

مختصر لغة

# Kotlin



أحمد الجعيد

2.....المحتويات

**بيئة العمل**.....ERROR! BOOKMARK NOT DEFINED.

9.....تثبيت JAVA 8

9.....تثبيت IDE

10.....استخدام code vs

13.....موجه الأوامر

**14.....تهيئة ECLIPSE لبرمجة KOTLIN**

14.....تثبيت الادوات

16.....مشروع Kotlin

17.....اهلاً بالعالم

**18.....لغة KOTLIN**

18.....أولاً : مقدمة

18.....ثانياً : جافا Java أم كوتلن Kotlin

19.....ثالثاً : مزايا لغة كوتلن Kotlin

**20.....المقدمة**

**21.....المتغيرات**

21.....أولاً : تعريف المتغيرات

23.....ثانياً : أنواع المتغيرات

24.....ثالثاً : تحويل المتغيرات

25.....رابعاً : برنامج "hello world"

**26.....ادخال البيانات**

27	أولا : دالة readLine.....
27	ثانيا : التحويل.....
29	<b>التعليقات</b> .....
31	<b>العمليات الرياضية</b> .....
31	أولا : الرموز الرياضية.....
32	ثانيا : Decrement & Increment.....
33	<b>العمليات المنطقية</b> .....
34	أولا : الرموز المنطقية.....
35	ثانيا : البوابات المنطقية "المقارنة".....
38	<b>السلاسل النصية</b> .....
42	<b>المصفوفات</b> .....
42	أولا : تعريف المصفوفة.....
43	ثانيا : إدخال قيم للمصفوفة.....
45	ثالثا : طباعة عناصر المصفوفة.....
47	رابعا : التعامل مع المصفوفات.....
49	<b>النطاق</b> .....
49	أولا : النطاق.....
51	ثانيا : القفزة أو step.....
51	ثالثا : التعامل مع النطاق.....
54	<b>VISIBILITY MODIFIERS</b> .....
54	أولا : رؤية المتغيرات.....
55	ثانيا : أهمية رؤية المتغيرات.....
55	<b>ESCAPE SEQUENCE</b> .....
55	أولا : ماهي الاختصارات أو الأوامر السريعة ESCAPE SEQUENCE.....
55	ثانيا : الرموز ومعانيها.....
58	<b>UNICODE</b> .....

## 62..... REGEX (REGULAR EXPRESSION)

62.....أولا : التعبيرات القياسية

63.....ثانيا : طريقة التعبير عن التعبيرات القياسية

66.....ثالثا: مثال : .....

68.....رابعا : الدوال المستخدمة مع التعبيرات القياسية

## 74 ..... بنى التحكم

### 74.....FOR LOOP

74.....أولا : الجملة for loop

76.....ثانيا : nested for والمقصود هنا الحلقات المتداخلة

### 78..... WHILE & DO WHILE

78.....الجملة while

79.....nested while

79.....الجملة do...while

### 81.....WHEN

### 84.....الجملة الشرطية

84.....الجملة if

85.....if ..else

86.....if ..else if

87.....If المتداخلة

88.....التعبيرات مع if

## 90 ..... الدوال

### 90.....الدوال

90.....أولا : طريقة كتابتها

90.....ثانيا : أنواع الدوال

92	.....	ثالثاً : الاستدعاء
94	.....	EXTENSION FUNCTION
96	.....	LAMBDA FUNCTION
98	.....	HIGH LEVEL FUNCTION
100	.....	JUMP & RETURN مفهوم

## 101 ..... برمجة كائنية التوجه

102	.....	أولاً : البرمجة كائنية التوجه
102	.....	ثانياً : مقدمة عن Classes & Object
103	.....	ثالثاً : الفرق بين Classes & Object
104	.....	CLASSES OOP
104	.....	أولاً : طريقة إنشاء الصنف class
106	.....	ثانياً : طريقة إنشاء كائن Object
106	.....	ثالثاً : الوصول إلى المتغيرات الخاصة بال Class
108	.....	رابعاً : الوصول إلى الدوال الخاصة بال Class
108	.....	خامساً : تمرير الوسائط في Class
110	.....	سادساً : كلمة this
111	.....	سابعاً : دالة البناء constructor
113	.....	تنويه
114	.....	INHERITANCE
115	.....	أولاً : مفاهيم متعلقة بالوراثة
116	.....	ثانياً : الوراثة في البرمجة
118	.....	ثالثاً : التعامل مع الوسائط الممررة للتصنيف
121	.....	رابعاً : الفرق بين super و this
121	.....	INTERFACE
121	.....	أولاً : تمهيد

122	.....	ثانياً : التعريف عن interface
122	.....	ثالثاً : الإعلان في interface
123	.....	رابعاً : استخدام interface
127	.....	خامساً : مثال interface
<b>129</b>	.....	<b>ABSTRACT</b>
129	.....	أولاً : مفهوم abstract
129	.....	ثانياً : استخدام abstract
129	.....	ثالثاً : انشاء تصنيف abstract
130	.....	رابعاً : مثال
133	.....	خامساً : مبدأ الوراثة المتعددة multi inheritance
<b>134</b>	.....	<b>OVERRIDE</b>
134	.....	أولاً : مفهوم Override
136	.....	ثانياً : كلمة final
<b>137</b>	.....	<b>OVERLOAD</b>
<b>139</b>	.....	<b>COMPANION OBJECT</b>
139	.....	أولاً : مفهوم companion object
139	.....	ثانياً : طريقة كتابته
139	.....	ثالثاً : مثال
141	.....	رابعاً : متغير يستقبل companion object
143	.....	خامساً : معلومات حول companion object
<b>145</b>	.....	<b>التصنيفات المتداخلة NASTED CLASSES</b>
145	.....	أولاً : مفهوم التصنيفات المتداخلة
145	.....	ثانياً : طريقة تعريف التصنيفات المتداخلة
146	.....	ثالثاً : أهمية التصنيفات المتداخلة
146	.....	رابعاً : مصطلحات مهمة في التصنيفات المتداخلة

146	.....	خامساً : إنشاء كائن من inner class
147	.....	سادساً : مثال
148	.....	<b>ENUM CLASS</b>
148	.....	أولاً : مفهوم Enum class
148	.....	ثانياً : استخدامها Enum class
148	.....	ثالثاً : تعريف Enum class
149	.....	<b>DATA CLASS</b>
149	.....	أولاً : مفهوم data class
150	.....	ثانياً : طريقة data class
150	.....	ثالثاً : مثال لاستخدام class
151	.....	رابعاً : اضافة data قبل تعريف class
152	.....	<b>POLYMORPHISM</b>
152	.....	أولاً : مبدأ Polymorphism
152	.....	ثانياً : شروط Polymorphism
153	.....	ثالثاً : أمثلة متنوعة لمفهوم Polymorphism
159	.....	<b>EXCEPTION- الاستثناءات</b>
159	.....	أولاً : ما هي الاستثناءات
159	.....	ثانياً : تكوين الاستثناءات
161	.....	ثالثاً : مثال
161	.....	<b>MULTI-THREADING</b>
161	.....	أولاً : مفهوم multi-thread
162	.....	ثانياً : عمل multi-thread
164	.....	ثالثاً : الدوال التي تعمل مع thread
167	.....	<b>ANY</b>
167	.....	أولاً : ما هو any class

# تقديم

لغة Kotlin احدى اللغات التي لاقت رواجاً في السنتين الماضيتين وزادت شعبيتها بعد دعمها بشكل رسمي في اندرويد استديو لبرمجة تطبيقات الاندرويد..

لغة Kotlin قادرة على برمجة تطبيقات الويب والـIOS بشكل طبيعي باستخدام أدوات للتطوير مدعومة مباشرة من الشركة المطورة لـKotlin

حاولت في هذا الكتيب ايجاز لغة Kotlin بشكل عام ليتمكن المستخدم المبتدئ في البرمجة من معرفة جميع جوانب اللغة.

وفي النهاية احمد الله على انتهاء هذا العمل وارجوا ان يكون نافعاً لاثراء المحتوى العربي في مجال البرمجة.

احمد الجعيد

معلم حاسب آلي - وزارة التربية والتعليم  
السعودية

تواصل: [aljo3aid@gmail.com](mailto:aljo3aid@gmail.com)

966504511433.



# بيئة العمل

للبدء في الدروس نحتاج الى تهيئة بيئة تشغيل للعمل علي البرمجة و Compiler للغة Kotlin. نحتاج للقيام ببعض الخطوات لنجعل جهازك جاهز للبرمجة.

## تثبيت JAVA 8

Kotlin تعمل على JVM ولهذا السبب نحتاج الى تثبيت JDK 8 على جهازك. اذهب الى الموقع الرسمي لاوراكل وقم بتحميل وتثبيت JDK 8 او احدث. اذا كنت قد جهزت بيئة لبرمجة JAVA فبطبيعة الحال لا تحتاج الى تثبيت JDK ولكن يجب ان تتأكد من رقم الاصدار لديك باستخدام الامر `java -version` في موجه الاوامر للوندوز او الطرفية للاجهز يونكس وماك.

## تثبيت IDE

يوجد الكثير من برامج IDE علي شبكة الانترنت ولكن الشهير منها ربما يكون محدود على الاصابع لذلك سأكتب لك الاشهر منها وتستطيع اختيار مايناسبك واستخدامه.



## استخدام vs code

---

حقيقة في البدايات انصح باستخدام محررات شفرات بسيطة لاتدعم الاكمال التلقائي ليس للتعقيد ولكن لجعل يدك ممارسه للكتابة السريعة وايضاً للفهم الجيد دون استخدام برامج الاكمال التلقائي.

من افضل محررات الاكواد التي استخدمتها مؤخراً وهو vs code محرر بسيط ولكن كمية الاضافات الموجودة في المتجر الخاص فيه تجعل منه منافس شرس للبرامج الكبيرة والمتخصصة.

## ❖ تحميل البرنامج



## ❖ اضافة Kotlin

اضافة تقوم بدعم Syntax highlighter في لغة kotlin حصلت الاضافة على 30 الف تحميل تقريباً ومدعومة بشكل كبير من المطور على صفحة github الخاصة حيث تتيح لك تشغيل الشفرة البرمجية من اي صفحة مفتوحة مباشرة.



الاضافة بدورها ستقوم بتحديد المتغيرات بالوان مختلفة وترتيب الشفرة البرمجية بشكل جذاب وجميل.

```
1 fun main(args:Array<String>){
2     print("اهلا بالعالم")
3 }
4
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[Done] exited with code=1 in 2.884 seconds

[Running] cd "/Users/ahmedaljoaid/.composer/" && kotlinc Untitled.kt -include-runtime -d Untitled.jar && java -jar Untitled.jar

اهلا بالعالم

[Done] exited with code=0 in 3.088 seconds

Ln 4, Col 2 Spaces: 4 UTF-8 LF Kotlin

## ❖ إضافة سمة ليلية

مايهم المبرمجين ان يكون المحرر مريح للعين فنعلم جميعاً اعتكافنا على اجهزة الحاسب لاقوات طويلة ولكن هذا الشي متعب للعين في حالة كانت الشاشة تشع باللون الابيض ولذلك سنحتاج الى اضافة سمات ليلية للبرنامج وساقوم باضافة السمة التي استخدمها . اسم السمة. Atom One Dark Theme



## ❖ اضافة ايقونات توضيحية

نعلم جميعاً ان اي لغة برمجة يكون لها ايقونة توضيحية وسنقوم باضافتها للبرنامج وذلك لاعطاء البرنامج شكل جميل ومرتب . الاضافة تخطت المليون تحميل لشعبيتها وجذابيتها ودعمها المستمر من المبرمج.



## مجموعة من اشكال الايقونات الخاصة بلغات البرمجة

Icon	Name	Icon	Name	Icon	Name	Icon	Name	Icon	Name
	Aurelia		Git		Lib		Phpmailer		Src
	Bower		Github		Log		Public		Sublime
	Components		Gittab		Markdown		React-components		Temp
	Config		Global		Ngrx-actions		Redux-actions		Test
	Css		I18n		Ngrx-effects		Redux-reducer		Tools
	Database		Images		Ngrx-reducer		Redux-store		Views
	Dist		Include		Ngrx-state		Resource		Vscode
	Docker		Javascript		Node		Sass		Vue
	Docs		Jinja		Php		Scripts		Webpack

الآن اصبح لديك محرر أكواد قوي ومنافس كبير.

## موجه الأوامر

الآن نحتاج لتحميل اضافات للنظام لعمل compiler للشفرة البرمجية لـ Kotlin باستخدام موجه الاوامر لان الاضافة تعتمد عليه بشكل اساسي.

قم بتحميل اخر اصدار من هنا وسيقوم بالتثبيت بشكل تلقائي



الآن يمكنك الانتقال للدروس ومتابعتها

## تهيئة Eclipse لبرمجة Kotlin

---

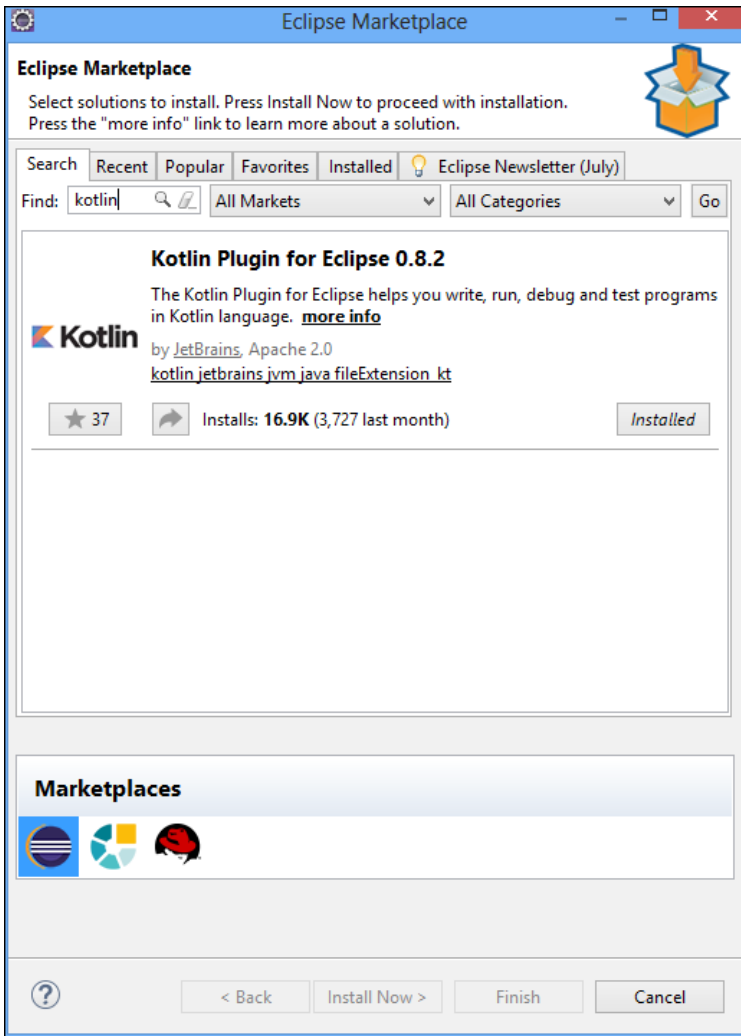
يعتبر Eclipse من أقوى IDE المجانية في البرمجة لاحتوائه على متجر وإضافات كثيرة تدعم العديد من لغات البرمجة والدعم الكبير من المطورين.

نحتاج إضافة بعض الأدوات للبرنامج حتى نستطيع استخدام Kotlin.

### تثبيت الأدوات

---

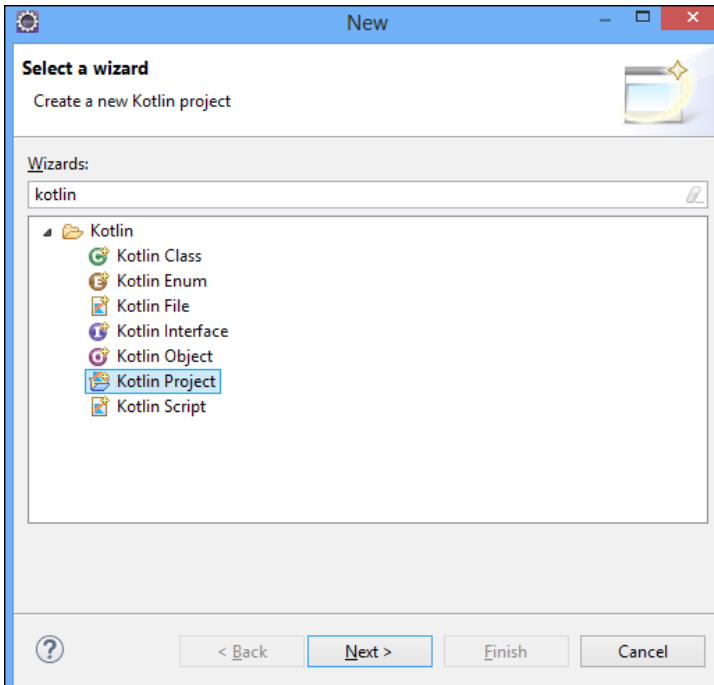
افتح برنامج Eclipse واذهب الى خيار "Eclipse Market Place" ستظهر لك نافذه مشابهه لما في الاسفل.



في مربع البحث اكتب kotlin وستظهر لك نتيجة مشابهة لما في الصورة قم بتثبيتها في الغالب ستحتاج الانتظار لبعض الوقت لتحميل وذلك يعتمد على سرعة الانترنت لديك . في الغالب بعد التحديث ستحتاج الى اعادة تشغيل Eclipse.

## مشروع Kotlin

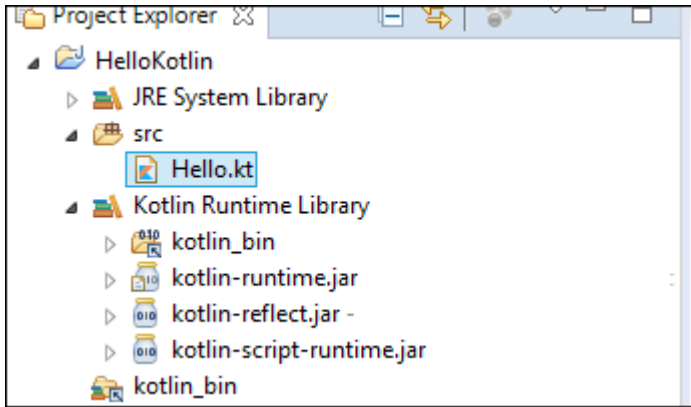
الآن بعد اعادة التشغيل ستستطيع استخدام Eclipse في برمجة مشاريع Kotlin . قائمة File -> New -> Others بعد ذلك قم باختيار "Kotlin project" من القائمة.





سينشئ لديك مشروع Kotlin جديد وستجد ملف Kotlin داخل مجلد . src عند الضغط بزر الماوس الايمن على مجلد src ستجد الكثير من الخيارات تخص Kotlin لانشاء فصول وايضاً interface الخ...

ستجد لديك صورة مشابهه لما في الاسفل.



## اهلاً بالعالم

عند فتح ملف Hello.kt ستجد شفرة برمجية التي لا طالما رأيتها في جميع لغات البرمجة. "Hello World"

