Kotlin

Reference Guide

for Beginners

Rapidly learn Kotlin's key features

Kotlin Reference Guide for Beginners

Learn the key features of Kotlin rapidly

Dedication

This book is dedicated to all developers alike. I hope that you not only learn Kotlin, but become master of it!

About This Book

I wrote this book because Kotlin is a pithy, powerful, and popular programming language that every developer should know. Whether you are a newbie or a seasoned developer, learning Kotlin will be a breeze compared to more established languages such as C++ and especially Java. While Kotlin is simple, you can't learn Kotlin simply by reading. That's why this book is very code-centric, as you are encouraged to study the code, make connections, and create your own programs.

Finally, given that I wrote this book with beginners in mind, I ommited discussion of more advanced subjects such as coroutines and reflection. Stay tune for more!

About the Author

Asaiah Toutouyoutte is an avid programmer who writes books to teach people to code faster and better. He is passionate about a variety of subjects, including Android development, Python scripting, psychology, language learning, and other topics. He is a lifelong learner who enjoys thinking and achieving.

GitHub: https://www.github.com/asaiahdev777

Email: asaiahdev777@gmail.com

Source Code

Find the source code at:
https://www.github.com/asaiahdev777/kotlin_reference_guide_for_beginners

Chapter 1: Variables

Creating Variables

Terms

- Variable: storage box for information
- Declaration: when a variable is named
- Assignment: when a variable is given a value
- Immutable variable: value can't be changed after assignment
- Mutable variable: value can be changed after assignment

Note: Variable names can't start with digits, contain spaces, or contain certain special characters.

Lines of code starting with "//" are single-line comments. Comments are notes; they are not part of the program.

Lines of code starting with "/*" and ending with "/*" are multi-line comments.

Variables/CreatingVariables.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

//To create mutable variable
var myAge = 30

//To print variable
println(myAge)

//To reassign mutable variable
myAge = 19
println(myAge)

//To create immutable variable
val myYearOfBirth = 1900
println(myYearOfBirth)
```

Variable types

- **String**: holds text
- **Char**: holds 1 character
- **Int**: holds whole numbers
- **Long**: holds very large whole numbers
- **Float**: holds decimal numbers
- **Double**: holds very large decimal numbers
- **Boolean**: holds true/false value

Primitive variables are either String, Char, Byte, Short, Int, Long, Double, Float, or Boolean.

Variables/VariableTypes.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

//String
val myString = "This is a string value"

//To insert variable in string
println("String: $myString")

//Character
val myCharacter = 'A'
println("Character: $myCharacter")

//Integer
val myInt = 1000
println("Int: $myInt")

//Long
val myLong1 = 7000000000000
println("Long: $myLong1")

//Long
val myLong2 = 7000L
println("Long: $myLong2")

//Float
val myFloat1 = 1.4F
println("Float: $myFloat1")

//Float
val myFloat2 = 1.4f
```

```
println("Float: $myFloat2")

//Double
val myDouble = 1.4
println("Double: $myDouble")

//Boolean (true)
val myBooleanTrue = true
println("Boolean: $myBooleanTrue")

//Boolean (false)
val myBooleanFalse = false
println("Boolean: $myBooleanFalse")
```

Explicit Types

It is possible to explicitly set variable types.

Variables/ExplicitTyping.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

val myString: String = "This is a string value"
println("String: $myString")

val myCharacter: Char = 'A'
println("Character: $myCharacter")

val myInt: Int = 1000
println("Int: $myInt")

val myLong: Long = 7000000000000
println("Long: $myLong")

val myFloat: Float = 1.4F
println("Float: $myFloat")

val myDouble: Double = 1.4
println("Double: $myDouble")

val myBooleanTrue: Boolean = true
println("Boolean: $myBooleanTrue")
```

```
val myBooleanFalse: Boolean = false
println("Boolean: $myBooleanFalse")
```

Static Typing

Variable types are static (usually) and set at declaration.

Variables/StaticTyping.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

var greeting = "Hello"

//Int can't be assigned to a String
/*greeting = 9*/

//Any = supertype of all variable types
//Possible to change type, because type is Any
var dynamicVariable: Any = 9

//Print the type of variable
println(dynamicVariable::class)

dynamicVariable = "Hello"
println(dynamicVariable::class)

//Unit is not null. Unit means empty!
dynamicVariable = Unit
println(dynamicVariable::class)
```

Null and Nothing

Null means the absence of a value.

1. All variables must be declared and assigned at the same time.
2. By default all variables are non-null, meaning they can't be set to null.

Variables/NullNothing.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

//To mark as nullable
```

```
var myString: String? = null
println(myString)

myString = "Dog"
println(myString)

//To make null
myString = null
println(myString)
```

Elvis

Kotlin has language features that make accessing null variables less error-prone.

Variables/Elvis1.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

println("Type age")

//To read typed text
val typedName = readLine()

//Elvis: if typedName not null, return typedName; else, return ""
val name = typedName ?: ""
val age = name.toIntOrNull()

//If age not null, return age; if not, return "Invalid age."
println(age ?: "Invalid")
```

More concise:

Variables/Elvis2.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

println("Type age")
val yourAge = readLine()?.toIntOrNull()
println(yourAge ?: "Invalid")
```

Even more concise:

Variables/Elvis3.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

println("Type age")
println(readLine()?.toIntOrNull() ?: "Invalid")
```

Casts

It is possible to convert one variable type to another.

Variables/Casts.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

val number : Any = 7
println(number)

//To cast to different type
val anythingAsAnInt = number as Int
println(anythingAsAnInt)

//Crashes; 7 can't be cast to String
val numberAsString = number as String
println(numberAsString)
```

Safe Casts

Variables/SafeCasts.ws.kts

```
package kotlin_reference_guide_for_beginners.variables

val number : Any = 7

//To return null if variable can't be cast to the new cast
val numberAsString = number as? String
println(numberAsString ?: "Ints not castable to String")
```

Type Checking

Variables/TypeChecking.ws.kts

```kotlin
package kotlin_reference_guide_for_beginners.variables

val number: Any = 7

//To find if variable is of a certain type
val isNumberString = number is String
println(isNumberString)
```

Chapter 2: Functions

What are Functions

Functions are reusable blocks of code. Functions can take inputs, called parameters/arguments, and return outputs.

Basic Functions

Functions/SimpleFunction.ws.kts

```
package kotlin_reference_guide_for_beginners.functions

fun printMyName() {
    println("My name is John")
}

//To call a function
printMyName()
```

Functions with Output Only

Functions/OutputOnly.ws.kts

```
package kotlin_reference_guide_for_beginners.functions

fun getMyAge() : Int {
    return 2022 - 1800
}

//To call a function
val age = getMyAge()
println(age)
```

Functions with Input Only

Functions/InputOnly.ws.kts

```
package kotlin_reference_guide_for_beginners.functions

fun printMyName(firstName: String, lastName: String) {
    println("My name is $firstName $lastName")
}

//To call a function
printMyName("John", "Doe")
```

Functions with Input and Output

Functions/InputOutput.ws.kts

```
package kotlin_reference_guide_for_beginners.functions

fun getLengthOfMyName(firstName: String, lastName: String): Int {
    return firstName.length + lastName.length
}

