**Functions**

Function without return

**fun** main(args: Array<String>) {

println("Hello, world!")

}

Function with return

**fun** max(a: Int, b: Int): Int {

**return** **if** (a > b) a **else** b

}

**fun** max(a: Int, b: Int): Int = **if** (a > b) a **else** b

**fun** max(a: Int, b: Int) = **if** (a > b) a **else** b

**Variables**

**val** answer: Int = 42

**val** answer = 42

**val** question = "Ask a question”

val (from value)—Immutable reference. A variable declared with val can’t be

reassigned after it’s initialized. It corresponds to a final variable in Java.

var (from variable)—Mutable reference. The value of such a variable can be changed.

This declaration corresponds to a regular (non-final) Java variable.

Even though the var keyword allows a variable to change its value, its type is fixed.

**string formatting**

If you need to include the $ character in a string, you escape it: println("\$x")

**val** name = “World”

println("Hello, $name!")

println("Hello, ${args[0]}!")

println("Hello, ${**if** (args.size > 0) args[0] **else** "someone"}!")

**Classes**

**class** Person(**val** name: String)

/\* Java \*/

**public** **class** Person {

**private** **final** String name;

**public** Person(String name){

**this**.name = name;

}

**public** String getName(){

**return** name;

}

}

In Kotlin, public is the default visibility

**Properties**

class Person(val name: String, var isMarried: Boolean)

name - Read-only property: generates a field and a trivial getter.

isMarried - Writable property: a field, a getter and a setter.

**val** person = Person("Bob", **true**)

println(person.name)

println(person.isMarried)

person.*isMarried* = **false**

**Directories and Packages**

Package declaration

**package** geometry.shapes

Imports the standard Java library class

**import** java.util.Random

Imports a function by name

**import** geometry.shapes.createRandomRectangle

Iimport all

**import** geometry.shapes.\*

**Enum classes**

**enum** **class** Color {

***RED***, ***ORANGE***, ***YELLOW***, ***GREEN***, ***BLUE***, ***INDIGO***, ***VIOLET***

}

**enum** **class** Color(

**val** r: Int, **val** g: Int, **val** b: Int

//Declares properties of enum constants

) {

***RED***(255, 0, 0), ***ORANGE***(255, 165, 0),

***YELLOW***(255, 255, 0), ***GREEN***(0, 255, 0), ***BLUE***(0, 0, 255),

***INDIGO***(75, 0, 130), ***VIOLET***(238, 130, 238);

//Specifies property values when each constant is created

**fun** rgb() = (r \* 256 + g) \* 256 + b //Defines a method on the enum class

}

**When**

**fun** getMnemonic(color: Color) =

**when** (color) {

Color.***RED*** -> "Richard"

Color.***ORANGE*** -> "Of"

Color.***YELLOW*** -> "York"

Color.***GREEN*** -> "Gave"

Color.***BLUE*** -> "Battle"

Color.***INDIGO*** -> "In"

Color.***VIOLET*** -> "Vain"

}

combine multiple values in the same branch if you separate them with commas

**fun** getWarmth(color: Color) =

**when** (color) {

Color.***RED***, Color.***ORANGE***, Color.***YELLOW*** -> "warm"

Color . GREEN -> "neutral"

Color.***BLUE***, Color.***INDIGO***, Color.***VIOLET*** -> "cold"

}

**fun** mix(c1: Color, c2: Color) =

**when** (setOf(c1, c2)) {

setOf(Color.***RED***, Color.***YELLOW***) -> Color.***ORANGE***

setOf(Color.***YELLOW***, Color.***BLUE***) -> Color.***GREEN***

setOf(Color.***BLUE***, Color.***VIOLET***) -> Color.***INDIGO***

**else** -> **throw** Exception("Dirty color")

}

If colors c1 and c2 are RED and YELLOW (or vice versa), the result of mixing them is ORANGE , and so on. To implement this, you use set comparison. The Kotlin standard library contains a function setOf that creates a Set containing the objects specified as itsarguments. A set is a collection for which the order of items doesn’t matter; two sets are

equal if they contain the same items.