```
fun main(args: Array<String>) {  
    println("Hello, World!")  
}
```

عند تشغيل المشروع ستظهر لك رسالة في console output وستكون النتيجة طباعة رسالة "اهلا بالعالم".

## لغة Kotlin

---

### أولاً : مقدمة

---

تُعد Kotlin لغة برمجية كائنية التوجه تعمل على JVM ومتوافقة مع حزمة JDK بشكل كامل، تم اطلاق وتطوير هذه اللغة عام ٢٠١١م من قبل شركة JetBrains الشركة المطورة لأكبر وأشهر مجموعة IDE مستخدمه حول العالم مع مختلف لغات البرمجة ، وتعتبر لغة مفتوحة المصدر . ولقد Google خلال فعاليات مؤتمر I/O للعام 2017م بدء الدعم لهذه اللغة وسيتم تضمينها في نسخة Android Studio 3.0 الجديدة بدون الحاجة لإعدادات التثبيت الإضافية.

### ثانياً : جافا Java أم كوتلن Kotlin

---

بالطبع وجود كوتلن لا يلغي أهمية الجافا ووجودها واختيارك للجافا أم للكوتلن يعود لك ولطبيعة عملك وللفريق الذي تعمل معه ، أما بالمجمل فتعتبر كوتلن لغة انسيابية وسهلة الفهم وتوفر الوقت والجهد فمطوري اللغة حرصو على تلافي الأخطاء وحل مشاكل الترميز.

## ثالثاً: مزايا لغة كوتلن Kotlin

---

- لغة مفتوحة المصدر.
- توفير الوقت والجهد.
- متوافقة مع android studio .
- أخطاء أقل وتتميز بقضائها على مشكلة NullPointerException .
- تتميز بالانسيابية.
- أكثر أمان.
- متعددة الاستخدامات فيمكنك من خلالها تطوير مواقع الانترنت ، السيرفرات ، سطح المكتب إلخ...

# المقدمة



سأقدم في هذه الجزئية مقدمة عن جميع العناصر التي تحتاجها لفهم تركيبية برمجة لغة Kotlin بدايةً من المتغيرات ومروراً على الخوارزميات الخاصة بها الدوارة والاختيار والاحتمالات.

بإذن الله سيكون هناك تفاصيل مذكورة ومميزات خاصة بلغة Kotlin التي تجعلها بديل جيد للغة JAVA وايضاً في وقت لاحق سنتعلم كيف نستخدمها مع JavaScript وايضاً في برمجة تطبيقات الاندرويد و الايفون. انت الان على بعد خطوات بسيطة من البداية وبإذن الله ستكون الدروس حافز لك للاطلاع على تفاصيل لم اذكرها وتجعلك تتوسع في تعلم اللغة وفق احتياجاتك.

- المتغيرات
- ادخال البيانات

- التعليقات
- العمليات الرياضية
- العمليات المنطقية
- السلاسل النصية
- المصفوفات
- النطاق
- Visibility Modifiers
- ESCAPE SEQUENCE
- UNICODE
- REGEX (Regular Expression)

## المتغيرات

---

عند تعلم أي لغة فلا بد من التطرق لأساسياتها ومن ضمن هذه الأساسيات المهمة هي كتابة المتغيرات بطريقة صحيحة وسنتعرف في هذا الدرس على تعريف المتغيرات، والثوابت، وشروط التسمية ، جملة الطباعة والدالة الرئيسية.

المتغيرات هي عبارة عن قيمة قابلة للتغيير اثناء كتابة البرنامج، لتعريف المتغيرات في لغة kotlin نستخدم كلمة var، وسنرى الآن أشكال مختلفة للتعريف بالمتغيرات.

### أولا : تعريف المتغيرات

---

## ❖ اسناد القيمة

يمكنك التعريف بالمتغير عن طريق تعريفه واسناد القيمة له مباشرة .

```
var myName="Ahmed"  
var myAge=27
```

في المثال أعلاه لم نذكر النوع ولكن اسندنا القيمة كيف يستطيع البرنامج معرفة نوع المتغير؟ بواسطة compiler الذي يقوم بمعرفة النوع من خلال القيمة المسندة له.

إذا قمت باسناد قيمة للمتغير من نوع Int مثلاً فانك لن تستطيع تغييرها الى اي نوع آخر.

```
var age = 12  
age = "12 years old" // Error: type mismatch
```

## ❖ تعريف بالمتغير والتعريف بنوعه واسناد القيمة مباشرة

يمكنك التعريف بالمتغير والتعريف بنوعه مع مراعاة كتابة أول حرف من نوعه بحرف كبير.

```
var myName : String ="Ahmed"  
var myAge:Int=27
```

## ❖ تعريف بالمتغير والتعريف بنوعه واسناد القيمة لاحقاً

يمكنك التعريف بالمتغير والتعريف بنوعه مع مراعاة كتابة أول حرف من نوعه بحرف كبير في سطر ويمكننا اسناد القيمة خلال اسطر البرنامج أو في السطر الذي يليه.

```
var myName : String  
myName="Ahmed"
```

يراعى عند تعريف المتغيرات في لغة kotlin أنها لا تبدأ بأي حرف من الرموز المخصصة باستثناء (.). ما تعرف بالشرطة التحتية أو underscore.

## ثانياً : أنواع المتغيرات

العدد الصحيح	Int
عدد صحيح قصير	Short
عدد صحيح طويل	Long
قيمة عشرية بسيطة	Float
قيمة عشرية أكثر دقة وتحديد	Double
حرف	Char
سلسلة نصية	String
متغير منطقي	Boolean

للتفريق بين Double و Float هو عندما نريد تعريف متغير عشري من نوع Float نضع في نهايته حرف f.

مع ملاحظة ان المتغير double يتكون من bit64 بعكس float الذي يتكون من bit32 فقط

```
var cost:Double=12.9001111
var cost = 14.1655f
```

## ❖ الثوابت

تنطبق عليه نفس مواصفات المتغيرات باستثناء أنه قيمة غير قابلة للتعديل أثناء البرنامج ولتعريف الثابت نستخدم كلمة `val` ، يمكننا الاستفادة منه في تعريف القوانين الرياضية أو لو كان هنالك رقم وظيفي فهو ثابت بالطبع لا يمكن تغييره أو رقم الهوية الوطني.

```
val id:Int=601121
id = 1234 // Error
```

## ثالثاً : تحويل المتغيرات

ربما كتبت متغير و اردت تحويل الى اي نوع آخر فانك لن تستطيع ذلك بسبب انك قد حددت نوع المتغير او ان Compiler قد قام بتحديد نوع المتغير اذا كان اسناد القيمة بشكل مباشر. تحويل المتغيرات ايسر مما تتخيل لن تحتاج الى اكثر من سطر او ذاكرة كبيرة لحفظ الطريقة. تابع الشفرة القادمة.



```
val number1: Int = 55
val number2: Long = number1.toLong()
```

لاحظ قمنا بتحويل Int الى Long باستخدام دالة toLong() وبشكل سلس وبسيط.

قائمة بجميع دوال التحويل :

toByte()

toShort()

toInt()

toLong()

toFloat()

toDouble()

toChar()

لاحظ لا يوجد تغيير للمتغير! Boolean

## رابعاً : برنامج "hello world"

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

```
println("hello world")
}
```

نرى هنا الفرق الواضح بينها وبين لغة الجافا من حيث الاختصار الشديد في كتابة الدالة والواضح في كتابة جملة الطباعة أيضاً.

جملة الطباعة تكتب بطريقتين اما print او println الفرق الوحيد بينهما ان الأخيرة تسمح بترك سطر اسفل الجملة المطبوعة.

## ❖ مثال

```
fun main(args : Array<String>)
{
    var id:Int=105541;
    var carName:String="Corolla";
    var price:Double= 1593.32;
    println("this car is : $carName and its price :$price")
}
```

في جملة الطباعة هذه أردنا كتابة جملة ومن ثم المتغير الذي يحمل قيمتها وفصلنا بينهما بعلامة + ، هنالك أيضاً طريقة أخرى سنتعرف عليها وهي :

```
println("this car is :$carName and its price: $price")
```

## ادخال البيانات

في هذا الدرس سنتعرف على طريقة استقبال البيانات من المستخدم وكيفية التحويل بين المتغيرات .

## أولاً : دالة readLine

---

نستخدم هذه الدالة عندما نريد استقبال قيمة من المستخدم ، مثلا طلبت من المستخدم أن يدخل اسمه و عمره والمرحلة الدراسية فهنا نقوم باستخدام هذه الدالة كما سنرى.

```
println("please enter your name")
var yourName:String= readLine()!!
println("Hello $yourName ")
```

في المثال أعلاه طلبنا من المستخدم أن يدخل اسمه من خلال جملة الطباعة ، ومن ثم عرفنا متغير وحددنا نوعه وقمنا بإسناد الدالة التي تقوم باستقبال هذه القيمة.

تعني قم بقراءة ما سوف يدخله المستخدم.

## ثانياً : التحويل

---

readLine هذه الدالة تستقبل قيمة فقط نصية أي من نوع String، ولكن فلنفرض أننا أردنا من المستخدم إدخال عمره والعمر متغير من نوع صحيح، ما العمل إذا ؟

الحل بسيط جداً كالتالي :

```
println("please enter your age")
var yourAge:Int= readLine()!!.toInt()
yourAge.toInt()
```

عرفنا متغير العمر من نوع عدد صحيح ومن ثم قمنا بإسناد الدالة لتقوم بقراءة ما يدخله المستخدم ومن ثم قمنا بالخطوة الجديدة التالية وهي التحويل :

```
.toInt()
```

هذا السطر يعني أن نقوم بتحويل العمر من متغير نصي إلى متغير من نوع عدد صحيح وذلك لأن دالة

تقرأ النص على أنه سلسلة نصية `readline`.

لو أردنا التحويل إلى متغير نوعه عدد عشري:

```
Age.toFloat()
```

هنا لو أردنا التحويل إلى متغير نوعه عدد عشري مزدوج :

```
Age.toDouble()
```

لا يقتصر التحويل عليهم فيمكننا التحويل إلى أي نوع، باستثناء :

المتغير من نوع عدد صحيح لا يمكن تحويله إلى منطقي وذلك لأن المنطقي يحمل قيمة صح أو خطأ.

```

fun main(args : Array<String>){

    println("please enter length for rectangle :")
    var n1:Int=readLine()!!.toInt()
    println("please enter width for rectangle:")
    var n2:Int=readLine()!!.toInt()
    var distance = n1*n2
    var circumference=2*(n1+n2)
    println("The distance = $distance and the circumference =
    $circumference")

}

```

please enter length for rectangle :

7

please enter width for rectangle:

4

The distance = 28 and the circumference = 22

## التعليقات

لا يخفى علينا أن كتابة التعليقات مفيدة جداً للمبرمجين فهي تعمل على تعريف القطعة البرمجية كفاءتها أو ما الغرض منها في حين أنه نسي ما الهدف الذي من أجله كتبها ، وهي لا تؤثر في عمل البرنامج كما نعلم فلا يتم تنفيذها من الأساس لأن البرنامج لا يستطيع قراءتها .

وللتعليقات نوعان اما ان تكون سطرية أي تعليق في سطر واحد فقط أو متعددة الأسطر، مثال:

### ❖ متعددة الأسطر

```
/*  
 * calculate distance and  
 * circumference for rectangle  
 */  
var length:Int=5  
var width:Int=7  
var distance=length * width  
println(distance)  
var circumference= 2*(length+width)  
println(circumference)
```

### ❖ سطرية

```
var num1=5 //value num1  
println(num1)  
num1--
```

```
println(num1)
```

## العمليات الرياضية

سنتطرق إلى العمليات الرياضية التي تلعب دور مهم وكبير في البرمجة ، سوف نقوم بشرح العمليات ، رموزها ، طرق مختصرة لإجراء العمليات وأيضاً سنتعلم كيفية كتابة التعليقات.

### أولاً : الرموز الرياضية

في البداية نحب أن نقوم بتعريف العمليات الأساسية وهي : الجمع، الطرح، الضرب، القسمة وباقي القسمة.

الرمز	العملية
+	الجمع
-	الطرح
*	الضرب
/	القسمة
%	باقي القسمة

في العمليات الرياضية يراعى عند تنفيذها إذا كانت تتضمن الأقواس، وعملية الضرب/ القسمة، والجمع/ الطرح التالي:

نبدأ بتنفيذ ما بين الأقواس ومن ثم عملية الضرب / القسمة ومن ثم الجمع / الطرح.

هذا مثال يحسب لنا مساحة المستطيل ومحيطه، نجد أن المساحة هي مجرد عملية ضرب عادية ، وبالنسبة للمحيط فهو يعطي أولوية التنفيذ إلى الأقواس ومن ثم يضرب الناتج في ٢.

```
var length:Int=5
var width:Int=7
var distance=length * width
println(distance)
var circumference= 2*(length+width)
println(circumference)
```

هنا قمنا باختصار فلدي متغير يحمل قيمة في البرنامج و اردنا اجراء عملية الضرب عليه فقمنا بكتابة `num *=3` بدلا من كتابة `num=num*3`.

كذلك الأمر لو أردنا اجراء طرح او قسمة نقوم بكتابة اسم المتغير ومن ثم رمز العملية علامة المساواة ومن ثم العدد الذي نرغب بإجراء العملية عليه.

```
var num=9
println(num)
num *=3
println(num)
```

## ثانيا : Decrement & Increment



لنفرض أن لدي متغير وأريد زيادته بعدد واحد فقط سوف نقوم بعملية بسيطة جداً وهي تعرف بالزيادة أو Increment.

```
var num1=5
println(num1)
num1++
println(num1)
```

هنا قمنا بالإختصار فبدلاً من كتابة :

```
num=num+1
```

قمنا بكتابة :

```
num1++
```

لو أردنا طرح عدد واحد من قيمة المتغير سوف نقوم بعملية النقصان أو ما تعرف بـ Decrement

```
var num1=5
println(num1)
num1--
println(num1)
```

## العمليات المنطقية

سننتقل إلى العمليات المنطقية التي هي أيضاً بدورها تلعب دور مهم وكبير في البرمجة، سوف نقوم بشرح العمليات، رموزها، طرق مختصرة لإجراء العمليات وأيضاً سنتعلم كيفية كتابة التعليقات.

## أولاً : الرموز المنطقية

في البداية نحب أن نقوم بتعريف العمليات الأساسية وهي: اصغر، اكبر، النفي، المساواة.

الرمز	العملية
<	اصغر
>	اكبر
=>	اصغر من او يساوي
=<	اكبر من او يساوي
==	يساوي
!	علامة النفي
!=	لا يساوي

### ❖ أمثلة

- هنا نطلب التحقق بين القيمتين number1 و number2 والناتج العائد لنا هو منطقي true / false.

```
var number1:Int=13
var number2:Int=20
println(number1<number2)
```

ناتج التنفيذ:

true

- عرفنا متغير منطقي يحمل قيمة false ولكن نلاحظ وقت الطباعة نفينا القيمة أي انه سيطبع العكس.

```
var isReal:Boolean=false
println ("Ahmed loves Kotlin :"+ !isReal)
```

ناتج التنفيذ:

Ahmed loves Kotlin :true

## ثانيا : البوابات المنطقية "المقارنة"

البوابات المنطقية في البرمجة هي: AND , OR ,NOT .

❖ بوابة AND

لو كان لدي قيمة اريد التحقق منها ولدي قيمة أخرى وأريد التحقق منها أيضاً، نستطيع فعل ذلك عن طريق بوابة AND ولكن يجب ان يكون ناتجهما جميعا متحقق أي انه لا يكون الشرط الأول صحيحا والآخر خطأ.

لذلك نستخدمها كثيرا في التحقق من اسم المستخدم وكلمة المرور عندما يقوم المستخدم من ادخال المعرف الخاص به وكلمة المرور الخاصة أيضا به وكانتا كلتاهما صحيحتان حينها يكون الناتج لدينا true بداخل البرنامج وناتج التنفيذ هو الدخول إلى البرنامج ولكن في حال انه ادخل المعرف الخاص به بشكل صحيح واخطاء في كتابة كلمة المرور فالناتج بالداخل سيكون خطأ وناتج التنفيذ الذي نراه لن يسمح لي بدخول البرنامج.

تمثل برمجياً بهذا الرمز (&&).

وفي المثال نجد أنه لدي قيمتين اريد التحقق منهما كلاهما بأنهما يعيدان لي true.

```
var number1: Int = 7
var number2: Int = 4
println(number1 < 10 && number2 < 5)
```

output : true

❖ بوابة OR

هذه البوابة تخالف AND في انها اذا كان احد الشرطين صحيح فهي تعيد قيمة صحيحة true أي انها لا تشترط ان يكون كلاهما صحيحين.

تمثل برمجياً بهذا الرمز (||).

اذا كان احدهما صحيح على الأقل فهي تعيد true اذا لا تشترط ان يكون كلاهما صحيح كما في AND.

```
var number1:Int = 7
var number2:Int = 4
println(number1 < 10 || number2 > 5)
```

output : true

## ❖ بوابة NOT

تقوم بنفي القيمة المدخلة كما شرحناه في المثال اعلاه.

## ❖ جدول مقارنة

بوابة OR	بوابة AND
تمثل برمجياً بهذا الرمز (  ).	برمجياً بهذا الرمز (&&). لمثال نجد أنه لدي قيمتين اريد ق منهما كلهما بأنهما يعيدان لي true.

إذا كان احدهما صحيح على الأقل فهي تعيد true إذا لا تشترط ان يكون كلاهما صحيح كما في AND.	5 > number2 && 10 > num
<pre>var number1:Int = 7 var number2:Int = 4 println(number1 &lt; 10    number2 &gt; 5)</pre>	<pre>var number1:Int = 7 var number2:Int = 4 println(number1 &lt; 10 &amp;&amp; number2 &gt; 5)</pre>
نتاج التنفيذ true	التنفيذ true

## السلاسل النصية

لا يخفى علينا أن السلاسل النصية هيا من أهم المدخلات والمخرجات أيضاً في أي لغة نتعلمها، كيف نستطيع التعامل معها بالحذف، الإضافة، اختبارها وحتى التعامل مع الأحرف وسوف نتطرق لهذا الحدث في درسنا اليوم.

سنذكر في البداية مثال بسيط تعريفى للسلاسل النصية:

```
var name:String="Ahmed"
println(name)
```

في هذا المثال البسيط قمنا بطباعة الاسم احمد، الآن سنستعرض أهم الدوال التي تتعامل مع السلاسل النصية مستخدمين المثال الحالي:

**length** ❖

عندما نريد معرفة طول السلسلة النصية.

```
var name:String="Ahmed"  
println(name.length)
```

plus ❖

عندما نريد إضافة سلسلة نصية للسلسلة السابقة.

```
var name:String="Ahmed"  
println(name.plus(" aljuaid"))
```

get ❖

عندما نريد استرجاع القيمة التي يشير إليها العنوان أو المؤشر [index].

```
var name:String="Ahmed"  
println(name.get(3))
```

equals ❖

القيمة العائدة من الدالة هيا منطقية فهي تستخدم للتحقق من تطابق القيم وسوف نرى ذلك بالمثال.

```
var name:String="Ahmed"  
println(name.equals("Khaled"))
```

## hashCode ❖

وهي لجلب الترميز الخاص بالسلسلة النصية.

```
var name:String="Ahmed"  
println(name.hashCode())
```

## replaceFirst ❖

تقوم باستبدال اول حرف من السلسلة النصية فقط.

```
var name:String="Ahmed"  
println(name.replaceFirst("A","G"))
```

## reversed ❖

تقوم بعكس السلسلة النصية.

```
var name:String="Ahmed"  
println(name.reversed())
```



## toLowerCase ❖

تقوم بتحويل حروف السلسلة النصية إلى حروف صغيرة.

