# محتوى اليوم الثاني

**محتوى الملف**

- المتغيرات Variables
- الثوابت Constants
- التعليقات Comments
- انواع البيانات Datatype
- العمليات Operators
- المصفوفات Array
- التكرار Loop
- جمل الشرط If Statements/Switch
- الدوال Functions

# مقدمة في Java

## المتغيرات Variables

أثناء عملك مع لغة java، ستحتاج في مرحلةٍ ما إلى التعامل مع البيانات. وعند رغبتك في تخزين تلك البيانات، فإنك ستحتاج الى شيء يقوم بتخزينها لك، وهذا هو عمل المتغيرات. يمكن تخيل المتغيرات والنظر إليها على أنها الأوعية أو الحاويات التي تحتوي وتُخزن كل ما يوضع فيها.
إذاً، يمكن النظر للمتغير على أنه أسلوب بسيط لتخزين البيانات واسترجاعها بشكل مؤقت أثناء عمل البرنامج.

### تعريف المتغيرات باستخدام int

. لتوضيح الفكرة لاحظ معي المثال التالي:

```
int age;
```

في المثال أعلاه قمنا بإنشاء و تعريف متغير بإسم `age`، وذلك لحفظ قيمة العُمر بداخله. الآن دعنا نقوم بإسناد قيمة العُمر `26` له:

```
age = 26;
```

لاحظ أننا قمنا باستخدام المتغير `age` في هذه المرة بدون استخدام التعبير `var`، وذلك لأنها تستخدم مرة واحدة فقط وهي أثناء تعريف المتغير، وبعد ذلك، سنتعامل مع المتغير بشكل مباشر من خلال اسمه فقط، وفي هذه الحالة هو `age`.
يمكنك أيضاً اختصار الخطوتين السابقتين في خطوة واحدة كالتالي:

```
int age = 26;
```

نلاحظ في المثال أعلاه أنه يمكننا تعريف المتغير وإسناد القيمة إليه في آنٍ واحد.
إذاً، يُمكن للمبرمج استخدام `age` في أماكن مُختلفة من البرنامج، وسيتم استبدالها بالقيمة `26`.
بالإضافة إلى ذلك، يمكن للمبرمج أن يقوم بتغيير قيمة المتغير أثناء البرنامج. فمثلاً، بعد تعريفنا للمتغير السابق `age`، يمكننا تغيير قيمته إن أردنا، ولتوضيح الفكرة لاحظ معي المثال التالي:

```
age = 30;
```

# الثوابت Constants

تختلف الثوابت `constants` عن المُتغيرات `variables` في أنه **لا يمكن تغيير قيمتها بعد إسناد أول قيمة لها**، وستظل قيمة الثابت كما هي طيلة فترة البرنامج. لتعريف ثابت في `java` نستخدم كلمة `final` متبوعة بنوع المتغير.
لتوضيح الفكرة، لاحظ معي المثال التالي:

```
final int weekDays = 7;
```

أيام الأسبوع هي دائماً 7 أيام، أي أنه لا يمكننا تغيير عدد أيام الأسبوع، وعليه، يمكن تمثيلها في ثابت في البرنامج كما سبق ورأينا. ومن الأمثلة على الثوابت؛ تاريخ الميلاد، فالتاريخ الذي ولدت فيه ثابت ولن يتغير. يوضح السطر التالي تعريف ثابت لتمثيل تاريخ الميلاد.

```
final String dateOfBrith = "10/02/1990";
```

# التعليقات Comments

في حالات معينة أثناء كتابة الكود، قد يحتاج المبرمج إلى وضع بعض الملاحظات أو التعليقات. فمثلاً، قد يحتاج إلى وضع ملاحظة لتذكيره بتعديل كود معين، فيقوم المبرمج حينها بكتابة بعض الملاحظات بجانب ذلك الكود للعودة إليه فيما بعد. وفي حالات أخرى، قد يعمل على الملف البرمجي أو المشروع البرمجي أكثر من شخص، وقد يحتاج أحد المبرمجين إلى أن يضع بعض الملاحظات لأعضاء الفريق، وهكذا. توفر التعليقات في java طريقة تُساعد المبرمج على كتابة ما يود من ملاحظات في البرنامج، وبالنسبة للغة java فإنها ستتجاهل تلك التعليقات، ولن تنظر لها على أنها تعليمات ستقوم بتنفيذها.

