<https://github.com/MindorksOpenSource/from-java-to-kotlin>

Java

System.out.print("Amit Shekhar");

System.out.println("Amit Shekhar");

Kotlin

print("Amit Shekhar")

println("Amit Shekhar")

Java

String name = "Amit Shekhar";

final String name = "Amit Shekhar";

Kotlin

var name = "Amit Shekhar"

val name = "Amit Shekhar"

Java

final String name = null;

String otherName;

otherName = null;

Kotlin

val name : String? = null

var otherName : String?

otherName = null

Java

if(text != null){

int length = text.length();

}

Kotlin

val length = text?.length

Java

String firstName = "Amit";

String lastName = "Shekhar";

String message = "My name is: " + firstName + " " + lastName;

Kotlin

val firstName = "Amit"

val lastName = "Shekhar"

val message = "My name is: $firstName $lastName"

Java

String text = "First Line\n" +

"Second Line\n" +

"Third Line";

Kotlin

val text = """

|First Line

|Second Line

|Third Line

""".trimMargin()

Java

String text = x > 5 ? "x > 5" : "x <= 5";

Kotlin

val text = if (x > 5)

"x > 5"

else "x <= 5"

Java

if(object instanceof Car){

}

Car car = (Car) object;

Kotlin

if (object is Car) {

}

var car = object as Car

Java

if(object instanceof Car){

Car car = (Car) object;

}

Kotlin

if (object is Car) {

var car = object // smart casting

}

Java

if(score >= 0 && score <= 300 ){}

Kotlin

if (score in 0..300) { }

Java

int score = // some score;

String grade;

switch (score) {

case 10:

case 9:

grade = "Excellent";

break;

case 8:

case 7:

case 6:

grade = "Good";

break;

case 5:

case 4:

grade = "Ok";

break;

case 3:

case 2:

case 1:

grade = "Fail";

break;

default:

grade = "Fail";

}

Kotlin

var score = // some score

var grade = when (score) {

9, 10 -> "Excellent"

in 6..8 -> "Good"

4, 5 -> "Ok"

in 1..3 -> "Fail"

else -> "Fail"

}

Java

for (int i = 1; i <= 10 ; i++) { }

for (int i = 1; i <= 10 ; i+=2) { }

for (String item : collection) { }

for (Map.Entry<String, String> entry: map.entrySet()) { }

Kotlin

for (i in 1..10) { }

for (i in 1..10 step 2) {}

for (item in collection) {}

for ((key, value) in map) {}

Java

final List<Integer> listOfNumber = Arrays.asList(1, 2, 3, 4);

final Map<Integer, String> keyValue = new HashMap<Integer, String>();

map.put(1, "Amit");

map.put(2, "Ali");

map.put(3, "Mindorks");

Kotlin

val listOfNumber = listOf(1, 2, 3, 4)

val keyValue = mapOf(1 to "Amit",

2 to "Ali",

3 to "Mindorks")

Java

for (Car car : cars) {

System.out.println(car.speed);

}

for (Car car : cars) {

if(cars.speed > 100) {

System.out.println(car.speed);

}

}

Kotlin

cars.forEach {

println(it.speed)

}

cars.filter { it.speed > 100 }

.forEach { println(it.speed)}

Java

void doSomething() {

// logic here

}

Kotlin

fun doSomething() {

// logic here

}

Java

int getScore() {

// logic here

return score;

}

Kotlin

fun getScore(): Int {

// logic here

return score

}

// as a single-expression function

fun getScore(): Int = score

Java

int getScore(int value) {

// logic here

return 2 \* value;

}

Kotlin

fun getScore(value: Int): Int {

// logic here

return 2 \* value

}

// as a single-expression function

fun getScore(value: Int): Int = 2 \* value

Java

public class Utils {

private Utils() {

// This utility class is not publicly instantiable

}

public static int getScore(int value) {

return 2 \* value;

}

}

Kotlin

class Utils private constructor() {

companion object {

fun getScore(value: Int): Int {

return 2 \* value

}

}

}

// other way is also there

object Utils {

fun getScore(value: Int): Int {

return 2 \* value

}

}

Java

public class Developer {

private String name;

private int age;

public Developer(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Developer developer = (Developer) o;

if (age != developer.age) return false;

return name != null ? name.equals(developer.name) : developer.name == null;