```
var name:String="Ahmed"  
println(name.toLowerCase())
```

## toUpperCase ❖

تقوم بتحويل حروف السلسلة النصية إلى حروف كبيرة.

```
var name:String="Ahmed"  
println(name.toUpperCase())
```

## removeRange ❖

لو طرأ في بالك حذف حرف من السلسلة النصية أو مجموعة من الحروف المتصلة فكل ما يتوجب عليك فعله هو تحديد العنوان الذي ستبدأ به [index] والذي ستنتهي عنده.

```
var name:String="Ahmed"  
println(name.removeRange(2,4))
```

var name:String="Ahmed" println(name.take(3))	فلنفرض مثلا أنك اردت ان تقتطع جزءً من لسلسلة النصية وتقوم بحذفها كل ما عليك هو د العنوان الذي تريد بدء اقتطاع السلسلة منه.
var name:String="Ahmed" println(name.first())	تعود بأول حرف من السلسلة النصية.
var name:String="Ahmed" println(name.last())	تعود بأول حرف من السلسلة النصية.
var name:String="Ahmed" println(name.drop(1))	هذه الدالة تقوم بحذف حرف من السلسلة بة عن طريق تحديد عنوانها.

## المصفوفات

المصفوفات تعتبر الحل السحري إذا كنت تريد ادخال او طباعة بيانات كثيرة ، فرئيس القسم يستطيع حصر بيانات من هم في قسمه داخل البرنامج عن طريق المصفوفة ، كذلك المعلم يستطيع طباعة بيانات اعداد كبيرة من طلبته فقط بضغطة الزر ، والمصفوفة معروف أنها مجموعة بيانات من نفس النوع . سوف نتعرف على طريقة تعريفها وتعبئتها وطباعتها:

### أولا : تعريف المصفوفة

```
var myArray=Array<Int>(5){0}
```

في البداية نعرف المتغير ومن ثم بعد علامة = نحدد أن نوع المتغير مصفوفة ومن ثم نحدد نوعها <Int> أنها من نوع عدد صحيح (5) تشير إلى طول المصفوفة أي أنها تحوي بداخلها ٥ عناصر، وفي النهاية {0} نحدد أن المؤشر أو index يبدأ من الصفر.

## ثانياً : إدخال قيم للمصفوفة

حسناً الآن عرفنا هذه المصفوفة أريد إدخال قيم بداخلها أي تعبئتها ، نستطيع تعبئتها بهذه الطريقة

```
myArray[0]=12
```

ولكن هذه الطريقة ستكون مرهقة لو كانت المصفوفة لدي طويلة أي تحمل عناصر كثيرة، والحل السحري كالعادة هو استخدام الدوارة أو loop ، فقمنا باستخدام for loop :  
طبعاً هذا المثال الذي سوف نطرحه هو لو أننا أردنا طباعة الأرقام من ٠-٤ بالترتيب من البرنامج:

```
for(i in 0..5)
{
    myArray[i]=i
    println(myArray[i])
}
```

أولاً :

```
for(i in 0..5)
```

الحلقة التي لدي عدد عناصرها ستة ، من ٠ إلى ٥.

ثانياً :

myArray[i]=i تعبر عن إذا كانت الحلقة تبدأ من ٠ فأجعل أيضا قيمة العنصر = ٠ ،  
وهكذا:

```
myArray[0]=0  
myArray[1]=1  
...
```

ثالثاً :

وهي الخطوة الأخيرة بعد التعبئة وهي طباعة العناصر.

ناتج التنفيذ :

```
0  
1  
2  
3  
4
```

حسناً ولكن لو أردنا من المستخدم تعبئة هذه المصفوفة ؟ الحل بسيط جداً :

```
println("Enter Number for ARRAY")
```

```
for (i in 0..4) {  
    myArray[i] = readLine()!!.toInt()  
}
```

```
myArray[i] = readLine()!!.toInt()
```

هذا يعني أنه بكل عنصر من ٠ - ٤ قم بطلب المستخدم أن يدخل رقم ولماذا من ٠ - ٤ وذلك لأن مصفوفتنا عدد عناصرها ٥.

## ثالثا : طباعة عناصر المصفوفة

حسننا هكذا تم أمر الإدخال، تبقى لنا أمر وهو لو أردنا طباعة هذه العناصر التي أدخلها ؟ نستطيع الطباعة إما بال while or for :

```
//printed by for  
println("printed by for")  
for(i in myArray){  
    println(i)  
}
```

```
//printed by while  
println("printed by while")  
var i=0
```

```
while (i<myArray.size){  
    println(myArray[i])  
    i++  
}
```

#### ▪ ناتج التنفيذ :

Enter Number for ARRAY

5

4

3

2

1

printed by for

5

4

3

2

1

printed by while

5

4

3

2

1

#### ▪ لو اردنا طباعة index للمصفوفه باستخدام for تكون الشفرة بهذا الشكل :

```
for ((index, value) in myArray.withIndex()) {  
    println("the element at $index is $value")  
}
```

```
}
```

#### ■ مصفوفة نصية :

```
var student=Array<String>(6){""}  
println("Enter name your student :")  
for(i in 0..5){  
    student[i]=readLine()!!  
}  
  
for(i in 0..4){  
    println(student[i])  
}
```

هنا عرفنا متغير من نوع سلسلة نصية لأسماء الطلاب مع مراعاة {""} القيمة الافتراضية هنا لا نستطيع كتابتها . أو أي ارقام وذلك لأن المصفوفة نصية فنقوم بوضع علامتي التنصيص ونضعها فارغة ، كما ترون هو نفس الأمر احتجنا إلى دورتان الأولى لإدخال الأسماء من المستخدم والأخرى لطباعة بيانات هذه المصفوفة.

## رابعا : التعامل مع المصفوفات

بالطبع اذا كان لدي مصفوفة وهي تجمع كبير للبيانات فسنحتاج إلى دوال سريعة وجاهزة للتعامل مع هذه المصفوفة، في الجدول ادناه اشهر الدوال المستخدمة للمصفوفة:

الاثنان لهم نفس الاستخدام وهو معرفة عدد عناصر المصفوفة فنقوم بكتابة:	count – size
<code>println(myArray.size)</code>	
لجلب عنصر في المصفوفة نقوم باستخدام <code>get</code> ونمرر لها <code>index</code> العنصر:	<code>get</code>
<code>println(myArray.get(index))</code>	
استبدال قيمة عنصر في المصفوفة بقيمة جديدة سنقوم باستخدام <code>set</code> ونمرر لها <code>index</code> العنصر والقيمة الجديدة، هكذا:	<code>set</code>
<code>println(myArray.set(index,new value))</code>	
وهي للتحقق من قيمة العنصر أي ان ناتج التنفيذ العائد سيكون منطقي اما <code>true or false</code> :	<code>equals</code>
<code>println(myArray.equals(value))</code>	
لترتيب عناصر المصفوفة:	<code>sort</code>
<code>println(myArray.sort())</code>	
لإيجاد اكبر عنصر في المصفوفة:	<code>max</code>
<code>println(myArray.max())</code>	
لإيجاد اصغر عنصر في المصفوفة:	<code>min</code>
<code>println(myArray.min())</code>	
لجلب اخر عنصر في المصفوفة:	<code>last</code>
<code>println(myArray.last())</code>	
لجلب اول عنصر في المصفوفة:	<code>first</code>
<code>println(myArray.first())</code>	
فلتر العناصر او البحث داخل المصفوفة:	<code>filter</code>
<code>val search = myArray.filter { x -&gt; x &lt;= 3 }</code>	



هنا سيقوم بطباعة العناصر داخل المصفوفة ذات القيم 3 او اقل.	
لعرض الترميز الخاص بالمصفوفة:	hashCode
<code>println(myArray.hashCode())</code>	
تقوم بعكس عناصر المصفوفة :	reserved
<code>println(myArray.reversed())</code>	

## النطاق

الحدود او النطاق من المسمى نعلم او نفهم ان لها قيمة بداية وقيمة نهاية محدودة وبداخلها ارقام او حروف الخ.

### أولا : النطاق

يعبر عن الأرقام مثلا الواقعة بين ١ إلى ١٠ أو الأحرف من أ إلى ي ، أي انه مجموعة قيم متسلسلة لها قيمة بداية ولها نهاية ، في تعبيراتنا البرمجية نحتاجها فهي توفر الوقت وتعتبر تعبير ممتاز وسريع.

### ❖ أمثلة

لطباعة الأرقام الزوجية من ١ إلى ٢٠ :

```
var evenNumbers=1..20
```

```
for(i in evenNumbers){  
    if(i%2==0)  
    {  
        println(i)  
    }  
}
```

output : 2 4 6 8 10 ..20

البحث عن قيمة ما إذا كانت موجودة في هذا النطاق ام لا:

```
var evenNumbers = 1..20  
var i = 30  
if (i in evenNumbers) {  
    println("the value in the range")  
} else {  
    println("not found")  
}
```

not found

تطبيق نفس المثال على السلاسل النصية:

```
var word:String="Hello , My name is Ahmed"  
  
var letter:Char='b'  
if(letter in word){  
    println("the value in the word")  
}
```

```
} else {  
    println("not found")  
}
```

not found

## ثانيا : القفزة أو step

القفزة هو بمعنى يكون لدي نطاق ولكن اريد عناصر معينة تبعد عن بعضها بمسافة ثابتة، بمعنى آخر نعلم أن الأرقام الزوجية تبعد عن بعضها رقمين فعندما اعمل قفز في النطاق استطيع الوصول إليه.

```
var numbers= 2..20  
for(i in numbers step 2){  
    println(i)  
}
```

output : 2 4 6 8 10 ..20

## ثالثا : التعامل مع النطاق

downTo ❖

لطباعة عناصر المصفوفة تنازليا، مثال:

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

until ❖

لطباعة العناصر تصاعديا، مثال:

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

هنا يبدأ يطبع من رقم ١ ويتوقف عند ٩ ، وذلك لأن until تعني حتى أي كأننا نقول حتى ترى ١٠ فتوقف لو أردنا طباعة الرقم ١٠ فسنجعل الرقم ينتهي عند ١١

دالة جاهزة لو اردنا البحث بداخل النطاق و القيمة العائدة من البحث منطقية، مثال:	contains
<code>var letter="A".. "Z"</code>	
<code>println(letter.contains("D"))</code>	
دالة للتحقق هل النطاق خالي ام لا والقيمة العائدة منطقية، مثال:	isEmpty
<code>var letter="A".. "Z"</code>	
<code>println(letter.isEmpty())</code>	
تقوم بطباعة العنصر الذي يبدأ فيه النطاق "نقط: البداية"، مثال:	start
<code>var letter="A".. "Z"</code>	
<code>println(letter.start)</code>	

طباعة اخر عنصر ينتهي عنده النطاق "نقط؛ النهاية"، مثال:	endInclusive
<code>var letter="A".."Z"</code>	
<code>println(letter.endInclusive)</code>	
الترميز الخاص بالنطاق نفسه، مثال:	hashCode
<code>var letter="A".."Z"</code>	
<code>println(letter.hashCode())</code>	
للتحقق من مطابقة القيم، مثال:	equals
<code>var letter="A".."Z"</code> <code>var newLetter="A".."Z"</code> <code>println(letter.equals(newLetter))</code>	
هنا عرفنا متغير اخر له نفس النطاق ، وقمنا بالاختبار هل هم متساويان ام لا.	
لطباعة عناصر المصفوفة تنازليا، مثال:	downTo
<code>for (i in 10 downTo 0){</code> <code>println(i)</code> <code>}</code>	
لطباعة العناصر تصاعديا، مثال:	until
<code>for (i in 1 until 10){</code> <code>println(i)</code> <code>}</code>	
هنا يبدأ يطبع من رقم ١ ويتوقف عند ٩ ، وذلك لأن until تعني حتى أي كأننا نقول حتى ترى ١٠ فتوقف لو أردنا طباعة الرقم ١٠ فسنجعل الرقم ينتهي عند ١١	

# Visibility Modifiers

## أولا : رؤية المتغيرات

يقصد برؤية المتغيرات وهي مجال رؤيتها في المشروع ولكي نعرفها لابد أن نعرف أنواع القيود على المتغيرات:

الدوال والمتغيرات الذي نقوم بتعريفها دائما هي بالأساس public أي أنه عندما نقوم بالتالي:	Public
<code>var num1:Int=0</code>	
سواء قمنا بوضع الكلمة أم لم نضعها هي تلقائيا هكذا تعتبر. public حدود رؤية المتغيرات المعرفة بهذه الطريقة تكون مرئية على مستوى المشروع كامل	
حدود رؤية المتغيرات والدوال المعرفة بهذه الطريقة تكون فقط في module الحالي:	internal
<code>internal var num2:Int=0</code>	
المتغيرات والدوال التي تعرف بهذا النوع لا يمكن رؤيتها إلا بالتصنيفات الوارثة:	Protected
<code>protected var num3:Int=0</code>	
لا يمكن رؤيته إلا بداخل التصنيف الذي تم تعريفه فيه فقط	private
<code>private var num4 :Int =0</code>	

## ثانيا : أهمية رؤية المتغيرات

---

وضع قيود على المتغيرات تجعل الوصول إلى هذه المتغيرات سهل ، فمثلا لو أردت أن أعرف مجموعة متغيرات وأريد فقط أن أصل إلى متغير واحد في المشروع ككل هنا نستطيع وضعه عام لنتمكن من رؤيته في كافة أجزاء المشروع ، وبقية المتغيرات نستطيع تعريفه على حسب استخدامها هل فقط سأكتفي بأن يكون المتغير في التصنيف الوارث أو في module الحالي.

## ESCAPE SEQUENCE

---

أولا : ماهي الاختصارات أو الأوامر السريعة

### ESCAPE SEQUENCE

---

هي رموز تساعدنا اثناء كتابة الأوامر البرمجية على أداء أمر معين، مثل: نزول سطر، ترك مسافة معينة، البدء من أول السطر...إلخ.

## ثانيا : الرموز ومعانيها

---

❖ الرمز \n

يشير إلى نزول سطر عند عملية الطباعة وهكذا يكتب:

```
print("Hello World\n")
```

## ❖ الرمز \t

لطباعة مساحة بين الكلمات عند عملية الطباعة وهكذا يكتب:

```
println("hello ,\t we learn Kotlin programming")
```

## ❖ الرمز \b

مسح آخر حرف من الكلمة الموضوع عندها الرمز ، وهكذا يكتب:

```
println("hello ,we learn Kotlin programming\b ")
```

## ❖ الرمز \r

تقوم بمسح السطر كاملاً لو وضعت آخر الأمر ، ولو كان في امر طباعة مثلاً ووضعته في المنتصف يمسح ما قبله ، نستطيع الاستفادة من هذا الأمر في عملية تحتاج إلى العد مثلاً لو اردنا قياس نسبة تحميل برنامج معين ألا يظهر لنا في البداية ١٥٪ ، ١٩٪ ، ٤٠٪... إلخ وهكذا نمسح العدد ونضع عدد جديد للتقدم:

```
for(x in 1..800){
```



```
for ( i in 1..500){  
    print("\r"+i)  
}
```

## ❖ الرمز \'

مخصص لطباعة علامة ' لو أردنا طباعتها في النص ، مثال:

```
println("hello ,we learn \'Kotlin\' programming ")
```

## ❖ الرمز \"

مخصص لطباعة علامة " لو أردنا طباعتها في النص ، مثال:

```
println("hello ,we learn \"Kotlin\" programming ")
```

## ❖ الرمز \\$

مخصص لطباعة علامة \$ لو أردنا طباعتها في النص ، مثال:

```
println("Dollar \$ is the currency for United State ")
```

## ❖ الرمز \\

مخصص لطباعة علامة \ لو أردنا طباعتها في النص ، مثال:

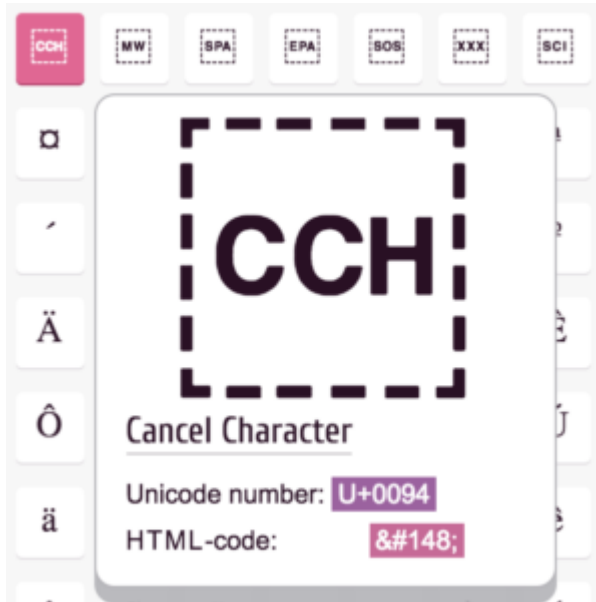
```
println("hello \\ welcome")
```

## UNICODE

تتنوع الرموز والحروف المستخدمة من بلد لآخر واليونيكود هو عبارة عن شفرة خاصة لحروف ورموز مخصصة كما هو موضح في الصورة التالية:

%	‰	°C	€	¥	%	‰	ε	⊕	°F	g	ℋ	ℳ	ℋ	h	h
ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ
SM	TEL	TM	ℳ	ℳ	ℳ	Ω	U	3	ı	K	Å	ℳ	ℳ	e	e
ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ
Σ	⊕	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ	ℳ
¼	½	¾	⅓	⅔	⅓	⅔	⅓	⅔	⅓	⅔	⅓	⅔	⅓	⅔	⅓
I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	L	C	D	M
i	ii	iii	iv	v	vi	vii	viii	ix	x	xi	xii	l	c	d	m
⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
←	↑	→	↓	↔	↑	↘	↙	↗	↖	↗	↖	↗	↖	↗	↖

هذه الرموز المخصصة يمكنك الوصول إليها من خلال الموقع UNICODE ولجلب ال Unicode الخاص بها نقوم بالتالي:



بمجرد الوقوف على الرمز والضغط ظهر لنا مربع اختصار فيه المعلومات التي نحتاجها ونلاحظ رمز Unicode ظهر لنا.

لكتابة الرمز في البرنامج والاستفادة منه نكتب هكذا \u2101:

لابد من رمز \u في البداية ومن ثم الرقم الخاص بالرمز، مثال:

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

```
var c='\u2101'  
println(c)  
}
```

وهنا قمنا بعملية طباعة للرموز الواقعة بين مدى معين:

```
fun main(args:Array<String>){  
    var e1='\u0200'  
    var e2='\u0218'  
  
    for ( i in '\u0200' .. '\u0218')  
    {  
        println(i)  
    }  
}
```

▪ **ناتج التنفيذ :**

À  
à  
Â  
â  
Ë  
ë  
Ê  
ê  
Ì  
ì

Î  
İ  
Ö  
ö  
Ï  
ï  
Ř  
ř  
ŕ  
Ů  
ů  
Ű  
ű  
Ş

## ❖ جلب Unicode الخاص بحرف أو رمز معين:

لو أردت كمستخدم أن تقوم بإدخال حرف معين لمعرفة Unicode الخاص به، مثال:

```
fun main(args:Array<String>){  
    println("enter any character :")  
    var ch:Char = readLine()!!.single()  
    println("unicode of this character is : ${ch.toInt()}")  
}
```