سنتحدث في هذا الجزء عن أنواع التعليقات في java، وهي:

- تعليق السطر الواحد Single Line Comment
- تعليق متعدد الأسطر Multi-Line Comment

## تعليق السطر الواحد Single Line Comment

عند رغبتنا في وضع تعليق في سطر واحد أو single-line comment، والذي سينتهي بنهاية السطر، سنستخدم `//` كعلامة لبداية التعليق. يوضح السطر التالي هذه الفكرة.

```
// This is a comment.
```

ليس بالضرورة أن يبدأ التعليق من بداية السطر، فقد يكون التعليق هو جزء من سطر برمجي. لتوضيح الفكرة، لاحظ معي المثال التالي:

```
int age = 25; // This is my age.
```

## تعليق متعدد الأسطر Multi-line Comments

في بعض الحالات، قد نحتاج إلى كتابة تعليق طويل، يمتد على أكثر من سطر. في هذه الحالة، يمكننا استخدام أسلوب التعليق متعدد الأسطر Multi-Line Comment. ونقوم بذلك عن طريق كتابة الملاحظات بين `/* */`. يوضح المثال التالي هذا الأمر.

```
/* Write your
   comments here..
*/
```

قد يستخدم البعض أسلوب التعليق متعدد الأسطر كتعليق سطر واحد. لتوضيح الفكرة، لاحظ معي المثال التالي:

```
/* Write your comment here.. */
```

وبنفس الأسلوب، يمكنك استخدامها عند وجود سطر برمجي. لتوضيح الفكرة، لاحظ المثال التالي:

```
int age = 25; /*This is my age.*/
```

## التسميات Naming

للتسميات في لغة java شروط ومن غير الممكن تسمية المتغيرات او الثوابت اذا خالفة هذه الشروط.

١. لا يمكن تسمية متغير يحتوي على كلمتين، لتوضيح الفكرة، لاحظ المثال التالي:

```
String my name = "khalid"; //Wrong
```

و عوضا عن ذلك نقوم بتسمية المتغيرات التي تحتوي على كلمتين عن طريق استخدام اسلوب كتابة Camel case, عن طريق كتابة اول كلمة بحرف ضغير ثم اول حرف من كل كلمة يكون حرفا كبيرا. لتوضيح الفكرة، لاحظ المثال التالي:

```
String myName = "Khalid"; //Correct
```

2. لا يمكن التسمية بأسماء تحتوي في داخلها على احدى الرموز الخاصة بالعمليات مثل علامة الجمع + و علامة لطرح الى آخره، لتوضيح الفكرة، لاحظ المثال التالي:

```
String +name = "Khalid" //Wrong
```

٣. لا يمكن التسمية بأسماء محجوزة في اللغة مثل كلمة `var` الخاصة بتعريف المتغيرات او كلمة `final` الخاصة بتعريف الثوابت

```
String final = "Khalid" //Wrong
```

# انواع البيانات Datatype

تدعم لغة java عدداً من أنواع البيانات. يوضح الجدول التالي هذه الأنواع.

| الوصف | النوع |
|---|---|
| لتمثيل الأعداد الصحيحة. | int |
| يستخدم لتمثيل الارقام التي تحتوي على النقطة العشرية | double |
| ويستخدم لتمثيل أنواع البيانات النصية مثل characters والنصوص strings. | String |
| أي بيانات من هذا النوع تكون ضمن قيمتين وهما true و false. | boolean |
| يستخدم لتمثيل الأحرف | char |

فيما يلي، ستتم مناقشة كل نوع من هذه الأنواع بشكل مفصل.

## تعريف متغير من نوع String

يمكننا استخدام ثلاث طرق مختلفة للتعامل مع النصوص على النحو التالي:

```
String message = "Welcome to java";
```

## طريقة دمج النصوص باستخدام علامة "+"

يمكننا دمج نصين أو أكثر ليكونا نصا واحدًا، كما يمكننا دمج متغير ونص على النحو التالي:

```
String itemTwo = "java";
String message = "Welcome to " + itemTwo;
```

## النصوص ومفهوم Escape Character

تستخدم هذه العمليات داخل النص String و كل واحدة منها تقوم بعملية محددة فعلى سبيل المثال \n يجعل ما بعده على سطر جديد و منها ، يوضح الجدول التالي عمليات Escape Character:

| وظيفتها | رمز العملية | اسم العملية |
|---|---|---|
| يضيف عدد من المسافات في مكان وضعه | **\t** | **Horizontal Tab** |
| يقوم بجعل ما بعده على سطر جديد | **\n** | **Newline** |
| تقوم باضافة ' مكان وضعها | **\'** | **Single quote** |
| تقوم باضافة " مكان وضعها | \" | **Double quote** |
| تقوم باضافة \ مكان وضعها | \\ | **Backslash** |

