

# Classes & Inheritance

## Classes & Inheritance



```
class class_name class_header {  
    class variables  
    secondary constructors  
    functions (methods)  
}
```

In kotlin, classes are more concise, explicit and fine-grained than Java

# Classes

Classes in Kotlin are declared using the keyword `class`:

```
class Invoice { ... }
```

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor etc.) and the class body, surrounded by curly braces. Both the header and the body are optional; if the class has no body, curly braces can be omitted.

```
class Empty
```

# Constructors

A class in Kotlin can have a **primary constructor** and one or more **secondary constructors**. The primary constructor is part of the class header: it goes after the class name (and optional type parameters).

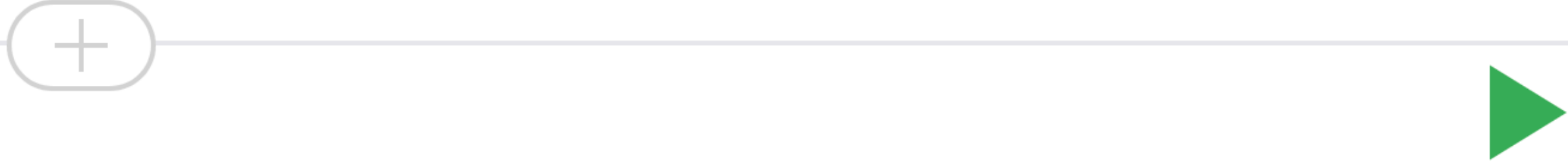
```
class Person constructor(firstName: String) { ... }
```

If the primary constructor does not have any annotations or visibility modifiers, the **constructor** keyword can be omitted:

```
class Person(firstName: String) { ... }
```

The primary constructor cannot contain any code. Initialization code can be placed in **initializer blocks**, which are prefixed with the `init` keyword.

During an instance initialization, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:



```
class InitOrderDemo(name: String) {  
    val firstProperty = "First property: $name".also(::println)  
  
    init {  
        println("First initializer block that prints ${name}")  
    }  
  
    val secondProperty = "Second property: ${name.length}".also(::println)  
  
    init {  
        println("Second initializer block that prints ${name.length}")  
    }  
}
```

Note that parameters of the primary constructor can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {  
    val customerKey = name.toUpperCase()  
}
```

In fact, for declaring properties and initializing them from the primary constructor, Kotlin has a concise syntax:

```
class Person(val firstName: String, val lastName: String, var age: Int) { ... }
```

Much the same way as regular properties, the properties declared in the primary constructor can be mutable (**var**) or read-only (**val**).



## Secondary Constructors

The class can also declare **secondary constructors**, which are prefixed with **constructor**:

```
class Person {  
    constructor(parent: Person) {  
        parent.children.add(this)  
    }  
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s).

Delegation to another constructor of the same class is done using the **this** keyword:

```
class Person(val name: String) {  
    constructor(name: String, parent: Person) : this(name) {  
        parent.children.add(this)  
    }  
}
```

## Creating instances of classes

To create an instance of a class, we call the constructor as if it were a regular function:

```
val invoice = Invoice()  
val customer = Customer("Joe Smith")
```

Note that Kotlin does not have a `new` keyword.

# Class Members

Classes can contain:

- Constructors and initializer blocks
- Functions
- Properties
- Nested and Inner Classes
- Object Declarations



# Inheritance

All classes in Kotlin have a common superclass `Any`, that is the default superclass for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

Note: `Any` is not `java.lang.Object`; in particular, it does not have any members other than `equals()`, `hashCode()` and `toString()`. Please consult the [Java interoperability](#) section for more details.

To declare an explicit supertype, we place the type after a colon in the class header:

```
open class Base(p: Int)  
class Derived(p: Int) : Base(p)
```

If the derived class has a primary constructor, the base class can (and must) be initialized right there, using the parameters of the primary constructor.

If the class has no primary constructor, then each secondary constructor has to initialize the base type using the `super` keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type:

```
class MyView : View {  
    constructor(ctx: Context) : super(ctx)  
  
    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)  
}
```

# Overriding Methods

As we mentioned before, we stick to making things explicit in Kotlin. And unlike Java, Kotlin requires explicit annotations for overridable members (we call them *open*) and for overrides:

```
open class Base {  
    open fun v() { ... }  
    fun nv() { ... }  
}  
class Derived() : Base() {  
    override fun v() { ... }  
}
```

The **override** annotation is required for `Derived.v()`. If it were missing, the compiler would complain. If there is no **open** annotation on a function, like `Base.nv()`, declaring a method with the same signature in a subclass is illegal, either with **override** or without it. In a final class (e.g. a class with no **open** annotation), open members are prohibited.

A member marked **override** is itself open, i.e. it may be overridden in subclasses. If you want to prohibit re-overriding, use **final**:

```
open class AnotherDerived() : Base() {  
    final override fun v() { ... }  
}
```



# Overriding Properties

Overriding properties works in a similar way to overriding methods; properties declared on a superclass that are then redeclared on a derived class must be prefaced with **override**, and they must have a compatible type. Each declared property can be overridden by a property with an initializer or by a property with a getter method.

```
open class Foo {  
    open val x: Int get() { ... }  
}  
  
class Bar1 : Foo() {  
    override val x: Int = ...  
}
```



You can also override a `val` property with a `var` property, but not vice versa. This is allowed because a `val` property essentially declares a getter method, and overriding it as a `var` additionally declares a setter method in the derived class.

Note that you can use the `override` keyword as part of the property declaration in a primary constructor.

```
interface Foo {  
    val count: Int  
}  
  
class Bar1(override val count: Int) : Foo  
  
class Bar2 : Foo {  
    override var count: Int = 0  
}
```

# Calling the superclass implementation

Code in a derived class can call its superclass functions and property accessors implementations using the **super** keyword:

```
open class Foo {
    open fun f() { println("Foo.f()") }
    open val x: Int get() = 1
}

class Bar : Foo() {
    override fun f() {
        super.f()
        println("Bar.f()")
    }

    override val x: Int get() = super.x + 1
}
```

# Abstract Classes

A class and some of its members may be declared **abstract**. An abstract member does not have an implementation in its class. Note that we do not need to annotate an abstract class or function with `open` – it goes without saying.

We can override a non-abstract open member with an abstract one

```
open class Base {  
    open fun f() {}  
}  
  
abstract class Derived : Base() {  
    override abstract fun f()  
}
```