//To call a function
val lengthOfMyName = getLengthOfMyName("John", "Doe")
println(lengthOfMyName)
```

One-line Functions

The { } in a function can be omitted if the body contains only 1 statement.

Functions/Oneline.ws.kts

```
package kotlin_reference_guide_for_beginners.functions

fun returnMyAge(dateOfBirth : Int) = 2021 - dateOfBirth

//Return type explicitly set (redundant)
fun getMyAge() : Int = 2021 - 1678

println(returnMyAge(1921))
println(getMyAge())
```

Named Arguments

Using named arguments makes calling a function easier.

Functions/NamedArguments.ws.kts

```
package kotlin_reference_guide_for_beginners.functions

fun introduce(name: String, age: Int, experience: Int) {
    println("I'm $name. I'm $age years old. I've known Kotlin for
$experience years.")
}

//Can name some parameters , and provide the rest as normal
introduce("John", age = 77, experience = 3)

//Can provide parameters out of order (must be named)
introduce(experience = 4, name = "Jack", age = 24)
```

Default Arguments

Using default arguments allows us to omit passing values for all arguments.

Functions/DefaultArguments.ws.kts

```
package kotlin_reference_guide_for_beginners.functions

fun introduce(name: String = "John", age: Int = 30, experience: Int)
{
    println("I'm $name. I'm $age years old. I've known Kotlin for
$experience years.")
}

//Can omit passing arguments that have a default value
introduce(experience = 3)
```

Extension Functions

It is possible to add more functionality to data types using extension functions.

Functions/Extensions.ws.kts

```
package kotlin_reference_guide_for_beginners.functions
```

```
//To declare extension function
fun String.changeCase(prefix: String = "Info"): String {

    //this: value on which function called
    val lower = this.lowercase()

    //Can omit "this" (implicitly uppercase on "this")
    val upper = uppercase()

    return "$prefix: $this $lower $upper"
}

println("John Doe".changeCase())
```

Chapter 3: Control Structures

Control structures refer to language features used to control code execution.

If statement

The if-statement runs code in the if-branch **only** if a condition is **true**.

Control/IfStatement.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type number of sides")

//toIntOrNull() returns null if text can't be converted to integer
val sides = readLine()?.toIntOrNull()

//If sides not = to null & numberOfSides >= 3
if (sides != null && sides >= 3) {
    println("Polygon")
}

//Braces not mandatory if only 1 statement
if (sides != null && sides >= 3)
    println("Polygon")
```

If-statement with Else

The if-statement runs code in the **if**-branch **only** if a condition is **true**, else it runs the code in the **else**-branch.

Control/IfStatementWithElse.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type number of sides")
val sides = readLine()?.toIntOrNull()

//If sides is not null AND numberOfSides >= 3
if (sides != null && sides >= 3) println("Polygon")
```

```
else println("Invalid")

//More concise version using if-expression
val message = if (sides != null && sides >= 3) "Polygon" else
"Invalid"
println(message)
```

If-statement with Else If

The if-statement runs code in the **if**-branch **only** if a condition is **true**, else it tests the conditions the **else if**-branches.

Control/IfStatementWithElseIf1.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type number of sides")
val sides = readLine()?.toIntOrNull()

if (sides == null) println("Invalid number.")
else if (sides < 3) println("Invalid ($sides sides)")
else if (sides == 3) println("Triangle")
else if (sides == 4) println("Quadrilateral")
else if (sides == 5) println("Pentagon")
else if (sides == 6) println("Hexagon")
else if (sides == 7) println("Heptagon")
else if (sides == 8) println("Octagon")
else if (sides == 9) println("Nonagon")
else if (sides == 10) println("Decagon")
else println("Too many!")
```

More concise version using if-expression:

Control/IfStatementWithElseIf2.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type number of sides")
val sides = readLine()?.toIntOrNull()

val message =
    if (sides == null) {
        "Invalid"
```

```
    } else if (sides < 3) {
        "Invalid ($sides sides)"
    } else if (sides == 3) {
        "Triangle"
    } else if (sides == 4) {
        println("A square is a kind of quadrilateral")
        "Quadrilateral" //This is what's returned
    } else if (sides == 5) {
        "Pentagon"
    } else if (sides == 6) {
        "Hexagon"
    } else if (sides == 7) {
        println("The lucky number")
        "Heptagon" //This is what's returned
    } else if (sides == 8) {
        "Octagon"
    } else if (sides == 9) {
        "Nonagon"
    } else if (sides == 10) {
        "Decagon"
    } else {
        "Too many!" //Else mandatory
    }

println(message)
```

Even more concise:

Control/IfStatementWithElseIf3.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type number of sides")
val sides = readLine()?.toIntOrNull()

val message =
    if (sides == null) "Invalid"
    else if (sides < 3) "Invalid ($sides sides)"
    else if (sides == 3) "Triangle"
    else if (sides == 4) {
        println("A square is a kind of quadrilateral")
        "Quadrilateral" //This is what's returned
    }
    else if (sides == 5) "Pentagon"
    else if (sides == 6) "Hexagon"
```

```
    else if (sides == 7) {
        println("The lucky number")
        "Heptagon" //This is what's returned
    }
    else if (sides == 8) "Octagon"
    else if (sides == 9) "Nonagon"
    else if (sides == 10) "Decagon"
    else "Too many!" //Else mandatory

println(message)
```

Boolean Operators

Boolean operators are used to evaluate boolean conditions.

Control/BooleanOperators.ws.kts

```
package kotlin_reference_guide_for_beginners.control

val iAmSmart = false
val iLikeMath = true

//If iAmSmart AND iLikeMath
if (iAmSmart && iLikeMath)
    println("I am smart and I like math")

//If iAmSmart OR iLikeMath
else if (iAmSmart || iLikeMath)
    println("I am smart or I like math")

//If I am NOT smart
else if (!iAmSmart)
    println("I am not smart")

//If I am NOT smart AND I don't like math
else if (!iAmSmart && !iLikeMath)
    println("I am not smart and I don't like math")

//If (I am NOT smart AND I like math) OR (I am smart, AND I don't li
ke math)
else if ((!iAmSmart && iLikeMath) || (iAmSmart && !iLikeMath))
    println("Very strange combination")
```

When statement

When-statement executes code **when** a condition is **true**.

Control/WhenStatement.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type the number of sides")
val sides = readLine()?.toIntOrNull()

when {
    sides == null || sides < 3 -> println("Invalid")
    sides == 3 -> println("Triangle")
    sides == 4 -> println("Quadrilateral")
    sides == 5 -> println("Pentagon")
    sides == 6 -> println("Hexagon")
    sides == 7 -> println("Heptagon")
    sides == 8 -> println("Octagon")
    sides == 9 -> println("Nonagon")
    sides == 10 -> println("Decagon")
    //Else not mandatory
    else -> println("Too many sides!")
}
```

When expression

When-expression executes code and returns a specific value **when** a statement is **true**.

Control/WhenExpression.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type number of sides")
val sides = readLine()?.toIntOrNull()

val message = when {
    sides == null || sides < 3 -> "Invalid"
    sides == 3 -> "Triangle"
    sides == 4 -> "Quadrilateral"
    sides == 5 -> "Pentagon"
    sides == 6 -> "Hexagon"
```

```
    sides == 7 -> "Heptagon"
    sides == 8 -> "Octagon"
    sides == 9 -> "Nonagon"
    sides == 10 -> "Decagon"
    //Else mandatory
    else -> "Too many sides!"
}
println(message)
```