## تعريف متغير من نوع Number

يُعرّف المتغير من نوع number كتعريف المتغير العادي، ويُسند إليه قيمة رقم. لاحظ معي المثال التالي:

```
int valueType = 2;
```

## تعريف متغير من نوع Double

يُعرّف المتغير من نوع Double كتعريف المتغير العادي، ويُسند إليه قيمة رقم عشري. لاحظ معي المثال التالي:

```
double valueType = 2.0;
```

## تعريف متغير من نوع Boolean

يُعرّف المتغير من نوع boolean كسائر المتغيرات، ولكن يتميز النوع boolean عن غيره من بقية الأنواع أنه يحتوي على قيمتين فقط، أي أن أي متغير يكون نوعه boolean فستكون قيمته إما `true` أو `false`. لتوضيح الفكرة، لاحظ معي المثال التالي:

```
boolean value = true;
```

### تعريف التغيرات بنوع المتغير

تحتوي لغة java على ميزة safe type، اي مبعنى انها تمتلك ميزة الأمان عند تعريف المتغيرات، لاحظ معي المثال التالي:

```
int intValue = 2;
```

في هذا المثال قنا بتعريف متغير من نوع عدد صحيح اي بمعنى انه لا يمكننا اسناد قيمة غير قيم الاعداد الصحيحة ولو اسندنا له قيمة عدد بخلاف هذا النوع فسوف يظهر لنا خطأ لأننا اسندنا قيمة لمتغير لا يقبل هذا النوع من القيم، لاحظ معي المثال التالي:

```
int intValue;
intValue = "Not int";
// Error: Type mismatch: cannot convert from String to intJava (16777233)
```

كذلك يمكننا تعريف بقية الأنواع الاخرى بإستخدام نوع المتغير لاحظ معي المثال التالي:

```
int intValue = 2;
double doubleValue  = 2.1;
String stringVale = "StringValue";
char characters = 's';
bool boolValue = true;
```

---

## العمليات Operators

هناك عدد من العمليات المختلفة التي يمكنك استخدامها أثناء البرمجة، مثل العمليات الرياضية وعمليات المقارنة والعمليات المنطقية وغيرها من العمليات المُختلفة. سنتحدث في هذا الجزء عن مجموعة من أهم العمليات التي توفرها لغة java.

### العمليات الحسابية Arithmetic Operators

ببساطة، يمكنك تنفيذ العمليات الرياضية المختلفة باستخدام الصيغة التالية:

```
result = left op right
```

حيث يمثل `op` نوع العملية الرياضية المُراد استخدامها، ويمثل كل من `left` و `right` القيمتين (أو المتغيرين أو الثابتين) اللذين سيتم تنفيذ العملية `op` عليهما. يوضح الجدول التالي أنواع العمليات الحسابية:

| اسم العملية | رمز العملية | وظيفتها |
|---|---|---|
| Addition | + | تقوم بتنفيذ عملية الجمع. |
| Subtraction | - | تقوم بتنفيذ عملية الطرح. |
| Division | / | تقوم بتنفيذ عملية القسمة. |
| Multiplication | * | تقوم بتنفيذ عملية الضرب. |

لتوضيح الفكرة، دعنا نقوم باستبدال `op` بأحد العمليات السابقة، وسنقوم هنا باختيار الجمع `+` كمثال يمكن تطبيقه على باقي العمليات الأخرى. يوضح السطر التالي هذا الأمر:

```
int result = 5 + 2;
```

في المثال أعلاه، قمنا بتنفيذ عملية الجمع باستخدام `+` وسيتم تخزين ناتج العملية -وهو في هذه الحالة 7- في المتغير `result`.

## عمليات المقارنة Comparison Operators

يمكنك تنفيذ عمليات المقارنة المختلفة باستخدام الصيغة التالية **(مع التنبيه على أنه يمكنك استخدامها في سياقات برمجية أخرى دون إسنادها إلى قيمة، مثل استخدامها كشرط مع جملة if كما سنرى لاحقًا)**:

```
result = left op right
```

يمثل `op` نوع عملية المقارنة المُراد استخدامها ويمثل كل من `left` و `right` القيمتين(أو المتغيرين أو الثابتين) اللذين سيتم تنفيذ العملية `op` عليهما. وستكون نتيجة عمليات المقارنة هي قيمة من نوع `boolean`، أي أن ناتج المقارنة سيكون إما `true` أو `false`. يوضح الجدول التالي هذا الأمر:

| اسم العملية | رمز العملية | وظيفتها |
|---|---|---|
| Greater Than | > | تعيد true في حال كان left **أكبر من** right. |
| Less Than | < | تعيد true في حال كان left **أصغر من** right. |

| | | |
|---|---|---|
| Greater Than or Equal | >= | تعيد true في حال كان left **أكبر من أو يساوي** right. |
| Less Then or Equal | <= | تعيد true في حال كان left **أصغر من أو يساوي** right. |
| Equal | == | تعيد true في حال كان left **يساوي** right من حيث القيمة فقط. |
| Not Equal | != | تعيد true في حال كان left لا **يساوي** right من حيث القيمة فقط. |

