Programació multimèdia
i dispositius mòbils

4. Estructures de control. Entrada i eixida.

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2021-22

Continguts

1 Estructures de control

Veiem les diferents estructures de control que podem utilitzar tant a Java com a Kotlin:

1.1 Estructures condicionals

1.1.1 if-else

Aquesta estructura de control és igaul en Java que en Kotlin:

```
if (expressióLògica) {
    bloc_de_sentències_si_expressió_avalua_a_true;
} else {
    bloc_de_sentències_si_expressió_avalua_a_false;
}
```

Recordeu, que a més, l'estructura `if-then-else` pot usar-se a Kotlin com a una expressió.

1.1.2 Switch i When

Els `switch` s'utilitzen en Java, i **no existeixen a Kotlin**. La seua sintaxi és:

```
switch (variable)
    case valor_1:
        sentències_si_variable==valor_1;
        break;
    case valor_2:
        sentències_si_variable==valor_2;
        break;
    ...
    case valor_n:
        sentències_si_variable==valor_n;
        break;
    default:
        sentències_si_cap_valor_es_compleix;
        break;
        }
    }
}
```

L'estructura que més s'assemblaria al `switch` de Java en Kotlin és el `when`. Els `when` en Kotlin podriem dir que es tracten de *`switch` supervitaminats*. Es tracta d'una construcció que pot utilitzar-se tant com a sentència com a expressió.

La sintaxi bàsica del `when` podría expressar-se de les formes següents:

- **Com a sentència**: Més o menys com fariem un `switch` amb altres llenguatges, però amb sintaxi diferent:

```
when (expressióValor){
    valor1 -> sentencies_si_Valor1
    valor2 -> sentencies_si_Valor2
    ...
    valorN -> sentencies_si_ValorN
    else -> sentencies_default
}
```

En aquesta construcció l'`else` seria com el `default` al `switch` de Java, i el seu ús no és obligatori.

- **Com a expressió**: Kotlin suporta el paradigma de la **programació funcional** de forma bastant bàsica. Als llenguatges de programació funcional, les estructures de control són expressions, de manera que el resultat de la seua avaluació pot ser retornat a qui l'invoca. Si aquest valor s'assigna a una variable, el compilador comprovarà que el tipus retornat és compatible amb l'esperat i ens informarà si no és el cas. Així doncs, podem utilitzar:

```
var x=when (expressióValor){
    valor1 -> valor_per_a_x_si_valor1
    valor2 -> valor_per_a_x_si_valor2
    ...
    valorN -> valor_per_a_x_si_valor1
    else -> valor_per_a_x_per_defecte
}
```

En aquest cas, s'avaluarà `expressióValor`, i en funció del resultat (`valor1`, `valor2`...) s'assignarà un o altre valor a `x`. En aquest cas, l'ús de `else` sí que és necessari, ja que necessàriament caldrà assignar un valor a `x`.

Veiem algun exemples d'ús per veure-ho més clar:

Exemple: Ús de `when` com a sentència. Suposant que estem dins un `main` amb arguments:

```
when (args[0]){
    "Hola Don Pepito" -> println("Hola Don José")
    "Pasó usted por mi casa" -> println("Por su casa yo pasé")
    "Y vio usted a mi abuela" -> println("A su abuela yo la vi")
    "Adiós don Pepito" -> println("Adiós don José")
}
```

Exemple: Ús de when com a expressió:

```
var resposta=when (args[0]){
    "Hola Don Pepito" -> "Hola Don José"
    "Pasó usted por mi casa" -> "Por su casa yo pasé"
    "Y vio usted a mi abuela" -> "A su abuela yo la vi"
    "Adiós don Pepito" -> "Adiós don José"
    else -> "No entenc la pregunta"
}
println(resposta)
```

Per altra banda, és possible que volgam afegir més d'una sentència segons les condicions. Aleshores, podriem utilitzar les claus {} per delimitar blocs de sentències:

```
resposta=when (args[0]){
    "Hola Don Pepito" -> {
        println("coincidència al bloc 1")
        "Hola Don José"}
    "Pasó usted por mi casa" -> {
        println("coincidència al bloc 2")
        "Por su casa yo pasé"}
    "Y vio usted a mi abuela" -> {
        println("coincidència al bloc 3")
        "A su abuela yo la vi"}
    "Adiós don Pepito" -> {
        println("coincidència al bloc 4")
        "Adiós don José"}
    else -> {
        println("coincidència al bloc del else -default-")
        "No entenc la pregunta"}
}
println(resposta)
```