When with Subject

Control/WhenWithSubject.ws.kts

```
package kotlin_reference_guide_for_beginners.control

val animal = readLine() ?: ""

//When statement with subject
when (animal) {
    "dog", "cat" -> println("Mammals")
    "parrot", "canary" -> println("Birds")
    else -> println("???")
}

//When expression with subject
val message = when (animal) {
    "dog", "cat" -> println("Mammals")
    "parrot", "canary" -> println("Birds")
    else -> println("???")
}

println(message)
```

Here's an example involving instance checks:

Control/WhenWithSubjectInstanceCheck.ws.kts

```
package kotlin_reference_guide_for_beginners.control

val x: Any = "chicken"

//When with subject
when (x) {
```

```
    is String -> println(x.uppercase()) //auto-casted to String
    is Int -> println(x.toFloat()) //auto-casted to Int
}
```

Here's an example involving ranges:

Control/WhenWithSubjectRanges.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type age\n")
val age = readLine()?.toIntOrNull()

val message = if (age != null) when (age) {
    in 0..1 -> "Baby"
    in 2..6 -> "Toddler"
    in 7..9 -> "Young Child"
    in 10..11 -> "Preteen"
    in 12..15 -> "Young adolescent"
    in 16..19 -> "Older adolescent"
    in 20..30 -> "Young adult"
    in 31..59 -> "Middle-age adult"
    in 60..100 -> "Senior adult"
    else -> "Centenarian"
} else "Invalid age"

println(message)
```

For loop

For loop can be used to loop over lists or to perform code a certain amount of times.

Control/ForLoop.ws.kts

```
package kotlin_reference_guide_for_beginners.control

val collection = listOf("Programmer", "John", "Kerry", "Doe")

//To loop over a list
for (word in collection) {
    println(word)
}
```

```
//Braces can be omitted if body contains 1 statement
for (word in collection) println("Word: $word")

//To loop over a range of numbers (counting upwards)
for (number in 1..5)
    println("Counting up: $number")
```

While loop

While loop perform a block of code so long as a specific condition is true.

Control/WhileLoop.ws.kts

```
package kotlin_reference_guide_for_beginners.control

var x = -1

//Keep looping while x < 5
while (x < 5) {
    x++ //increment x
    println(x)
}
```

Return statement

Return statement is used to abort execution of a function.

Control/Return.ws.kts

```
package kotlin_reference_guide_for_beginners.control

fun typeYourName() {
    println("Type your name or type Exit to end")

    val name = readLine()

    if (name != null) {
        if (name.equals("Exit", ignoreCase = true)) {
            println("Exiting...")
            //Returns from function (stop executing)
            return
        } else println("Hello $name!")
```

```
    }
}

typeYourName()
```

Break statement

Break statement is used to abort execution of a loop.

Control/Break.ws.kts

```
package kotlin_reference_guide_for_beginners.control

val numbers = listOf(1, 5, 6, 7, 9, 8, 10)

for (num in numbers) {
    //If the number is even
    if (num % 2 == 0) {
        println("I found an even number: $num")
        //Stop the loop
        break
    }
}

//Loop labels help specify which loop we are specifying
loopA@ for (i in 1..10) {
    loopB@ for (j in 1..10) {
        if (i * j == 15) {
            println("$i times $j equals 15.")
            //Break the outer loop
            break@loopA
        }
    }
}
```

Continue statement

Continue statement is used to move to next iteration of a loop.

Control/Continue.ws.kts

```
package kotlin_reference_guide_for_beginners.control

val numbers = listOf(1, 5, 6, 7, 9, 8, 10)

for (num in numbers) {
    //If the number is even
    if (num % 2 == 0) {
        println("I found an even number $num")
        //Continue to the next iteration of the loop
        continue
    } else if (num % 3 == 0) {
        println("Be careful with the number 3. Half of 6 is 3")
        //Stop the loop
        break
    }
}

//Loop labels help specify which loop we are specifying
loopA@ for (i in 1..10) {
    loopB@ for (j in 1..10) {
        if (i * j == 15) {
            println("$i times $j equals 15.")
            //Skip to the next iteration of the outer loop
            continue@loopA
        }
    }
}
```

Exceptions

Exceptions are events that are thrown when an error happens in code. It is possible to catch exceptions and handle them accordingly.

Control/Exceptions.ws.kts

```
package kotlin_reference_guide_for_beginners.control

println("Type a number to be converted to string")

try {
    val number = readLine()?.toInt()
    println("The number is $number")
} catch (e: NumberFormatException) {
```

```
    println("Invalid number")
} finally {
    println("Finally block always executed")
}


/*
//To throw an exception
throw Exception()

//Or more specific
throw ArithmeticException()*/
```

Chapter 4: Classes and Properties

What Are Classes

Classes are code structures used to represent concepts. Classes can contain methods and properties. Objects are actual instances of classes.

Classes/Basic.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//To declare a class
class Simple

//To instantiate a class
val simpleObject = Simple()

//The simpleObject is an instance of the Simple class
println(simpleObject)
```

Member Properties and Constructors

Constructors are used to create a class with specific properties. Properties are variables that are part of a class definition.

Classes/Constructors.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//Constructor parameters are between the ()
//Member properties start with val/var
//Default arguments allowed

class Person(
    val fName: String = "John",
    val lName: String = "Doe",
    var age: Int,
    mName: String = "" //Constructor parameter
) {
    //Member property
```

```
    val fullName = "$fName $mName $lName"

    //Initializer block called when object created
    init {
        println("I, $fName am alive!")
    }
}

//To create object
val person = Person("Jack", age = 12)

//To access class properties
println(person.fName)
println(person.lName)
println(person.fullName)
println(person.age)

person.age = 19
println(person.age)


//Won't work; middleName not a member property
//println(person.middleName)
```

Secondary Constructors

Secondary constructors are alternative constructors for creating a class. All secondary constructors must call the primary/main constructor.

Classes/SecondaryConstructors.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

class Person(
    var fName: String = "John",
    var lName: String = "Doe"
) {
    //Full name must have 2 names (or crash)
    constructor(fullName: String) : this() {
        val names = fullName.split(" ")
        fName = names[0]
        lName = names[1]
    }
```

```
    //Member property
    val fullName = "$fName $lName"
}

//Instantiate class as usual
val person1 = Person("John Kerry Larry")
println(person1.fName)
println(person1.lName)
println(person1.fullName)

//Instantiate class as usual
val person2 = Person("Jim", "Brown")
println(person2.fName)
println(person2.lName)
println(person2.fullName)
```

Member Functions/Methods

Member functions are functions that are part of a class definition and accesible from an object.

Classes/MemberFunctions.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

class Person(var fName: String = "John",
             var lName: String = "Doe",
             var age: Int) {

    //Member function
    fun getName() = "$fName $lName"

    //Member function
    fun introduceYourself() = println("I'm ${getName()}, and I'm
$age.")