دعنا نقوم الآن باستبدال `op` بأحد العمليات السابقة، وسنقوم هنا باستبدالها بأكبر من `>` ، ونستخدمها بصيغة java على النحو التالي:

```
boolean result = 5 > 2;
```

في هذه الحالة، قمنا بتنفيذ عملية المقارنة باستخدام `>` وسيتم تخزين الناتج -وهو `true`- في المتغير `result`.

## العمليات المنطقية Logical Operators

هناك ثلاث عمليات منطقية، اثنتان منهما تكتب بالصيغة التالية(**مع التنبيه على أنه يمكنك استخدامها في سياقات برمجية أخرى دون إسنادها إلى قيمة، مثل استخدامها كشرط مع جملة if كما سنرى لاحقًا**):

```
result = left op right
```

يمثل `op` نوع العملية المنطقية المُراد استخدامها ويمثل كل من `left` و `right` القيمتين(أو المتغيرين أو الثابتين) اللذين سيتم تنفيذ العملية `op` عليهما. وستكون نتيجة العمليات المنطقية هي قيمة من نوع `boolean`، أي أن ناتج المقارنة سيكون إما `true` أو `false`.

أما بالنسبة للعملية المتبقية، أي العملية الثالثة، فهي تكتب بالصيغة التالية :

```
result = op right
```

يوضح الجدول التالي العمليات المنطقية:

| وظيفتها | رمز العملية | اسم العملية |
|---|---|---|
| ستكون النتيجة true في حالة واحدة فقط، وهي إن كانت left و right كلاهما | && | And |

| | | |
|---|---|---|
| | | .true |
| Or | \|\| | ستكون النتيجة false في حالة واحدة فقط، وهي إن كانت left و right كلاهما false. |
| Not | ! | يقوم بعكس قيمة right. في حال كانت right تساوي true فستكون النتيجة هي false، والعكس صحيح. |

توضح الأسطر التالية استخدام العمليات السابقة برمجياً:

```
boolean first = true;
boolean second = false;
boolean andResult = first && second; // false
boolean orResult = first || second; // true
boolean notResult = !(5 == 10); // true
```

## نظرة على Increment و Decrement

من العمليات المتكررة أثناء البرمجة، عملية زيادة واحد على قيمة المتغير الحالية أو إنقاص واحد منها. تسمى عملية الزيادة في هذه الحالة Increment وتُسمى عملية الإنقاص Decrement. لتوضيح الفكرة العامة، لاحظ معي الأسطر التالية:

```
int number = 5;
number = number + 1; // 6
number = number - 1; // 5
```

في السطر الثاني، قمنا بزيادة واحد على قيمة `number`، لتصبح القيمة 6، وهذا هو المقصود بمفهوم Increment. وقمنا في السطر الثالث بإنقاص واحد من قيمة `number`، لتصبح 5، وهذا هو المقصود بمفهوم Decrement.

توفر لغة java طريقة مُختصرة لتنفيذ كلتا العمليتين السابقتين، وذلك من خلال استخدام معامل الزيادة `++` لزيادة واحد على قيمة المتغير، ومعامل الإنقاص `--` لإنقاص واحد من قيمة المتغير. لتوضيح الفكرة، دعنا نقوم بإعادة كتابة المثال السابق بالطريقة المُختصرة في المثال التالية:

```
int number = 5;
number++; // number = number + 1 (increment)
number--; // number = number - 1 (decrement)
```

# المصفوفات Array

فكر في القوائم على أنها متغير أو ثابت يتكون من مجموعة من القيم، ويمكن الوصول لكل خانة أو قيمة من تلك القيم من خلال رقم يدعى `index` وهو ترتيب القيمة بين القيم.
لتعريف مصفوفة في java سنستخدم الأقواس المربعة `[]` بجانب اسم النوع الخاص بالمصفوفة ثم نقوم بإسناد مجموعى قيم بداخل اقواس متعرجة `{}`، ونفصل بين كل قيمة والأخرى بفاصلة `,` . لتوضيح الفكرة، دعنا نقوم بتعريف مصفوفة تحتوي على ثلاثة ألوان، الأحمر `red` والأخضر `green` والأزرق `blue` كما هو موضح في السطر التالي:

```
String[] array = { "Red", "Blue", "Green" };
```