}

@Override

public int hashCode() {

int result = name != null ? name.hashCode() : 0;

result = 31 \* result + age;

return result;

}

@Override

public String toString() {

return "Developer{" +

"name='" + name + '\'' +

", age=" + age +

'}';

}

}

Kotlin

data class Developer(val name: String, val age: Int)

Java

public class Utils {

private Utils() {

// This utility class is not publicly instantiable

}

public static int triple(int value) {

return 3 \* value;

}

}

int result = Utils.triple(3);

Kotlin

fun Int.triple(): Int {

return this \* 3

}

var result = 3.triple()

<https://gist.github.com/dodyg/5823184>

#Intro

Kotlin is a new programming language for the JVM. It produces Java bytecode, supports Android and generates JavaScript. The latest version of the language is Kotlin M5.3

Kotlin project website is at [kotlin.jetbrains.org](http://kotlin.jetbrains.org/).

All the codes here can be copied and run on [Kotlin online editor](http://kotlin-demo.jetbrains.com/).

Let's get started.

#Basics

* You do not need ; to break statements.
* Comments are similar to Java or C#, /\* This is comment \*/ for multi line comments and // for single line comment.
* Unlike Java, you do not need to match your file name to your class name.
* Like JavaScript, you can create functions outside classes. So there is no need to stuff your functions as static members of classes like what you do in C# or Java.
* Kotlin has string templates, which is awesome. e.g. "$firstName $lastName" for simple variable name or "${person.name} is ${1 \* 2}" for any expressions. You can still do the string concatenation if you like e.g. "hello " + "world", but that means being stupid.
* It has no tuple although Kotlin's data classes is an option to use in place of tuple.

##Variables

* There are two keywords for variable declaration, **var** and **val**.
* Use **var** when the variable value is to be modified and **val** where the variable value will not change after first assigned.
* This **val** is similar to **readonly** keyword in C# or **final** keyword in Java.
* **val** variable must be initialized at declaration.
* Unlike Java or C#, you declare the type of a variable after the name, e.g. var firstName : String
* Number primitive types are as follows: Double, Float, Long, Int, Short, Byte. There is no automatic conversion between types. You have to explicitly convert them.
* More primitive types: Char, String, Boolean.
* All variable declarations in Kotlin must be initialized.
* The keyword void common in Java or C# is called Unit in Kotlin.

###Null

In Kotlin you have to decide whether a variable can be assigned null or not. This applies to both primitives or class types. A nullable variable is marked by assigning ? after the type, e.g. var firstName: String?

You can assign a value from not-nullable to nullable type without any problem.

fun main(args : Array<String>) {

val firstName : String = "Adam"

val name : String? = firstName

print("$name")

}

The other way around though requires that you declare that this nullable variable does not contain null at the point of assignment with !! operator (which pretty much declares : "I am sure this nullable variable is not null at this point")

fun main(args : Array<String>) {

val name : String? = "Adam"

val firstName : String = name!!

print("$firstName")

}

###Type inference

Kotlin is pretty smart about inferring what type a variable is, whether it is primitives or class. This is similar to the var keyword in C#.

fun main(args : Array<String>) {

val firstName = "Adam"

val middle = 'c'

val lastName = "Brown"

val age = 15

println("$firstName $middle $lastNameis $age")

}

You will encounter in further examples of more capabilities of Kotlin's type inference.

#Functions We are going to spend a considerable time in discussing function because it has many different forms and subtleties. Here is a list of facilities that Kotlin provides for functions

* Single expression function.
* Optional parameter.
* Positional argument and named argument.
* Variable argument.
* Single expression function.
* Function type.
* Function literals.
* Callable references.
* Extension functions.
* Infix function call.
* Local function.
* Closure.
* Generic function.
* Operator overloading.