```
var ch:Char = readLine()!!.single()
```

الحرف هو الوحيد الذي لا تستطيع كتابته هكذا toChar, إذا لتحويل القيمة المدخلة إلى حرف نقوم باستخدام single.

`${toInt()}`

عن طريق هذا الأمر سيظهر لنا Unicode.

## ❖ لجلب الحرف أو رمز معين عن طريق إدخال Unicode

```
fun main(args:Array<String>){  
    println("enter the unicode :")  
    var uniNum:Int= readLine()!!.toInt()  
    println("unicode of this character is : ${uniNum.toChar()}")  
}
```

## REGEX (Regular Expression)

### أولا : التعبيرات القياسية

التعبيرات القياسية مفيدة بشكل كبير خصوصا لو أردت أن يدخل المستخدم بيانات ولكن بقيود معينة ، مثلا البريد الإلكتروني تريده أن يقوم بإدخاله بصيغته الصحيحة كالتالي :

[Ahmed1431@gmail.com](mailto:Ahmed1431@gmail.com)

## ثانيا : طريقة التعبير عن التعبيرات القياسية

---

لابد من التعرف على الرموز المستخدمة للتعبير عن الرموز القياسية:

▪ `d\`

مخصص لإدخال رقم من ٩-٠ فقط أي لو قمت بإدخال أي حرف او رمز ستكون النتيجة false.

▪ `w\`

لإدخال أي من القيم التالية 0-9 ,A-Z ,a-z : الحروف سواء كانت كبيرة أو صغيرة وأي رقم من ٩-٠.

▪ `s\`

لطباعة مسافة.

▪ `[]`

للتنسيق بمعنى لو أردت أن تطلب من المستخدم إدخال حروف من a-z ونلاحظ أننا طلبنا حروف صغيرة نقوم بوضعها بين هذه الأقواس المربعة هكذا [a-z] :

▪ {}

نستخدمها لتحديد عدد الخانات المدخلة فمثلا بعد طلبك من المستخدم ادخال اسمه سنقوم بتحديد عدد الحروف المدخلة بحيث أن اسمه لا يتجاوز ٢٠ حرف ولا يقل عن ٥ مثلا هكذا:

[a-z]{5,20}

▪ \* \

لو أردت أن يقوم المستخدم بإدخال خانات كثيرة وفي حال أيضا تركت له خيار بأن لا يقوم بالإدخال أيضا. مثلا:

\w\*

▪ + \

لو أردت أن يقوم المستخدم بإدخال خانات كثيرة أو على الأقل لابد من إدخال خانة واحدة بمعنى لا مجال لتركه فاضي إما إن تقوم بملء الفراغ بخانات كثيرة أو على الأقل أدخل خانة. مثلا:

\w+



▪ \?

يدع المجال للمستخدم إما أن لا يدخل أبداً أو يدخل خانة واحدة فقط ، مثلا:

\w?

▪ ()

لتجميع التنسيق الذي تخطط له ، مثلا:

("\\d{4}")

نلاحظ أننا وضعنا قوسين دائريين ومن ثم علامة التنصيص.

▪ |

لنفرض أنك طلبت من المستخدم إدخال بيان معين ويحتمل إدخاله أكثر من شكل (تنسيق) فنستخدم هذه العلامة للتخيير بمعنى لو أدخل التنسيق الذي عرفناه أو التنسيق الآخر فهو صحيح، مثلا:

لو طلبت من المستخدم إدخال رقم الهاتف النقال فأنت تتوقع إدخاله بطريقتين إما أن يقوم بإدخال ١٠ ارقام كما هو معتاد أو إما عن طريق وضع مفتاح الدولة ومن ثم ٩ ارقام هكذا:

```
var exp = Regex("\\d{10}|")
```

```
var exp1 = Regex("\\+966\\d{9}")
```

وعند الاختيار سنقوم هكذا:

exp | exp1

▪ للوصول إلى الرموز ك (& ، \* ، ؟ ، # ، @ ، ! ، ، ، ... إلخ)

\\?

\\\$

\\&

\\\*

▪ . Dout

تستخدم للتجاهل فمثلا تريد تجاهل خانة أو أكثر في القيمة المدخلة، مثلا:

```
\\d..\\d{4}
```

نلاحظ طلبنا أنه يدخل في البداية رقم ثم . لتجاهل خانة و . مرة أخرى لتجاهل خانة ثانية ومن ثم رقم.

ثالثا: مثال:

```
fun main(args:Array<String>){  
  
    println("Hello , please complete this form :\n Enter your name :")  
    val nameEx = Regex("\\w{3,10}\\s\\w{3,15}")
```

```

val name = readLine()!!.toString()
if(nameEx.matches(name)){
    println(" Enter your phone number :")
    val phoneEx = Regex("\\+966\\d{9}")
    val phone = readLine()!!.toString()
    if(phoneEx.matches(phone)){
        println(" Enter your Email :")
        val emailEx = Regex("(^[a-zA-Z0-9_+]{2,25})+@[a-zA-Z0-9-]{5,12}+\\. [a-zA-Z0-9-]{2,3}")
        val email = readLine()!!.toString()
        if(emailEx.matches(email)){
            println("thank you")
        }
        else {
            println("please enter correct email")
        }
    }
    else {
        println("please enter correct phone name beginning with +966")
    }
}
else {
    println("please enter correct name")
}
}

```

## ■ ناتج التنفيذ

Hello , please complete this form :  
Enter your name :

ahmed aljuaid  
Enter your phone number :  
+966123456789  
Enter your Email :  
ahmed12.ahmed@gmail.com  
thank you

## رابعاً : الدوال المستخدمة مع التعبيرات القياسية

---

Find ❖

القيمة العائدة [matchedResult – null].

تتحقق هذه الدالة من تطابق النص المدخل مع التنسيق المخصص له طالما كان متطابق يعود بالقيمة المدخلة، وعند عدم التطابق يعود بـ null.

▪ مثال

```
println("enter your phone number:")  
var phone = readLine()!!.toString()  
  
val phoneEx :String? = Regex("\\+966\\d{9}") // like : +966561234567  
    .find(phone)?.value  
println(phoneEx)
```

▪ ناتج التنفيذ

ادخلنا قيمة غير مطابقة للتنسيق فنلاحظ انه سيطبع null.

## ❖ findAll

القيمة العائدة : تعود بجميع القيم المتطابقة مع التنسيق المطلوب. مثلا:

```
val matchedResults = Regex(pattern = "\\d+").findAll(input = "1as232v4")

for (i in matchedResults) {
    print(i.value + " ")
}
```

## ▪ ناتج التنفيذ

التنسيق المطلوب في المثال هو رقم فقط فنقوم بعمل حلقة تعود بالأرقام الموجودة في النص مع طباعة مسافة:

1 232 4

نلاحظ أنه طبع رقم ١ ومن ثم مسافة ومن ثم طبع الأرقام ٢٣٢ متتالية بدون مساحة لأنها في النص أتت مع بعضها لم يفصلها حرف أو رمز ومن ثم مساحة وهكذا يستمر عملها.

## ❖ matchEntire

القيمة العائدة: [matchedResult – null].

للتحقق من القيمة المدخلة عند التطابق مع التنسيق المخصص له طالما كان متطابق يعود بالقيمة المدخلة ، وعند عدم التطابق يعود بـ null، مع التنويه أن المسافة تحتسب كحرف فعند إذ تعود بـ null لو كان التنسيق المطلوب رقم فقط.

```
println("enter your age:")  
  
val age= Regex("\\d+").matchEntire("25")?.value  
  
println(age)
```

## ▪ ناتج التنفيذ

عند التطابق طبع العمر وعند الاختلاف يعود null

```
enter your age:  
25
```

**matches** ❖

القيمة العائدة: [true – false].

للتحقق من تطابق القيمة المدخلة كاملة مع التنسيق المخصص له طالما كان متطابق يعود بـ true ، وعند عدم التطابق يعود بـ false.

## ▪ مثال

```
val regex = Regex(pattern = "\\d+")
println(regex.matches(input = "50 dollars"))
```

## ▪ ناتج التنفيذ

false

## containsMatchIn ❖

القيمة العائدة: [true – false].

إذا كان جزء من القيمة المدخلة متطابقة مع التنسيق المخصص له يعود بـ true، وعند عدم التطابق يعود بـ false، وهي عكس الدالة matchEntire التي تعد أن القيمة غير متطابقة.

## ▪ مثال

```
var age = Regex("\\d+").containsMatchIn("25 years")
println(age)
```

## ▪ ناتج التنفيذ

true

نلاحظ أنه عاد بـ true وذلك لأن الوظيفة الرئيسية ليست تطابق جميع القيمة المدخلة إنما يتحقق هل أدخلت رقم كما هو مطلوب في التنسيق.

القيمة العائدة : تعيد لنا قائمة تخلو من التنسيق الذي أنت حددته. مثال:

```
println("enter your age:")
var testSplit = readLine()!!.toString()
val a = Regex("[a-zA-Z]").split(testSplit)
println(a)
```

طلبنا منه أن يعود بجميع ما تقوم بإدخاله باستثناء Regex("[a-zA-Z]") جميع الحروف لا تعود.

#### ▪ ناتج التنفيذ

```
enter your age:
28
[28]
```

القيمة العائدة: القيمة النصية مع استبدال ما تم تحديده في التنسيق من حرف او رقم او رمز بقيمة أخرى. مثال:

```
val testReplace = Regex("""\d+""").replace("ab12cd34ef", "x")
println(testReplace)
```



نلاحظ أن `Regex("""\d+""")` حددنا أننا نريد استبدال كل رقم في النص `replace("ab12cd34ef","x")` حددنا ومن ثم حددنا الحرف الذي نريده أن يحل مكان الرقم الذي نريد تبديله.

إذا تقوم باستبدال قيمة معينة في النص بقيمة أخرى عن طريق تحديدها في التنسيق.

#### ▪ **ناتج التنفيذ**

abxcdxef

# بنى التحكم

## For Loop

فلنفترض أننا أردنا كتابة جميع الأرقام الواقعة من ١-١٠ في برنامجنا، هل هذا ممكن؟ نعم، ممكن حسناً ولو أردنا كتابة الأرقام من ١-٢٠، هل هذا ممكن؟ سوف نقول نعم ولكن لنفكر قليلاً بأرقام أكبر مثلاً من ١ - ١٠٠ سوف تجاوب وبسرعة إنه أمر مرهق أن اكتب كل الأعداد بداخل البرنامج، حسناً مثال آخر لو وضعت شرط أردنا تكرار ظهور رسالة معينة مادام هذا الشرط صحيح، هنا تظهر فائدة الحلقات التكرار التي تقوم بتكرار الأمر البرمجي في حال وضعت له شرط أو كما في مثال الأرقام الذي ذكرناه أردنا طباعة مجموعة الأرقام.

أنواع الحلقات التكرارية: (for loop – while – do..while)

### أولاً : الجملة for loop

نستخدمها عندما يكون لدي عدد معلوم للتكرار الذي أريده.

الصيغة العامة:

```
for (item in collection) {  
    // body of loop  
}
```

هنا أردنا طباعة الأعداد من ١-١٠ أي أن النطاق عندي معلوم فقط ١٠ ارقام اريد طباعتها مع مراعاة ان النقطتين بين العددين مهمين لأننا نريد الأعداد التي تقع في هذا النطاق.

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

هنا نريد من المستخدم أن يدخل اسمه ومن ثم نريد طباعة هذا الاسم حرف حرف أي في كل سطر حرف

```
println("please enter your name :")  
var name:String= readLine()!!  
for(i in name)  
{  
    println(i)  
}
```

ونستطيع تطبيق ما تعلمناه في درس السلاسل النصية وهو استخدام الدوال الخاصة بها.

```
println("please enter your name :")
```

```
var name:String= readLine()!!
for(i in name.reversed())
{
    println(i)
}
```

## ثانيا : nested for والمقصود هنا الحلقات المتداخلة

مثلا لو أردنا طباعة جدول الضرب من ١-٥ جدول ١ و جدول ٢ و جدول ٣ وهكذا وصولاً لـ ٥ ، ولكننا لا نريد طباعة جدول الضرب كاملا من ١-١٠ سوف نكتفي من ١-٣، سوف نستخدم حلقتين متداخلتين كما في المثال.

قيمة المتغير i في الحلقة الأولى تعبر عن عدد جداول الضرب الذي نريدها فنحن نريد من جدول ١ إلى جدول ٥ ، ومن ثم قمنا بفتح الأقواس المربعة وكتبنا بداخلها الحلقة الثانية وهي ما تعرف بالمتداخلة لأنها دخلت على الأولى ، قيمة المتغير x تعبر عن أننا نريد بدء الضرب من رقم ١ والتوقف عند ٣ لا نريد طباعة الجدول كامل.

```
for(i in 1..5){
    for(x in 1..3) {
        println("$i * $x = " + i*x)
    }
    println("-----")
}
```

## ❖ برنامج اختبار الرقم زوجي ام فردي

```
for( i in 1..3) {  
    println("please enter number :")  
    var num: Int = readLine()!!.toInt()  
    if (num % 2 == 0)  
    { println("number is even")}  
    else  
    { println("number is odd")}  
}
```

## ❖ ناتج التنفيذ

please enter number:

4

number is even

please enter number:

21

number is odd

please enter number:

10

number is even

# while & do while

## الجملة while

يؤدي الأمر while الى تنفيذ الشفرة البرمجية مراراً وتكراراً طالما ان الشرط متحقق بـ True

```
while( condition)
{
    statements
    .
    .
    counter
}
```

هنا افترضنا ان المتغير قيمته ٢ ومن ثم قلنا طالما أن i اقل من ١٠ قم بطباعة قيمة ، حسنا ولكن ماذا تعني i++ هذا يعني أننا نريد زيادة العدد ومن ثم نختبره مرة أخرى حتى يصل إلى ٩ يقوم بطابعته ومن ثم يتوقف البرنامج لماذا؟ لأن الشرط يريد طباعة الاعداد التي تقل عن ١٠ فعندما نصل للعشرة لن يتحقق الشرط أي يصبح غير صحيح ومن ثم يتوقف ، لو اننا قلنا الشرط هكذا  $i \leq 10$  هنا سوف يتحقق من قيمة ١٠ ومن ثم يتوقف وذلك لأن علامة المساواة متواجدة أي هل الرقم اقل

من ١٠ او يساويها اذا كان نعم فالشرط صحيح وإذا العكس فتوقف.

```
var i=2
while(i<10)
{
    println(i)
    i++
}
```

## nested while

---

المهم هو ان لا ننسى كتابة العداد  $x++,y++$  وعندما لا نكتبها كليها او احدهما سوف يقوم بطباعة للما لانهاية لان الرقم لن يتغير وسيضل صحيح وهي ما تسمى بـ infinitive loop.

```
var x =1
var y =1
while(x<=5)
{
    while(y<=3)
    {
        println("$x*$y="+x*y)
        y++
    }
    x++
}
```

## الجملة do...while

---

في هذه الحلقة الأمر مختلف قليلاً هو أنه حتى لو الشرط غير متحقق فسوف يقوم بتنفيذ الأمر لمرة واحدة ، مثال توضيحي على ذلك.

هنا في البداية عرفنا المتغير x وقيمته ٥ ، البرنامج سوف يدخل إلى do في البداية وينفذ ما بداخلها وهي طباعة القيمة ولا ننسى أيضاً هنا كتابة العداد الذي هو الزيادة ومن ثم يقوم بالتحقق من الشرط وهو قيمة x بعد الزيادة هل هي اكبر من ١٠ ام لا والشرط هنا غير متحقق لان ٦ اصغر من ١٠ فيتوقف البرنامج وبذلك يكون فقط القراءة الأولى هي التي تنفذ بدون النظر إلى الشرط.

```
var x = 5
do{
    println(x)
    x++
}
while(x>10)
```

هنا فتحنا do ومن ثم طلبنا من المستخدم ادخال سلسلة نصية اسمه وسوف يقوم بطابعته ومن ثم يقوم بالدخول إلى while للتحقق من الشرط في حال ان المستخدم ادخل اسم حروفه تتجاوز ١٠ احرف سوف تتكرر ظهور الرسالة ، ولكن لو ادخل اسم حروفه اقل من ١٠ سوف يتوقف البرنامج لأن الشرط لم يتحقق.

```
do{
    println("please enter your name :")
    var x:String= readLine()!!
    println(x)
```



```
}  
while(x.length > 10)
```

## when

when هي الجزء المطور والبديل عن switch-case في لغات البرمجة بحيث انها اكثر مرونة ومنطقية وقابلة للتخصيص والتعديل باكثر من نوع من المتغيرات . والجميل ايضاً اننا يمكننا استخدامها ك extension او ك statment بشكل سريع.

### ❖ الصيغة العامة لها تكون بهذا الشكل

```
var dayOfWeek = 4  
when(dayOfWeek) {  
    1 -> println("Monday")  
    2 -> println("Tuesday")  
    3 -> println("Wednesday")  
    4 -> println("Thursday")  
    5 -> println("Friday")  
    6 -> println("Saturday")  
    7 -> println("Sunday")  
    else -> println("Invalid Day")  
}  
// Displays - "Thursday"
```

نلاحظ اننا قمنا ايضاً باستخدام else اذا قام المستخدم بادخال رقم خارج الحدود سيطلع له ان اليوم غير صحيح.

## ❖ الدمج بين اكثر من قيمة او شرط

اذا كان لديك اكثر من حالة وتود اختبارها في when واحدة سيتحتم عليك وضع فاصلة بين كل احتمال وآخر كما ستلاحظ في الشفرة القادمة.

```
var dayOfWeek = 6
when (dayOfWeek) {
    1, 2, 3, 4, 5 -> println("Weekday")
    6, 7 -> println("Weekend")
    else -> println("Invalid Day")
}
// Displays - Weekend
```

## ❖ استخدامها مع النطاق

```
var dayOfMonth = 5
when(dayOfMonth) {
    in 1..7 -> println("We're in the first Week of the Month") !
    in 15..21 -> println("We're not in the third week of the Month")
    else -> println("none of the above")
}
```

```
// Displays - We're in the first Week of the Month
```

## ❖ استخدامها كـ is

إذا اردت التحقق من نوع المتغير يمكنك استخدام is مع when بالشكل التالي.

```
var x : Any = 6.86
when(x) {
    is Int -> println("$x is an Int")
    is String -> println("$x is a String")
    !is Double -> println("$x is not Double")
    else -> println("none of the above")
}
// Displays - none of the above
```

## ❖ استخدامها كبديل لـ if-else-if

يمكن استخدام when كبديل لـ if-else-if ستكون افضل وارتب للكود وأكثر منطقية عندما تكون الاحتمالات كثيرة كما ستلاحظ في الشفرة القادمة.

```
var number = 20
when {
    number < 0 -> println("$number is less than zero")
    number % 2 == 0 -> println("$number is even")
    number > 100 -> println("$number is greater than 100")
    else -> println("None of the above")
}
```

## الجملة الشرطية

كما نعلم جميعنا أن الاحتمالات والمقارنات أمر وارد في الحياة كما أن اختبار الأشياء يجعلنا نتحقق من قيمتها كمثال بسيط عندما يكون لديك مادة دراسية معدل النجاح بها من ٦٠ فأنت هنا سوف تبدأ اختبار درجتك التي حققتها هل هي اكبر او اقل من معدل النجاح ، وإن كانت أكبر فمن الطبيعي سوف يتبادر لذهنك هل أنا ناجح بدرجة ممتاز أم جيد ؟.

من هذه المقدمة يتضح لنا جلياً أن الدرس سوف يكون عن الجمل الشرطية وطريقة كتابتها.

في البداية سنوضح أن الجمل الشرطية في لغة Kotlin هي ( when – else ..if ).

## الجملة if

الصيغة العامة:

```
if (testExpression) {  
    // codes to run if testExpression is true  
}
```

```
else {  
    // codes to run if testExpression is false  
}
```

يمكن كتابة if الشرطية لتحقيق من شرط واحد وتكون بشكل بسيط.

إذا تحقق الشرط نفذ ما بداخل if إذا لم يتحقق تجاوز بدون ان يكون هناك اي احتمال آخر.