## الوصول لقيمة من خلال الرقم Index

ذكرنا سابقاً أن المصفوفة تحتوي على أكثر من قيمة، وأن كل قيمة مُرتبطة برقم يُسمى `index`، والذي يساعدنا على الوصول إلى تلك القيمة سواء لجلبها أو لتغييرها إلى قيمة أخرى. **يبدأ ترقيم خانات وقيم المصفوفة من اليسار لليمين، ويبدأ العد من index رقم 0.** لتوضيح الفكرة، سنقوم بوضع رقم `index` فوق كل خانة أو قيمة من قيم المصفوفة كما هو موضح في المثال التالي:

```
      //index:        0       1       2
String[] array = { "Red", "Blue", "Green" };
```

الآن، لو أردنا الوصول للقيمة `green` لطباعتها مثلاً، فسنجد أن رقم `index` الخاص بها هو 1، لذلك، سنستخدم اسم المصفوفة array مع الرقم 1 للوصول إليها، على النحو التالي:

```
 System.out.println(array[1]);
```

أعلاه، قمنا بطباعة قيمة المتغير array الموجود في index رقم `1`.
المخرج:

```
Blue
```

لاحظ أنها تمت طباعة القيمة `green` وذلك لأن الرقم index المرتبط بقيمة `green` يساوي `1`.

## تعديل قيمة من قيم المصفوفة

عند رغبتنا في تعديل أو تحديث قيمة معينة من قيم المصفوفة سنقوم باستخدام الرقم index بنفس الطريقة السابقة للوصول إلى المكان الذي نُريد وضع القيمة فيه. على سبيل المثال، لو أننا أردنا **تعديل قيمة اللون blue لتصبح black في مصفوفة**

array **السابقة**، عندها سنقوم بكتابة السطر التالي:

```
array[2] = "Blue";
```

## عدد عناصر المصفوفة وإستخدام length

يمكننا معرفة عدد العناصر الموجودة داخل مصفوفة معينة عن طريق استخدام خاصية length. لتوضيح الفكرة، لاحظ معي المثال التالي:

```
String[] array = { "Red", "Blue", "Green" };
System.out.println(array.length);
```

المُخرجات:

```
3
```

## قراءة عناصر القائمة

يمكنك قراءة جميع العناصر الموجودة بداخل القائمة عن طريق إستخدام for-loop على النحو الموضح في السطر التالي:

```
String[] array = { "Red", "Blue", "Green" };
for (var int i =0; i<array.length; i++){
     System.out.println(array[i]);
}
```

المخرجات

```
Red
Blue
Green
```

# التكرار Loop

## تكرار العملية باستخدام for

قد نحتاج في بعض الحالات إلى تكـرار مجموعـة من التعليمات البرمجية عدد من المرات، 10، 20، أو حتـى 100 مـرة أو أكثـر، فبـدلًا مـن تكرار كتابــة نفـس التعليمــات البرمجيــة يمكنــنا استخدام for

```
int len = 5;
for(var i = 0; i < len; i++){
     System.out.println(i);
}
```

المخرجات

```
0
1
2
3
4
```

## تكرار العملية باستخدام while

يمكننا تكرار تنفيذ أمر برمجي وفقًا لشرط معين باستخدام `while` على النحو التالي:

- كتابة المتغير الذي سيتم قياس تحقق الشرط على قيمته
- استخدام while لكتابة الشرط الذي سيتم تكرار تنفيذ الأمر حال تحققه وهو أن تكون قيمة المتغير number أكبر من أو تساوي 1
- الأوامر التي سيتم تكرار تنفيذها حال تحقق الشرط وهي كالتالي:
  a. طباعة قيمة المتغير number
  b. طرح 1 من المتغير number

```
int number = 5;
while(number >= 1){
```

```
      System.out.println(number);
      number--;
}
```

المخرجات

```
5
4
3
2
1
```

## تكرار العملية باستخدام do-while

يمكننا تكرار تنفيذ أمر برمجي وفقًا لشرط معين باستخدام `do-while` على النحو التالي:

- تختلف do-while عن while بحيث انها سوف تنفذ الأمر الموجود بداخلها مرة واحدة بغض النظر عن الشرط

في المثال التالي سوف يكون الشرط ان number >=1 و لكن قيمة number تساوي -١ حيث انه لن يتم تنفيذ الشرط

```
int number = -1;
do{
      System.out.println(number);
      number--;
}while(number >= 1);
```

