Contents

[Kotlin 2](#_Toc526571320)

[Constructors 2](#_Toc526571321)

[Properties and Fields 2](#_Toc526571322)

[Getters and Setters 2](#_Toc526571323)

[Enums 2](#_Toc526571324)

[Data Classes 2](#_Toc526571325)

[Collections: List, Set, Map 3](#_Toc526571326)

[No Multimap in Kotlin 3](#_Toc526571327)

[Ranges 3](#_Toc526571328)

[Map 3](#_Toc526571329)

[Extending Data Classes 3](#_Toc526571330)

[Transform method on Set 3](#_Toc526571331)

[Assertions with Java 3](#_Toc526571332)

# Kotlin

## 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 Card constructor(rank: Rank, suit: Suit) : Comparable<Card> {  
  
 private val **rank** = rank;  
 private val **suit** = suit;

class Card constructor(private val **rank**: Rank, private val **suit**: Suit) : Comparable<Card> {

## Properties and Fields

Classes in Kotlin can have properties. These can be declared as mutable, using the var keyword or read-only using the val keyword.

there's no 'new' keyword in Kotlin

To use a property, we simply refer to it by name, as if it were a field in Java:

## Getters and Setters

The initializer, getter and setter are optional. Property type is optional if it can be inferred from the initializer (or from the getter return type, as shown below).

The full syntax of a read-only property declaration differs from a mutable one in two ways: it starts with val instead of var and does not allow a setter

## Enums

Each enum constant is an object. Enum constants are separated with commas.

Just like in Java, enum classes in Kotlin have synthetic methods allowing to list the defined enum constants and to get an enum constant by its name. The signatures of these methods are as follows (assuming the name of the enum class is EnumClass):

## Data Classes

We frequently create classes whose main purpose is to hold data. In such a class some standard functionality and utility functions are often mechanically derivable from the data. In Kotlin, this is called a *data class* and is marked as data:

* equals()/hashCode() pair;
* toString() of the form "User(name=John, age=42)";
* [componentN() functions](https://kotlinlang.org/docs/reference/multi-declarations.html) corresponding to the properties in their order of declaration;
* copy() function (see below).

## Collections: List, Set, Map

Unlike many languages, Kotlin distinguishes between mutable and immutable collections (lists, sets, maps, etc). Precise control over exactly when collections can be edited is useful for eliminating bugs, and for designing good APIs.

It is important to understand up front the difference between a read-only view of a mutable collection, and an actually immutable collection. Both are easy to create, but the type system doesn't express the difference, so keeping track of that (if it's relevant) is up to you.

## No Multimap in Kotlin

## Ranges

Range expressions are formed with rangeTo functions that have the operator form .. which is complemented by in and !in. Range is defined for any comparable type, but for integral primitive types it has an optimized implementation. Here are some examples of using ranges:

## Map

Access using index or get? IntelliJ prompts to change to index method.

## Extending Data Classes

The truth is: data classes do not play too well with inheritance. We are considering prohibiting or severely restricting inheritance of data classes. For example, it's known that there's no way to implement equals() correctly in a hierarchy on non-abstract classes.

So, all I can offer: don't use inheritance with data classes.

Reference: <https://stackoverflow.com/a/26467380>

## Transform method on Set

public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {  
 return *mapTo*(ArrayList<R>(*collectionSizeOrDefault*(10)), transform)  
}

## Assertions with Java

In the code the assertions are done after calling toString() so that we can assert that the code is correct. This has to be done using toString() because in Kotlin you cannot extend a class and implement an interface at the same time.

References: Kotlin docs and Stackoverflow (link pasted)