patterns.

# 4. Collections and Loops

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| `mutableListOf(...)` | Creates a list that can be modified | `mutableListOf(rex, bella)` |
| `listOf(...)` | Creates a read-only list | `listOf("a", "b", "c")` |
| `mutableMapOf(...)` | Creates a map that can be modified | `mutableMapOf()` |
| `mapOf(...)` | Creates a read-only map | `mapOf("Rex" to rex)` |
| `list.add(x)` | Adds an element to a mutable list | `dogs.add(rex)` |
| `list.remove(x)` | Removes an element from a mutable list | `dogs.remove(rex)` |
| `map[key] = value` | Adds or updates a map entry | `registry["Rex"] = rex` |
| `map[key]` | Looks up a value by key (returns nullable) | `registry["Rex"]` |
| `.size` | Number of elements | `dogs.size` |
| `.isEmpty()` | True if the collection has no elements | `dogs.isEmpty()` |
| `.contains(x)` | True if the element is present | `dogs.contains(rex)` |
| `for (x in collection)` | Iterates over every element | `for (dog in dogs)` |
| `for (i in a..b)` | Iterates over a range of values | `for (i in 1..5)` |
| `while (condition)` | Repeats while a condition is true | `while (i < 10)` |
| `do { } while (condition)` | Runs once, then repeats while true | `do { } while (x > 0)` |

---

So far, every object we have written has held a fixed number of things: one name, one age, one owner, etc. But a shelter holds many dogs. A library holds many books. A classroom holds many students. When the number of things is not fixed in advance, you need a collection.

Kotlin's two most common collection types are **`List`** – an ordered sequence – and **`Map`** – a lookup table that pairs keys with values. This unit introduces both, along with the loops for working through their contents.

---

## `List`: ordered sequence

A `List` holds elements in order, one after another, and allows duplicates. To create a list you can add to and remove from, use **`mutableListOf`**:

```kotlin
val dogs: MutableList<Dog> = mutableListOf()
```

The type `MutableList<Dog>` is a mutable list that holds `Dog` objects. The angle brackets are Kotlin's syntax for specifying what a collection contains.

A list starts empty by default. You can add elements to it using `add`:

```kotlin
val rex   = Dog("Rex", 3, null)
val bella = Dog("Bella", 7, alice)
val buddy = Dog("Buddy", 2, null)

dogs.add(rex)
dogs.add(bella)
dogs.add(buddy)
```

You can also start with some elements already in the list:

```kotlin
val dogs = mutableListOf(rex, bella, buddy)
```

You can delete elements by using `remove` and passing the object to be removed:

```kotlin
dogs.remove(bella)   // removes the first occurrence of bella
```

A handful of other properties and methods come up constantly:

```kotlin
println(dogs.size)           // number of elements
println(dogs.isEmpty())      // true if the list has no elements
println(dogs.contains(rex))  // true if rex is in the list
println(dogs[0])             // the first element, by index
```

---

## Read-only collections

`mutableListOf` gives you a list you can change. But often, once a list is built, you do not want it to change accidentally. For this, Kotlin has **`listOf`**:

```
Kotlin
val breeds = listOf("Labrador", "Poodle", "Beagle")
```

A list created with `listOf` is **read-only**: It has no `add` or `remove` methods. You can still read from it – index into it, loop over it, or call `contains` – but you cannot modify it.

> 💡 **`List` and `MutableList` are distinct types.**
>
> In Java, `ArrayList` is always mutable – there is no language-level read-only list type (you can wrap one in `Collections.unmodifiableList`, but that is a runtime wrapper, not a type distinction). In Kotlin, `List` and `MutableList` are separate interfaces: `List` provides read-only access, while `MutableList` extends it with modification methods. This distinction is enforced at compile time. The same applies to `Map` and `MutableMap`.

---

## `Map`: looking things up by key

A `Map` associates keys with values. Instead of looking up an element by its position in a sequence, you look it up by a name, an identifier, or any other key.

This is how you can create a mutable map whose keys are `String` and whose values are `Dog` objects:

```
Kotlin
val registry: MutableMap<String, Dog> = mutableMapOf()
```

Adding an entry uses square-bracket syntax:

```
Kotlin
registry["Rex"]   = rex
registry["Bella"] = bella
```

Looking up a value works the same way:

```
Kotlin
println(registry["Rex"]?.name)   // Rex
```

Notice the `?.`. Looking up a key that does not exist returns `null`, so the value type of a map lookup is always nullable – `registry["Rex"]` has type `Dog?`, not `Dog`. This connects directly back to 2. Objects Working Together: The map tells you honestly that the key might not be there, and the compiler requires you to handle that.