Below is an example of functions

fun main(args : Array<String>) {

greet(englishGreeting())

greet(italianGreeting())

}

fun greet(msg : String){

println(msg)

}

fun englishGreeting() : String = "Hello world"

fun italianGreeting() : String{

return "bon giorno"

}

* Functions can exists on their own.
* It is marked by **fun** keyword.
* If a function returns value, you declare it after the function name.
* englishGreeting() is a single expression function.
* A void function such as greet() returns Unit type but you are not required to declare it.
* All parameters in a Kotlin function are read only. You are actually not allowed to mark it with either val or varkeyword.

##Single expression function

This is a shorthand form in defining a function when you only have a single expression to be executed.

fun main(args : Array<String>) {

val res = add(1,1)

show("$res")

}

fun add(a : Int, b : Int) = a + b

fun show(msg : String) = println("$msg")

As you can see above, in a single expression function, the function return type is inferred. You can declare the return type if you want to such as below.

fun main(args : Array<String>) {

val res = add(1,1)

show("$res")

}

fun add(a : Int, b : Int) : Int = a + b

fun show(msg : String) : Unit = println("$msg")

##Optional parameters

Kotlin allows you to assign default values for your parameters, making them optional.

fun main(args : Array<String>) {

show()

show("Good morning")

}

fun show (msg : String = "Hello World"){

println("$msg")

}

If you are mixing mandatory parameter and optional parameter, the mandatory parameters must be listed first.

##Arguments

fun main(args : Array<String>) {

greet(firstName = "Frasensco", lastName = "Merini")

greet(lastName = "John", firstName = "Stamos")

greet("Borat", "Ismail")

greet("Crystal", lastName = "Stamos")

call("Xavier", age = 20, location = "Portugal")

}

fun greet(firstName : String, lastName : String){

println("Good morning $firstName $lastName")

}

fun call(name : String, location : String, age : Int){

println("Call $name who lives at $location and he is $age old")

}

Kotlin allows positional argument, named argument and the mix between the two. When you mix named and positional argument, you must start with positional argument.

###Variable arguments

Use the keyword **vararg**.

fun main(args : Array<String>) {

names("John", "Adam", "Joy")

}

fun names(vararg names : String){

for(n in names){

println("$n")

}

}

If **vararg** parameter is not the last parameter, named argument must be used to supply the function argument.

fun main(args : Array<String>) {

names("John", "Adam", "Joy", age = 20)

}

fun names(vararg names : String, age : Int){

for(n in names){

println("$n is $age old")

}

}

###vararg produces array of argument

fun main(args : Array<String>) {

names("John", "Adam", "Joy")

}

fun names(vararg names : String){

println("Argument length is ${names.size}")

println("${names[0]}")

val nns : Array<String> = names

println("${nns[1]}")

}

###Using array to supply variable arguments Use the \* operator in front of the array variable

fun main(args : Array<String>) {

val n = array("John", "Adam", "Joy")

names(\*n)

}

fun names(vararg names : String){

println("Argument length is ${names.size}")

println("${names[0]}")

val nns : Array<String> = names

println("${nns[1]}")

}

###Passing one varargs argument to another

fun main(args : Array<String>) {

val n = array("John", "Adam", "Joy")

fugitives(\*n)

}

,

fun fugitives(vararg escapees: String){

names(\*escapees)

}

fun names(vararg names : String){

println("Argument length is ${names.size}")

println("${names[0]}")

val nns : Array<String> = names

println("${nns[1]}")

}

Since **vararg** creates an array, you simply use the \* operator to pass one **vararg** to another.

##Function Types and Function Literals

A function type is a type consisted of a function signature and function return type that are separated by -> operator. In its simplest form, it looks as follows:

() -> Unit

Above is a type for a function that takes no parameter and returns a Unit (void in other language parlance)

() -> String

Above is a type for a function that takes no parameter and return a String

(String) -> Unit

Above is a type for a function that takes a string and returns nothing.

(String, Float) -> Unit

Above is a type for a function that takes two parameters (String and Float) and returns nothing.

Because a function type is just a type, it means that you can assign it to a variable, you can pass it as a function argument and you can return it from a function.

###Different ways to write function literals

val m = { (x : String) -> println("$x") }

val n : (String) -> Unit = { x -> println("$x") }

val o : (String) -> Unit = { (x : String) -> println("$x") }

fun main(args : Array<String>) {

m("good morning")

n("good morning")

o("good morning")

}

Above code is an example of function literals. All m, n and o represent the same function.

