**Chapter 6. Abstract Classes And Interface**

• An abstract class can’t be instantiated. It can contain both abstract and non-abstract properties and functions.

• Any class that contains an abstract property or function must be declared abstract.

• A class that’s not abstract is called concrete.

• You implement abstract properties and functions by overriding them.

• All abstract properties and functions must be overridden in any concrete subclasses.

• An interface lets you define common behavior outside a superclass hierarchy so that independent classes can still benefit from polymorphism.

• Interfaces can have abstract or non-abstract functions.

• Interfaces properties can be abstract, or they can have getters and setters. They can’t be initialized, and they don’t have access to a backing field.

• A class can implement multiple interfaces.

• If a subclass inherits from a superclass (or implements an interface) named A, you can use the code:

super<A>.myFunction

to call the implementation of myFunction that’s defined in A.

• If a variable holds a reference to an object, you can use the is operator to check the type of the underlying object.

• The is operator performs a smart cast when the compiler can guarantee that the underlying object can’t have changed between the type check and its usage.

• The as operator lets you perform an explicit cast.

• A when expression lets you compare a variable against an exhaustive set of different options.

**Chapter 7. Data Classes**

• The behavior of the == operator is determined by the implementation of the equalsfunction.

• Every class inherits an equals, hashCode and toString function from the Any class because every class is a subclass of Any. These functions can be overridden.

• The equals function tells you if two objects are considered “equal”. By default, it returns true if it’s used to test the same underlying object, and false if it’s used to test separate objects.  
• The === operator lets you check whether two variables refer to the same underlying object  
irrespective of the object’s type.

• A data class lets you create objects whose main purpose is to store data. It automatically overrides the equals, hashCode and toString functions, and includes copy and componentN functions.  
• The data class equals function checks for equality by looking at each object’s property values. If two data objects hold the same data, the equals function returns true.

• The copy function lets you create a new copy of a data object, altering some of its properties. The original object remains intact.

• componentN functions let you destructure data objects into their component property values.

• A data class generates its functions by considering the properties defined in its primary constructor.

• Constructors and functions can have default parameter values. You can call a constructor or function by passing parameter values in order of declaration or by using named arguments.

• Classes can have secondary constructors.

• An overloaded function is a different function that happens to have the same function name. An overloaded function must have different arguments, but may have a different return type.

RULES FOR DATA CLASSES

\* There must be a primary constructor.

\* The primary constructor must define one or more parameters.

\* Each parameter must be marked as val or var.

\* Data classes must not be open or abstract.

**Chapter 8. nulls and exceptions**

• null is a value that means a variable doesn’t hold a reference to an object. The variable exists, but it doesn’t refer to anything.

• A nullable type can hold null values in addition to its base type. You define a type as nullable by adding a ? to the end of it.

• To access a nullable variable’s properties and functions, you must first check that it’s not null.

• If the compiler can’t guarantee that a variable is not null in between a null-check and its usage, you must access properties and functions using the safe call operator (?.).

• You can chain safe calls together.

• To execute code if (and only if) a value is not null, use ?.let.

• The Elvis operator (?:) is a safe alternative to an if expression.

• The not-null assertion operator (!!) throws a NullPointerException if the subject of your  
assertion is null.

• An exception is a warning that occurs in exceptional situations. It’s an object of type Exception.

• Use throw to throw an exception.

• Catch an exception using try/catch/finally.

• try and throw are expressions.

• Use a safe cast (as?) to avoid getting a ClassCastException.

**Chapter 9. collections**

• Create an array initialized with null values using the arrayOfNulls function.

• Useful array functions include: sort, reverse, contains, min, max, sum, average.

• The Kotlin Standard Library contains pre-built classes and functions grouped into packages.

• A List is a collection that knows and cares about index position. It can contain duplicate values.

• A Set is an unordered collection that doesn’t allow duplicate values.

• A Map is a collection that uses key/value pairs. It can contain duplicate values, but not duplicate keys.

• List, Set and Map are immutable. MutableList, MutableSet and MutableMap are mutable  
subtypes of these collections.

• Create a List using the listOf function.

• Create a MutableList using mutableListOf.

• Create a Set using the setOf function.

• Create a MutableSet using mutableSetOf.

• A Set checks for duplicates by first looking for matching hash code values. It then uses  
the === and == operators to check for referential or object equality.

• Create a Map using the mapOf function, passing in key/value pairs.

• Create a MutableMap using mutableMapOf.

**Chapter 10. Generics**

• Generics let you write consistent code that’s type-safe. Collections such as MutableList use generics.

• The generic type is defined inside angle brackets <>, for example:

class Contest<T>

• You can restrict the generic type to a specific supertype, for example:

class Contest<T: Pet>

• You create an instance of a class with a generic type by specifying the “real” type in angle brackets, for example:

Contest<Cat>

• Where possible, the compiler will infer the generic type.

• You can define a function that uses a generic type outside a class declaration, or one that uses a different generic type, for example:

fun <T> listPet(): List<T>{

...  
}

• A generic type is invariant if it can only accept references of that specific type. Generic types are invariant by default.

• A generic type is covariant if you can use a subtype in place of a supertype. You specify that a type is covariant by prefixing it with out.

• A generic type is contravariant if you can use a supertype in place of a subtype. You specify that a type is contravariant by prefixing it with in.

**Chapter 11. Lambdas and Higher-order functions**

• A lambda expression, or lambda, takes the form:

{ x: Int -> x + 5 }

The lambda is defined within curly braces, and can include parameters, and a body.

• A lambda can have multiple lines. The last evaluated expression in the body is used as the lambda’s return value.

• You can assign a lambda to a variable. The variable’s type must be compatible with the type of the lambda.

• A lambda’s type has the format:

(parameters) -> return\_type

• Where possible, the compiler can infer the lambda’s parameter types.

• If the lambda has a single parameter, you can replace it with it.

• You execute a lambda by invoking it. You do this by passing the lambda any parameters in  
parentheses, or by calling its invoke function.

• You can pass a lambda to a function as a parameter, or use one as a function’s return value. A function that uses a lambda in this way is known as a higher-order function.

• If the final parameter of a function is a lambda, you can move the lambda outside the function’s parentheses when you call the function.

• If a function has a single parameter that’s a lambda, you can omit the parentheses when you call the function.

• A type alias lets you provide an alternative name for an existing type. You define a type alias using typealias.

**Chapter 12. built-in higher-order functions**

• Use minBy and maxBy to find the lowest or highest value in a collection. These functions take one parameter, a lambda whose body specifies the function criteria. The return type matches the type of items in the collection.

• Use sumBy or sumByDouble to return the sum of items in a collection. Its parameter, a lambda, specifies the thing you want to sum. If this is an Int, use sumBy, and if it’s a Double, use sumByDouble.

• The filter function lets you search, or filter, a collection according to some criteria. You specify this criteria using a lambda, whose lambda body must return a Boolean. filterusually returns a List . If the function is being used with a Map, however, it returns a Mapinstead.

• The map function transforms the items in a collection according to some criteria that you specify using a lambda. It returns a List.

• forEach works like a for loop. It allows you to perform one or more actions for each item in a collection.

• Use groupBy to divide a collection into groups. It takes one parameter, a lambda, which defines how the function should group the items. The function returns a Map, which uses the lambda criteria for the keys, and a List for each value.

• The fold function lets you specify an initial value, and perform some operation for each item in a collection. It takes two parameters: the initial value and a lambda that specifies the operation you want to perform.