> 💡 **Map lookup returns a nullable value.**
>
> In Java, `map.get(key)` returns `null` for a missing key, but the return type is not nullable – it is `Dog`, which means the compiler will not warn you that the result might be null. In Kotlin, `map[key]` returns `Dog?`, making the possibility of absence explicit and requiring you to handle it.

As with lists, a read-only version is available:

```
Kotlin
val fixed = mapOf("Rex" to rex, "Bella" to bella)
```

The `to` keyword creates a key-value pair. A `mapOf` result has no `put` or `remove`; it can only be read.

Other useful map operations:

```
Kotlin
println(registry.size)                // number of entries
println(registry.containsKey("Rex"))  // true
registry.remove("Bella")              // removes the entry
```

---

# Loops

A loop repeats a block of code – for each element in a collection, for each value in a range, or for as long as a condition remains true.

## `for`: iterating over a collection

To do something with each element in a list, use a **`for` loop**:

```
Kotlin
for (dog in dogs) {
    println(dog.name)
}
```

`dog` is a new variable, holding the current element. The loop runs once per element, in order, and stops when the list is exhausted. The same syntax works for any collection type in Kotlin, including maps:

```kotlin
for ((name, dog) in registry) {
    println("$name: age ${dog.age}")
}
```

The `for ((key, value) in map)` form unpacks each entry into its key and value, which is often the most readable way to iterate over a map.

## `for`: iterating over a range

`for` also iterates over a range of values:

```kotlin
for (i in 1..5) {
    println(i)   // prints 1 2 3 4 5
}
```

`1..5` is a range from 1 to 5, inclusive. This is often the clearest way to repeat something a fixed number of times, or to work through positions:

```kotlin
for (i in dogs.indices) {
    println("${i + 1}. ${dogs[i].name}")
}
```

`dogs.indices` is the range of valid indices for the list, from `0` to `dogs.size - 1`.

## `while`: looping on a condition

When you do not know in advance how many times to repeat an action, use a **`while` loop**:

```kotlin
var i = 0
while (i < dogs.size && dogs[i].owner != null) {
    i++
}

if (i < dogs.size) {
    println("First unclaimed dog: ${dogs[i].name}")
} else {
    println("All dogs have owners")
}
```

`while` tests its condition before each iteration. If the condition is already false when the loop starts, the body never runs. It is the right choice when the number of iterations depends on something that changes during the loop. Here, both the index bound and a per-element condition must hold simultaneously, which a `for` loop cannot express as naturally.

Kotlin also has **`do`** / **`while`**, which tests the condition *after* the body runs, guaranteeing at least one execution:

```kotlin
var day = 1
do {
    println("Performing shelter check for day $day.")
    day++
} while (day <= 7)
```

`do` / `while` is less common in practice but useful when an action must happen before the condition can meaningfully be checked. Unlike `while`, the body is guaranteed to run at least once – if `day` had started at 8, the condition would be false immediately, but the first check would still execute.

---

## Further reading

**Collections overview** – A thorough introduction to Kotlin's collection hierarchy, including the distinction between read-only and mutable interfaces. The section on collection types is directly relevant; the later sections on sequences go well beyond this unit.

**List-specific operations** – Covers the full set of list operations, including sorting and searching. Useful as a reference when students ask what else a list can do.

**Map-specific operations** – The equivalent reference for maps. The `getOrDefault` and `getOrElse` functions are particularly useful alternatives to the `?: fallback` pattern when working with maps.

**Conditions and loops** – The official reference for `for`, `while`, and `do`/`while` in Kotlin, alongside `if` and `when`. The loop sections are directly relevant here.

**Ranges and progressions** – Covers the full range API, including `downTo`, `step`, and `until`. For this unit, `in a..b` is what matters; the rest is useful background information for how to step through ranges in non-standard patterns.

