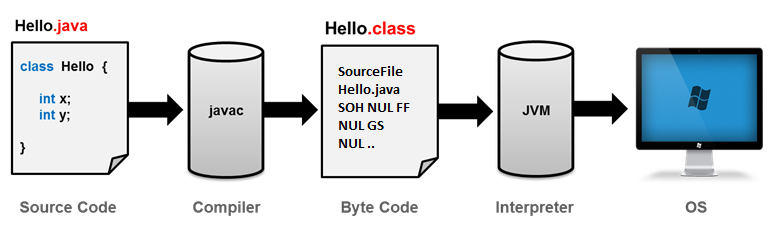
**1. مقدمة حول لغة الجافا**

هي لغة برمجة كائنية التوجه، ابتكرها [جيمس غوسلينغ](https://ar.wikipedia.org/wiki/%D8%AC%D9%8A%D9%85%D8%B3_%D8%BA%D9%88%D8%B3%D9%84%D9%8A%D9%86%D8%BA) في عام [1992م](https://ar.wikipedia.org/wiki/1992) أثناء عمله في مختبرات شركة [صن ميكروسيستمز](https://ar.wikipedia.org/wiki/%D8%B5%D9%86_%D9%85%D9%8A%D9%83%D8%B1%D9%88%D8%B3%D9%8A%D8%B3%D8%AA%D9%85%D8%B2) وذلك لاستخدامها بمثابة العقل المفكر المستخدم لتشغيل أجهزة التطبيقات الذكية مثل التليفزيون التفاعلي، وقد كانت لغة الجافا تطويرًا للغة سي ++، وعند ولادتها أطلق عليها مبتكرها **«أواك»** بمعنى شجرة [السنديان](https://ar.wikipedia.org/wiki/%D8%B3%D9%86%D8%AF%D9%8A%D8%A7%D9%86)؛ وهي الشجرة التي كان يراها من نافذة مكتبه وهو يعمل في مختبرات صن ميكروسيستمز، ثم تغير الاسم إلى جافا، وهذا الاسم (على غير العادة في تسمية لغات البرمجة) ليس الحروف الأولى من كلمات جملة معينة أو تعبيرًا بمعنى معين، ولكنه مجرد اسم وضعه مطورو هذه اللغة لينافس الأسماء الأخرى.

**1.1 كيف تعمل لغة الجافا**



**1.2 مميزات لغة الجافا:**

* لغة كائنية التوجه
* مستقلة عن نظام التشغيل ولها بيئة تشغيل خاصه بها JavaVirtualMachine(JVM)
* سهلة التعلم
* توفر إمكانية إضافة برامج إلى صفحات الويب
* توفر إمكانيات ومكتبات للتشبيك
* تستخدم في الألعاب والبرامج والتطبيقات وقواعد البيانات
* أكثر من 3 مليار جهاز حول العالم يعمل بالجافا
* وغيرها الكثير

**1.3 التركيبة اللغوية للجافا (Syntax)**

بنية جافا تستمد الكثير من بنية لغة البرمجة سي: بنية الإسناد، التعابير، التحكم، الكتل الجملية (blocks). جافا بنيت خصيصًا لتكون لغة كائنية التوجه، حيث أن كامل الكود يكون مضمناً داخل كائنات (Objects) فيما عدا المتغيرات الأساسية (الأرقام، والقيم المنطقية والأحرف), حيث أنها لم توضع ككائنات لأسباب تتعلق بالأداء.

**مثال:** برنامج Hello World التقليدي مكتوباً بلغة الجافا:

public class HelloWorld {  
 public static void main(String[] args) {  
 System.out.println("Hello, World!");  
 }  
}

## **2. أساسيات لغة الجافا**

**2.1 المتغيرات Variables**

ستحتاج في مرحلةٍ ما إلى التعامل مع البیانات. وعند رغبتك في تخزین تلك البیانات، فإنك ستحتاج إلى شيء یقوم بتخزینھا لك، وھذا ھو عمل المتغیرات. یمكن تخیل المتغیرات والنظر إلیھا على أنھا الأوعیة أو الحاویات التي تحتوي وتخُزن كل ما یوضع فیھا.

إذاً، یمكن النظر للمتغیر على أنه أسلوب بسیط لتخزین البیانات واسترجاعھا بشكل مؤقت أثناء عمل البرنامج.

### **تعریف المتغیرات باستخدام int**

لتوضیح الفكرة لاحظ معي المثال التالي:

int age;

في المثال أعلاه قمنا بإنشاء و تعریف متغیر باسم age، وذلك لحفظ قیمة العُمر بداخله. الآن دعنا نقوم بإسناد قیمة العُمر 26 له:

age=26;

لاحظ أننا قمنا باستخدام المتغیرage في ھذه المرة بدون استخدام int ، وذلك لأنھا تستخدم مرة واحدة فقط وھي أثناء تعریف المتغیر، وبعد ذلك، سنتعامل مع المتغیر بشكل مباشر من خلال اسمه فقط، وفي ھذه الحالة ھو age.