Però when encara ens guarda algunes funcionalitats més bastant potents:

When sense arguments

When pot utilitzar-se sense arguments, de manera que ens servisca com a alternativa al `if-then-else`:

```
when{
    temperatura < 15 -> println("Fa fred")
    temperatura in 15..24 -> println("S'està bé")
    temperatura > 25 -> println("Fa calor")
}
```

Operadors `is` i `in`

- Amb l'operador `is` podem determinar la classe d'una variable. Quan l'utilitzem a la part esquerra de la `->`, a la part dreta ja tindrem el càsting fet. Cal dir que per tal de fer aquest *smart cast* correctament i sense errors, la variable ha de ser de tipus *Any*.

```
var variable:Any=Valor

when(variable){
    is Int -> println("${variable} és un enter")
    is Char -> println("${variable} és un caràcter")
    is String -> println("${variable} és una cadena")
    else -> println("${variable} és d'altre tipus")

}
```

- Amb l'operador `in` podem comprovar si el valor està dins un rang. Per exemple:

```
when (mes){
    in 1..3 -> println("Estem en hivern")
    in 4..6 -> println("Estem en primavera")
    in 7..9 -> println("Estem en estiu")
    in 10..12 -> println("Estem en tardor")
}
```

Veiem en aquest exemple com és de senzill definir rangs amb Kotlin fent ús de l'operador `...`

- A més, també podem especificar valors concrets dins un mateix cas:

```
var dia = 4
when(dia) {
    1, 2, 3, 4, 5 -> println("Hui es treballa")
    6, 7 -> println("Hui és cap de setmana")
    else -> println("El dia no és correcte")
}
```

Podeu trobar més informació sobre el when a Kotlin a l'article https://devexperto.com/expresion-when-kotlin/

1.2 Estructures repetitives

1.2.1 Bucles for

- Es repeteix un bloc de sentències mentre es compleix la condició de repetició.
- La inicialització, la condició i la iteració es realitzen en la mateixa instrucció.

En ***Java***, la sintaxi del `for` és la següent:

```
for (inicialització; condició_de_repetició; iteració)
    Bloc_de_sentències
```

En **Kotlin**, la sintaxi és lleugerament diferent, i fa ús de l'operador `in`:

```
for (element in conjuntDeValors)
    Instruccions
```

Anb açò podem fer coses com:

- Recórrer una cadena de caràcters o un vector:

```
for (caracter in "Hola Mon")  println(caracter)
```

- Recórrer un rang de valors:

```
for (i in 1..10) {
    println(i)
}
```

O bé, fent ús de la funció d'extensió `rangeTo()`, que seria equivalent:

```
for (i in 1.rangeTo(10)) {
    println(i)
}
```

- Recórrer un rang de valors saltejant valors:

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

- Utilitzar les expressions downTo per iterar números en ordre invers:

```
for (i in 10 downTo 0 step 3) {
    println(i)
}
```

Finalment, fixeu-vos que no hem indicat en cap moment el tipus de la variable que fa d'índex dels bucles. Aquest tipus és inferit per Kotlin segons els valors que li donem, i de fet, **no ens permet indicar-lo en la declaració**.

1.2.2 Bucles while i do-while

Els bucles `while` i `do-while` tenen la mateixa sintaxi a Kotlin que a Java.

Recordeu que als bucles `while`:

- La inicialització es realitza prèviament a la sentència while.
- Si la primera vegada l'expressió s'avalúa a fals, no s'executa el bloc de sentències cap vegada.

```
[inicialització;]
while (expressió) {
    Bloc_de_sentències;
    [iteració]
}
```

Per la seua banda, als bucles `do-while`

- El bloc de sentències s'exeuta al menys una vegada, ja que l'expressió s'avalúa al finalitzar aquest.

```
do {
    Bloc_de_sentències;
    [iteració]
} while (expressió);
```

1.2.3 Sentències break i continue

- Poden trobar-se dins el bloc de sentències dels bucles.
- **break** força l'eixida del bucle.
- **continue** fa que es passe a la següent iteració, sense acabar d'executar la resta de sentències.