    //Member function
    fun changeYourName(newFirstName: String, newLastName: String) {
        fName = newFirstName
        lName = newLastName
    }
}

val person = Person("John", "Dove", 17)
```

```
person.introduceYourself()

person.changeYourName("Bill", "Hal")
person.introduceYourself()
```

Static Functions/Methods and Properties

Static functions/methods and properties are methods and properties that are accesible without needing an object instance.

Classes/StaticFunctionsProperties.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

class Country(
    val name: String,
    val language: String,
    val continent: String,
    val nickname: String = ""
) {
    //Static methods/properties go in companion object
    companion object {
        //Use const for static immutable primitives
        const val africa = "Africa"
        const val northAmerica = "North America"
        const val southAmerica = "South America"
        const val asia = "Asia"
        const val europe = "Europe"
        const val oceania = "Oceania"

        val continents = listOf(africa, northAmerica, southAmerica,
asia, europe, oceania)

        fun printAllContinents() {
            println(continents)

            //Member methods/properties inaccessible
            //println(nickname)
        }
    }
}

//To access static methods/functions and properties
println(Country.continents)
```

```
println(Country.africa)
Country.printAllContinents()

//Accessing object properties
val country = Country("China", "Chinese", "Asia", "Mighty Beijing")
println(country.name)
println(country.language)
println(country.continent)
println(country.nickname)
```

Visibility Modifiers

Visibility modifiers are used to control access to code elements. This helps keep code clean and modularized.

Classes/VisibilityModifiers.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//Private modifier prevents outside access
//By default, methods/properties are public
class Person(
    private val firstName: String,
    private val lastName: String,
    val age: Int
) {
    private fun getFullName() = "$firstName $lastName"

    fun printName() {
        println(getFullName())
    }
}

val person = Person("John", "Doe", 4)
person.printName()
println(person.age)

//Won't work (firstName is private)
//println(person.firstName)
```

Data Classes

Data classes are a special type of class used to store data.

Classes/DataClasses.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//Regular class
class Person(
    val firstName: String,
    val lastName: String
) {
    fun printName() = println("$firstName $lastName")
}

//Data class
data class PersonData(
    val firstName: String,
    val lastName: String
) {
    fun printName() = println("$firstName $lastName")
}

//Instantiate data classes
val personData1 = PersonData("John", "Doe")
val personData2 = PersonData("John", "Doe")

//Equal because constructor properties are same
println(personData1 == personData2)

//Only data classes have copy method
val personData3 = personData1.copy(firstName = "Jim")
personData3.printName()


//Instantiate regular classes
val person1 = Person("John", "Doe")
val person2 = Person("John", "Doe")

//Not equal because they're different objects
println(person1 == person2)
person1.printName()
person2.printName()
```

Enum Classes

Enum classes are a special type of class used to represent constants.

Classes/EnumClasses.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

enum class Continent {
    Africa,
    NorthAmerica,
    SouthAmerica,
    Europe,
    Asia,
    Oceania;

    //Enum constants can also have methods
    fun printYourself() {
        //name and ordinal are built-in enum properties
        println("Continent ${ordinal + 1}: $name")
    }
}

println(Continent.Africa)
println(Continent.NorthAmerica)
println(Continent.SouthAmerica)
println(Continent.Asia)

Continent.Oceania.printYourself()
Continent.Europe.printYourself()

println(Continent.values().toList())
```

Singletons

Singletons are special classes that are instantiated only once and remain instantiated until the termination of the program. Singletons are instantiated upon first access. All methods and properties in a singletons are accessed statically.

Classes/Singleton.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//To declare a singleton
object MyFirstSingleton {
    var counter = 0
```

```
    init {
        println("Singleton initialized for 1st time")
    }

    fun incrementAndPrint() {
        counter++
        println("Counter: $counter")
    }
}

//To access a singleton's properties/methods
MyFirstSingleton.incrementAndPrint()

//Counter will be 2 (singleton)
MyFirstSingleton.incrementAndPrint()
```

Customer Getters/Setters

Custom getters/setters can be used to control modifications of variables.

Classes/GetterSetters.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//Setter can be defined with/without getter and vice-versa
var name = ""
    set(value) {
        //Log the new value
        println("New name is $value")
        //Set variable equal to new value
        field = value
    }


//Variable type omitted because compiler can infer return type
val nameMessage1 get() = "The name is $name"

//Explicit typing mandatory (curly braces used)
val nameMessage2: String
    get() {
        return "The person's name is $name"
    }
```

```
//Can add (redundant) type annotation.
val nameMessage3 get(): String = "Your name is $name!"

name = "John"
println(nameMessage1)
println(nameMessage2)
println(nameMessage3)

name = "Jack"
println(nameMessage1)
println(nameMessage2)
println(nameMessage3)
```

Lazy Initialization

The lazy property modifier delays initialization of a variable until it is first accessed.

Classes/LazyInit.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//numberOfPeople won't be computed until first accessed
val numberOfPeople by lazy {
    println("Counting # of people")
    //Pause execution for 5 seconds (simulating work)
    Thread.sleep(5000)
    100_000_000
}

fun performCensus() {
    //First access of variable causes initialization
    val number = numberOfPeople
    println("Total people: $number")
    println("Count again: $numberOfPeople") //Already initialized
}

performCensus()
```

Late Initialization

Late initialization is used when a default value for a variable can't be provided, and it is undesirable to make the property nullable. Only non-primitive types (String, Char, Byte, Short, Int, Long, Float, Double) can be annotated with the lateinit modifier.

Classes/Lateinit.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//To create lateinit variable
lateinit var words: List

//Accessing words causes crash (not initialized):
//println(words)

fun loadWordsAndPrint() {
    //To see if a lateinit variable is initialized
    val isInitialized = ::words.isInitialized
    println("Initialized: $isInitialized")

    if (!isInitialized) {
        println("Initializing words")
        words = listOf("Kotlin", "Java", "Groovy", "Scala")
    }
    println(words)
}

loadWordsAndPrint()
loadWordsAndPrint()
```

Extension Properties

It is possible to add extra functionality to variable types using extension properties. Extension properties are like properties that are added to a class definition.

Classes/Extensions.ws.kts

```
package kotlin_reference_guide_for_beginners.classes

//Implicit return type
val String.reversedAndAllCaps get() = reversed().uppercase()
```

```
val String.wordCount
    //Explicit typing mandatory, because {} are used
    get(): Int {
        return split("").size
    }

//Explicit typing (redundant)
val String.distinctWordCount get(): Int = split("").distinct().size


println("John".reversedAndAllCaps)
println("Lorem impsum lorem impsum".distinctWordCount)
println("As a man thinketh so is he".wordCount)
```

Chapter 5: Collections

Collections are data structures that can contain multiple pieces of data in a list.

Note: Index of first element in collection is 0.

Lists

Lists are collections that store data in the order in which the data was described.

Collections/Lists.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val animals = listOf("Cat", "Dog", "Bird", "Squirrel", "Badger",
"Ferret")

//To access element at index
println(animals[0])
//To find if element contained
println("Cat" in animals)
//To find if element contained
println(animals.contains("Cat"))
//To get index of element (if element not found, -1 returned)
println(animals.indexOf("Pig"))

//To get first element
println(animals.first())
//To get last element
println(animals.last())
//To get last index
println(animals.lastIndex)

//To return size
println(animals.size)
//To find if empty
println(animals.isEmpty())
//To find if not empty
println(animals.isNotEmpty())

//To get list of first 2 elements
println(animals.take(2))
```

```
//To get list without first 2 elements
println(animals.drop(2))
//To get list of last 2 elements
println(animals.takeLast(2))
//To get list without last 2 elements
println(animals.dropLast(2))

//To get sublist
println(animals.subList(1, 3))
//To get reversed version
println(animals.reversed())
//To get shuffled version
println(animals.shuffled())
//To get random item
println(animals.random())

//To return copy
println(animals.toList())
//To return mutable copy
println(animals.toMutableList())
//To return set containing elements
println(animals.toSet())
//To return mutable set containing elements
println(animals.toMutableSet())

//To return string of elements
println(animals.joinToString("|"))

//To return list with element removed
println(animals - "Badger")
//To return list with element added
println(animals + "Canary")
```

Mutable Lists

Mutable lists are lists that allow adding and removing elements.