```
if ( 5 < 10 )  
    print( "العدد خمسة اقل من العدد عشرة ، متفاجئ حقيقة" )
```

## if ..else

ربما يكون لدينا احتمالين لل if كما سنلاحظ في المثال القادم ولذلك سنحتاج الى استخدام else طريقة كتابة شرط بسيط عبارة الشرط وإذا لم يتحقق فسوف يتنفذ أمر آخر ،

### ▪ مثال

هنا سوف يطلب من المستخدم إدخال اسم حسناً و ومن ثم يقوم بمقارنة الاسم المدخل هل هو أحمد؟

إذا كان نعم فسيطبع له مرحباً أحمد وإذا لم يتحقق الشرط وأدخل المستخدم اسم آخر سوف يطبع له خطأ.

```
println("Please enter your name :")
var Name:String= readLine()!!

if(Name == "Ahmed")
    println("Welcome $Name")
else
    println("Error")
```

#### ▪ ناتج التنفيذ

Welcome Ahmed

## if ..else if

ربما تحتاج او لديك اكثر من شرط تريد التحقق اي منهم قد تنفذ

#### ▪ مثال

مثلا كموضوع المادة الدراسية التي طرحناها في المقدمة ، جميعنا نعلم أن من يحصل على ٩٠ وأكثر يأخذ ممتاز ، ومن يحصل من ٨٠ إلى ٨٩ يأخذ جيد جداً ... وهكذا.

```
println("Please enter your mark :")
var mark: Float = readLine()!!.toFloat()
```

```
if(mark >= 90)
    println("$mark Excellent")
else if(mark >= 80 && mark < 90)
    println("$mark Very Good")
else if(mark >= 70 && mark < 80)
    println("$mark Good")
else if(mark >= 60 && mark < 70)
    println("$mark study hard")
else
    println("$mark failed")
```

## ▪ ناتج التنفيذ

Please enter your mark :

78

78.0 Good

## If المتداخلة

بعض الشروط او المتطلبات تعتمد على متطلبات متداخلة.

الصيغة العامة:

```
if( boolean_expression 1) {

    /* Executes when the boolean expression 1 is true */
```

```

if(boolean_expression 2) {

    /* Executes when the boolean expression 2 is true */

}

}

```

مثلاً اردنا التحقق من رقمين s & a اذا كان a يساوي 100 تحقق الشرط سيدخل داخل if ليجد if اخرى تريد التحقق من هل s تساوي 200 وهكذا .. الخ.

```

var a = 100
var s = 200
/* check the boolean condition */
if( a == 100 ) {
    /* if condition is true then check the following */
    if( s == 200 ) {
        /* if condition is true then print the following */
        print("Value of a is 100 and s is 200\n" );
    }
}
}

```

## التعبيرات مع if

من الاشياء الجميلة في kotlin ولا توجد في جافا هي استخدام if في تعريف المتغيرات مباشرة واعادة قيمة للمتغير المعروف.



يعكس هذا اختصار كبير في كتابة الشفرات البرمجية وترتيبها.

## ■ مثال

```
val a = 100
val b = 200

val max = if (a < b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
print("max = $max")
```

عرفنا متغيرين واسندنا لهم قيم وفي المتغير الثالث عرفنا max سيحصل على القيمة الاعلى بين a & b.

يجب ان تحتوي if على احتمالين او اكثر لخدمتها ك Expression.

# الدوال

## الدوال

الدوال اشهر من نار على علم في عالم البرمجة جميعنا نعلم أهمية استخدامها وانها تختصر الوقت علينا في تكرار العمليات وتوفير الجهد.

### أولا : طريقة كتابتها

تعريفها جدا بسيط نسمي الدالة ويفضل تسميتها كما نعلم جميعا باسم يمثلها أي يمثل الغرض الذي من اجله انشأتها ، Unit:هذا يعني نوع البيانات العائدة وهنا يقصد أنها لا تعود بشيء ومن الممكن أن لا نكتبها وهو سيعرف انها دالة لا تعيد قيمة ، ومن ثم نقوم بكتابة الأوامر التي اريدها بداخل الاقواس.

```
fun functionName( parameter ):Unit  
{  
    return ...  
}
```

### ثانيا : أنواع الدوال

## ❖ نمرر لها بيانات وتعود بقيمة

```
fun maxNumber( number1:Int ,number2:Int):Int
{
    if(number1<number2)
    {
        return number2
    }
    else {
        return number1
    }
}
```

مررنا لها متغيرين وهما number1 ,number2 وتعود بقيمة العدد الأكبر.

## ❖ نمرر لها بيانات ولا تعود بقيمة

```
fun circleCirc(radius:Float):Unit
{
    val bai=3.14
    var circumference = bai *2 * radius
    println("the circumference of circle is :"+circumference)
}
```

## ❖ لا نمرر لها بيانات وتعود بقيمة

```

fun minNumber( ):Int
{
    var number1:Int = readLine()!!.toInt()
    var number2:Int = readLine()!!.toInt()
    if(number1<number2)
    {
        return number1
    }
    else {
        return number2
    }
}

```

❖ لا نمرر لها بيانات ولا تعود بقيمة

```

fun welcome(){
    return println("welcome world")
}

```

## ثالثاً : الاستدعاء

حسناً نحن عرفنا الآن الدوال وطريقة كتابتها، تبقى علينا معرفة طريقة استدعائها وهي جداً بسيطة كالتالي:

نقوم بكتابة اسم الدالة وان كانت تمرر لها قيم فلا ننسى كتابة هذه القيم.

## ▪ مثال:

دالة تحسب مساحة المربع:

```
fun distance( a :Int ):Int{  
    return a*a  
}
```

دالة تحسب محيط المربع:

```
fun circumference(a:Int):Int{  
    return a*4  
}
```

في الدالة الرئيسية:

```
fun main(args:Array<String>){  
  
    println("enter the length side for square :")  
    var len:Int= readLine()!!.toInt()  
    println("the distance is :"+distance(len))  
    println("the circumference is :"+circumference(len))  
}
```

## ▪ ناتج التنفيذ:

enter the length side for square :

3

the distance is :9  
the circumference is :12

## Extension function

---

تعتبر دالة عادية كما سبق وتعرفنا ولكن كيف تكون امتداد او extension ؟ او لنوضح السؤال أكثر ما فائدتها ؟

ماذا لو اردنا اضافة دالة غير موجودة مسبقاً في class string مثلاً ؟ حتماً ستحتاج الى extension وسأوضحها هنا في مثالين:

حسننا نستطيع بناء دالة تكون امتداد لطبقة ما class دون التعريف عنها ، مثال على ذلك:

### ❖ طبقة {class} خاصة بالسيارة

لنفرض اننا قمنا بانشاء class لـ car وكان يحتوي على func واحدة فقط وهي لحساب السرعة اذا كانت منخفضة.

```
class Car{  
    fun LowSpeed(sp:Int):Boolean{  
        return sp < 100  
    }  
}
```

## ❖ دالة extension للطبقة car

الان بعد فترة من الوقت احتجت ان اقوم باضافة دالة جديدة ل class car ولكن لا اريد التعديل عليه ؟ ساقوم باستخدام extension كما ستلاحظ في الشفرة التالية:

```
fun Car.highSpeed(sp:Int):Boolean{  
    return sp > 100  
}
```

نلاحظ أنها دالة عادية ولكن لتكون امتداد لطبقة ما نقوم بوضع اسم الطبقة ومن ثم اسم الدالة كما هو ملاحظ لدينا Car.highSpeed

## ❖ في الدالة الرئيسية

```
fun main(args:Array<String>){  
  
    var car1= Car()  
    println("please enter the speed:")  
    var speed :Int = readLine()!!.toInt()  
    println("speed is : "+car1.LowSpeed(speed))  
    println("speed is : "+car1.highSpeed(speed))  
  
}
```

لدينا أنواع المتغيرات التي هي (Int – String – Char ...)

إلخ) وهي تعتبر classes - طبقات بالأساس ولها دوالها الخاصة بها وخصائصها ومن هنا نستطيع القول أنه ينطبق عليها أيضا ما فعلنا سابقا حيث نستطيع انشاء دالة امتداد تكون لطبقة String مثلا أو Int أو أي طبقة.

## ❖ المثال الثاني

اريد اضافة extension لـ string class ليقوم بتحويل المسافات الى camelCase لاحظ الشفرة القادمة

```
fun String.spaceToCamelCase() { ... }
```

كتبنا اسم الكلاس String وقمنا بانشاء الدالة الجديدة spaceToCamelCase والتي ستقوم بتحويل النص وطريقة استخدامها كالتالي

```
"Convert this to camelcase".spaceToCamelCase()
```

## lambda function

افضل ان اسميها الدالة المضمنة وذلك لأن من الوهلة الأولى عند النظر إليها تستنكر أنها دالة فنحن كما تعلمنا أننا نقوم بأنشاء متغير وفتح قوسين ومن الممكن تمرير متغيرات لهذا القوسين الدائريين ومن ثم الأقواس المربعة وبداخلها الجمل البرمجية



المراد تنفيذها ، حتى تأتينا lambda وتكسر هذه القاعدة وتصبح مضمنة بداخل متغير أي أننا نعرف متغير القيمة الذي يحملها هي دالة.

الصيغة العامة تكون بهذا الشكل

```
val printMessage = {  
    println("Hello, world!")  
}
```

يمكن استدعاء الدالة او تنفيذها بكتابة الشفرة

```
printMessage()  
// or  
printMessage.invoke()
```

اما اذا اردنا تمرير متغيرات الى الدالة فاننا سنستخدم الشفرة القادمة وسيتم تضمين المتغيرات داخلها كما ستلاحظ

```
var printName = { name :String -> println("Hello $name")}
```

نقوم بفتح أقواس مربعة وبداخلها نعرف المتغيرات التي سيتم تمريرها ومن ثم السهم ويليه جسم الدالة وهي الأوامر المراد تنفيذها.

## ■ مثال

```
val total : (Int , Int)->Int = {num1 , num2 -> num1+num2}
```

## ▪ وداخل الدالة الرئيسية

```
fun main(args:Array<String>){  
    println("the total is :"+total(6,5))  
}
```

## high level function

---

الدالة من المستوى العالي وهذا يعني أن هذه الدالة لا تكتفي فقط بتمرير متغيرات فردية وإنما أيضا تقوم بتمرير دالة ومن الممكن ان ترجع دالة.

### ▪ لكي نستطيع القول أنها دالة عالية المستوى لابد من توفر الاتي

- تمرر لها دالة كوسيط.
- تعيد دالة.
- نستطيع فعل الخطوتين السابقتين أيضا معاً.

ويرتبط استخدام `lambda` ارتباط وثيق بالدالة عالية المستوى فهس تعمل على ترتيب الجمل البرمجية وجعله اسهل كما لا يخفى علينا تقلل من الأسطر البرمجية مما يعطينا شكل بسيط ونظيف.

سنقوم بعرض مثال لتوضيح مفهوم الدالة عالية المستوى:

## ❖ الخطوة الأولى

انشأنا دالة بسيطة تقوم بتمرير متغيرين:

– متغير من نوع عدد صحيح ونريد ان نطبع العدد الذي يسبقه والعدد التالي له.

– المتغير الثاني : دالة تقوم بطباعة العدد المدخل (مع العلم نستطيع أن ننشئ دالة لا علاقة لها بالمتغير السابق كأن تقوم بإنشاء دالة تلقي التحية مثلاً).

```
fun calculat( num1:Int , printNumber:(Int)->Unit){  
  
    var PreviousNumber = num1 - 1  
    var NextNumber = num1 + 1  
    printNumber(num1)  
    println("the previous number is : $PreviousNumber and the next  
    number is : $NextNumber")  
  
}
```

لنركز قليلا بالدالة الممررة وهي `printNumber:(Int)->Unit` : وهذا يعني أننا عندما نمررها فقط نكتفي بكتابة نوع البيان المدخل وهو كما هو موضح لدينا من نوع عدد صحيح ومن ثم السهم الذي هو أساسي ومن ثم نوع الخرج أو البيان العائد لنا وهو في المثال لا يعود بقيمة

## ❖ الخطوة الثانية

سننشئ دالة مضمنة بداخل متغير:

```
val myNumber : (Int)->Unit = {number -> println(number)}
```

حسناً سنتحدث عن هذا الآن قليلاً، هذه هي الصيغة التي تعلمناها كما سبق لنا ولكن نريد أن نوضح أن أهميتها تكمن في أننا كما سنرى في الخطوة الثالثة بعد قليل أنها توفر علينا الوقت بدل من أن ننشئ دالة جديدة أخرى عند الاستدعاء وسوف نرى ذلك.

## ❖ الخطوة الثالثة

الاستدعاء وهي الخطوة النهائية:

```
fun main (args:Array<String>){  
    println("please enter any number :")  
    var number1 : Int = readLine()!!.toInt()  
    calculat(number1, myNumber)  
}
```

## مفهوم jump & return

ببساطة شديدة نستطيع القول عن jump هو انك عندما تريد استدعاء دالة لتنفيذ ما بداخلها فهو في عملية الاستدعاء يقفز او يقوم بعملية الـ jump إلى الدالة نفسها لتنفيذ ما بداخلها.

حسناً القيمة الراجعة من تنفيذ الدالة تسمى عملية return او ارجاع أي بمعنى لو لدي دالة تجمع عددين وتعود بالناتج لطباعته على الشاشة فهذا حصلت عملية العودة او الرجوع بقيمة معينة ولذلك عندما تكون لدينا دالة تعيد قيمة معينة يجب علينا تعريف متغير يتسقبل هذه القيمة بداخله ، وسنمثل ما شرحناه سابقا بمثال بسيط للتوضيح:

```
fun sumFunction(num1:Int,num2:Int):Int{  
    return num1 + num2  
}
```

هذه الدالة وبكل بساطة تقوم بإعادة ناتج الجمع كما وضحنا مسبقا ، سنرى الآن كيف يتم الاستدعاء:

```
fun main(args:Array<String>){  
  
    var result = sumFunction(4,5)  
    println(result)  
}
```

هنا نرى في جملة الاستدعاء انها أسندت إلى متغير وذلك لتخزين القيمة الراجعة ، لو لم تكن الدالة تعود بقيمة فهذا نستطيع ان نكتفي بالاستدعاء فقط بدون الاسناد لمتغير.

## برمجة كائنية التوجه

## أولاً : البرمجة كائنية التوجه

---

سوف نتطرق لمفهوم جديد وهو البرمجة كائنية التوجه أو شيءية ( OOP ) — (Object-oriented programming) التي تهدف لجعل البرنامج عبارة عن مجموعة من الكائنات او بالمفهوم المتداول عنه ” كل شيء عبارة عن كائن ” . كل كائن عبارة عن حزمة من البيانات، ويتم بناء البرنامج بواسطة استخدام الكائنات وربطها مع بعضها البعض وواجهة البرنامج الخارجية باستخ دام هيكلية البرنامج وواجهات الاستخدام الخاصة بكل كائن .وتشمل البرمجة كائنية التوجه مبادئ رئيسية وهي:

- التغليف.(Encapsulation)
- إخفاء البيانات.(Data Hiding)
- الميراث.(Inheritance)
- تعدد الأشكال.(Polymorphism)
- من اللغات التي تعمل بالكائنية:
- لغة سي++
- لغة جافا
- لغة بايثون
- لغة دلفي (لغة برمجة)

## ثانياً : مقدمة عن Classes & Object

---

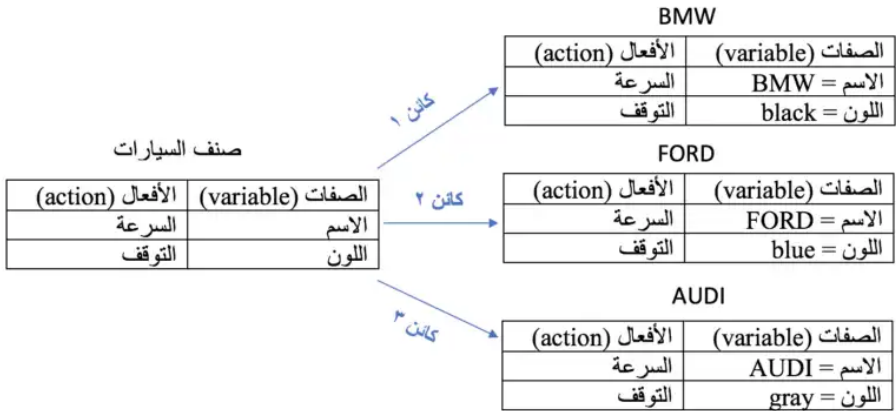
حسناً ما هو الكائن Object ؟

كل شيء تراه أمامك هو كائن ، الكتاب يعتبر كائن ، السيارة كائن ، القلم كائن وحتى أنت يا عزيزي القارئ في برمجتنا هذه تعتبر كائن ، وكل كائن يتكون من شيئين رئيسيين وهما الصفات والأفعال ، فالصفات هي كل سمة تمتلكها من حيث الاسم ، عمرك ، لون شعرك ، لون عينيك ، الطول ، الوزن والكثير من الصفات أيضا غيرهم ، أما الأفعال وهي الوظائف الذي تقوم بها مثل الكتابة ، المشي ، وينطبق نفس الحديث عن أي كائن آخر نقوم بتحديد صفاته ومن ثم وظائفه ونعبر عنها برمجيا الصفات بالمتغيرات أما الأفعال فنعبر عنها بالدوال.

Classes بالعربية يعني تصنيفات وهذا يعني أننا نضع الأشياء في تصنيف معين ، لنفترض أننا نتحدث عن السيارات حسناً هنالك سيارة BMW وله لون معين وسرعات معينه وشركة خاصة بإنتاجها وأيضاً لدينا AUDI وأيضاً لها شركة خاصة ولون معين...إلخ وهكذا لدينا مجموعة متنوعة ومختلفة ولكنهم جميعهم يعتبرون صورة لصنف السيارات ، أي أننا في برنامجنا سنتحدث عن السيارات صحيح ؟ إذا نقوم بإنشاء تصنيف للسيارات نعرف به الصفات العامة بها مثل الاسم ، الشركة المصنعة له ، اللون ونضع كذلك داخل التصنيف الدوال الخاصة بالسيارة كالسرعة والتوقف مثلاً

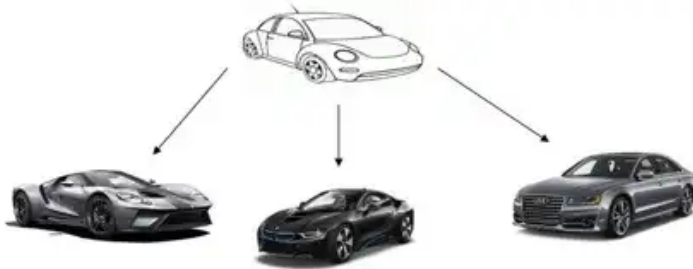
## ثالثاً : الفرق بين Classes & Object

عرفنا ما هو التصنيف [Class] وما هو الكائن [Object] وبشكل بسيط سأوضح لك الفرق بينهم ، الآن بعد ما وضعنا تصنيف خاص بالسيارات يحمل المتغيرات والدوال ، والسيارات الذي ذكرناها في الأعلى FORD , AUDI , BMW وغيرهم ما هم إلا كائنات من هذا الصنف أي أنهم صورة لهذا الصنف.



## Classes OOP

### أولاً: طريقة إنشاء الصنف class