Amb Kotlin, a més podem fer ús d'***etiquetes*** per a aquestes sentències. Una etiqueta no és més que un identificador seguit del signe @, al que podem fer referència quan fem un `break` o un `continue`, posant l'@ dvant. Veiem-ho amb alguns exemples:

```
bucleExtern@ for (i in 1..10) {
    for (j in 1..10) {
        if (condició) break@bucleExtern
    }
}
```

El següent codi:

```
for (i in 1..5){
  for (j in 1..5){
      println("${i}, ${j}")
      if (j>2) break
  }
}
```

Tindrà l'eixida:

```
1, 1
1, 2
1, 3
```

```
2, 1
2, 2
2, 3
3, 1
3, 2
3, 3
4, 1
4, 2
4, 3
5, 1
5, 2
5, 3
```

Mentre que si fem:

```
bucleExt@ for (i in 1..5){
  for (j in 1..5){
    println("${i}, ${j}")
      if (j>2) break@bucleExt
  }
}
```

Obtindrem:

```
1, 1
1, 2
1, 3
```

2 Tractament d'excepcions

Les excecions representen situacions anòmales o problemes al nostre codi. En Java, podem distingir en general dos tipus d'excepcions:

- Els errors: situacions irrecuperables que provoquen l'aturada del programa.
- Les excepcions en sí, provocades per alguna situació anòmala, i que podem controlar.

Al següent diagrama de classes podem vore la jerarquía de les diferents classes d'errors que ens podem trobar a Java:

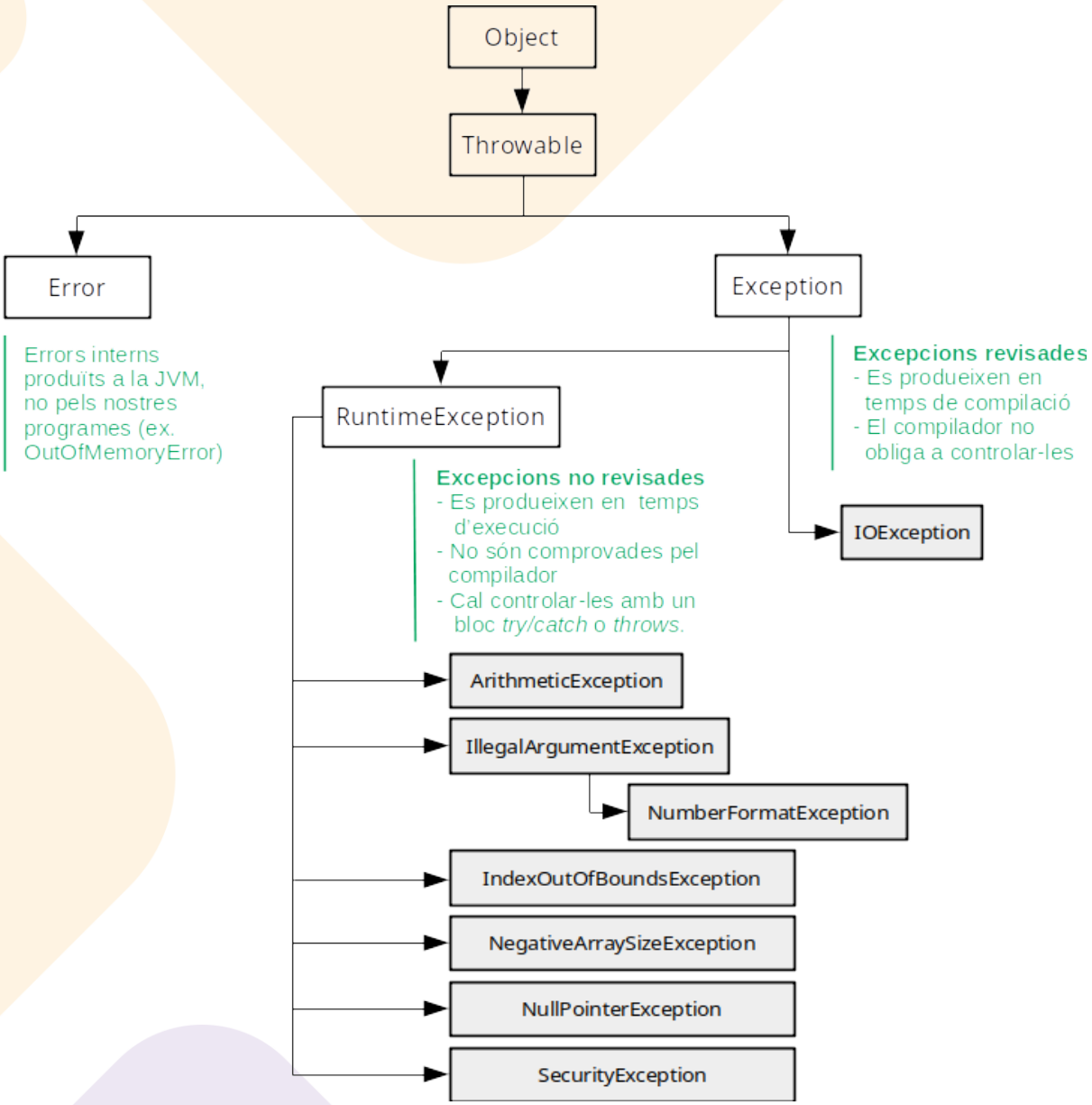

Figura 1: Excepcions en java

Veiem com totes les excepcions deriven de la classe `Throwable`, i d'aquesta deriven les classes `Error` i `Exception`.

A més, podem distingir dos tipus diferents d'excepcions a partir de la classe `Exception`:

- Les excepcions **revisades**, que hereten directament de `java.lang.Exception` i que són **detectades en temps de compilació**, de manera que el compilador ens obliga a controlar-les amb un bloc *try/catch* o rellançar-les amb *throws*. Els mètodes que puguen provocar aquest

tipus d'excepcions ho han d'indicar amb la paraula clau `throws`, així com els mètodes que invoquen a mètodes que puguen llançar estes excepcions, hauran bé de capturar-les o rellançar-les. Exemples d'aquest tipu sd'excepcions són `IOException`, `ClassNotFoundException`, `FileNotFoundException`, `SQLException`, `NoSuchMethodException`.

- Les excepcions **no revisades**, que hereten de la classe `java.lang.RuntimeException`, són errades de codi, i es detecten **en temps d'execució**. Aquestes no són controlades pel compilador, pel que no ens obliga a tractar-les. Les excepcions no revisades més comunes inclouen:
  - **ArithmeticException**: Desbordament o divisió per zero.
  - **NumberFormatException**: Conversió il·legal de tipus.
  - **IndexOutOfBoundsException**: Accés a un element inexistent d'un vector.
  - **NegativeArraySizeException**: Intent de creació d'un vector de longitud negativa.
  - **NullPointerException**: Intent d'ús d'una referència nul·la.
  - **SecurityException**: Violació de la seguretat en temps d'execució.

Les excepcions no revisades poden previndre's codificant correctament el codi, pel que, com hem dit, no necessiten ser capturades.

La manera de tractar les excepcions amb Java és amb els blocs *try/catch*:

```
try {
    bloc_de_sentències_que_pot_llençar_excepcions
} catch (ClasseExcepció_1 objecteExcepció) {
    bloc_de_sentències_per_tractar_excepció_1;
} catch(ClasseExcepció_2 objecteExcepció) {
    bloc_de_sentències_per_tractar_excepció_1;
} finally {
    bloc_de_tasques_comunes;
}
```

2.0.1 Tractament d'Excepcions a Kotlin

La principal diferència en Kotlin respece a Java és que en Kotlin **totes les excepcions són no revisades**. A nivell pràctic, això implica que no s'han d'incloure *throws* a les declaracions de funcions on es puguen produir.

Això no vol dir que no es puguen tractar excepcions ni llançar excepcions al nostre codi.