Collections/MutableLists.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val animals = mutableListOf("Cat", "Dog", "Bird")
val browsers = listOf("Firefox", "Edge", "IE", "Chrome")
```

```
//To remove element
animals.remove("Cat")
//To remove element at index
animals.removeAt(0)
//To add an element
animals.add("Pig")
//To add elements from another collection
animals.addAll(browsers)
//To see if contains elements in another collection
animals.containsAll(browsers)
//To set an item at specific index
animals[1] = "Doggy"
println(animals)

//To reverse
animals.reverse()
//To shuffle
animals.shuffle()
println(animals)

//To remove first element
animals.removeFirst()
//To remove last element
animals.removeLast()
println(animals)

//To replace all list items with a value
animals.fill("Animal")
println(animals)

//To remove all elements in a list
animals.clear()
```

Sets

Sets are collections that do not contain duplicate elements. The order of elements in sets usually doesn't matter.

Collections/Sets.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val mySet = setOf("Cat", "Dog", "Bird", "Squirrel", "Badger")
```

```
//To access item at index
println(mySet.elementAt(0))
//To find if item contained
println("Cat" in mySet)
//To find if item contained
println(mySet.contains("Cat"))
//To get index of item
println(mySet.indexOf("Pig"))

//To get first item
println(mySet.first())
//To get last item
println(mySet.last())

//To get size
println(mySet.size)
//To find if empty
println(mySet.isEmpty())
//To find if not empty
println(mySet.isNotEmpty())

//To get list of first 2 items
println(mySet.take(2))
//To get list without first 2 items
println(mySet.drop(2))

//To get reversed version (as list)
println(mySet.reversed())
//To get shuffled version (as list)
println(mySet.shuffled())
//To get random item
println(mySet.random())

//To return a copy (as list)
println(mySet.toList())
//To return mutable copy (as list)
println(mySet.toMutableList())
//To return set containing items
println(mySet.toSet())
//To return mutable set containing items
println(mySet.toMutableSet())

//To return string of set's items
println(mySet.joinToString("|"))
//To return result of merging 2 sets
println(mySet union setOf("Pencil", "Pen"))
```

```
//To return what's shared in 2 sets
println(mySet intersect setOf("Pencil", "Cat", "Dog", "Pen"))
//To return what's not shared in 2 sets
println(mySet subtract setOf("Pencil", "Cat", "Dog", "Pen"))
```

Mutable Sets

Mutable sets are sets that allow adding and removing elements.

Collections/MutableSets.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val animals = mutableSetOf("Cat", "Dog", "Bird")
val browsers = setOf("Firefox", "Edge", "IE", "Chrome", "Cat")

//To add an element
animals.add("Pig")

//To add elements from another collection
animals.addAll(browsers)

//To see if a set contains elements in another set
animals.containsAll(browsers)

//To remove an element
animals.remove("Cat")

println(animals)

//To remove all elements in a set
animals.clear()
```

Maps

Maps are Collections that **map** a **key** to a **value**

Collections/Maps.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val carmakers = mapOf(
```

```
    "Volvo" to "Sweden",
    "Audi" to "Germany",
    "Renault" to "France",
    "Ford" to "USA",
) //key to value; keys must be unique


//To find value for key
println(carmakers["Volvo"])
//To find if key contained
println("Ford" in carmakers)
//To find if key contained
println(carmakers.contains("Audi"))
//To find if value contained
println(carmakers.containsValue("Japan"))

//To get size
println(carmakers.size)
//To find if empty
println(carmakers.isEmpty())
//To find if not empty
println(carmakers.isNotEmpty())

//To return list with keys and values
println(carmakers.toList())
//To return copy
println(carmakers.toMap())
//To get keys
println(carmakers.keys)
//To get values
println(carmakers.values)

//To get a map without a specific key
println(carmakers - "Audi")
//To get a map without specific keys
println(carmakers - listOf("Audi", "Ford"))
//To get map with a key/value pair added
println(carmakers + ("Mazda" to "Japan"))
```

Mutable Maps

Mutable maps are maps that allow adding and removing elements.

Collections/MutableMaps.ws.kts

```kotlin
package kotlin_reference_guide_for_beginners.collections

val carMakers = mutableMapOf(
    "Volvo" to "Sweden",
    "Audi" to "Germany",
    "Renault" to "France",
    "Ford" to "USA",
) //key to value; keys must be unique

val asianCarmakers = mapOf(
    "Honda" to "Japan",
    "KIA" to "South Korea",
    "Suzuki" to "Japan",
    "Nissan" to "Japan"
)

val americanCarmakers = mapOf(
    "Ford" to "USA",
    "Tesla" to "USA",
    "General Motors" to "USA",
)

//To add a key with its value
carMakers["Mazda"] = "Japan"
//To add contents of one map to another
carMakers.putAll(asianCarmakers)
//To add contents of one map to another
carMakers += americanCarmakers
//To remove a key (and its associated value)
carMakers.remove("Ford")
//To remove a key (and its associated value)
carMakers -= "KIA"

println(carMakers)

//To empty a map
carMakers.clear()
```

Ranges

Ranges represent a range of values.

Collections/Ranges.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val range1 = 1..20

//To return start of range
println(range1.first)
//To return end of range
println(range1.last)
//To return step of range
println(range1.step)

//To find if range contains number
println(range1.contains(34))
//To return random element from range
println(range1.random())

//To return list containing range elements
println(range1.toList())
//To return mutable list containing range elements
println(range1.toMutableList())
//To return set containing range elements
println(range1.toSet())
//To return mutable set containing range elements
println(range1.toMutableSet())

//To create a range with step
val range2 = 'A'..'Z' step 5
//To return step of range
println(range2.step)
println(range2.toList())
println((50 downTo 7).toList())
println((50 downTo 7 step 3).toList())
```

Filtering Operations on Collections

Collections/FilterCollections.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

//Predicate: lambda function { } returning Boolean for each item
val list = (1..50).toList()

//To return a list items matching predicate
println(list.filter { it % 2 == 0 }) //"it" = list item
```

```
//To override default name "it" in a lambda
println(list.filter { number -> number != 5 })
//To return list items by index and value matching predicate
println(list.filterIndexed { index, i -> index != 0 && i != 30 })
//To return list of elements that don't match predicate
println(list.filterNot { it % 2 == 0 })


val strangeList = listOf("Cracker", 45, 1..4, "Tomato", 1245, null,
null)
//To return list items of a specific type
println(strangeList.filterIsInstance(String::class.java))
//To return list of non-null items
println(strangeList.filterNotNull())

//To find if all list items match criteria
println(strangeList.all { it is String || it is Int || it is
IntRange || it == null })
//To find if any list items match criteria
println(strangeList.any { it == "Dog" })
//To find if no list items match criteria
println(strangeList.none { it is Char })
```