# 5. Encapsulation

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| `private` | Restricts access to inside the class | `private var age: Int` |
| `private set` | Property is publicly readable but only writable from inside the class | `var age: Int = age`<br>`private set` |
| `public` | Allows access from anywhere (the default) | `val name: String` |
| `init { }` | Runs at construction time, after parameters are received | `init { require(...) }` |
| `require(cond) { msg }` | Throws an exception if the condition is false, with the given message | `require(age >= 0) { "..." }` |
| Custom getter | Computes a value on each read, stored as a property | `val isAdult: Boolean`<br>`get() = age >= 2` |

---

Previously, the `Dog` class had `age` as a publicly settable property:

```kotlin
class Dog(val name: String, var age: Int) {
    fun haveBirthday() {
        age = age + 1
        println("$name is now $age years old.")
    }
}
```

`haveBirthday()` is the natural, meaningful way to change a dog's age. But nothing stops external code from bypassing it entirely:

```kotlin
rex.age = -5
rex.age = 0
rex.age = rex.age - 3
```

Nothing in the class prevents it. A dog's age can be set to any integer, by any code, at any point. The class has no opinion about what a valid age is, and no ability to enforce one. As a program grows and more code interacts with the same objects, it becomes increasingly difficult to trace where an unexpected value came from.

Encapsulation is the practice of letting a class take responsibility for its own data – deciding what can be seen from outside, what can be changed, and under what conditions changes may occur.

## `private` and `public`

Making a property **`private`** means that only code inside the class can access or modify it. Code outside the class cannot even read it:

```kotlin
class Dog(val name: String, private var age: Int) { /* ... */ }
```

`private var age` is now the class's internal business. If an attempt is made to read or write it from outside, the compiler will deny access:

```kotlin
println(rex.age)   // compile error — age is private
rex.age = 5        // compile error — age is private
```

The class now owns its data. Meanwhile, `name` remains public – anyone can read a dog's name.

> 💡 **`public` is the default in Kotlin; `package-private` is the default in Java.**
>
> In Java, a field or method with no visibility modifier is accessible within the same package, which is known as *package-private* access. In Kotlin, the default is `public`. This means that if you declare a property without any modifier in Kotlin, it is accessible from everywhere. To restrict access, you must explicitly write `private` (or `protected` or `internal`). This is one of the few cases where Kotlin's default is *less* restrictive than Java's, not more.

Making `age` fully private is appropriate when external code should neither read nor write it directly. More commonly, a class needs to expose a value for reading while preventing external modification. Kotlin's **`private set`** modifier handles this: The property remains publicly readable, but only code inside the class can assign to it.

The property is declared in the class body rather than in the constructor header, so the `private set` modifier can be attached:

```kotlin
class Dog(val name: String, age: Int) {
    var age: Int = age
        private set

    fun haveBirthday() {
        age++
        println("$name is now $age years old.")
    }
}
```

`rex.age` can now be read from anywhere. `rex.age = 5` is a compile error – the setter is private. `haveBirthday()` is the only way to advance the dog's age – one year at a time, as intended. The class is in control of what changes, and when.

---

## Validation with `init` and `require`

Private properties give the class control over its data, but control is only useful if the class actually exercises it. The right place to enforce constraints on construction-time values is an **`init` block**. Code in `init` runs when the object is first created, after the constructor parameters have been received:

```kotlin
class Dog(val name: String, age: Int) {
    var age: Int = age
        private set

    init {
        require(name.isNotBlank()) { "Name cannot be empty" }
        require(age >= 0) { "Age cannot be negative, got $age" }
    }

    /* ... */
}
```

**`require`** checks that a condition is true. If it is not, it immediately throws an `IllegalArgumentException` with the message provided. If either `require` in `init` fails, the object is never created – the constructor call throws an exception, and the caller receives an error rather than a dog with an age of −5.

The same `require` call works inside any method body, wherever input arrives that the class needs to validate before acting on it. The pattern is always the same: State the precondition, provide a clear message, and let the method fail immediately rather than silently store, or act on, a bad value.

---

## Custom property getters

Sometimes the most useful thing to expose from a class is not a stored value but a computed one – something derived from the object's current state. Kotlin lets you define a property with a **custom getter**:

```kotlin
class Dog(val name: String, age: Int) {
    var age: Int = age
        private set
```

```
    val isAdult: Boolean
        get() = age >= 2

    val status: String
        get() = if (isAdult) "Adult" else "Puppy"

    /* ... */
}
```

`isAdult` and `status` look like properties to the caller – `rex.isAdult`, `rex.status` – but they are not stored anywhere. Each access re-runs the `get()` expression against the current `age`. The caller does not know or care that the values are computed; they just read a property. And because they depend only on `age`, they are always in sync – there is no way for them to go stale.