Per exemple, tenim el següent codi:

```
import java.io.File

fun mostraFitxer (fitxer:String){
    File(fitxer).forEachLine() { println(fitxer) }
}

fun main() {
    mostraFitxer("/tmp/noexisteix.txt");
}
```

Com a peculiaritats, en primer lloc, podem vore com podem utilitzar indistintament les llibreríes de Java dins els nostres projectes. Per altra banda, també podem fixar-nos amb com de senzill resulta llegir el contingut d'un fitxer amb Kotlin.

De tota manera, el que ens interessa és que en compilar aquest programa amb Kotlin no ens mostra cap missatge. Si havérem fet l'equivalent amb Java ens diría que el nostre codi podría disparar una excepció del tipus IOException, aquesta hauria d'haver segut llançada per la funció `mostraFitxer()`, i capturada per `main`. però Kotlin no ens diu res, ja que no revisa les excepcions.

En canvi, quan executem el programam, aquest s'interromp i ens mostra el missate:

```
Exception in thread "main" java.io.FileNotFoundException: /tmp/noexisteix.txt (No s
```

Com veiem, el fet que no es revisen les possiles excepcions en temps de compilació no significa que aquestes no puguen ocórrer. Per tractar-les, tenim els mateixos mecanismes que amb Java, amb el *try-catch-finally*.

L'exemple anterior podríem haver-lo expressat de les següents maneres, segons on volguérem tractar l'error:

```
// Tractem l'error dins el mètode mostraFitxer
fun mostraFitxer (fitxer:String){
    try{
        File(fitxer).forEachLine() { println(fitxer) }
    } catch (ex: IOException){
        println(ex.message);
    }
}

fun main() {
    mostraFitxer("/tmp/noexisteix.txt");
}
```

```
// Tractem l'error al propi main, en aquest cas, tot i que
// la funció mostraFitxer no emet explícitament cap excepció,
// podem capturar-la des de la funció que la invoca.
fun mostraFitxer (fitxer:String){
        File(fitxer).forEachLine() { println(fitxer) }
}

fun main() {
    try{
        mostraFitxer("/tmp/noexisteix.txt");
    } catch (ex:Exception){
        println(ex.message);
    }
}
```

En els dos casos, l'eixida del programa (que no interromprà l'execució) serà la següent:

```
/tmp/noexisteix.txt (No such file or directory)
```

Per altra banda, recordem que amb Kotlin, tot són expressions que poden tindre un resultat, i la construcció *try-catch-finally* no é suna excepció. Així podem fer coses com:

```
fun mostraFitxer (fitxer:String){
    // Assignem el resultat del bloc try-catch a una constant
    val result = try{
        File(fitxer).forEachLine() { println(fitxer) }
        true    // Si no ha botat cap excepció el resultat serà true
    } catch (ex: IOException){
        println(ex.message);
        false   // Si ha botat alguna excepció d'E/E, el resultat serà fals
    }

    if (result) println("El fitxer s'ha llegit correctament");
    else println("El fitxer no s'ha trobat");
}

fun main() {
    mostraFitxer("/tmp/noexisteix.txt");
}
```

> **La classe Nothing i les excepcions**
>
> La classe `Nothing` representa *un valor que mai existirà*, i s'utilitza com a tipus de retorn per a aquells mètodes/funcions que **sempre** retornen una excepció (i per tant no acaba d'una forma normal).

2.0.2 Interoperabilitat amb Java

Com hem dit, Java i Kotlin s'executen sobre la JVM, i podem treballar amb els dos llenguatges en una mateixa aplicació, tal i com hem vist a l'exemple inicial amb Gradle. Però… si en Kotlin totes les excepcions són no revisades, i les funcions que poden llançar excepcions no tenen un `throws`, com sabem des de Java que una funció Kotlin pot llançar excepcions?

Ho fem mitjançant anotacions. Concretament disposem de l'anotació `@Throw` en Kotlin que afegim a la definició de les funcions. Amb aquesta anotació, avisem a les invocacions que es realitzen des de Java per a que siguen conscients que la funció pot llençar excepcions i aquestes s'han de capturar.

Anem a veure l'exemple següent, amb un projecte Gradle anomenat ***KFileReader***.