المخرجات

```
-1
```

## إيقاف التكرار واستخدام **break**

تقوم break بإنهاء عمليـة التكرار بشـكل كامـل . يوضـح المثـال التالـي كيفيـة الخـروج مـن for عندمـا نحصل على الرقـم الزوجـي الأول:

```
for(var i = 1; i < 10; i++){
    if(i % 2 == 0){
        break;
    }
     System.out.println(i);
}
```

## تجاوز خطوة من التكرار واستخدام **continue**

تقوم continue بإقاف التعليمـات البرمجيـة و الانتقال لتكـرار التعليمـات البرمجيـة للعنصـر التالي . يوضـح المثـال التالـي كيـف يمكننـا اسـتخدام عبـارة continue لطباعـة الأرقـام الفرديـة فقـط :

```
for(var i = 1; i < 10; i++){
    if(i % 2 == 0){
        continue;
    }
     System.out.println(i);
}
```

# الشروط **Conditions**

## تعريف الشروط **Conditions**

الشروط ( **conditions** ) تستخدم لتحديد طريقة عمل البرنـامج نسبةً للمتغيرات التي تطرأ على الكود.
كمثال بسيط, يمكنك بناء برنـامج لمشاهدة المسلسلات, عند الدخول إليه يطلب من المستخدم في البداية أن يدخل عمره لكي يقوم بعرض أفلام تناسب عمره.
يمكنك وضع العدد الذي تريده من الشروط في البرنـامج الواحد, و تستطيع وضع الشروط بداخل بعضها البعض أيضاً.

## أنواع جمل الشرط

| اسم الجملة | وظيفتها |
| --- | --- |
| **if - else - else if statements** | نستخدمهم إذا كنا نريد تنفيذ كود معين في حال تحقق الشرط أو مجموعة من الشروط التي وضعناها. |
| **switch statement** | نستخدمها إذا كنا نريد إختبار قيمة متغير معين مع لائحة من الإحتمالات نقوم نحن بوضعها, و إذا تساوت هذه القيمة مع أي إحتمال وضعناه ستتنفذ الأوامر التي وضعناها في هذا الإحتمال فقط. |

طريقة الكتابة

```
if ( condition )
{
                    // إذا كان الشرط صحيحاً نفذ هذا الكود
}

else if ( condition )
{
                    // إذا كان الشرط صحيحاً نفذ هذا الكود
}

else
{
 // نفذ هذا الكود في حال لم يتم التعرف على الكود في أي شرط
}
```

## جملة الشرط if

تعني باللغة العربية "اذا".و هي تستخدم فقط في حال كنت تريد تنفيذ كود معين حسب شرط معين.

```
int number = 1;

    if( number < 6 )
    {
    System.out.print("number is smaller than 6");
    }
```

المخرجات

```
number is smaller than 6
```

## جملة الشرط else

في اللغة العربية تعني "أي شيء آخر". و هي تستخدم فقط في حال كنا نريد تنفيذ كود معين في حال كانت نتيجة جميع الشروط التي قبلها تساوي false
يجب وضعها دائماً في الأخير, لأنها تستخدم في حال لم يتم تنفيذ أي جملة شرطية قبلها.
إذاً, إذا نفذ البرنامج الجملة **if** أو **else if** فإنه سيتجاهل الجملة **else**.
و إذا لم ينفذ أي جملة من الجمل **if** و **else if** فإنه سينفذ الجملة **else**

```
int age = 11;

        if( age == 10 ) {
            System.out.print("age is equal to 10");
        }

        else {
            System.out.print("age is not equal to 10");
        }
```

المخرجات

```
age is not equal to 10
```

## جملة الشرط else if

جملة else if تستخدم اذا كنت تريد وضع اكثر من احتمال (شرط)
جملة او جمل الelse if يوضعون في الوسط بين else و if

```
int number = 1;


   if( number == 0 ) {
     System.out.print("Zero");
   }
   else if(  number == 2 ) {
     System.out.print("One");
   }
    else {
     System.out.print("Negative Number");
   }
   }
```

المخرجات

```
One
```

## تعريف Switch

نستخدمها إذا كنا نريد إختبار قيمة متغير معين مع لائحة من الإحتمالات نقوم نحن بوضعها فيها, و إذا تساوت هذه القيمة مع أي إحتمال وضعناه ستتنفذ الأوامر التي وضعناها في هذا الإحتمال فقط.
كل إحتمال نضعه يسمى **case**.

```
switch(expression) {
   case value:
        // Statements
        break;
    case value:
        // Statements
        break;
    default:
        // Statements
        break;
}
```

بعض القواعد المهمة Switch statements:

- غير مسموح بقيم Case المكررة
- يجب ان تكون قيمة Case من نفس نوع المتغير
- يجب ان تكون قيمة Case ثابتة او حرفية. المتغيرات غير مسموح بها
- يتم استخدام Break داخل switch لانهاء تسلسل العبارة
- عبارة break اختيارية اذا تم حذفها، فسيستمر التنفيذ في الحالة التالية
- العبارة الافتراضية و يمكن ان تظهر في اي مكان داخل Switch block في حاله اذا لم تكن في النهاية فيجب الاحتفاظ بتعليمة break بعد العبارة الافتراضية لحذف تنفيذ case التالية

```
int day = 6
        String dayString;
        switch (day) {
        case 1:
            dayString = "Monday";
            break;
        case 2:
            dayString = "Tuesday";
            break;
        case 3:
            dayString = "Wednesday";
            break;
```

```
        case 4:
            dayString = "Thursday";
            break;
        case 5:
            dayString = "Friday";
            break;
        case 6:
            dayString = "Saturday";
            break;
        case 7:
            dayString = "Sunday";
            break;
        default:
            dayString = "Invalid day";
        }
        System.out.println(dayString);
```

المخرجات

```
Saturday
```

# الدوال Functions

## تعريف دالة Function

لتعريف دالة يمكننا استخدام كلمة `static` ثم نوع الدالة - مثل `void` و هي دالة لا تعيد لنا اي قيمة - متبوعةً باسم الدالة، وفي حالتنا هنا اسم الدالة `printlnMessage`. لتوضيح الفكرة، لاحظ المثال التالي:

```
static void printHelloWorld(){
    System.out.println("Hello World");
}
```

قمنا ببناء دالة باسم `printlnMessage` ، والتي تحتوي على أمر طباعة واحد حيث ان اي سطر برمجي يكتب داخل `{}` يكون تابع للدالة، وعند تنفيذ هذه الدالة، ستتم طباعة الرسالة الظاهرة بداخل الدالة.

## استدعاء الدالة Function Call

لاستدعاء الدالة نقوم بكتابة اسم الدالة متبوعاً `()` ، على النحو التالي:

```
public static void main(String[] args) {
    printHelloWorld();
}
```

---

## الدالة والمدخلات ومفهوم Parameters

لإنشاء الدالة مع `parameter` نقوم بتحديد نوع و اسم المدخل الذي سوف يستقبل القيمة بين الأقواس `(parameter)` ، و يصبح بإمكاننا استخدامه داخل الدالة. كما هو موضح بالشكل التالي مدخل من نوع `String` و اسمه `Name` :

```
static void printMessageWithParam(String message){
    System.out.println("Welcome to " + message);
}
```

## الدالة والمدخلات ومفهوم Arguments

لاستدعاء الدالة السابقة نقوم بكتابة اسم الدالة ثم بين القوسين نكتب القيمة المطلوب إسنادها ، كما هو موضح بالشكل التالي :

```
public static void main(String[] args) {
    printMessageWithParam("منصة سطر");
}
```

ملاحظة: يجب ان يكون ترتيب قيم Arguments مبني على Parameters

## الفرق بين *Parameter* و *Argument*

تسمى المدخلات التي يتم كتابتها عند تعريف الدالة parameter بينما تسمى القيم الممررة للدالة عند الاستدعاء argument

## إنشاء الدالة مع Return

لإنشاء دالة تعود بقيمة نقوم بالبداية بتعريف الدالة بنوع القيمة المرجعة ثم نستخدم عبارة `return` متبوعةً بالقيمة التي سوف تعود بها الدالة .يوضح المثال التالي كيفية إنشاء دالة تعود بناتج جمع عددين:

```
static int sum(int one,int two ){
  return  one + two;
}
```

# مصفوفات ArrayList

تتميز مصفوفات `ArrayList` بقابليتها على اضافة و زيادة حجمها مقارنة `Array` العادية
لتعريف مصفوفة ArrayList في java نقوم بإستدعاء المكتبة الخاصة بها بهذا الشكل

```
import java.util.ArrayList;
```

ثم نقوم بتعريف ArrayList بهذا الشكل

```
ArrayList<Type> arrayList = new ArrayList<Type>();
```

ترمز كلمة Type بداخل الاقواس المتعرجة الى نوع المتغير, مثلا اذا اردنا تعريف متغير من نوع String نقوم بكتابتها بهذا الشكل

```
ArrayList<String> arrayList = new ArrayList<String>();
```