Custom getters are useful whenever a value is directly determined by other properties. They keep the class interface clean – one place to ask a question about the object – without storing anything redundant.

---

## Further reading

**Visibility modifiers** – The official reference for `private`, `protected`, `internal`, and `public`. For this unit, `private` and the default `public` are what matter; `protected` becomes relevant in the inheritance units.

**Properties – getters and setters** – Covers custom getters and setters in depth, including the backing field (`field`) used when a setter needs to read the current value. The getter section is directly relevant here.

**`init` blocks and constructors** – The reference for primary and secondary constructors, including `init` block ordering. For this unit, the `init` section is the relevant part.

**`require` and related functions** – The API documentation for `require`. Related functions `check` (for internal state invariants) and `error` (for unconditional failure) are worth knowing as you encounter them.

# 6. Inheritance

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| `open class` | Allows the class to be subclassed | `open class Animal(val name: String)` |
| `open fun` | Allows the method to be overridden | `open fun speak()` |
| `: Parent(args)` | Declares inheritance and calls the parent constructor | `class Dog(name: String) : Animal(name)` |
| `override fun` | Replaces the parent's version of a method | `override fun speak() { ... }` |
| `super.method()` | Calls the parent's version of a method | `super.describe()` |
| `is` | Checks whether an object is an instance of a class | `if (animal is Dog)` |
| Smart cast | After `is`, Kotlin treats the variable as the checked type | `animal.breed` inside `if (animal is Dog)` |

---

Suppose a program needs to model both dogs and cats. Each has a name, each can eat, each makes a sound. The naive approach is to write two independent classes:

Kotlin

```kotlin
class Dog(val name: String) {
    fun eat() { println("$name is eating.") }
    fun speak() { println("$name says: Woof!") }
}

class Cat(val name: String) {
    fun eat() { println("$name is eating.") }
    fun speak() { println("$name says: Meow!") }
}
```

The `eat()` method is identical in both. Any future change to it – adding a message, changing the format – must be made in two places. Add a `Bird`, and it will need to be made in three. This is the problem inheritance solves: When two or more classes share an "is-a" relationship, common behavior can be defined once in a parent class and inherited by all subclasses.

---

## open: opting in to inheritance

1. Objects, Classes, and State mentioned in passing that Kotlin classes are final by default – they cannot be subclassed unless explicitly marked otherwise. Here is where that matters.

To allow a class to be inherited from, mark it **open**:

```kotlin
open class Animal(val name: String) {
    fun eat() {
        println("$name is eating.")
    }

    open fun speak() {
        println("...")
    }
}
```

There are two things to notice here. The class itself is `open`, which allows subclassing. The `speak` method is also marked `open`, which allows subclasses to replace its behavior. The `eat` method is not `open` – it is the same for all animals and cannot be overridden.

This is intentional design: in Kotlin, inheritance and overriding are both opt-in. A class that is not `open` cannot be subclassed, and a method that is not `open` cannot be overridden. The parent class is in control of which parts of its behavior are fixed and which can be specialized.

---

## Extending a class

To inherit from `Animal`, use a colon followed by the parent class name and its constructor call:

```kotlin
class Dog(name: String, val breed: String) : Animal(name)

class Cat(name: String) : Animal(name)
```

`Dog` and `Cat` are **subclasses** of `Animal`. `Animal` is their **parent class** (also called superclass).

Notice that `Dog`'s constructor takes `name` as a plain parameter – not `val name` – and immediately passes it to `Animal(name)`. The `name` property already exists on `Animal`; writing `val name` in `Dog`'s header would create a second, shadowing property, which is not what we want. The subclass forwards the value up to the parent, where it is stored.

A `Dog` inherits everything from `Animal` automatically: the `name` property, the `eat()` method, and the default `speak()` implementation. `breed` is `Dog`-specific – it exists only on `Dog` objects.

---

## `override`: specializing behavior

The **`override`** keyword replaces the parent class's implementation of a method with the subclass's own version:

```kotlin
class Dog(name: String, val breed: String) : Animal(name) {
    override fun speak() {
        println("$name says: Woof!")
    }
}
```

`override` is required – Kotlin will not let you silently replace a parent method without it. This is another opt-in: Both the parent (with `open fun`) and the subclass (with `override fun`) must agree that replacement is happening.