Below is a function that returns a function type

fun main(args : Array<String>) {

val greet = greetingFrom("Cairo, Egypt")

greet("Brown")

}

fun greetingFrom(location : String) : (String) -> Unit{

return { name -> println ("Hello $name from $location")}

}

Below shows that you can specify a function type as an argument and supply it with function literal with corresponding function signature and return type.

fun evening(): String = "Good Evening"

fun main(args : Array<String>){

say({ "good morning"})

say { val msg = "good afternoon" msg }

say({evening()})

}

fun say(greet : () -> String){

println("${greet()}")

}

##Callable references

How about if you already have a function that you want to pass as a parameter? You prefix the function name with '::'

fun main(args : Array<String>) {

calcAndShow(10,10, ::add) //20

calcAndShow(10,10, ::multiply) /100

calcAndShow(10,19, { x, y -> x - y }) //-9

}

fun calcAndShow(a : Int, b : Int, func : (a : Int, b : Int) -> Int){

val result = func (a, b)

println("$result")

}

fun add(a : Int, b : Int) : Int = a + b

fun multiply (a : Int, b : Int) : Int = a \* b

##Function expansion

When you call a function which has a function type as the last argument, you can expand it by { }

fun main(args : Array<String>) {

val a = calculate(1) { x -> 10 + x } //11

val b = calculate(2) { x -> 20 \* x } //40

println("a = $a, b = $b")

}

fun calculate(a : Int, calc : (Int) -> Int) : Int{

return calc(a)

}

##Closure

Kotlin support Closure as highlighted by the example below

fun main(args : Array<String>) {

val total = add(1)(2)

println("Total value is $total")

}

fun add(a : Int) : (Int) -> Int{

return { x -> a + x }

}

##Local function

You can declare a function inside a function. It will have access to the local variable at the parent function.

fun main(args : Array<String>){

accumulate()

}

fun accumulate(){

var i = 0

fun add(){

i++

}

for (i in 1..10){

add()

}

println("i is now $i")

}

//It prints "i is now 10"

##Extension function

Extension function enables a function to be accessed from the type function. It works in the form of **type.function** Inside the function, the keyword this refers to the instance.

For example

fun Int.show(){

println("This number is $this")

}

fun main(args : Array<String>){

3.show()

}

Above example shows how the Int built in type has been enriched by show extension function. Notice the use of thiskeyword that refers to the 3 number.

**Notice** You can extend a function on a nullable type and it will be accessible for both nullable and non nullable type. The reverse though does not apply.

fun Int?.show(){

println("This number is $this")

}

fun Int.show2(){

println("This number is $this")

}

fun main(args : Array<String>){

var number : Int? = null

number.show()

5.show()

//number.show2() will not compile

}

### Extension function expressed in function literals

val show = { Int.() -> println("This is number $this") }

val add = { Int.(number : Int) : Int ->

val now = this + number

now

}

fun main(args : Array<String>){

5.add(10).show()

}

Both show and add extension functions are expressed in literal format. Please notice that add function returns an Int.

### Extension function in infix form

fun main(args : Array<String>) {

val res = 1 add 2

println("$res")

}

fun Int.add (one : Int) : Int = this + one

If the extension function only takes one argument, you can call them in infix form (you drop the . between the type and the function). So instead of 1.add(2), you can call it in the form of 1 add 2. This makes certain constructs looks natural (more like an operator than a function call) and especially useful in construction DSL in Kotlin.

##Variable arguments and function type argument

vararg parameter can also be naturally combined with a function type parameter.

fun main(args : Array<String>) {

names("John", "Adam", "Joy"){

name -> println ("$name")

}

}

fun names(vararg names : String, print : (String) -> Unit){

for(n in names){

print(n)

}

}

above code can also be expressed in this matter (using named argument)

fun main(args : Array<String>) {

names("John", "Adam", "Joy", print = {name -> println ("$name")})

}

fun names(vararg names : String, print : (String) -> Unit){

for(n in names){

print(n)

}

}

#Control Structures

##If statement

Kotlin **if** statement should look familiar with other language

fun main(args : Array<String>) {

val total = 10

if (total > 5){

println("$total is greater than 5")