L'estructura d'aquest projecte és el següent:

```
.
|-- build.gradle
|-- gradle
|   `-- wrapper
|       |-- gradle-wrapper.jar
|       `-- gradle-wrapper.properties
|-- gradlew
|-- gradlew.bat
|-- settings.gradle
`-- src
    `-- main
        |-- java
        |   `-- com
        |       `-- ieseljust
        |           `-- dam
        |               `-- FileReader.java
        |-- kotlin
        |   `-- com
        |       `-- ieseljust
```

```
|           `-- dam
|               `-- mostraFitxer.kt
`-- resources
```

Com veiem, tenim dos rutes de codi font diferents per al mateix paquet, la de Java i la de Kotlin.

En aquest exemple, com veiem al fitxer `build.gradle`, la classe principal és la de Java (`FileReader`):

```
plugins {
    // Plugins de Kotlin
    id 'org.jetbrains.kotlin.jvm' version '1.3.72'

    // Tipus de projecte: Aplicació CLI
    id 'application'
}

repositories {
    jcenter()
}

dependencies {
    // Dependències de Kotlin
    implementation platform('org.jetbrains.kotlin:kotlin-bom')
    implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk8'
    testImplementation 'org.jetbrains.kotlin:kotlin-test'
    testImplementation 'org.jetbrains.kotlin:kotlin-test-junit'
}

application {
    // La classe principal serà la que classe
    // que hem creat en Java
    mainClassName = 'com.ieseljust.dam.FileReader'
}
```

***Cas 1: Exemple sense tractament d'excepcions**

Si no anem a tractar les excepcions, el codi font dels fitxers és el següent:

- **Fitxer `src/main/kotlin/com/ieseljust/dam/mostraFitxer.kt`**

```kotlin
package com.ieseljust.dam

import java.io.File
import java.io.IOException;

// Cas 1: Funció mostraFitxer sense
// tractament d'excepcipns

fun mostraFitxer (fitxer:String){
    File(fitxer).forEachLine() { println(it) }
}
```

Com veiem, consisteix només a la funció que llegeix un fitxer i el mostra línia per línia. Per a això, fem ús del mètode `forEachLine`, de la classe `File`, a la que li passem el fitxer a llegir. Per tal de mostrar cada línia, passem una *funció lambda* que mostra cadascuna de les línies que retorna `forEachLine`, fent ús d'un argument especial anomenat `it`. Més endavant veurem què són les funcions *lambda* i l'argument `it`.

- **Fitxer `src/main/java/com/ieseljust/dam/FileReader.java`**

```java
package com.ieseljust.dam;


class FileReader
{
    public static void main(String args[])
    {
        // Cas 1:  Podem utilitzar directament la funció mostraFitxer
        //         ja que aquesta no llança errors.
        //         Si el fitxer no existeix, donarà un
        //         error d'execució.


        MostraFitxerKt.mostraFitxer("/tmp/noexisteix.txt");

    }
}
```

Aquesta classe, que és la que llança l'aplicació, al seu mètode main invoca el mètode `mostraFitxer` de la classe `MostraFitxerKt`, que és la que generarà el compilador automàticament a partir del codi que hi ha al fitxer `mostraFitxer.kt`.

Si compilem amb `gradle build`, no obtindrem cap missatge d'error. Si llancem el projecte i el fitxer `/tmp/noexisteix.txt` sí que existeix, mostrarà aquest per pantalla. Però en cas que no existisca, donarà un error d'execució i aturarà aquesta:

```
$ gradle run

> Task :run FAILED
Exception in thread "main" java.io.FileNotFoundException: /tmp/noexisteix.txt (El f
        at java.base/java.io.FileInputStream.open0(Native Method)
    at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
    at kotlin.io.FilesKt__FileReadWriteKt.forEachLine(FileReadWrite.kt:190)
    at kotlin.io.FilesKt__FileReadWriteKt.forEachLine$default(FileReadWrite.kt:
    at com.ieseljust.dam.MostraFitxerKt.mostraFitxer(mostraFitxer.kt:9)
        at com.ieseljust.dam.FileReader.main(FileReader.java:7)

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':run'.
> Process 'command '/usr/lib/jvm/java-11-openjdk-amd64/bin/java'' finished with no
zero exit value 1

* Try:
Run with --stacktrace option to get the stack trace. Run with --
-info or --debug option to get more log output. Run with --
scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 789ms
3 actionable tasks: 1 executed, 2 up-to-date
```

Per tal d'evitar aquests errors d'execució, tenim les *excepcions revisades* en Java, que detecten possibles errors en temps de compilació, i ens obliguen a controlar-los amb `try-catch`. Al següent cas, al mateix exemple, anem a vore com ho faríem per tal que Kotlin indicara que la funció `mostraFitxer` pot generar una excepció, i conseqüentment hajam de tractar-la des de Java.