> 💡 **`override` is a keyword, not an annotation.**
>
> In Java, overriding a method is implicit – you simply write a method with the same signature, and optionally annotate it with `@Override` as a documentation aid that the compiler will check. In Kotlin, `override` is mandatory. Without it, Kotlin will not allow you to provide a new implementation for an inherited method. The effect is the same, but the intent is declared explicitly on both sides: the parent marks the method `open`, the subclass marks the replacement `override`.

---

## `super`: reusing the parent's behavior

Sometimes a subclass does not want to *replace* a method entirely – it wants to *extend* it. `super` calls the parent class's version of a method from within the override:

```kotlin
open class Animal(val name: String) {
    open fun describe() {
        println("I am $name.")
    }
}

class Dog(name: String, val breed: String) : Animal(name) {
    override fun describe() {
        super.describe()
        println("I am a $breed.")
    }
}
```

```kotlin
val rex = Dog("Rex", "Labrador")
rex.describe()
// I am Rex.
// I am a Labrador.
```

`super.describe()` runs `Animal`'s version of `describe()`, then the rest of `Dog`'s `describe()` adds breed-specific information. This is the right pattern when the subclass behavior is the parent behavior plus something extra.

---

## Runtime dispatch: object decides

Here is the part of inheritance that causes the most confusion, and also the part that makes it genuinely powerful.

Consider a variable declared as type `Animal`:

```kotlin
val animal: Animal = Dog("Rex", "Labrador")
```

This is valid: a `Dog` is an `Animal`, so a `Dog` object can be assigned to an `Animal` variable. Now consider what happens when we call `speak()`:

```kotlin
animal.speak()   // Rex says: Woof!
```

The variable's type is `Animal`. But the method that runs is `Dog`'s version of `speak()`, not `Animal`'s. This is because Kotlin determines which method to run based on the **actual type of the object** at runtime – not the type of the variable holding it.

This matters most when working with a collection of mixed types:

```kotlin
val animals: List<Animal> = listOf(
    Dog("Rex", "Labrador"),
    Cat("Whiskers"),
    Dog("Buddy", "Poodle")
)

for (animal in animals) {
    animal.speak()
}
// Rex says: Woof!
// Whiskers says: Meow!
// Buddy says: Woof!
```

The loop variable `animal` has type `Animal`, but each call to `speak()` dispatches to the right subclass version automatically. You do not need to ask what type each animal is – the object handles it.

This behavior – where a method call resolves to different implementations depending on the runtime type of the object – is called **polymorphism**, and it is one of the core reasons inheritance is useful.

---

## `is` and smart casts

Sometimes you need to know the actual type of an object – to call a method that only exists in a specific subclass, for instance. Kotlin's **`is`** operator checks whether an object is an instance of a given class:

```kotlin
fun describeAnimal(animal: Animal) {
    if (animal is Dog) {
        println("${animal.name} is a ${animal.breed}.")
    } else {
        println("${animal.name} is some other animal.")
    }
}
```

Notice that inside the `if (animal is Dog)` block, `animal.breed` compiles without any cast. Kotlin knows that within that block `animal` must be a `Dog` – the `is` check has already established it – so it automatically treats `animal` as type `Dog`. This is called a **smart cast**.

Smart casts extend to `when`:

```kotlin
fun announce(animal: Animal) {
    when (animal) {
        is Dog -> println("${animal.name} the ${animal.breed} barks.")
        is Cat -> println("${animal.name} the cat meows.")
        else   -> println("${animal.name} makes a sound.")
    }
}
```

Each branch automatically gives access to the subclass-specific properties. The `when` form is usually preferable when checking against multiple types.

Use `is` checks sparingly. If a method needs to ask what type of object it is dealing with before deciding what to do, that is often a sign that the behavior should be pushed into the class hierarchy instead – an `open` method that each subclass implements differently, rather than a chain of `is` checks in external code.

---

## Further reading

**Inheritance** – The official reference for `open`, `override`, `super`, and the rules around inheritance in Kotlin. Covers secondary constructor inheritance and method overriding rules in detail.

**Type checks and casts** – Covers `is`, smart casts, and the unsafe cast operator `as`. Smart casts have a few edge cases (they do not apply to `var` properties, for instance) that are worth being aware of as you write more complex code.