}else if (total > 10){

println("$total is greater than 10")

}else{

println("$total is less than 6")

}

}

#About

My name is Dody Gunawinata and I primarily develop using C#. I stumbled upon Kotlin last year as I was researching the way to write app for Android platform. I found the language easy to learn and easy read. I implemented Android Rivers, a fast River of News style RSS reader, in Kotlin. You can find the project at GitHub

With Android Rivers, you will learn about:

* Using SQLite database.
* High performance XML parsing.
* High performance JSON parsing.
* Implementing a media player.
* Writing Android Service.
* Accessing Android Clipboard.
* Fragments.
* Implementing Holo theme in Android 2.2 above.
* Using Library modules.
* First class Outliner UI support.
* Asynchronous operations.
* Managing file system.
* Creating custom dialogs.
* Implementing Sliding Menu.
* Asynchronous HTTP calls.
* Integrating Android Java library.
* Implementing Arabic language support.

<https://fabiomsr.github.io/from-java-to-kotlin/index.html>

BASICS

Print

Java

System.out.print("Hello, World!");

System.out.println("Hello, World!");

Kotlin

print("Hello, World!")

println("Hello, World!")

Variables I

Java

final int x;

final int y = 1;

Kotlin

val x: Int

val y = 1

Variables II

Java

int w;

int z = 2;

z = 3;

w = 1;

Kotlin

var w: Int

var z = 2

z = 3

w = 1

Null I

Java

final String name = null;

String lastName;

lastName = null

Kotlin

val name: String? = null

var lastName: String?

lastName = null

var firstName: String

firstName = null *// Compilation error!!*

Null II

Java

if(text != null){

int length = text.length();

}

Kotlin

val length = text?.length

val length = text!!.length *// NullPointerException if text == null*

Strings I

Java

String name = "John";

String lastName = "Smith";

String text = "My name is: " + name + " " + lastName;

String otherText = "My name is: " + name.substring(2);

Kotlin

val name = "John"

val lastName = "Smith"

val text = "My name is: $name $lastName"

val otherText = "My name is: ${name.substring(2)}"

Strings II

Java

String text = "First Line\n" +

"Second Line\n" +

"Third Line";

Kotlin

val text = """

|First Line

|Second Line

|Third Line

""".trimMargin()

Ternary Operator

Java

String text = x > 5 ? "x > 5" : "x <= 5";

Kotlin

val text = if (x > 5)

"x > 5"

else "x <= 5"

BASICS

Bits Operations

Java

final int andResult = a & b;

final int orResult = a | b;

final int xorResult = a ^ b;

final int rightShift = a >> 2;

final int leftShift = a << 2;

Kotlin

val andResult = a and b

val orResult = a or b

val xorResult = a xor b

val rightShift = a shr 2

val leftShift = a shl 2

Is As In

Java

if(x instanceof Integer){ }

final String text = (String) other;

if(x >= 0 && x <= 10 ){}

Kotlin

if (x is Int) { }

val text = other as String

if (x in 0..10) { }

Smart Cast

Java

if(a instanceof String){

final String result = ((String) a).substring(1);

}

Kotlin

if (a is String) {

val result = a.substring(1)

}

Switch / When

Java

final int x = *// value;*

final String xResult;

switch (x){

case 0:

case 11:

xResult = "0 or 11";

break;

case 1:

case 2:

*//...*

case 10:

xResult = "from 1 to 10";

break;

default:

if(x < 12 && x > 14) {

xResult = "not from 12 to 14";

break;

}

if(isOdd(x)) {

xResult = "is odd";

break;

}

xResult = "otherwise";

}

final int y = *// value;*

final String yResult;

if(isNegative(y)){

yResult = "is Negative";

} else if(isZero(y)){

yResult = "is Zero";

}else if(isOdd(y)){

yResult = "is Odd";

}else {

yResult = "otherwise";

}

Kotlin

val x = *// value*

val xResult = when (x) {

0, 11 -> "0 or 11"

in 1..10 -> "from 1 to 10"