- **Fitxer `src/main/kotlin/com/ieseljust/dam/mostraFitxer.kt`**

```kotlin
package com.ieseljust.dam

import java.io.File
import java.io.IOException;

// Cas 2: Funció mostraFitxer amb tractament
// d'excepcions.

@Throws(IOException::class)
fun mostraFitxer (fitxer:String){
    File(fitxer).forEachLine() { println(it) }
}
```

En aquest codi, mitjançant `@Throws(IOException::class)` estem indicant que la funció que anem a definir pot llançar una excepció de tipus IOException. Això farà que quan es compile a byte-code de la JVM siga com si havérem indicat un `throws IOException` en Java. Açò ens obligarà a capturar aquesta excepció des de la classe principal en Java.

En aquest cas, si compilàrem només amb les modificacions que hem fet al fitxer de Kotlin, obrindríem el següent error de compilació:

```
$ gradle build

> Task :compileJava FAILED
.../KFileReader/src/main/java/com/ieseljust/dam/FileReader.java:7: error: unrepor
        MostraFitxerKt.mostraFitxer("/tmp/noexisteix.txt");
                                    ^
1 error
...
```

Aquest error de compilació ens indica que l'excepció `IOException` que llança el mètode `mostraFitxer` ha de ser tractada amb `try-catch` o rellançada am un `throws`. Al nostre cas, Al nostre cas, haurem de capturar-la des de la classe principal de l'aplicació en Java.

Veiem ara les modificacions a la classe principal:

```kotlin
package com.ieseljust.dam;


class FileReader
```

```
{
    public static void main(String args[])
    {
        // Cas 2: Per al segon cas, ara sí que se'n obliga a
        //        capturar l'excepció en temps de compilació.

        try {
            MostraFitxerKt.mostraFitxer("/tmp/noexisteix.txt");
        } catch (Exception e){
            System.out.println(e.getMessage());
        }

    }
}
```

Amb les modificacions realitzades als dos fitxers font, la compilació ja és correcta, i quan executem, veurem com se'ns mostra el missatge d'error que hem indicat al `catch`, en aquest cas, el que torne el `getMessage()` de la pròpia excepció:

```
$ gradle run

> Task :run
/tmp/noexisteix.txt (El fitxer o directori no existeix)

BUILD SUCCESSFUL in 784ms
```

3 Entrada i eixida des de consola

En Java l'entrada i eixida (E/E) es gestiona al paquet `java.io`, que proporciona un conjunt d'streams per llegir i escriure dades, tant a fitxers com a altres tipus de fonts. Per gestionar l'eixida i l'entrada estàndard, tenim tres tipus de classes:

- System.in: Entrada estàndard, que proporciona entre d'altres el mètode `readLine()`, per llegir línies d'un búffer.
- System.out: Eixida estándard, que proporciona els mètodes `System.out.print()` i `System.out.println()`.
- System.err: Eixida d'errors.

A més, dins el paquet d'utilitats de Java (`java.util`) tenim la classe `java.util.Scanner`.

3.1 Exemples d'ús de readLine i Scanner

3.1.1 Readline

Per llegir línies, necessitem un flux d'entrada de dades. La manera de llegir una línia des de teclat mitjançant `readLine` seria la següent:

```
import java.io.*;

public class ExempleReadLine {
    public static void main( String[] args ) {
        BufferedReader entrada = new BufferedReader (new
        ↪  InputStreamReader(System.in));

        try{
            System.out.println("Escriu una línia");
            String cadena = entrada.readLine();
            System.out.println("Has escrit: "+cadena);
            } catch( IOException e) {
                System.out.println("Error de lectura");
            }

    }
}
```

Com veiem, per tal de llegir una línia mijançant `readLine`, necessitem fer-ho mitjançant un objecte de tipus `BufferedReader`, generat a partir d'un `InputStreamReader` de l'entrada estàndard (`System.in`). Tot açò ho fem amb la línia:

```
BufferedReader entrada = new BufferedReader (new
↪  InputStreamReader(System.in));
```

Per poder llegir la línia posteriorment amb `entrada.readLine();`.