**Visibility modifiers** – `protected` – visible inside the class and its subclasses – becomes relevant here. 5. Encapsulation introduced `private` and `public`; `protected` is the third modifier that matters in an inheritance hierarchy.

# 7. Abstract Classes

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| `abstract class` | Cannot be instantiated; designed to be subclassed | `abstract class Animal(val name: String)` |
| `abstract fun` | Declares a method that subclasses must implement | `abstract fun speak(): String` |
| `List<AbstractParent>` | Holds any mix of concrete subclass objects | `List<Animal>` holding `Dog` and `Cat` |

---

In 6. Inheritance, the `Animal` class had a `speak()` method with a placeholder body:

```kotlin
open class Animal(val name: String) {
    open fun speak() {
        println("...")
    }
}
```

The `"..."` exists only because Kotlin requires a method to have a body. But there is no meaningful sound a generic `Animal` makes. Every concrete subclass – `Dog`, `Cat`, every animal that actually exists – will override it. The placeholder is a lie: it pretends `Animal` can speak when it cannot.

There is a deeper problem. Nothing in this code prevents someone from instantiating `Animal` directly:

```kotlin
val a = Animal("Generic")
a.speak()   // prints: ...
```

An `Animal` with no specific type is not a meaningful object in the domain. The parent class should not be usable on its own – it should exist solely to define a structure that subclasses must fill in.

---

## `abstract class`: class that cannot be instantiated

Marking a class **`abstract`** removes the ability to create objects from it directly:

```kotlin
abstract class Animal(val name: String)
```

```kotlin
val a = Animal("Generic")   // compile error – Animal is abstract
```

An abstract class can still have properties, concrete methods, and `init` blocks. What it cannot do is produce objects. It is a blueprint for other blueprints – a description of structure that concrete subclasses complete and instantiate.

Abstract classes are implicitly `open`: there is no need to write both `abstract` and `open`. The compiler understands that a class that cannot be instantiated directly must be designed to be inherited.

---

## `abstract fun`: method without implementation

An **`abstract fun`** declares that a method must exist but provides no implementation. The subclass is required to supply it:

```kotlin
abstract class Animal(val name: String) {
    abstract fun speak(): String
}
```

`speak()` has a return type and a signature – it says what a method called `speak` must look like – but no body. Any concrete class that extends `Animal` must implement `speak()`, or the code will not compile.

```kotlin
class Dog(name: String, val breed: String) : Animal(name) {
    override fun speak(): String = "Woof!"
}

class Cat(name: String) : Animal(name) {
    override fun speak(): String = "Meow!"
}
```

`Animal` can no longer be instantiated. `Dog` and `Cat` can, because they provide a concrete implementation for every abstract method.

---

## Parent defines what, subclass defines how

Abstract methods become most useful when a parent class has concrete behavior that *depends on* an abstract method. The parent defines the overall structure of an action; each subclass fills in the part that varies.

```kotlin
abstract class Animal(val name: String) {
    abstract fun speak(): String

    fun describe() {
        println("$name says: ${speak()}")
    }
}
```

`describe()` is a concrete method in the abstract class – it is fully implemented, and it is the same for all animals. But it calls `speak()`, which is an abstract method. When `describe()` runs on a `Dog` object, it calls `Dog`'s `speak()`. When it runs on a `Cat`, it calls `Cat`'s `speak()`.

The parent defines *what* the describe sequence does: announce the animal's name and print its sound. The subclass defines *how* – what that sound is:

```kotlin
val rex = Dog("Rex", "Labrador")
rex.describe()     // Rex says: Woof!

val whiskers = Cat("Whiskers")
whiskers.describe()   // Whiskers says: Meow!
```

This pattern – concrete methods in the parent delegating to abstract methods that subclasses implement – is one of the central uses of abstract classes. It lets the parent class coordinate behavior that only subclasses can complete.

---

## Further reading

**Abstract classes** – A short section on abstract classes and methods in Kotlin, covering the rules around abstract members and their interaction with `open`.