!in 12..14 -> "not from 12 to 14"

else -> if (isOdd(x)) { "is odd" } else { "otherwise" }

}

val y = *// value*

val yResult = when {

isNegative(y) -> "is Negative"

isZero(y) -> "is Zero"

isOdd(y) -> "is odd"

else -> "otherwise"

}

For

Java

for (int i = 1; i < 11 ; i++) { }

for (int i = 1; i < 11 ; i+=2) { }

for (String item : collection) { }

for (Map.Entry<String, String> entry: map.entrySet()) { }

Kotlin

for (i in 1..10) { }

for (i in 1..10 step 2) {}

for (item in collection) {}

for ((index, item) in collection.withIndex()) {}

for ((key, value) in map) {}

Collections

Java

final List<Integer> numbers = Arrays.asList(1, 2, 3);

final Map<Integer, String> map = new HashMap<Integer, String>();

map.put(1, "One");

map.put(2, "Two");

map.put(3, "Three");

*// Java 9*

final List<Integer> numbers = List.of(1, 2, 3);

final Map<Integer, String> map = Map.of(1, "One",

2, "Two",

3, "Three");

Kotlin

val numbers = listOf(1, 2, 3)

val map = mapOf(1 to "One",

2 to "Two",

3 to "Three")

Collections

Java

for (int number : numbers) {

System.out.println(number);

}

for (int number : numbers) {

if(number > 5) {

System.out.println(number);

}

}

Kotlin

numbers.forEach {

println(it)

}

numbers.filter { it > 5 }

.forEach { println(it) }

Collections

Java

final Map<String, List<Integer>> groups = new HashMap<>();

for (int number : numbers) {

if((number & 1) == 0){

if(!groups.containsKey("even")){

groups.put("even", new ArrayList<>());

}

groups.get("even").add(number);

continue;

}

if(!groups.containsKey("odd")){

groups.put("odd", new ArrayList<>());

}

groups.get("odd").add(number);

}

Kotlin

val groups = numbers.groupBy {

if (it and 1 == 0) "even" else "odd"

}

Collections

Java

final List<Integer> evens = new ArrayList<>();

final List<Integer> odds = new ArrayList<>();

for (int number : numbers){

if ((number & 1) == 0) {

evens.add(number);

}else {

odds.add(number);

}

}

Kotlin

val (evens, odds) = numbers.partition { it and 1 == 0 }

Collections

Java

final List<User> users = getUsers();

Collections.sort(users, new Comparator<User>(){

public int compare(User user, User otherUser){

return user.lastname.compareTo(otherUser.lastname);

}

});

*// or*

users.sort(Comparator.comparing(user -> user.lastname));

Kotlin

val users = getUsers()

users.sortedBy { it.lastname }

<https://fabiomsr.github.io/from-java-to-kotlin/functions.html>

FUNCTIONS

Basic Function

Java

public void hello() {

System.out.print("Hello, World!");

}

Kotlin

fun hello() {

println("Hello, World!")

}

Arguments

Java

public void hello(String name){

System.out.print("Hello, " + name + "!");

}

Kotlin

fun hello(name: String) {

println("Hello, $name!")

}

Default Values

Java

public void hello(String name) {

if (name == null) {

name = "World";

}

System.out.print("Hello, " + name + "!");

}

Kotlin

fun hello(name: String = "World") {

println("Hello, $name!")

}

Return

Java

public boolean hasItems() {

return true;

}

Kotlin

fun hasItems() : Boolean {

return true

}

Single-Expression

Java

public double cube(double x) {

return x \* x \* x;

}

Kotlin

fun cube(x: Double) : Double = x \* x \* x

FUNCTIONS

Vararg

Java

public int sum(int... numbers) { }

Kotlin

fun sum(vararg x: Int) { }

Main

Java

public class MyClass {

public static void main(String[] args){

}

}

Kotlin

fun main(args: Array<String>) {

}

Named Arguments

Java

public static void main(String[]args){

openFile("file.txt", true);

}

public static File openFile(String filename, boolean readOnly) { }

Kotlin

fun main(args: Array<String>) {

openFile("file.txt", readOnly = true)