```

class Car {
    var name :String = ""
    var color :String = ""
    var model :Int = 0
    var price :Int = 0

    fun speed(sp:Int){
        if(sp > 100){
            println("Speed is slow")
        } else{
            println("High speed")
        }
    }
}

```

هنا مثال لـ Class السيارة وضعنا بداخله الصفات التي عرفنا اننا نمثلها برمجياً بالـ Variables وهي الاسم واللون والموديل والسعر مثلا ولكن نريد توضيح أمر عند تعريف المتغيرات بداخل التصنيف لابد من أن تقوم بتمهيد قيمة لهذا المتغير لأن لغة الـ kotlin لا تسمح لك بتعريفه بدون ان تمهد له ، ونلاحظ أيضا اننا لا نقوم بإعطاء قيمة حقيقية كما عرفنا فالقيم تعطى عند انشاء كائن أما هنا فنحن فقط نعطي الصفات والافعال.

طريقة أخرى لتمهيد قيم المتغيرات كالتالي:

```

var name :String? = null

```

الآن ستقول لي ما هو ناتج تنفيذ هذا الكود ؟ سأقول لك لا شيء يا عزيزي . نعم لا شيء ، لماذا لأننا فقط قمنا بإعلام البرنامج بإنشاء تصنيف يمرر له بيانات معينة فقط ، متى سينفذ الأمر ؟ وهذا هو الجزء الثاني المهم في درسنا وهو الكائن.

## ثانياً : طريقة إنشاء كائن Object

---

الكائن يا عزيزي هو عبارة عن صورة للتصنيف الذي أنشأناه سأوضح لك الأمر مرة أخرى وهي أن السيارات جميعها بمختلف أنواعها تجتمع بأن لها صفات مشتركة مثل الاسم فكل سيارة تحمل اسم وكل سيارة تحمل موديل ولون وسرعة فهنا ننشئ مصنف يضم كل الصفات والافعال التي تشترك فيها كل السيارات حسناً أين الاختلاف ؟ الاختلاف هو نوع البيان المدخل فسيارة من نوع فورد مختلفة عن سيارة نوع اودي وهكذا ، نقوم بإنشاء الكائن في داخل الدالة الرئيسية:

```
var object1 = Car()
```

هكذا نحن انشأنا كائن عرفناه كما نقوم بتعريف المتغيرات ولكن الاختلاف هو أنه بعد علامة المساواة نضع اسم الصنف Class الذي يتبعه الكائن الذي انشأناه ، يمكنك أيضاً تعريف أكثر من كائن لنفس الصنف بشرط أن يكون الاسم مختلف.

## ثالثاً : الوصول إلى المتغيرات الخاصة بال Class

---

طبعا الآن تعرف لدينا كائن من صنف السيارات إذا جميع ما عرفناه في داخل صنف السيارة ينطبق تماما على الكائن الذي انشأناه من اسم وموديل وسعر ولون ، وهنا السؤال كيف نستدعيهم ونقوم بتعبئتهم ؟

```
object1.name = "BMW"  
object1.color = "White"  
object1.model = 2017  
object1.price = 90872.87f
```

كما نرى في كل مرة نريد الوصول إلى متغير أو دالة في داخل Class نقوم بكتابة:

```
Object .Attribute
```

ومن ثم نقوم بالتعبئة كما نريد.

حسنا الآن تم تعبئة بيانات هذا الكائن سنقوم بعملية الطباعة لنرى ما سيحدث:

```
println(object1.name)  
println(object1.color)  
println(object1.model)  
println(object1.price)
```

## ▪ ناتج التنفيذ

```
BMW  
White  
2017
```

90872.87

## رابعاً : الوصول إلى الدوال الخاصة بال Class

---

من الممكن ان نستدعيها بداخل متغير ومن الممكن اجراء امر الطباعة مباشرة،  
سوف نقوم باتباع نفس الطريقة:

```
var speed1 = object1.speed(150)println(speed1)
```

## خامساً : تمرير الوسائط في Class

---

بإمكاننا إنشاء تصنيف Class ونمرر له وسائط معينه:

```
class labtop(id:Int,name:String) {  
  
    var serialNum= id  
    var companyName = name  
  
    fun getInfo(){  
        println("The labtop is : $companyName , and the serial number  
is : $serialNum")  
    }  
  
}
```

نلاحظ قمنا بإنشاء تصنيف class ومررنا له وسائط وهذا يعني أنه عندما نقوم بإنشاء كائن من هذا التصنيف يستوجب علينا تمرير قيم وسنرى ذلك في الدالة الرئيسية ، وفي داخل التصنيف عرفنا متغيرات واسندنا لها قيم هذي الوسائط الممررة وذلك لأن هذه وسائط بداخل تصنيف لن يستطيع التعامل معها مباشرة ، ومن ثم قمنا بإنشاء دالة تطبع المعلومات.

### ▪ في الدالة الرئيسية

```
fun main (args:Array<String>){  
  
    var dell = labtop(1015,"Dell")  
    dell.getInfo()  
}
```

نرى أنه ألزمتنا عند إنشاء الكائن بتمرير قيم لهذه الوسائط التي ادخلناها في التصنيف فوق.

لو نريد أن نتعامل مع الوسائط مباشرة بدون تعريف متغيرات في داخل تصنيف :class

```
class labtop(var id:Int,var name:String) {  
  
    fun getInfo(){  
        println("The labtop is : $name , and the serial number is : $id")  
    }  
  
}
```

فقط نقوم بتعريفهم بداخل اقواس التصنيف.

## سادساً : كلمة this

```
class labtop(var id:Int,var name:String) {  
  
    fun setValues(id:Int , name:String){  
        this.id = id  
        this.name = name  
    }  
  
    fun getInfo(){  
        println("The labtop is : $name , and the serial number is : $id")  
    }  
  
}
```

حسننا الآن سنتعرف على فائدة this نلاحظ هنا أن انشأنا دالة ومررنا لها نفس المتغيرات للتصنيف class وبداخل الدالة:

```
this.id = id  
this.name = name
```

هذه الكلمة تعني id الخاص بالتصنيف وليس id الممر للدالة ، لو كانت المتغيرات الممررة للدالة ذات مسميات مختلفة عن التي تم تمريرها للتصنيف فلن نحتاج لهذه الكلمة ، باختصار شديد كلمة this خاصة للوصول إلى محتويات التصنيف.

## سابعاً : دالة البناء constructor

في البداية لابد من توضيح ماهية دالة البناء وعملها ؟

دالة البناء هي عبارة عن قطعة برمجية يتم تنفيذها مباشرة عند إنشاء كائن مشتق من التصنيف ،ويمكن إنشاء أكثر من دالة بناء بداخل التصنيف

لدينا طريقتان لعمل دالة البناء:

### ❖ الطريقة الأولى : دالة تعريف جاهزة init

- هذه الدالة اختصار لكلمة initialization.
- لا نستطيع تمرير متغيرات بداخلها ابداً.
- لها الأولوية في التنفيذ لو قمنا بإنشاء أكثر من دالة بناء.
- نقوم بداخلها بتمهيد القيم وعند إنشاء كائن يتم استدعاؤها تلقائياً بدون القيام بكتابتها بنفسك وتنفيذ ما بداخلها.

### ▪ مثال

```
class employee(var firstName:String,var lastName:String,var
titleJob:String,var salary :Double){
    init{
        println("the name of employee is :$firstName $lastName, his title
job :$titleJob ,and his salary :$salary ")
    }
}
```

```
}
```

في المثال عرفنا المتغيرات بين اقواس التصنيف ومن ثم نلاحظ بداخل init فقط نريد طباعة هذه الجملة ، أي عن انشاء كائن من هذا التصنيف سوف يقوم بوضع قيم للمتغيرات وعند تنفيذ البرنامج سيطبع الجملة التي اردنا طباعتها في دالة التهيئة.

#### ▪ في الدالة الرئيسية

```
var employee1 = employee("Ahmed","Aljuaid","Teacher",7000.0)
```

#### ▪ ناتج التنفيذ

the name of employee is :Ahmed Aljuaid, his title job :Teacher ,and his salary :7000.0

### ❖ الطريقة الثانية : دالة constructor

نستطيع تمرير متغيرات بداخلها.

أيضا نقوم بداخلها بتمهيد القيم وعند إنشاء كائن يتم استدعائها تلقائيا بدون القيام بكتابتها بنفسك وتنفيذ ما بداخلها.

#### ▪ مثال

سوف نقوم بتطبيق نفس المثال المذكور في init:

```
class employee{
```



```

var firstName:String ? = null
var lastName:String ? = null
var titleJob:String ? = null
var salary :Double ? = null

constructor(fName:String,lName:String ,tJob:String ,sal :Double){
    firstName = fName
    lastName = lName
    titleJob = tJob
    salary = sal
    println("the name of employee is :$firstName $lastName, his
title job :$titleJob ,and his salary :$salary ")
}
}

```

قمنا بتعريف متغيرات بداخل التصنيف ومن ثم انشأنا دالة بناء تمرر لها متغيرات وقمنا بإسناد هذه المتغيرات ومن ثم أمر الطباعة.

#### ▪ في الدالة الرئيسية

```
var employee1 = employee("Ahmed","Aljuaid","Teacher",7000.0)
```

#### ▪ ناتج التنفيذ

the name of employee is :Ahmed Aljuaid, his title job :Teacher ,and his salary :7000.0

تنويه

لو أردنا تمرير متغيرات عند إنشاء التصنيف فدالة البناء الـ constructor لا نستطيع التمرير لها ولكن نستعيز عنها بفعل الآتي:

```
class employee (var firstName:String,var lastName:String,var
titleJob:String,var salary :Double){

    constructor():this("Ahmed","Aljuaid","Teacher",7000.0){

        println("the name of employee is :$firstName $lastName, his title
job :$titleJob ,and his salary :$salary ")

    }

}
```

نلاحظ أننا استعملنا كلمة this التي تقوم بالوصول إلى المتغيرات التي عرفناها عند إنشاء التصنيف.

وفي الدالة الرئيسية نقوم فقط بإنشاء كائن بدون تمرير:

```
var employee1 = employee()
```

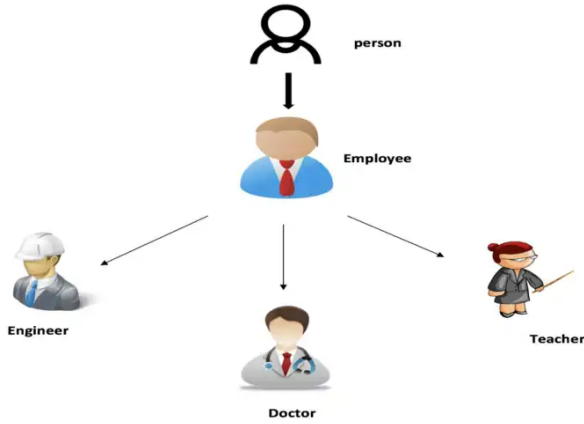
تتنوع هذه الطرق وكلاً بحسب حاجتك أنت عند إنشاء البرنامج وبحسب ما تراه.

## Inheritance

لو تكلمنا علمياً بهذا الموضوع وارادنا تبسيط مفهوم الوراثة سنقول أن تنتقل بعض من صفات الوالدين أو جميعها للطفل ، ونفس الشيء يحدث برمجيا نستطيع إنشاء تصنيف class يرث من تصنيف class آخر.

## أولا : مفاهيم متعلقة بالوراثة

سنتعرف على طريقة الوراثة برمجيا ولكن بعد توضيح بعض المفاهيم العامة كالتالي:



فلنفترض أن لدينا class Person يحتوي على الآتي :

{ اسم الشخص - تاريخ الميلاد - الرقم المدني } ، وانشأنا class Employee يرث من class Person يحتوي على الآتي : { الرقم الوظيفي } ، فهنا class Employee يستطيع

الوصول إلى محتويات class Person من الاسم وتاريخ الميلاد والرقم المدني بالإضافة إلى ما يتضمنه هذا التصنيف من الرقم الوظيفي.

حسناً نكمل الشرح ونلاحظ أن الوظائف أنواع فلدينا المعلم ولدينا المهندس ولدينا الطبيب وغيرها الكثير وجميعهم يرثون من class Employee وبالطبع يا عزيزي بما أنهم يرثون منه وهو بدوره يرث من class Person فهم يستطيعون الوصول إلى محتويات class Person أيضاً ، هذا الشيء يشبه كثيرا ما تقدم شرحه في التمهيد فوق بأن الطفل يستطيع أن يرث من والديه ولكن هل فقط والديه ؟ لا أيضا قد يرث من جديه صفات. إذا نحن الآن نعلم أن في الوراثة نستطيع الوصول إلى كافة محتويات class الموروثة.

من هو الوارث ومن هو الموروث ؟

الموروث في المثال أعلاه هو class person هو الأب يعتبر ونطلق عليه super ، أما الوارث فهو class Employee ونسميه sub أي فرعي. وبالطبع المعلم والطبيب والمهندس يعتبرون وارثين ويعتبرون sub، والموروث هو class Employee الذي يعد بالنسبة لهم super.

## ثانياً : الوراثة في البرمجة

التصنيف الأب أو super class لكي نستطيع أن نجعل تصنيف آخر يرث منه لابد من وضع كلمة open عند إنشائه.

❖ مثال

### ▪ super class

```
open class person(){
    var name:String? = null
    var id :Int? = null

    fun getInfo(){
        println("The name is :$name your Id is : $id")
    }

    fun getId(id:Int){
        println("your id is :$id")
    }
}
```

إذا نلاحظ أنه تصنيف عادي قمنا بإنشائه كما تعلمنا ولكن لأن هناك تصنيف آخر سيرث منه فيجب علينا جعله مفتوح ليستطيع الوارث أن يصل لمحتويات هذا التصنيف.

أما بالنسبة للتصنيف الوارث فعلى هذا النحو يتم كتابته:

### ▪ sub class

```
class Employee(): person()
{
    var idj :Int?=null

    fun printAllData(){
        println("the ID for your job : $idj")
    }
}
```

```
}
```

عند إنشاء التصنيف نقوم فوراً بعملية الوراثة `person()` من `class Employee()` نقطتين رأسيّتين ومن ثم اسم التصنيف الموروث.

## ❖ في الدالة الرئيسية

نقوم بإنشاء كائن من `class Employee` وذلك لتوضيح كيف تتم عملية الوراثة:

```
fun main(args :Array<String>){  
    var job1 = Employee()  
    job1.  
        printAllData() Unit  
        idj Int?  
        id Int?  
        getId(id: Int) Unit  
        getInfo() Unit  
        hashCode() Int  
        name String?  
        equals(other: Any?) Boolean  
}
```

نرى أن الكائن المشتق من `class Employee` استطاع الوصول إلى محتويات `class Person` وهذا الغرض الرئيسي من عملية الوراثة أن الكائن يستطيع الوصول إلى محتويات `super class`.

## ثالثاً : التعامل مع الوسائط الممررة للتصنيف

لو كان class Person ممرر له وسائط وهذا الشيء طبيعي كما تعرفنا عليه مسبقا عند إنشاء تصنيف ، كيف سنتعامل معه في الوراثة ؟

الجواب كالتالي:

- نقوم بإنشاء تصنيف الموروث او ما نسميه بـ super class ونمرر له وسائط. مثال:

```
open class person(id :Int , name :String){  
  
    var id:Int = id  
    var name :String = name  
    .  
    .  
    .  
}
```

- نقوم بإنشاء sub class ونمرر له أيضا نفس الوسائط الذي سيرثها. مثال:

```
class Employee(id :Int , name :String) : person(id , name)  
{  
    .  
    .  
    .  
}
```

نلاحظ مررنا نفس الوسائط للتصنيف الذي سيرث ومن ثم وضعنا اسم التصنيف الموروث ومررنا المتغيرات بشكل طبيعي بدون تعريف لها أو لنوعها.

حسنًا نفس الشيء ينطبق لو كان لدي constructor دالة بناء:

#### ▪ super class

```
open class person{  
  
    var id:Int? = null  
    var name :String? = null  
  
    constructor(id:Int,name:String){  
        this.id = id  
        this.name=name  
    }  
}
```

#### ▪ sub class

```
class Employee : person  
{  
    constructor(id:Int,name:String):super(id , name){  
  
        this.id = id  
        this.name=name  
    }  
}
```



ونلاحظ هنا أننا قمنا باستخدام كلمة super ليتضح لنا أنها keywords في لغة الـ kotlin، ونقصد بها هنا الوصول إلى دالة البناء الخاصة بالتصنيف الأب.

## رابعاً : الفرق بين this و super

---

الفرق بسيط جداً :

نستخدم this في التصنيف نفسه وللتعبير عن محتوياته كما تعلمنا مسبقاً.

نستخدم super للوصول إلى محتويات التصنيف الأب أي أنه خاص بالتصنيف الموروث للوصول إليه.

## interface

---

### أولاً : تمهيد

---

هل قد خطر في بالك أثناء تصميم مشروعك أنك تعلن عن دالة ولكن لا تعلم ما لذي ستضعه بداخلها أو أنك تريد الإعلان فقط وستستخدمها لاحقاً ؟ في بيئة Kotlin نستطيع فعل ذلك عن طريق إما abstract أو interface ، هذه العناصر تمكنني من الإعلان عن متغيرات ودوال بدون أن أكتب قطع برمجية بمعنى أنني فقط أعرف عن دالة ولكني ما تتضمنه هذه الدالة سأؤجل كتابته إلى حين أحاجه.

## ثانياً: التعريف عن interface

---

كما قلنا سابقا نستطيع فيه الإعلان عن الدوال التي سأستخدمها لاحقاً.