# 8. Interfaces

*Core concepts summary:*

| Concept | What it does | Example |
|---|---|---|
| `interface` | Declares a set of methods that any implementer must provide | `interface Trainable { fun train(cmd: String) }` |
| `class Foo : MyInterface` | Implements an interface | `class Robot : Trainable` |
| `class Foo : Parent(), Interface` | Inherits and implements together | `class Dog : Animal(name), Trainable` |
| `override fun` in interface implementation | Provides the concrete implementation | `override fun train(cmd: String) { ... }` |
| Default implementation | A method body in the interface, used when not overridden | `fun introduce() { println("...") }` |
| Interface as type | Allows code to work with any implementing class | `fun train(subject: Trainable)` |

---

The `Animal` hierarchy from 6. Inheritance and 7. Abstract Classes works well within its own family. A `Dog` is an `Animal`, a `Cat` is an `Animal`, and code written for `Animal` works with both.

But suppose a program also has a `Robot` – a machine that can be trained to perform tasks. A `Robot` is not an `Animal`. Inheriting from `Animal` would be wrong, because the is-a relationship does not hold. Yet both a `Dog` and a `Robot` can be trained. The *capability* is shared even though the *identity* is not.

Abstract classes cannot model this. A class can only inherit from one parent, and both `Dog` and `Robot` cannot share a parent class unless they genuinely share a type, which they do not.

An **interface** solves this. It defines a capability – a set of methods that any class can promise to implement – without imposing a shared type identity.

---

## `interface`: contract without implementation

An interface declares what methods an object must have, without providing their bodies:

Kotlin

```kotlin
interface Trainable {
    fun train(command: String)
    fun canDo(command: String): Boolean
}
```

This says: Anything that is `Trainable` must have a `train` method and a `canDo` method. The interface makes no statement about what those methods do, what data the implementing class holds, or how the two classes relate to each other.

An interface cannot be instantiated directly – `Trainable()` is a compile error. It exists only to be implemented.

---

## Implementing interface

A class implements an interface using the same colon syntax as inheritance:

Kotlin

```kotlin
class Dog(name: String, val breed: String) : Animal(name), Trainable {
    private val learnedCommands: MutableList<String> = mutableListOf()

    override fun train(command: String) {
        learnedCommands.add(command)
        println("$name learned: $command")
    }

    override fun canDo(command: String): Boolean {
        return learnedCommands.contains(command)
    }
}
```

Kotlin

```kotlin
class Robot(val id: String) : Trainable {
    private val programmedActions: MutableList<String> = mutableListOf()

    override fun train(command: String) {
        programmedActions.add(command)
        println("Robot $id programmed with: $command")
    }

    override fun canDo(command: String): Boolean {
        return programmedActions.contains(command)
    }
}
```

`Dog` inherits from `Animal` and implements `Trainable`. `Robot` has no parent class and implements `Trainable`. Both must provide concrete implementations for every method the interface declares – the compiler will refuse to compile either class if any method is missing.

> 💡 **`implements` does not exist in Kotlin.**
>
> Java distinguishes between `extends` (for classes) and `implements` (for interfaces). Kotlin uses the same colon syntax for both: `class Foo : Parent()` and `class Foo : MyInterface` look the same, except that inheriting from a class requires parentheses (for the constructor call) while implementing an interface does not. When both appear together, the parent class comes first: `class Foo : Parent(), Interface1, Interface2`.

The `override` keyword is required here, too, just as with method overrides in inheritance.

---

## Using interface types

Once a class implements an interface, it can be referred to by the interface type. This is where the contract becomes useful.

A variable of type `Trainable` can hold any object whose class implements `Trainable`:

Kotlin

```kotlin
val rex  = Dog("Rex", "Labrador")
val unit = Robot("R2")

val trainer: Trainable = rex    // valid – Dog implements Trainable
val trainer2: Trainable = unit  // valid – Robot implements Trainable
```

More importantly, a function that takes a `Trainable` parameter works equally with any implementing class:

Kotlin

```kotlin
fun runTrainingSession(subject: Trainable, commands: List<String>) {
    for (command in commands) {
        subject.train(command)
    }
    println("Training complete.")
}

runTrainingSession(rex, listOf("sit", "fetch"))
runTrainingSession(unit, listOf("patrol", "scan"))
```

`runTrainingSession` does not know or care whether its argument is a `Dog`, a `Robot`, or any other class. It knows only that the argument is `Trainable` – and that is enough to call `train`. This is the

same runtime dispatch from 6. Inheritance and 7. Abstract Classes: The actual type of the object determines which implementation runs.