## اضافة عناصر

لأضافة عناصر الى هذه المصفوفة نقوم بإستخدام دالة add و بداخلها القيمة التي نريد اضافتها

```
ArrayList<String> arrayList = new ArrayList<String>();
arrayList.add("Red");
arrayList.add("Blue");
arrayList.add("Green");
```

## الوصول لقيمة من خلال الرقم Index

ذكرنا سابقاً في المصفوفة العادية انه يمكننا الوصول الى العناصر التي بداخل المصفوفة بإستخدام index يمكننا هنا كذلك استخدام نفس الفكرة بإستخدام دالة get
الآن، لو أردنا الوصول للقيمة `green` لطباعتها مثلاً، فسنجد أن رقم `index` الخاص بها هو ٢، لذلك، سنستخدم اسم المصفوفة arrayList مع الرقم ٢ للوصول إليها، على النحو التالي:

```
System.out.println(arrayList.get(2));
```

المخرج:

```
green
```

## تعديل قيمة من قيم المصفوفة

عند رغبتنا في تعديل أو تحديث قيمة معينة من قيم المصفوفة سنقوم باستخدام دالة set ثم نقوم بإسناد رقم الرقمindex الذي نريد تعديله ثم القيمة التي نريد اضافتها كما في المثال:
**لو تعديل قيمة اللون blue لتصبح black في مصفوفة `arrayList` السابقة، عندها سنقوم بكتابة السطر التالي:**

```
arrayList.set(2,"Black");
```

## عدد عناصر المصفوفة وإستخدام size

يمكننا معرفة عدد العناصر الموجودة داخل مصفوفة معينة عن طريق استخدام دالة size(). لتوضيح الفكرة، لاحظ معي المثال التالي:

```
ArrayList<String> arrayList = new ArrayList<String>();
arrayList.add("Red");
arrayList.add("Blue");
arrayList.add("Green");

System.out.println(arrayList.size());
```

المُخرجات:

```
3
```

### قراءة عناصر القائمة

على عكس المصفوفة العادية يمكننا في ArrayList قراءة جميع عناصر المصفوفة بمجرد طباعة المصفوفة

```
ArrayList<String> arrayList = new ArrayList<String>();
arrayList.add("Red");
arrayList.add("Blue");
arrayList.add("Green");

System.out.println(arrayList);
```

المخرجات

```
[Red, Blue, Green]
```

# مصفوفات HashMap

في فصل ArrayList ، تعلمت أن ArrayList تخزن العناصر كمجموعة مرتبة ، ويجب عليك الوصول إليها برقم index . في `HashMap` يخزن العناصر في أزواج "مفتاح / قيمة" ، ويمكنك الوصول إليها من خلال index من نوع آخر مثل String.
عند تعريف `HashMap` يتم تحديد نوع المفتاح و القيمة وقت التعريف كما في المثال التالي

```
HashMap<String,String> hashMap = new HashMap<String,String>();
```

قمنا بتعريف HashMap يحتوي على مفتاح من نوع String و قيمة من نوع String

### اضافة عناصر

لأضافة عناصر الى HashMap نقوم بإستخدام دالة put ثم اضافة المفتاح و القيمة كما في المثال التالي

```
hashMap.put("key1", "Value1");
hashMap.put("key2", "Value2");
```

```
hashMap.put("key3", "Value3");
hashMap.put("key4", "Value4");
```

## الوصول لقيمة من خلال الرقم المفتاح

للوصول للقيم نستخدم دالة get و نقوم بإسناد اسم المفتاح الذي نريد الحصول على القيمة الخاصة فيه كما في المثال التالي

```
System.out.println(hashMap.get("key1"));
```

المخرج

```
Value1
```

## عدد عناصر hashMap وإستخدام size

يمكننا معرفة عدد العناصر الموجودة داخل hashMap عن طريق استخدام دالة size(). لتوضيح الفكرة، لاحظ معي المثال التالي:

```
HashMap<String,String> hashMap = new HashMap<String,String>();
hashMap.put("key1", "Value1");
hashMap.put("key2", "Value2");
hashMap.put("key3", "Value3");
hashMap.put("key4", "Value4");

System.out.println(hashMap.size());
```

المُخرجات:

```
4
```

## قراءة عناصر hashMap

يمكننا في hashMap قراءة جميع المفاتيح و القيم بمجرد طباعة hashMap

```
HashMap<String,String> hashMap = new HashMap<String,String>();
hashMap.put("key1", "Value1");
hashMap.put("key2", "Value2");
hashMap.put("key3", "Value3");
hashMap.put("key4", "Value4");

System.out.println(hashMap);
```

المخرجات

```
{key1=Value1, key2=Value2, key3=Value3, key4=Value4}
```

-