Sorting Operations on Collections

Collections/SortCollections.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val words = mutableListOf("zone", "kabob", "Alexandra", "apple",
"even")

//To return sorted version of list (ascending)
println(words.sorted())

//To return sorted version of list (descending)
println(words.sortedDescending())

//To return list sorted by length (ascending)
println(words.sortedWith(compareBy { it.length }))

//To return list sorted by length (descending)
println(words.sortedWith(compareBy { 1 - it.length }))
```

```
//To return list sorted by length (ascending)
println(words.sortedBy { it.length })

//To return list sorted by length (descending)
println(words.sortedByDescending { it.length })
```

Sorting Operations on Mutable Collections

Collections/SortMutableCollections.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val letters = mutableListOf("z", "g", "mlm", "almh", "Aqwery")

//To sort list (ascending)
letters.sort()
println(letters)

//To sort list (descending)
letters.sortDescending()
println(letters)

//To sort list length (ascending)
letters.sortWith(compareBy { it.length })
println(letters)

//To sort list by length (descending)
letters.sortWith(compareBy { 1 - it.length })
println(letters)

//To sort list by length (ascending)
letters.sortBy { it.length }
println(letters)

//To sort list by length (descending)
letters.sortByDescending { it.length }
println(letters)
```

Aggregate Operations on Collections

Collections/AggregateCollections.ws.kts

```
package kotlin_reference_guide_for_beginners.collections

val numbers = listOf(5, 123, 98, 234, 3, 9)

//To get max of list items (null returned if collection empty)
println(numbers.maxOrNull())

//To get min of list items (null returned if collection empty)
println(numbers.minOrNull())

//To get average of list items
println(numbers.average())

//To get sum of list items
println(numbers.sum())
```

Chapter 6: Strings and Characters

Strings hold text. Below are important methods for working with Strings.

Concatenation Operations

Strings/Concatenation.ws.kts

```
package kotlin_reference_guide_for_beginners.strings

val greeting = "Hello world"

//To concatenate strings
val addedStrings1 = greeting + ". How are you?"
println(addedStrings1)

//To concatenate strings
val addedStrings2 = "$greeting. How are you?"
println(addedStrings2)

//To concatenate strings
val addedStrings3 = "${greeting.uppercase()}. How are you?"
println(addedStrings3)
```

Casing Operations

Strings/Casing.ws.kts

```
package kotlin_reference_guide_for_beginners.strings

val greeting = "Hi! My name is John."

//To make uppercase
println(greeting.uppercase())

//To make lowercase
println(greeting.lowercase())

//To reverse
println(greeting.reversed())
```

Substring Operations

Strings/Substring.ws.kts

```
package kotlin_reference_guide_for_beginners.strings

val greeting = "Hi! My name is John"

//To get first character
println(greeting.first())

//To get last character
println(greeting.last())

//To get x first characters
println(greeting.take(4))

//To get x last characters
println(greeting.takeLast(4))

//To get string without x first characters
println(greeting.drop(4))

//To get string without x last characters
println(greeting.dropLast(4))

//First element/character = at index 0
//To get a section of a string
println(greeting.substring(0, 2)) //from 1st char up to, but not 3rd char

//To get a section of a string
println(greeting.substring(1)) //from 2nd char to the end
```

Search and Replace Operations

Strings/SearchReplace.ws.kts

```
package kotlin_reference_guide_for_beginners.strings

val text = "Lorem ipsum text"

//To find if string starts with some text (case-sensitive)
```

```
println(text.startsWith("lor"))

//To find if string starts with some text (case-sensitive)
println(text.startsWith("lor", ignoreCase = false))

//To find if string starts with some text (case-insensitive)
println(text.startsWith("lor", ignoreCase = true))

//To find if string ends with some text (case-sensitive)
println(text.endsWith("T"))

//To find if string ends with some text (case-sensitive)
println(text.endsWith("T", ignoreCase = false))

//To find if string ends with some text (case-insensitive)
println(text.endsWith("T", ignoreCase = true))

//To find if string contains some text (case-sensitive)
println(text.contains("Ipsum"))

//To find if string contains some text (case-sensitive)
println(text.contains("Ipsum", ignoreCase = false))

//To find if string contains some text (case-insensitive)
println(text.contains("Ipsum", ignoreCase = true))

//To find if a string contains some text (case-sensitive)
println("ipsum" in text)

//To replace some text in a string (case-sensitive)
println(text.replace("lorem", "Merol"))

//To replace some text in a string (case-sensitive)
println(text.replace("lorem", "Merol", ignoreCase = false))

//To replace some text within a string (case-insensitive)
println(text.replace("lorem", "Merol", ignoreCase = true))
```

Whitespace Operations

Strings/Whitespace.ws.kts

```
package kotlin_reference_guide_for_beginners.strings
```

```
val myString = "   Hi my name is John  "

//To find if has no characters
println(myString.isEmpty())
//To find if have characters
println(myString.isNotEmpty())
//To find if only whitespace
println(myString.isBlank())
//To find if not only whitespace
println(myString.isNotBlank())

//To trim whitespace from start
println(myString.trimStart())
//To trim whitespace from end
println(myString.trimEnd())
//To trim whitespace from both sides
println(myString.trim())
```

Conversion Operations

Strings/Conversions.ws.kts

```
package kotlin_reference_guide_for_beginners.strings

val numberString = "89"

//To convert to Byte if possible or crash
println(numberString.toByte())
//To convert to Byte if possible or null
println(numberString.toByteOrNull())
//To convert to Short if possible or crash
println(numberString.toShort())
//To convert to Short if possible or null
println(numberString.toShortOrNull())

//To convert to Int if possible or crash
println(numberString.toInt())
//To convert to Int if possible or null
println(numberString.toIntOrNull())
//To convert to Long if possible or crash
println(numberString.toLong())
//To convert to Long if possible or null
println(numberString.toLongOrNull())
```

```
//To convert to Float if possible or crash
println(numberString.toFloat())
//To convert to Float if possible or null
println(numberString.toFloatOrNull())
//To convert to Double if possible or crash
println(numberString.toDouble())
//To convert to Double if possible or null
println(numberString.toDoubleOrNull())

val booleanText = "tRue"

//To convert to Boolean (lenient)
//toBoolean returns true if: string not null; and text = "true" (cas
e-insensitive)
println(booleanText.toBoolean())

//To convert to Boolean (strict, exception-safe)
//toBooleanStrictNull returns null if text is neither "true" or "fal
se" (case-sensitive)
println(booleanText.toBooleanStrictOrNull())

//To convert to Boolean (strict)
//toBooleanStrict crashes if text is neither "true" or "false" (case
-sensitive)
println(booleanText.toBooleanStrict())
```

Other String Operations

Strings/Other.ws.kts

```
package kotlin_reference_guide_for_beginners.strings

val greeting = "Hi! My name is John"

//To get length
println(greeting.count())

//To get random character
println(greeting.random())

//To repeat
println(greeting.repeat(3))

//To string by delimiter(s)
```

```
println(greeting.split(" ", "!"))

//To chunk into similarly sized pieces (max length = 4)
println(greeting.chunked(4))
```