Cal tindre en compte, que quan llegim de l'entrada estàndard (teclat), obtenim dades de tipus *String*. Per tal d'utilitzar altre tipus de dades, utilitzarem els wrappers o classes de cobertura:

Wrapper.Mètoode	Tipus bàsic al què es converteix
Byte.parseByte(cadena)	byte
Short.parseShort(cadena)	short

Wrapper.Mètoode	Tipus bàsic al què es converteix
Integer.parseInt(cadena)	int
Long.parseLong(cadena)	long
Boolean.parseBoolean(cadena)	boolean
Float.parseFloat(cadena)	float
Double.parseDouble(cadena)	double
Character.parseChar(cadena)	char

3.1.2 Scanner

El mètode anterior mostra el procés necessari per llegir dades des de l'entrada estàndard, però resulta un codi bastant llarg i farragós. Per tal de simplificar el procés de lectura, disposem de la classe `Scanner`, que a més, ens permet llegir directament valors dels diferents tipus de dades, sense haver d'utilitzar *wrappers*

Veiem-ho amb un exemple senzill:

```
import java.util.Scanner;

class ExempleScanner {

    public static void main(String arg[]){
        Scanner MyScanner = new Scanner(System.in);
        String cadena = MyScanner.nextLine();
        String paraula = MyScanner.next();
        int num_enter = MyScanner.nextInt();
        float num_real=MyScanner.nextFloat();
   }
}
```

Amb això, creem un objecte `Scanner` a partir de l'entrada estàndard (`System.in`) i amb ell, podem llegir tant cadenes(`nextLine`), paraules (`next`), o valors enters (`nextInt`) i en coma flotant (`nextFloat`).

3.2 Entrada i eixida des de consola en Kotli

3.2.1 Eixida estàndard

Per a mostrar informació per l'eixida estàndard, Kotlin utilitza les funcions `print()` i `println()` de Java. La principal diferència entre les dos, és que `println` mou el cursos al començament de la pròxima línia, i amb `print`, el cursos es queda al final de la línia.

Internament, tant `print()` com `println()` són crides als mètodes `System.out.print()` i `System.out.println()` de Java.

Cal recordar que amb Kotlin, podem fer ús de les *template strings* (`${}`) per mostar variables i expressions amb elles:

```
println("cadena")
println("$valor")
println("valor = $valor")
println("valor+3=${valor + 3}")
```

3.2.2 Entrada estàndard

Per tal de llegir informació de l'entrada estàndard fem ús de la funció `readLine()`, de la següent manera:

```
val cadena = readLine()!!
```

La funció `readline()` llig l'entrada estàndard, i ens retorna un valor de tipus String *nullable*, és a dir, que pot ser *nul*. Per tant, per tal d'assegurar-nos de tractar correctament aquesta valor de tornada, hem d'utilitzar l'operador d'*asserció no nul* `!!`. D'aquesta manera, si el valor que ens retorna `readLine()` és nul, Kotlin llançaria una excepció de tipus *Kotlin null pointer Exception*, que pot ser capturada i tractada de forma adeqüada.

Si volem llegir un tipus específic de dades fent ús de `readLine()`, hauriem de convertir aquests valor de forma explícita, ja que per defecte, ens retorna un *String*.

Una altra opció interessant, és fer ús de l'objecte `Scanner` de Java. Veiem com l'utilitzariem en un exemple:

```
import java.util.Scanner
```

```kotlin
fun main(args: Array<String>) {

    // Creem una instància d'Scanner, que agafe com a
    // entrada l'entrada estàndard.
    val reader = Scanner(System.`in`)
    print("Inserix un número: ")

    // A l'igual que Java, podem utilitzar nextInt()
    // per llegir un enter del teclat
    var integer:Int = reader.nextInt()

    println("Has introduït el número: $integer")
}
```