}

fun openFile(filename: String, readOnly: Boolean) : File { }

Optional Arguments

Java

public static void main(String[]args){

createFile("file.txt");

createFile("file.txt", true);

createFile("file.txt", true, false);

createExecutableFile("file.txt");

}

public static File createFile(String filename) { }

public static File createFile(String filename, boolean appendDate) { }

public static File createFile(String filename, boolean appendDate,

boolean executable) { }

public static File createExecutableFile(String filename) { }

Kotlin

fun main(args: Array<String>) {

createFile("file.txt")

createFile("file.txt", true)

createFile("file.txt", appendDate = true)

createFile("file.txt", true, false)

createFile("file.txt", appendDate = true, executable = true)

createFile("file.txt", executable = true)

}

fun createFile(filename: String, appendDate: Boolean = false,

executable: Boolean = false): File { }

Generic Methods

Java

public void init() {

List<String> moduleInferred = create("net");

}

public <T> List<T> createList(T item) { }

Kotlin

fun init() {

val module = createList<String>("net")

val moduleInferred = createList("net")

}

fun <T> createList(item: T): List<T> { }

Data Classes - Destructuring

Java

public static void main(String[]args) {

Book book = createBook();

System.out.println(book);

System.out.println("Title: " + book.title);

}

public static Book createBook(){

return new Book("title\_01", "author\_01");

}

public class Book {

final private String title;

final private String author;

public Book(String title, String author) {

this.title = title;

this.author = author;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

**@Override**

public String toString() {

return "Title: " + title + " Author: " + author;

}

}

Kotlin

fun main(args: Array<String>) {

val book = createBook();

*// or*

val (title, author) = createBook()

println(book)

println("Title: $title")

}

fun createBook() : Book{

return Book("title\_01", "author\_01")

}

data class Book(val title: String, val author: String)

<https://fabiomsr.github.io/from-java-to-kotlin/classes.html>

CLASSES

Constructor Call

Java

final File file = new File("file.txt");

Kotlin

val file = File("file.txt")

Class

Java

public final class User {

}

Kotlin

class User

Open Class

Java

public class User {

}

Kotlin

open class User

Final Attributes

Java

final class User {

private final String name;

public User(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

Kotlin

class User(val name: String)

Primary Constructor

Java

final class User {

private String name;

public User(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Kotlin

class User(var name: String)

Optional Arguments in Constructors

Java

final class User {

private String name;

private String lastName;

public User(String name) {

this(name, "");

}

public User(String name, String lastName) {

this.name = name;

this.lastName = lastName;

}

*// And Getters & Setters*

}

Kotlin

class User(var name: String, var lastName: String = "")

Properties

Java

public class Document {

private String id = "00x";

public String getId() {

return id;

}

public void setId(String id) {

if(id != null && !id.isEmpty()) {

this.id = id;

}

}

}

Kotlin

class Document{

var id : String = "00x"

set(value) {

if(value.isNotEmpty()) field = value

}

}

FUNCTIONS

Abstract Class

Java

public abstract class Document{

public abstract int calculateSize();

}

public class Photo extends Document{

**@Override**

public int calculateSize() {

}

}

Kotlin

abstract class Document {

abstract fun calculateSize(): Int

}

class Photo : Document() {

override fun calculateSize(): Int {

}

}

Singleton

Java

public class Document {

private static final Document INSTANCE = new Document();

public static Document getInstance(){

return INSTANCE;

}

}

Kotlin

object Document {

}

Extensions

Java

public class ByteArrayUtils {

public static String toHexString(byte[] data) {

}

}

final byte[] dummyData = new byte[10];

final String hexValue = ByteArrayUtils.toHexString(dummyData);

Kotlin

fun ByteArray.toHex() : String {

}

val dummyData = byteArrayOf()

val hexValue = dummyData.toHex()

Inner Class

Java

public class Documment {

class InnerClass {

}

}

Kotlin

class Document {

internal inner class InnerClass

}

Nested Class

Java

public class Documment {

public static class InnerClass {

}

}

Kotlin

class Document {

class InnerClass

}

Interface

Java

public interface Printable {

void print();

}

public class Document implements Printable {

**@Override**

public void print() {

}

}

Kotlin

interface Printable{

fun print()

}

class Document : Printable{

override fun print() {

}

}