Escape Sequences and Raw Strings

Strings/Escape.ws.kts

```
package kotlin_reference_guide_for_beginners.strings

//Escape sequences used to add special characters

//To add linebreak
val stringWithLinebreaks = "Hi\nMy\nname\nis John"
println(stringWithLinebreaks)

//To add tab
val stringWithTabs = "Stockholm\tSweden\t\nGothenburg\tSweden"
println(stringWithTabs)

//To add quote
val stringWithQuotes = "John said \"How awesome\""
println(stringWithQuotes)

//To add reverse backslashes
val stringWithBackSlashes = "dogs.com\\whatever\\whatever"
println(stringWithBackSlashes)

//Raw string can contain newlines and any text (no escaping)
val rawString = """
    Hi,     my name is John.
    Quote from John "I love Kotlin"
    Kotlin\Java\Groovy\Scala
    I want a million ${'$'}
""".trimIndent() //to remove extra whitespace in front of each line in the block
println(rawString)
```

Chapter 7: Numbers and Math

Number Types

There are different number types for holding Numbers of different sizes

Type	Min value	Max value	Decimal
Byte	-128	127	No
Short	-32768	32767	No
Int	-2,147,483,648	2,147,483,647	No
Long	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	No
Float	1.4E-45F	3.4028235E38F	Yes
Double	4.9E-324	1.7976931348623157E308	Yes

The default number type is Int. However, if the number in question is too large to be an Int, the type will be set to Long. Defining a Byte or a Short requires explicit typing.

Whole Numbers

Numbers/WholeNumbers.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

//To create a byte (rarely used)
val myByte: Byte = 1
println(myByte)

//To create a short (rarely used)
val myShort: Short = 1
println(myShort)

//To create an integer
val myInteger = 1
println(myInteger)
```

```
//To create a long
val myLong1 = 1L
println(myLong1)

//To create a long (number too large for Int)
val myLong2 = 10000000000000000
println(myLong2)

//To separate digits for any number type
val myLong3 = 1_000_000_000_000_000
println(myLong3)
```

Decimal Numbers

Numbers/DecimalNumbers.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

//To create float
val myFloat1 = 1F
println(myFloat1)

//To create float
val myFloat2 = 1f
println(myFloat2)

//To create double (mustn't end in f/F)
val myDouble = 1.0
println(myDouble)
```

Arithmetic Operations

PEDMAS:

Numbers/Arithmetic.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

import kotlin.math.pow

//To perform PEDMAS operations (to multiply, use *)
val calculation = 4 * 2 + (4 - 5) / 19
println(calculation)
```

```
//To find the remainder of a quotient
val remainder = 13F % 7F
println(remainder)

//To raise to power (number must be fractional)
val fiveRaisedTo10 = 5.0.pow(10)
println(fiveRaisedTo10)
```

Incrementing and Decrementing

Numbers/IncrementDecrementAssignment.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

var number = 50

//To increment by 1
number++
println(number)

//To decrement by 1
number--
println(number)

//To increment by any amount
number += 5
println(number)

//To decrement by any amount
number -= 5
println(number)

//To multiply by any amount and set to that product
number *= 5
println(number)

//To divide by any amount and set to that quotient
number /= 10
println(number)
```

Quirky Division

Numbers/Division.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

//Dividing integers returns integer
val twoDividedByFive = 2 / 5 //will equal zero, not 0.4
println(twoDividedByFive)

//If we want the actual answer, use fractional types
val twoDividedByFiveDouble = 2.0 / 5.0
println(twoDividedByFiveDouble)

//If we want the actual answer, use fractional types
val twoDividedByFiveFloat = 2F / 5F
println(twoDividedByFiveFloat)
```

Mixing Number Types

Numbers/MixingNumberTypes.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

//Resulting number will be of largest fractional type used
val equation1 = 30F + 5 / 9.0 * 90L //Double
println(equation1::class)

val equation2 = 30F + 5 / 9 * 90L //Float
println(equation2::class)
```

Converting Numbers

To convert between number types, use conversion methods. Every number type has the following conversion methods:

- toByte() (converts number to Byte)
- toShort() (converts number to Short)
- toInt() (converts number to Int)
- toLong() (converts number to Long)
- toFloat() (converts number to Float)
- toDouble() (converts number to Double)

Numbers/Conversions.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

val numberToConvert = 500

val myByte = numberToConvert.toByte()
//500 is too big to fit in a Byte
println("Type: ${myByte::class}, Value: $myByte")

val myShort = numberToConvert.toShort()
println("Type: ${myShort::class}, Value: $myShort")

//Unnecessary conversion, numberToConvert is already an Int
val myInt = numberToConvert.toInt()
println("Type: ${myInt::class}, Value: $myInt")

val myLong = numberToConvert.toLong()
println("Type: ${myLong::class}, Value: $myLong")

val myFloat = numberToConvert.toFloat()
println("Type: ${myFloat::class}, Value: $myFloat")

val myDouble = numberToConvert.toDouble()
println("Type: ${myDouble::class}, Value: $myDouble")
```

Constants

Numbers/Constants.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

//To obtain PI constant
val pi = Math.PI
println(pi)

//To obtain Euler's number
val euler = Math.E
println(euler)
```

Signs and Absolute Value

Numbers/SignsAbsoluteValue.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

import kotlin.math.abs
import kotlin.math.absoluteValue
import kotlin.math.sign

//To find absolute value
val absValue1 = abs(-107)
println(absValue1)

//To find absolute value
val absValue2 = 500.absoluteValue
println(absValue2)

//To find sign (fractional)
val sign1 = sign(-900.0)
println(sign1)

//To find sign
val sign2 = 900.sign
println(sign2)
```

Minimum/Maximum and Rounding

Numbers/MinimumMaximumRounding.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

import kotlin.math.ceil
import kotlin.math.floor

//To find maximum of 2 numbers
val max = maxOf(70, 100)
println(max)

//To find minimum of 2 numbers
val min = minOf(-90, 78)
println(min)

//To round down (fractional)
val floor = floor(4.24)
println(floor)
```

```
//To round up (fractional)
val ceiling = ceil(4.24)
println(ceiling)
```

Logarithms

Numbers/Logarithms.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

import kotlin.math.ln
import kotlin.math.log
import kotlin.math.log10
import kotlin.math.sqrt

//To find square root (fractional)
val sqrt = sqrt(78.toDouble())
println(sqrt)

//To find common log (fractional)
val commonLog = log10(45.0)
println(commonLog)

//To find natural log of a number (fractional)
val naturalLog = ln(45F)
println(naturalLog)

//To find log of number (number & base must be fractional)
val log = log(45F, 2F)
println(log)
```

Trigonometry

Numbers/Trigonometry.ws.kts

```
package kotlin_reference_guide_for_beginners.numbers

import kotlin.math.*

println(Math.toRadians(5.0))
println(Math.toDegrees(0.08726646259971647))
```

```
//To find sin (angle in radians, fractional)
val sin = sin(0.08726646259971647)
println(sin)

//To find cos (angle in radians, fractional)
val cos = cos(0.08726646259971647)
println(cos)

//To find tan (angle in radians, fractional)
val tan = tan(0.08726646259971647)
println(tan)