Interface types can also appear as return types:

```kotlin
fun createTrainee(type: String): Trainable {
    return if (type == "dog") Dog("Rex", "Labrador") else Robot("R1")
}
```

The caller receives a `Trainable` and can call any method the interface defines, without knowing or caring which concrete type was actually returned.

---

## Default method implementations

An interface method can have a default implementation – a body that implementing classes inherit but may override:

```kotlin
interface Trainable {
    fun train(command: String)
    fun canDo(command: String): Boolean

    fun introduce() {
        println("I am a trainable subject.")
    }
}
```

`introduce()` is not abstract – it has a body. A class that implements `Trainable` may override it, but does not have to. If it does not, the default implementation is used.

Default implementations are useful when an interface has a method with a sensible general behavior that most implementers will want but that specific classes might still want to specialize. They keep the interface useful without forcing every implementing class to repeat the same boilerplate:

```kotlin
class Dog(name: String, val breed: String) : Animal(name), Trainable {
    override fun introduce() {
        println("I am $name, a $breed.")
    }

    /* ... */
}
```

`Dog` overrides `introduce` with something specific. `Robot` does not override it, so it uses the default.

---

## Implementing multiple interfaces

A class can implement any number of interfaces. Each one is listed after the colon, separated by commas:

```kotlin
interface Taggable {
    fun tag(): String
}

class Dog(name: String, val breed: String) : Animal(name), Trainable, Taggable
{
    override fun tag(): String {
        return "$name ($breed)"
    }

    /* ... */
}
```

`Dog` now has three types simultaneously: it is an `Animal`, it is `Trainable`, and it is `Taggable`. Code written for any of those three types will accept a `Dog` object.

When a class implements multiple interfaces that both define a method with the same signature, the class must override that method and resolve the conflict explicitly – typically by calling one of the default implementations via `super<InterfaceName>.method()`, or by providing its own. The compiler will flag the ambiguity if you don't.

---

## Further reading

**Interfaces** – The official reference for interface syntax, default implementations, interface properties, and the rules around implementing multiple interfaces with conflicting defaults. The section on resolving override conflicts is worth reading before you encounter it in practice.

# What's next?

This document covers the core of introductory OOP in Kotlin. Once you and your students are comfortable with the material, the following resources provide natural next steps – from expanding the Kotlin language scope to moving into professional tools and project-based learning.

## Programming in Kotlin curriculum

JetBrains offers a **Programming in Kotlin curriculum** – a comprehensive teaching toolkit that goes beyond the scope of this document, covering functional programming, collections transformations, coroutines, and more. The course comes with slides, lecture notes, and assessment resources, and is designed to be adapted or used as-is – you can pick the modules that match your syllabus and adjust the pace to your class. This is a practical choice if you want to move from an introductory OOP to a fuller Kotlin course, or if you would like pre-built assessment material to accompany the concepts covered here.

## Practice inside IntelliJ IDEA

Students who are ready to work on larger projects can move on from BlueJ to IntelliJ IDEA, a professional IDE that is free for educators and students.

**The JetBrains Academy plugin** offers a set of Kotlin courses that run inside IntelliJ IDEA, allowing students to get immediate feedback on their code without leaving the IDE. The Kotlin Onboarding series – Introduction and Object-Oriented Programming – offers hands-on exercises building toward complete projects, and maps directly onto the topics covered in this document, making it a good choice for further practice.

Students seeking a systematic path through the language may prefer Bruce Eckel and Svetlana Isakova's book, **Atomic Kotlin**. It moves from programming basics through objects, nullability, and functional and object-oriented programming to more advanced Kotlin features. JetBrains Academy provides a companion course with in-IDE exercises for the book, so the reading and practice can follow the same sequence.

For self-paced, project-based learning, **Hyperskill** hosts Kotlin tracks that guide students through building real programs while learning the language incrementally. The tracks are graded and include automated checking.

## JetBrains tools for education

Students and teachers can access JetBrains educational tools free of charge through the Student and Teacher Packs, which include access to JetBrains IDEs and team tools.

## Stay connected

To explore more resources for learning and teaching Kotlin, and to join the Kotlin educators community, visit the Teach Computer Science With Kotlin page. You are also welcome to contact us with questions, feedback, and examples of how you use Kotlin in the classroom at **education@kotlinlang.org**.

Good luck, and have `fun` teaching Kotlin!