```
interface myInterface{  
  
}
```

## ثالثاً: الإعلان في interface

---

سنوضح الآن كيفية تجهيز المتغيرات والاعلان عنها وعن الدوال:

```
interface myInterface{  
  
    fun sum()  
    fun sub()  
    fun mul(num1:Int,num2:Int){  
        var result = num1*num2  
    }  
    var num1:Int  
}
```

اعلنا عن دالة جمع ولكن بدون أن نبني القطعة البرمجية الخاصة بها.

أيضا اعلنا عن دالة الطرح ولم نستخدمها أيضاً.

اعلنا عن دالة الضرب وقمنا ببنائها وتمرير قيم ووضحنا عملها وهذا يدل ان interface ليس حكرا فقط لتجهيز الدوال والمتغيرات أيضا نستطيع بناء دالة كاملة، ولكن الاختلاف يكمن في طريقة العمل التي سنوضحها بعد قليل.

المتغيرات التي يتم تعريفها بداخل interface لا يتم اسناد قيم لها ابدا ، كما لاحظنا عرفنا متغير ولم نسند له قيم.

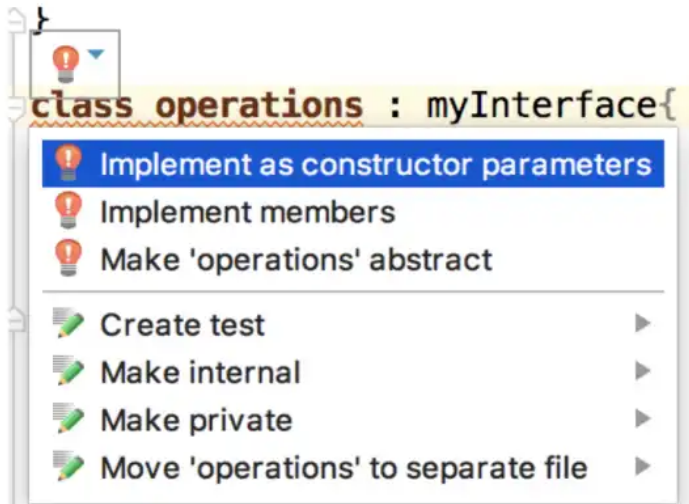
## رابعاً : استخدام interface

---

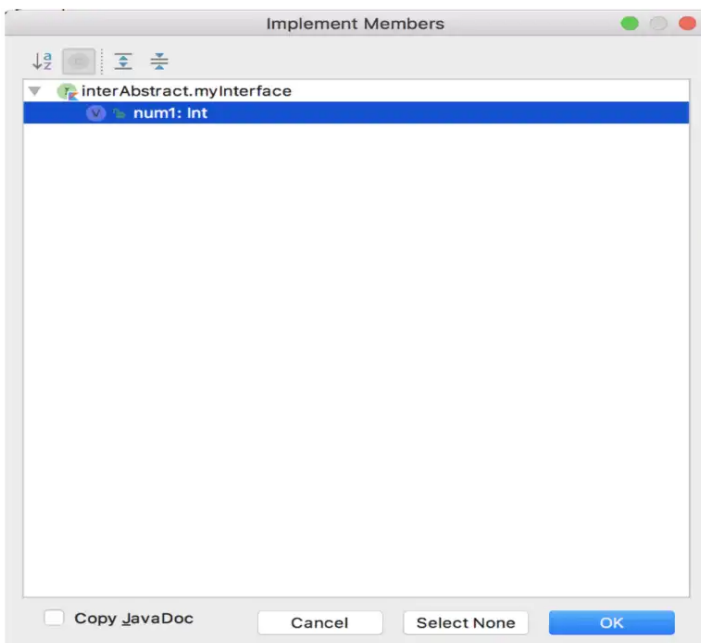
Interface لا يتم استخدامها مباشرة في الدالة الرئيسية وإنما نستخدمها بعملية implements وهي بالضبط مماثلة لعملية الوراثة، هكذا:

```
class operations : myInterface{  
  
}
```

نرى أنه عندما جعلنا التصنيف يرث من الـ interface قام بوضع خط احمر ، لماذا ؟ لأنه عندما تجعل التصنيف يرث من interface وبداخله دوال لم تبني بعد إجباري أن تدرجهم بداخل التصنيف كلهم أو أحدهم على الأقل، حسنا ولكن كيف يعني ادرجهم ؟ بمعنى أن تقوم بعمل override للدالة كما تعلمنا مسبقا، سنوضح ذلك الآن:



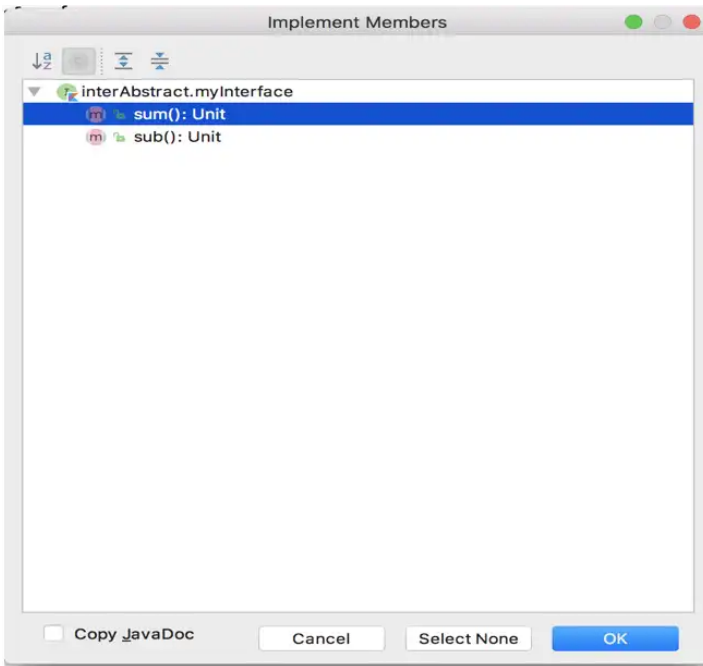
- بعد الضغط عليها نلاحظ أن يطلب implement as constructor parameters في أول مرة ويقصد به أننا عرفنا متغير في interface فهنا يجب في التصنيف الوارث ان نستخدمه.



بعد إدراج المتغير:

```
class operations(override var num1: Int) :myInterface{  
}
```

- **implement members** وهو يعني قم بإضافة الدوال التي عرفتتها ولم تستخدمها ، وهنا ننوه بأنه إجباري استخدامها وكتابتها بما أنك قمت بجعل التصنيف يرث من interface، أما الدالة الجاهزة الخاصة بعملية الضرب فنحن بنيناها في interface فلك الحرية بأن تستدعيها في التصنيف أو لا.



بعد إدراج الدوال:

```
class operations(override var num1: Int) :myInterface{
    override fun sub() {
        TODO("not implemented") //To change body of created functions
use File | Settings | File Templates.
    }

    override fun sum() {
```

```
    TODO("not implemented") //To change body of created functions  
use File | Settings | File Templates.  
}  
  
}
```

## خامساً : مثال interface

```
interface myInterface{  
  
    fun sum(num1:Int,num2:Int)  
    fun sub(num1:Int,num2:Int)  
    fun mul(num1:Int,num2:Int){  
        var result = num1*num2  
        println("the result :$result")  
    }  
  
}  
  
class operations:myInterface{  
    override fun sub(num1:Int,num2:Int) {  
        var result = num1 - num2  
        println("the result :$result")  
    }  
  
    override fun sum(num1:Int,num2:Int) {  
        var result = num1 + num2  
        println("the result :$result")  
    }  
}
```

```

    fun dev(num1:Int,num2:Int){
        var result = num1 / num2
        println("the result :$result")
    }

}

fun main(args:Array<String>){
    var op1 = operations()
    println("the result of sum :")
    op1.sum(4,3)
    println("the result of sub :")
    op1.sub(4,3)
    println("the result of mul :")
    op1.mul(3,4)
    println("the result of dev :")
    op1.dev(4,3)
}

```

## ▪ ناتج التنفيذ

the result of sum :  
 the result :7  
 the result of sub :  
 the result :1  
 the result of mul :  
 the result :12  
 the result of dev :  
 the result :1



## abstract

---

### أولاً: مفهوم ال abstract

---

نستطيع فيه جعل الدوال والمتغيرات معلنة فقط ولكن بشرط أن نضع كلمة abstract أمامها ، بمعنى أن الدوال في interface كنا نستطيع أن نعلن عنها فقط بدون أن ننشئ قطعة برمجية لها لأننا سنقوم بذلك في تصنيف آخر ، هنا أيضا نستطيع أن نعلن فقط عن الدالة بداخل هذا التصنيف ولكن يشترط أن نضع كلمة abstract لأن التصنيف أصلا هو. abstract

### ثانياً: استخدام ال abstract

---

هو عبارة عن تصنيف class ولكنه لا يقبل أن ننشئ منه كائن ، تصنيف نصممه للوراثة (أي لنجعل جميع محتوياته قابلة للوراثة لتصنيفات أخرى).

### ثالثاً: انشاء تصنيف abstract

---

```
abstract class operation{  
  
}
```

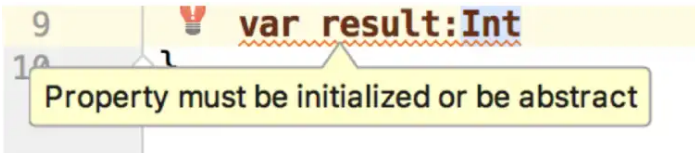
أنشأنا تصنيف `abstract` إذا ما بداخله من دوال لكي يتم الإعلان عنها فقط بدون عملية بناء لها نجعلها `abstract` كما سيتم توضيحه الآن:

```
abstract class operation{

    abstract fun sum(num1:Int , num2:Int)
    abstract fun sub(num1:Int , num2:Int)
    abstract fun mul(num1:Int , num2:Int)
    abstract fun dev(num1:Int , num2:Int)

}
```

حسنًا لو أردنا تعريف المتغيرات:



نرى أنه عندما أردنا تعريف متغير اعترض ووضع خط أحمر وعند الوقوف عليه ظهر السبب وهو أنه عندما تريد تعريف متغير إما أن تجعل له قيمة مبدئية أو أن تجعله `abstract` لكي تستطيع ألا تسند له قيمة أي يعني سوف تستخدمه وتسند له فيما بعد.

## رابعاً : مثال

```
abstract class operation{
```

```

abstract fun sum(num1:Int , num2:Int)
abstract fun sub(num1:Int , num2:Int)
abstract fun mul(num1:Int , num2:Int)
abstract fun dev(num1:Int , num2:Int)

fun sayWelcome(){
    println("Welcome")
}

}

class oper : operation(){

    override fun dev(num1: Int, num2: Int) {
        var result = num1 / num2
        println("the result :$result")
    }

    override fun mul(num1: Int, num2: Int) {
        var result = num1 * num2
        println("the result :$result")
    }

    override fun sub(num1: Int, num2: Int) {
        var result = num1 - num2
        println("the result :$result")
    }

    override fun sum(num1: Int, num2: Int) {
        var result = num1 + num2
    }
}

```

```

        println("the result :$result")
    }

}

fun main(args:Array<String>){
    var op1 = oper()
    println("the result of sum :")
    op1.sum(4,3)
    println("the result of sub :")
    op1.sub(4,3)
    println("the result of mul :")
    op1.mul(3,4)
    println("the result of dev :")
    op1.dev(4,3)

    println("-----")
    op1.sayWelcome()
}

```

## ▪ ناتج التنفيذ

the result of sum :  
 the result :7  
 the result of sub :  
 the result :1  
 the result of mul :  
 the result :12

the result of dev :

the result :1

-----  
Welcome

## خامساً : مبدأ الوراثة المتعددة multi inheritance

---

الوراثة المتعددة تعني أن تصنيف class يرث أكثر من تصنيف classes ولكن لغة kotlin ترفض مبدأ الوراثة المتعددة ولحل هذه المعضلة.

وجدت لدينا interface وهي ما تعلمناه مسبقاً بحيث نستطيع أن ننشئ تصنيف class يرث من تصنيف class وأيضاً نقوم بعمل interface J implements, مثال:

```
interface showDepartment{
    fun showDept()
}

abstract class salary{
    abstract fun getSal()
}

class employee : salary(), showDepartment{
    override fun showDept() {
    }
    override fun getSal() {
    }
}
```

# Override

## أولاً: مفهوم Override

تكمُن أهمية هذا المفهوم بأنه تتيح لنا الفرصة بأن ننشئ دالة في التصنيف الوارث هي نفسها موجودة في التصنيف الموروث ، لتبسيط الأمر فلنفترض أن لدينا تصنيف class Person وبداخله دالة تقوم بالطباعة:

- يتوجب علينا أن نجعل الدالة في التصنيف الموروث open.

```
open class Person(){  
  
    open fun getInfo(name:String){  
        println("welcome $name")  
    }  
    fun printWelcome(){  
        println("welcome")  
    }  
  
}
```

ولدي التصنيف الوارث class Employee يرث جميع محتويات class Person وبما أنه يرثها فهذا يعني أنه سيمنعني من تكرار نفس مسمى الدالة ولكن لتجنب هذا الأمر نستخدم مفهوم override:

```
class Employee():Person(){
    override fun getInfo(jobTitle:String){
        println(" your job title :$jobTitle ")
    }
}
```

- نلاحظ أن الدالة الموجودة بداخل التصنيف class Employee هي نفسها الموجودة بداخل التصنيف الموروث class Person وكتبنا قبلها كلمة override وهذا يعني أنه عند إنشاء كائن من هذا التصنيف لن نستطيع الوصول إلى الدالة getInfo الموجودة في class Person.
- نريد التوضيح بأننا لو أردنا الدالة أن تنفذ نفس الأوامر ولكن نريد الزيادة عليها بمعنى أن الدالة لا تكتفي فقط بالأوامر البرمجية التي تمت كتابتها في التصنيف الموروث وإنما نريد أن نضيف أوامر أخرى:

```
open class Employee():Person(){
    final override fun getInfo(jobTitle:String){
        super.getInfo("Ahmed")
        println(" your job title :$jobTitle ")
    }
}
```

## ▪ في الدالة الرئيسية

```
fun main(args:Array<String>){
    var emp = Employee()
    emp.
}
```

getInfo(jobTitle: String)	Unit
printWelcome()	Unit
hashCode()	Int
equals(other: Any?)	Boolean

إذا نلاحظ أنه يستطيع الوصول لأي دالة ماعدا تلك التي عملنا لها override حينها لن يستطيع الوصول إلا للدالة الموجودة في التصنيف `Employee` class.

## ثانياً : كلمة **final**

استخدام هذه الكلمة مع التصنيف يعني أنه غير قابل للوراثة وهذا الشيء يرجع لاختياراتك الشخصية فنحن لنجعل التصنيف يقبل الوراثة لابد من وضع كلمة `open`، كذلك من أجل جعله تصنيف نهائي غير قابل للوراثة نضع `final`، وينطبق نفس الشيء على الدوال.

مثال:

```
open class Employee():Person(){

    final override fun getInfo(jobTitle:String){
        super.getInfo("Ahmed")
        println(" your job title :$jobTitle ")
    }
}
```



```
}
```

هنا جعلنا هذه الدالة لا تقبل أن تكون مرة أخرى فيما بعد `override`، بمعنى لو أنشأت تصنيف آخر جديد فيما بعد وأردت أن أجلب الدالة نفسها عن طريق `override`، فلن نستطيع وذلك لأنها `final`.

## Overload

في بعض الأحيان نصادف أكثر من شخص يحملون نفس الاسم ولهم نفس اسم الأب وللمصادفة أيضا نفس اسم العائلة صحيح ولكنك تستطيع التفريق بين صديقك أي واحد هو منهم وذلك لأنه يمتلك خصائص خاصة به وسمات تدل على أنه هو المقصود، أيضا نفس الشيء يحدث لدينا عند البرمجة تواجه أكثر من دالة لهم نفس المسمى بالضبط نسميه `overload` ولكن لنركز انتباهنا قليلا انه هذه الدوال فقط لهم نفس الاسم ولكن يختلفون من حيث تمرير المتغيرات ونوع الدالة هل ترجع بقيمة ام لا وهكذا.

مثال:

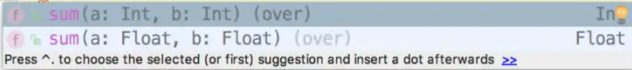
```
fun sum(a:Int , b:Int):Int{
    return a+b
}

fun sum(a:Float, b:Float):Float{
    return a+b
}
```

انشأنا دالتين لهم نفس المسمى ولكن كل واحدة منهم تختلف من حيث نوع المتغيرات الممررة ، أيضا كان بالإمكان تعريف الدالة الثانية وان نوع المتغيرات int ولكن بدل من أن تقوم بجمع متغيرين تقوم مثلا بجمع أكثر.

## ▪ في الدالة الرئيسية

```
fun main(args:Array<String>){  
    var result = sum  
}
```



نلاحظ أنه عندما اردنا استدعاء الدالة باسمها طلب منا ان نحدد أي دالة منهم هل التي متغيراتها Int او تلك التي متغيراتها Float.

```
fun main(args:Array<String>){  
  
    var result = sum(4,9)  
    println ("the first function result is : $result")  
    var result1 = sum(2.3f,1.9f)  
    println ("the second function result is : $result1")  
}
```

## ▪ ناتج التنفيذ

the first function result is : 13  
the second function result is : 4.2

مفهوم overload نستطيع تنفيذه حتى بداخل التصنيف فيمكننا من إنشاء أكثر من دالة بناء ولكن مع مراعاة ما ذكرناه سابقا.

## Companion object

### أولاً: مفهوم companion object

هذا المفهوم يعبر عن إمكانية وصولك لدوال و متغيرات معينة تم تعريفها بداخل التصنيف بدون اشتقاق كائن، هذه المتغيرات عملها مستقل عن ما بداخل التصنيف يعني أنني أنشأتها لعمل معين بدون ان ترتبط بالمتغيرات الأخرى أو الدوال التي بداخل التصنيف، ويتم تعريف هذه المتغيرات والدوال بداخل قطعة برمجية تسمى companion object.

### ثانياً: طريقة كتابته

```
companion object {  
  
}
```

### ثالثاً: مثال

```
class labtop {
```

```

companion object {
    fun sayHello(){
        println("Welcome in my project")
    }
}

var serialNum:Int = 0
var company:String = " "
fun getData(){
    println(" labtop : $company \n the serial number is :
$serialNum")
}
}

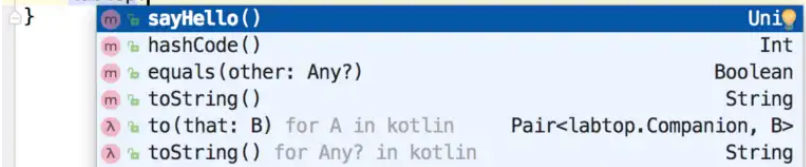
```

ستلاحظ بأنها قطعة برمجية مستقلة انشأنا بداخلها دالة تقوم بالترحيب ، كما قلنا أنه ما يتم كتابته بداخل هذه القطعة لن تستخدمه خارجه أي أنه لا علاقة له بمحتويات التصنيف الباقية.

## ▪ في الدالة الرئيسية

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

```
    labtop.
```



نلاحظ أننا قمنا بكتابة اسم التصنيف class ومن ثم ظهرت لنا الدالة التي انشأناها بداخل companion object وهنا يتجلى لنا الهدف منها أننا نستطيع ان نضع العمليات والمتغيرات بداخل هذه القطعة البرمجية للوصول إليها سريعاً من اسم التصنيف.