//To find arcsin of a value (fractional)
val arcsin = Math.toDegrees(asin(0.08715574274765817))
println(arcsin)

//To find arccos of a value (fractional)
val arccos = Math.toDegrees(acos(0.9961946980917455))
println(arccos)

//To find arctan of a value (fractional)
val arctan = Math.toDegrees(atan(0.087488663525924))
println(arctan)
```

Chapter 8: Polymorphism

Polymorphism refers to grouping classes together in a hiearchy. Classes can inherit methods and properties from other classes. Interfaces define common structures for classes.

- Any: the superclass of all Kotlin classes.
- Abstract class: a class that defines a common protocol for subclasses and can **not** be instantiated.
- Concrete class: opposite of abstract; can be instantiated.

Interfaces

Interfaces are code structures that define common methods and properties for subclasses. Properties in an interface can not have default values. Classes can inherit any number of interfaces, but can only inherit from a maximum of 1 class.

Polymorphism/Interfaces.ws.kts

```
package kotlin_reference_guide_for_beginners.polymorphism

interface Disk {
    //Interface properties can't be pre-assigned
    //Must be implemented in a concrete subclass
    var diskName: String
}

interface Screen {
    //Must be implemented in a concrete subclass
    var brightness: Int

    //Must be implemented in a concrete subclass
    fun changeBrightness(newBrightness: Int)

    //No need to implement (already implemented)
    fun turnScreenOn() {
        println("Screen on! Brightness: $brightness")
    }
}
```

```
interface Processor {

    fun getCores() = 4
}


class Computer : Processor, Disk, Screen {

    override var diskName = "C:"

    override var brightness = 0

    //Must implement this method (no default method available)
    override fun changeBrightness(newBrightness: Int) {
        brightness = newBrightness
    }

    //Can override a default method
    override fun getCores() = 8
}

val computer = Computer()

println(computer.getCores())
println(computer.diskName)

computer.turnScreenOn()
println(computer.brightness)

computer.changeBrightness(5)
println(computer.brightness)

println(computer is Disk)
println(computer is Screen)
println(computer is Processor)
```

Open Classes

Open classes allow for overriding and can participate in a class hiearchy.

Polymorphism/Open.ws.kts

```
package kotlin_reference_guide_for_beginners.polymorphism
```

```
open class Animal(val age: Int) {
    //Protected: accessible only by Animal + subclasses
    //Open means can be overridden
    protected open val name = "Animal"

    open fun eatFood() = println("fEASTing on Food")

    fun printSpecies() = println("I'm a: $name")
}

class Dog(age: Int) : Animal(age) /*pass constructor arguments*/ {

    //Can override val property with a var (NOT VICE-VERSA)
    override var name = "Doggy"

    override fun eatFood() {
        //To call superclass implementation of eatFood
        super.eatFood()
        println("Very tasty")
    }

    fun changeName(newName: String) {
        name = newName
    }
}

val animal = Animal(78)
animal.eatFood()
animal.printSpecies()

//Not all animals are dogs!
println(animal is Dog)

val dog = Dog(120)

dog.eatFood()
dog.printSpecies()
dog.changeName("Bloodhound")
dog.printSpecies()

println(dog.age)
println(dog is Animal)
```

Abstract Classes

Abstract classes are special classes that define common methods and properties for subclasses and are open by default.

Polymorphism/Abstract.ws.kts

```
package kotlin_reference_guide_for_beginners.polymorphism

abstract class Democracy {

    //Abstract members must be implemented in concrete subclasses
    abstract val type: String

    //Abstract functions have no body
    abstract fun holdElections()

    abstract fun holdReferendum()

    fun challengeLaw() {
        println("Challenge the unconstitutional law!")
    }
}

class Canada : Democracy() {
    //Must implement all abstract members
    override val type = "Federation/constitutional monarchy"

    override fun holdElections() {
        println("Choose your leader!")
    }

    override fun holdReferendum() {
        println("Government asks citizen a question")
    }
}

//Can't instantiate an abstract class:
//val democracy = Democracy()
val canada = Canada()
println(canada.type)

canada.holdElections()
canada.holdReferendum()
canada.challengeLaw()
```

Classes with Multiple Parent Classes and Interfaces

It is possible for a class to inherit from a parent class that inherits from another class and so on.

Polymorphism/MultipleInheritance.ws.kts

```
package kotlin_reference_guide_for_beginners.polymorphism

abstract class Country {
    abstract val officialName: String

    abstract fun holdElections()

    abstract fun electHeadOfState(president: String)

    open fun sendAid(country: String) {
        println("Sending aid to $country")
    }
}

abstract class FormerUSSRState : Country() {
    abstract val percentageOfRussianSpeakers: Int

    open fun switchToCapitalism() {
        println("This is going to hurt!")
    }
}

interface GasExporter {
    val gasUtility: String

    fun exportGas(country: String) {
        println("$gasUtility exporting gas to $country")
    }
}

class Russia : FormerUSSRState(), GasExporter {
    override val officialName = "Russian Federation"
    override val percentageOfRussianSpeakers = 80
    override val gasUtility = "Gazprom"

    override fun holdElections() = println("Choose your leader")
```

```
    override fun electHeadOfState(president: String) {
        println("$president is president of $officialName")
    }
}

val russia = Russia()
println(russia is GasExporter)
println(russia is FormerUSSRState)
println(russia is Country)

println(russia.gasUtility)
println(russia.officialName)
println(russia.percentageOfRussianSpeakers)

russia.holdElections()
russia.electHeadOfState("Vladimir Putin")
russia.switchToCapitalism()
russia.sendAid("Syria")
russia.exportGas("Germany")
```

Chapter 9: File IO

File I/O refers to performing operations on files.

Working with Files

Following are important methods for working with files:

FileIO/File.ws.kts

```
package kotlin_reference_guide_for_beginners.fileio

import java.io.File

//Represents a file called "names.txt". Does not create file!
val path = File("names.txt")

//To return name
println(path.name)

//To return name minus extension
println(path.nameWithoutExtension)

//To return extension if any
println(path.extension)

//To return if file exists
println(path.exists())

//To return if path points to an actual file (the opposite of a fold
er)
println(path.isFile)

//To return if path points to directory
println(path.isDirectory)

//To return the absolute path of file
println(path.absolutePath)

//To return text read from file
println(path.readText())
```

```
//To create empty file
path.createNewFile()

//To write text to file
path.writeText("Steve\nJobs\nBill\nGates")
```

Working with Folders

Following are important methods for working with folders:

FileIO/Folder.ws.kts

```
package kotlin_reference_guide_for_beginners.fileio

import java.io.File

//Represents a folder called "folder". Does not create folder!
val path = File("folder")

//To return name
println(path.name)

//To return name minus extension
println(path.nameWithoutExtension)

//To return if folder exists
println(path.exists())

//To return if path points to an actual file (opposite of a folder)
println(path.isFile)

//To return if path points to a directory
println(path.isDirectory)

//To return absolute path of folder
println(path.absolutePath)

//To return files and subdirectories in folder as Files
println(path.listFiles()?.toList())

//To return files and subdirectories in folder as Strings
println(path.list()?.toList())
```

Conclusion

Thanks for reading this book. I hope you enjoyed the learning experience. I would greatly appreciate if you could leave a positive review online to help spread the word.