No argument for when

**fun** mixOptimized(c1: Color, c2: Color) =

**when** {

(c1 == Color.***RED*** && c2 == Color.***YELLOW***) ||

(c1 == Color.***YELLOW*** && c2 == Color.***RED***) ->

Color.***ORANGE***

(c1 == Color.***YELLOW*** && c2 == Color.***BLUE***) ||

(c1 == Color.***BLUE*** && c2 == Color.***YELLOW***) ->

Color.***GREEN***

(c1 == Color.***BLUE*** && c2 == Color.***VIOLET***) ||

(c1 == Color.***VIOLET*** && c2 == Color.***BLUE***) ->

Color.***INDIGO***

**else** -> **throw** Exception("Dirty color")

}

**Smart casts: combining type checks and casts**

**interface** Expr

**class** Num(**val** value: Int) : Expr

**class** Sum(**val** left: Expr, **val** right: Expr) : Expr

**fun** eval(e: Expr): Int {

**if** (e **is** Num) {

**val** n = e **as** Num //This explicit cast to Num is redundant.

**return** n.value

}

**if** (e **is** Sum) {

**return** eval(e.right) + eval(e.left) //The variable e is smart-cast.

}

**throw** IllegalArgumentException("Unknown expression")

}

The '**is**' check is similar to instanceof in Java.

The smart cast works only if a variable couldn’t have changed after the **is** check. The property has to be a val.

Same function with when

**fun** eval(e: Expr): Int =

**when** (e) {

**is** Num ->

e.value

**is** Sum ->

eval(e.right) + eval(e.left)

**else** ->

**throw** IllegalArgumentException("Unknown expression")

}

Both if and when can have blocks as branches. In this case, the last expression in the

block is the result.

**The while Loop**

**while (**condition**) {**

/\*...\*/

}

**do** {

/\*...\*/

} **while** (condition)

**Iterating over numbers: ranges and progressions**

A range is essentially just an interval between two values

**val** oneToTen = 1..10

**for** (i **in** 1..100) {

/\* ------ \*/

}

iterating over a progression that has a step, which allows it to skip some

numbers

**for** (i **in** 1..20 step 2) {

println(i)

}

The step can also be negative, in which case the progression goes backward

rather than forward.

**for** (i **in** 100 downTo 1 step 2) {

}

Iterate overhalf-closed ranges, which don’t include the specified end point

**val** str = "Kotlin"

**for** (i **in** 2..str.length - 1) {

println(i)

}

**for** (i **in** 2 until str.length) {

println(i)

}

**for** (c **in** 'A'..'F') {

// it works with characters also

}

**Iterating over Collection**

**val set = setOf(1, 7, 53)**

**val** list = listOf(1, 7, 53)

**val** map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")

**val binaryReps = TreeMap<Char, String>()**

binaryReps['a'] = "A"

binaryReps['b'] = "B"

**for** ((key, value) **in** binaryReps) {

println("$key = $value")

}

**val list = arrayListOf("10", "11", "1001")**

**for (element in list) {**

println(" $element")

}

**for** ((index, element) **in** list.withIndex()) {

println("$index: $element")

}

**Using an 'in' check**

**fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'**

**fun isNotDigit(c: Char) = c !in '0'..'9'**

**fun recognize(c: Char) = when (c) {**

**in** '0'..'9' -> "It's a digit!"

**in** 'a'..'z', **in** 'A'..'Z' -> "It's a letter!"

**else** -> "I don't know..."

}

**println("Kotlin" in setOf("Java", "Scala"))**

**Exceptions in Kotlin**

**throw IllegalArgumentException(**

"A percentage value must be between 0 and 100")

Unlike in Java, in Kotlin the throw construct is an expression and can be used as a part of other expressions:

**val** percentage =

**if** (number **in** 0..100)

number

**else**

**throw** IllegalArgumentException(

"A percentage value must be between 0 and 100: $number")

In this example, if the condition is satisfied, the program behaves correctly, and the percentage variable is initialized with number . Otherwise, an exception is thrown, and the variable isn’t initialized.

**fun readNumber(reader: BufferedReader): Int? {**

**try** {

**val** line = reader.readLine()

**return** Integer.parseInt(line)

} **catch** (e: NumberFormatException) {

**return** **null**

} **finally** {

reader.close()

}

}

Just like many other modern JVM languages, Kotlin doesn’t differentiate between checked and unchecked exceptions. You don’t specify the exceptions thrown by a function, and you may or may not handle any exceptions.

**'try' as an expression**

**val** number = **try** {

Integer.parseInt(reader.readLine())

} **catch** (e: NumberFormatException) {

**return**

}