```
fun main(args:Array<String>){  
    labtop.sayHello()  
  
    var lab1=labtop()  
    lab1.  
}
```

company	String
serialNum	Int
getData()	Unit
hashCode()	Int
equals(other: Any?)	Boolean

هنا قمنا باشتقاق كائن من التصنيف ونلاحظ أنه لا يستطيع الوصول إلى ما بداخل companion object.

## رابعاً : متغير يستقبل companion object

نستطيع أن نعرف متغير في الدالة الرئيسية قيمته عبارة عن companion object هكذا:

```
var quiz: Float = 0f  
var midterm: Float = 0f  
var final: Float = 0f  
class student {  
  
    companion object {
```

```

fun totalQuiz() {
    println("Enter first Degree : ")
    var quiz1: Float = readLine()!!.toFloat()
    println("Enter second Degree : ")
    var quiz2: Float = readLine()!!.toFloat()
    var totalQuiz: Float = (quiz1 + quiz2) / 2
    println("your quiz degree is : $totalQuiz")
}

fun totalDegree(quiz: Float, midterm: Float, final: Float) {
    var result = quiz + midterm + final

    if (result >= 90 && result <= 100) {
        println("A")
    } else if (result >= 80 && result <= 89) {
        println("B")
    } else if (result >= 70 && result <= 79) {
        println("C")
    } else if (result >= 60 && result <= 69) {
        println("D")
    } else {
        println("F")
    }
}
}

```

في الدالة الرئيسية

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

```
var tQuiz = student.Companion
```

```
tQuiz.totalQuiz()
```

```
}
```

نعرف متغير ونرى أننا اسندنا له companion object.

## خامساً : معلومات حول companion object

في حال أننا عرفنا متغيرات بداخله واسندنا لها قيم فإنه عند استدعاؤها وتغير قيمتها في البرنامج سوف تتغير بشكل عام أي ليس في الأمر فقط المحدد ، سنوضح ذلك:

```
class compan {  
    companion object {  
        var yourName = "Ahmed"  
        var familyName = "Aljuaid"  
        fun show() {  
            println("Hello $yourName $familyName")  
        }  
    }  
    var yourName = "Ahmed"  
    var familyName = "Aljuaid"  
    fun show() {  
        println("Hello $yourName $familyName")  
    }  
}
```

```
fun main(args:Array<String>){
    var name = compan.Companion
    var name1 = compan.Companion

    name.show()
    name1.show()
}
```

التنفيذ

Hello Ahmed Aljuaid  
Hello Ahmed Aljuaid

حسننا لنلاحظ قليلا لو أردنا تغيير الاسم الأول في متغير name ما لذي سيحدث ، في الدالة الرئيسية:

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

    var name = compan.Companion
    var name1 = compan.Companion

    name.yourName="Ali"
    name.show()
    name1.show()

}
```

التنفيذ



Hello Ali Aljuaid  
Hello Ali Aljuaid

تم تغيير الاسم في المتغيرين name and nam1 بالرغم من أننا استدعينا name وعدلنا بها ، وهذا يدل على أنه غير مستقل بحيث أنه قيمة متغير في كائن واحد يجعلك عندما تستدعي نفس المتغير لكائن جديد هيا نفسها القيمة.

## التصنيفات المتداخلة Nasted classes

---

### أولاً: مفهوم التصنيفات المتداخلة

---

لغة Kotlin تدعم طريقة التصنيفات المتداخلة هي عبارة عن تصنيف كبير وبداخله مجموعة من التصنيفات، يعني تعريف class بداخل class.

### ثانياً: طريقة تعريف التصنيفات المتداخلة

---

```
class school{  
  
    class employee{  
    }  
  
    class student{  
    }  
}
```

```
}
```

## ثالثاً: أهمية التصنيفات المتداخلة

---

تفيد في عملية التغليف Encapsulation.

تعتبر طريقة عملية لو لدينا عدد من التصنيفات المرتبطة ببعض.

## رابعاً: مصطلحات مهمة في التصنيفات المتداخلة

---

التصنيف الخارجي الكبير يسمى:

OUTER CLASS

التصنيفات الداخلية التي بداخل outer class تسمى:

INNER CLASS

## خامساً: انشاء كائن من inner class

---

في الدالة الرئيسية

```
var obj= outerClass.innerClass()
```

```
class school{

    class employee{
        val id:Int = 0
        var name :String = " "
        fun getInfo(){
            println("The Id of Employee is :$id \n Name :$name")
        }
    }

    class student{
        var grade:Char = ' '
        var level:Int =0
        var name :String = " "
        fun getInfo(){
            println("The student name is :$name \n Level : $level \n
Grade:$grade")
        }
    }

}

fun main (args:Array<String>){
    var stu = school.student()
    stu.name="Khalid Ali"
    stu.level=5
    stu.grade='A'
```

```
stu.getInfo()  
}
```

## ENUM class

---

### أولاً : مفهوم Enum class

---

هو تصنيف نستخدمه للتعداد والأشياء الثابتة الغير قابلة للتغيير ، مثلا الأسبوع يبدأ من الأحد وينتهي السبت وأسماء الأيام ثابتة ومعروفة ولا يمكن لك تغييرها.

### ثانياً : استخدامها Enum class

---

هو تصنيف نعرفه ونضع بداخله الشيء الذي نريد أن نثبته ، بمعنى أنني أريد أن اصمم مشروع يتحدث عن العمل في أيام الأسبوع مثلاً فأعرف تصنيف واضح به أيام الأسبوع السبعة وعندما أقوم بالعمل داخل المشروع لن يقبل مني يوم آخر غيرهم.

### ثالثاً : تعريف Enum class

---

```
enum class weekDays{  
  
    Sunday , Monday , Tuesday , Wednesday , Thursday , Friday , Saturday  
  
}
```

## ▪ في الدالة الرئيسية

```
fun main(args:Array<String>){  
    var myDay : weekDays //نعرف متغير من التصنيف/  
    myDay = weekDays.Tuesday // هنا لو أردنا أن ننادي يوم من الأيام  
    myDay = weekDays.Ahmed // هنا اعترض لأنه ليس من ضمن العناصر الموجودة  
    في التصنيف  
}
```

```
myDay = weekDays.Ahmed
```

## data class

### أولاً: مفهوم data class

ماذا لو كان لديك class. و اردت طباعة القيم التي داخله او طباعته كـ Object فانت عند القيام بامر الطباعة سيقوم المترجم بطباعة hashcode للـ Object وليس للبيانات! حتى لو اردت المقارنة بين اكثر من Object فسيقوم المترجم بالمقارنة بين hashcode

وليس بين القيم للـ Object. ايضاً اذا اردت نسخ بيانات Object الى Object جديد فانت لست قادر على ذلك مع class بصيغته العامة.

لذلك كان لابد من ايجاد طريقة جديدة او مفهوم جديد وهو استخدام data class والذي سيقوم بجميع المهام من مقارنة ونسخ وطباعة بشكل سريع ودون كتابة الكثير من الدوال والشفرات البرمجية.

## ثانياً : طريقة data class

---

```
data class employee(var id:Int , var name:String , var department:String){  
}
```

## ثالثاً : مثال لاستخدام class

---

لاحظ في هذه الشفرة انشئنا class وقمنا بارسال له الاسم والقيمة ونريد عمل مقارنات وطباعة للقيم في class ولكن يتضح لنا هنا انه لايقوم بالطباعة والمقارنة مثل ماكان متوقع فهو يقوم بالمقارنة hashCode او يقوم بطباعته.

```
class Employee(name: String, salary: Int) {  
    fun display() {  
        print("Hello Employee")  
    }  
}
```

```

fun main(args: Array<String>) {
    val emp1 = Employee("Ahmed", 11200)
    val emp2 = Employee("Ahmed", 11200)
    //compare
    print(emp1 == emp2) //false
    //copy
    val emp3 = emp1.copy() // unresolved reference
    //print
    print(emp1) // hashCode 'Employee@27c170f0'
}

```

## رابعاً : اضافة data قبل تعريف class

الان بعد ان اتضح لك ان بعض العمليات لايمكن للـ class القيام بها ويجب عليك كتابتها وتعريفها للطباعة والنسخ . ولكن مع kotlin كل ما في الامر انك تحتاج الى اضافة data قبل اسم class وستكون قادر على القيام بجميع العمليات من نسخ وطباعة ومقارنة كما ستلاحظ في الشفرة القادمة.

```

data class Employee(val name: String, val salary: Int) {
    fun display() {
        print("Hello Employee")
    }
}

fun main(args: Array<String>) {
    val emp1 = Employee("Ahmed", 11200)
    val emp2 = Employee("Ahmed", 11200)
    //compare

```

```
print(emp1 == emp2) //true
//copy
val emp3 = emp1.copy(salary = 2000) // Employee(name=Ahmed,
salary=2000)
//compare
print(emp1 == emp3) //false
//print
println(emp1) // Employee(name=Ahmed, salary=11200)
println(emp3) // Employee(name=Ahmed, salary=2000)
}
```

تلاحظ في الشفرة البرمجية اني قمت بنسخ emp1 مع تعديل قيمة salary مباشرة ونسخ اسم الموظف دون تعديل.

## Polymorphism

### أولاً : مبدأ Polymorphism

هذا المفهوم يعني بأن لدينا دالة في الـ super class إذا هي دالة رئيسية ، ولدينا أيضا sub classes نقوم بعمل override للدالة التي في super class.

إذا بشكل عام هي دالة لها أكثر من وجه.

### ثانياً : شروط Polymorphism



- لا بد من تواجد super class و sub classes.
- الوراثة، إذ تعتبر الوراثة عامل مهم لنتمكن من قول أن العملية التي أمامنا polymorphism.
- الدالة التي بالـ super class نجعلها open، ونقوم بعملية override لها في sub classes.

## ثالثاً : أمثلة متنوعة لمفهوم Polymorphism

### ❖ مثال (١)

```
open class schools(){

    open fun getInfo(){
        var name:String = readLine()!!
        println("Welcome in to our schools $name")
    }

    class employees():schools(){
        override fun getInfo() {
            println("every employee in this school have ID")
        }
    }

    class students():schools(){
        override fun getInfo() {
            println("we are proud of our students")
        }
    }
}
```

```

    }
}
}

```

- نلاحظ أننا قمنا بعمل sub classes بداخل super class وهذا ليس للتعقيد إنما نريد توضيح بأنك تستطيع صنع نفس العمل بأكثر من طريقة (كل الطرق تؤدي إلى روما) المهم أن يكون العمل صحيح ومنطقي.
- نلاحظ أن التصنيف الرئيسي super class يحتوي على دالة ترحيبية وافترضنا انك ستمرر قيمة التي هي اسم المدرسة ومن ثم تقوم بطباعة رسالة ترحيبية باسم المدرسة التي مررتها.
- قمنا بعمل override لنفس الدالة في sub classes ولكن كل دالة تقوم بعمل معين حسب التصنيف التي تتبعه فمثلا تصنيف الموظفين هي نفس الدالة فيه ولكن تقوم بطباعة معلومة أن كل موظف لديهم يحمل رقم وظيفي.
- في الدالة الرئيسية :

نستطيع أن نقوم بطريقتين لاستدعاء الدالة ذات الوجوه المتعددة:

#### ▪ الطريقة الأولى : عمل دالة خارجية ويتم الاستدعاء من خلالها

```

fun showInfo(s:schools){
    s.getInfo()
}

fun main (args:Array<String>){

```

```
var sch = schools()
showInfo(sch)

}
```

نلاحظ قمنا بتمرير كائن من التصنيف الرئيسي وبداخل الدالة هذه جلبنا الدالة التي تطبع البيانات ، ومن ثم عرفنا كائن من التصنيف الرئيسي واستدعينا الدالة التي عرفناها ومررنا لها نفس الكائن.

- ناتج التنفيذ:

```
happy kids
Welcome in to our schools happy kids
```

## ▪ الطريقة الثانية : تعريف كائن مرجعه الـ **super class**

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

    var emp:schools = schools.employees()
    emp.getInfo()
}
```

- ناتج التنفيذ:

```
every employee in this school have ID
```

## ❖ مثال (٢)

```
open class primaryOperator() {
```

```

open fun operator() {
    println("primary operator")
    println("please choose between this operators(+ , -) :")
    var c = readLine()!!
    if (c == "+") {
        println("Enter first number :")
        var a: Int = readLine()!!.toInt()
        println("Enter second number :")
        var b: Int = readLine()!!.toInt()
        var result = a + b
        println("The result is = $result")
    } else if (c == "-") {
        println("Enter first number :")
        var a: Int = readLine()!!.toInt()
        println("Enter second number :")
        var b: Int = readLine()!!.toInt()
        var result = a - b
        println("The result is = $result")
    }
}
}

```

```

open class secondaryOperator() : primaryOperator() {
    override fun operator() {
        println("secondary operator")
        println("please choose between this operators(* , /) :")
        var c = readLine()!!
        if (c == "*") {
            println("Enter first number :")

```

```

    var a: Int = readLine()!!.toInt()
    println("Enter second number :")
    var b: Int = readLine()!!.toInt()
    var result = a * b
    println("The result is = $result")
} else if (c == "/") {
    println("Enter first number :")
    var a: Int = readLine()!!.toInt()
    println("Enter second number :")
    var b: Int = readLine()!!.toInt()
    var result = a / b
    println("The result is = $result")
}
}
}

fun main(args:Array<String>){
    var secOp: primaryOperator=seconderyOperator()
    secOp.operator()
}

```

• ناتج التنفيذ:

secondary operator

please choose between this operators(\* , /) :

\*

Enter first number :

3

Enter second number :

8

The result is = 24

```
open class shapes(){  
  
    open fun getShape(){  
        println("shape")  
    }  
  
}  
  
class rectangle():shapes(){  
    override fun getShape() {  
        println("Rectangle")  
    }  
}  
  
class circle():shapes(){  
    override fun getShape() {  
        println("Circle")  
    }  
}  
  
class triangle():shapes(){  
    override fun getShape() {  
        println("Triangle")  
    }  
}  
  
fun showShapes(s:shapes){
```

```
s.getShape()
}  
  
fun main (args:Array<String>){  
    var shape1 = triangle()  
    showShapes(shape1)  
}
```

• ناتج التنفيذ:

Triangle

## الاستثناءات - exception

### أولاً : ما هي الاستثناءات

هو عبارة عن خطأ حدث ولمعالجة هذا الخطأ وجدت ما تسمى بالاستثناءات.

### ثانياً : تكوين الاستثناءات

تتكون جملة الاستثناء من قطعتين برمجيتين أساسيتين هي try , catch : وقطعة  
ثالثة اختيارية وهي : finally.

try ❖

- نكتب في هذه القطعة البرمجية الأوامر التي نحتمل وقوع الخطأ فيها بمعنى لسنا متأكدين من مدى صحة ما كتبناه، ومن المحتمل أن لا يكون بها خطأ.
- في حال قام البرنامج بقراءة هذه القطعة ووجد خطأ يقوم بتجاهل باقي الأوامر وينتقل إلى قطعة catch
- في حال تمت قراءة قطعة try ولم يكن فيها خطأ لا يدخل الـ catch وينتقل إلى finally في حال توفرها.

```
try{
}
```

## catch ❖

- في البداية نعرف متغير بين القوسين ونعين نوع المتغير وهو نوع الخطأ منطقي ، رياضي ... إلخ
- بداخل القطعة البرمجية نكتب أوامر خاصة بنوع الخطأ الذي قمنا بتمريره بمعنى لو كان نوع الخطأ منطقي ، وحدث خطأ في البرنامج نوعه رياضي لن نستطيع هذه القطعة حل المشكلة ولحل المشكلة نقوم بإنشاء قطعة catch أخرى مخصصة لنفس نوع الخطأ.

```
catch(variable:exceptionType){
}
```

## finally ❖



قطعة اختيارية تتضمن أوامر معينة عادية ليس لها علاقة بالخطأ.

## ثالثاً : مثال

```
try{
    println("enter first number :")
    var number1:Int = readLine()!!.toInt()
    println("enter second number :")
    var number2:Int = readLine()!!.toInt()
    var result = number1+number2
    println("The result is: $result")
}

catch(numEx:NumberFormatException){
    println("only integer number entered")
}
```

ربما للوهلة الأولى ترى البرنامج فتستغرب بأنه صحيح لا يوجد خطأ ، ولكن لنفترض أن المستخدم بدل من أن يدخل ارقام صحيحة قام بإدخال نص فلا بد هنا من معالجة الخطأ.

## Multi-threading

### أولاً : مفهوم multi-thread

هو كيفية تنفيذ عمل أكثر من دالة في آن واحد.

## ثانياً : عمل multi-thread

لنتعامل مع عذا المفهوم سنستعين بـ Thread class الذي يحتوي على العديد من الدوال التي تفيدنا في موضوعنا هذا. مثال:

```
class first :Thread() {

    override fun run() {
        var num = 0

        for (i in 1..3) {
            num = num + i
            println(num)
            Thread.sleep(1000)
        }

        println("-----\n The result is :$num")
    }
}

class second :Thread() {

    override fun run() {
        var num = 1

        for (i in 1..3) {
            num = num * i
        }
    }
}
```

```
println(num)
Thread.sleep(1000)
}
println("-----\n The result is :$num")}}
```

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

    var F = first()
    var S = second()

    F.start()
    S.start()

}
```

- قمنا بإنشاء تصنيفين مختلفين أحدهما لجمع الأرقام الواقعة بين ٠ و ٥ ، وتصنيف لنواتج ضرب الأعداد الواقعة بين ٠ و ٥.
- تمت عملية الوراثة في التصنيفين من Thread.
- في كل من التصنيفين استعملنا دالة منشأة في Thread للعمل وتجهيز الأوامر بداخلها وهي run وبما أنها منشأة من قبل في تصنيف Thread فنقوم بعمل override لها.
- لجعل الدالتين في التصنيفين يعملون بالتزامن بمعنى أننا لن ننتظر دالة run في تصنيف first للعمل ومن بعد انتهائها ننتقل إلى الدالة run في التصنيف second ، نستخدم دالة sleep التي نحدد لها الوقت بالـ (milliseconds) ، تقوم هذه الدالة بتعطيل عمل الدالة الحالية مدة زمنية معينة ومن ثم تعود لعملها من جديد.
- الآن للتحقق من عملنا نقوم بعمل كائن من كلا التصنيفين والعمل بهما.

- نلاحظ أننا لم نستدعي دالة run لتنفيذها وهذا صحيح لأن المسؤول عن تنفيذها هي الدالة الموضحة بالمثل وهي

start : start() .

### ▪ ناتج التنفيذ

1  
1  
3  
2  
6  
6

-----  
The result is :6

-----  
The result is :6

## ثالثاً : الدوال التي تعمل مع thread

### Start ❖

- يعود بقيمة منطقية
- إذا كانت قيمته true يتم تشغيل ال thread.

### run ❖

- لكتابة الأوامر التي سيتم تنفيذها عند عمل ThreadJ start.

## sleep ❖

- لإيقاف عمل الدالة مدة معينة نقوم بتحديدتها بالmilliseconds .

## Suspend ❖

- لإيقاف عمل الدالة مدة غير محددة ، ولاستئناف عملها نقوم باستدعاء الدالة .resume

## resume ❖

- تكمل تنفيذ الأوامر التي توقفت بسبب الدالة Suspend.

## stop ❖

- توقف عمل الكائن الذي يرث Thread.

## id ❖

- للحصول على id الخاص بكائن Thread.

## setName ❖

- لتعيين اسم بالكائن الخاص بـ Thread.

## getName ❖

- للحصول على اسم الكائن الخاص بـ Thread.

## getPriority ❖

- يعود بقيمة من ١-١٠.
- يعبر عن الأولوية التي يمتلكها الكائن الخاص بـ Thread.
- الأولوية الافتراضية هي ٥ ولكن لو أردنا تغيير قيمتها نستخدم setPriority التي تسمح لنا بتعيين قيمة جديدة للأولوية.

## toString ❖

- تعود بمجموعة من المعلومات عن الكائن الخاص بـ Thread ( name – id – group).

## Interrupt ❖

- لمقاطعة عمل الكائن الخاص بـ Thread.

## ❖ join

- تهتم بإنجاز جميع الأوامر الخاصة بكائن الـ Thread وبعد الانتهاء منها الانتقال إلى ما تبقى في main.

## ❖ State

- يعود بحالة كائن الـ Thread.

## any

---

### أولاً: ما هو class any

---

هو عبارة عن تصنيف رئيسي موجود في لغة Kotlin يحتوي بداخلها على دوال واحداث جاهزة للتنفيذ ، في حال إنشائك تصنيف جديد فإن هذا التصنيف تلقائياً يرث من تصنيف any . إذا يعتبر class any هو super class لجميع التصنيفات classes.

لذلك عند اشتقاق كائن من التصنيف الذي انشأنه فإننا بالتأكيد سنلاحظ وجود دوال جاهزة أيضا غير تلك التي كتبناها ، مثل: `hashCode` , `toString` , `equals`,...etc.