یمكنك أیضاً اختصار الخطوتین السابقتین في خطوة واحدة كالتالي:

int age=26;

نلاحظ في المثال أعلاه أنه یمكننا تعریف المتغیر وإسناد القیمة إلیه في آنٍ واحد

يمكن للمبرمج استخدام age في أماكن مختلفة من البرنامج، وسيتم استبدالها بالقيمة 26. بالإضافة إلى ذلك، یمكن للمبرمج أن یقوم بتغییر قیمة المتغیر أثناء البرنامج. فمثلاً، بعد تعریفنا للمتغیر السابق age، یمكننا تغییر قیمته إن أردنا، ولتوضیح الفكرة لاحظ معي المثال التالي:

age=30;

#### **2.2 التعلیقات Comments**

في حالات معینة أثناء كتابة الكود، قد یحتاج المبرمج إلى وضع بعض الملاحظات أو التعلیقات. فمثلاً، قد یحتاج إلى وضع ملاحظة لتذكیره بتعدیل كود معین، فیقوم المبرمج حینھا بكتابة بعض الملاحظات بجانب ذلك الكود للعودة إلیه فیما بعد. وفي حالات أخرى، قد یعمل على الملف البرمجي أو المشروع البرمجي أكثر من شخص، وقد یحتاج أحد المبرمجین إلى أن یضع بعض الملاحظات لأعضاء الفریق، وھكذا.

توفر التعلیقات في java طریقة تُساعد المبرمج على كتابة ما یود من ملاحظات في البرنامج، وبالنسبة للغة java فإنھا ستتجاھل تلك التعلیقات، ولن تنظر لھا على أنھا تعلیمات ستقوم بتنفیذھا.

سنتحدث في ھذا الجزء عن أنواع التعليقات في java، و ھي:

Single Line Comment تعلیق السطر الواحد

Multi-Line Comment تعلیق متعدد الأسطر

##### Single Line Comment تعلیق السطر الواحد

عند رغبتنا في وضع تعلیق في سطر واحد أو single-line comment، والذي سینتھي بنھایة السطر، سنستخدم //كعلامة لبدایة التعلیق. یوضح السطر التالي ھذه الفكرة.

// This is a comment.

لیس بالضرورة أن یبدأ التعلیق من بدایة السطر، فقد یكون التعلیق ھو جزء من سطر برمجي. لتوضیح الفكرة، لاحظ معي المثال التالي:

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

##### Multi-line Comments تعلیق متعدد الأسطر

في بعض الحالات، قد نحتاج إلى كتابة تعلیق طویل، یمتد على أكثر من سطر. في ھذه الحالة، یمكننا استخدام أسلوب التعلیق متعدد الأسطر Multi-Line Comment. ونقوم بذلك عن طریق كتابة الملاحظات بین /\* \*/. یوضح المثال التالي ھذا الأمر.

/\* Write your  
 comments here  
\*/

###### **2.3 التسمیات Naming**

للتسمیات في لغة java شروط ومن غیر الممكن تسمیة المتغیرات او الثوابت إذا خالفة ھذه الشروط.

1. لا یمكن تسمیة متغیر یحتوي على كلمتین، لتوضیح الفكرة، لاحظ المثال التالي:

String my name = "khalid"; //Wrong

و عوضا عن ذلك نقوم بتسمیة المتغیرات التي تحتوي على كلمتین عن طریق استخدام اسلوب كتابة Camel case ، عن طریق كتابة اول كلمة بحرف صغير ثم أول حرف من كل كلمة یكون حرفا كبیرا. لتوضیح الفكرة، لاحظ المثال التالي:

String myName = "Khalid"; //Correct

2 .لا یمكن التسمیة بأسماء تحتوي في داخلھا على احدى الرموز الخاصة بالعملیات مثل علامة الجمع + و علامة لطرح الى آخره، لتوضیح الفكرة، لاحظ المثال التالي:

String +name = "Khalid" //Wrong

3. لا یمكن التسمیة بأسماء محجوزة في اللغة مثل كلمة final الخاصة بتعریف الثوابت:

String final = "Khalid" //Wrong

#### **2.4 انواع البیانات Datatype**

تدعم لغة Javaعدداً من أنواع البیانات. یوضح الجدول التالي ھذه الأنواع.

|  |  |
| --- | --- |
| النوع | الوصف |
| int | لتمثیل الأعداد الصحیحة. |
| double | یستخدم لتمثیل الارقام التي تحتوي على النقطة العشریة |
| String | ویستخدم لتمثیل أنواع البیانات النصیة مثل characters والنصوص strings. |
| boolean | أي بیانات من ھذا النوع تكون ضمن قیمتین وھما true و false. |
| char | یستخدم لتمثیل الأحرف |

##### تعریف متغیر من نوع String

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

String message = "Welcome to java";

##### طریقة دمج النصوص باستخدام علامة "+"

یمكننا دمج نصین أو أكثر لیكونا نصا واحدًا، كما یمكننا دمج متغیر ونص على النحو التالي:

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

##### Escape Character النصوص ومفھوم

تستخدم ھذه العملیات داخل النص String و كل واحدة منھا تقوم بعملیة محددة فعلى سبیل المثال n\ یجعل ما بعده على سطر جدید و منھا ، یوضح الجدول التالي عملیات Escape Character:

|  |  |  |
| --- | --- | --- |
| اسم العملیة | رمز العملیة | وظیفتھا |
| Horizontal Tab | \t | یضیف عدد من المسافات في مكان وضعه |
| Newline | \n | یقوم بجعل ما بعده على سطر جدید |
| Single quote | \' | تقوم باضافة ' مكان وضعھا |
| Double quote | \” | تقوم باضافة " مكان وضعھا |
| Backslash | \\ | تقوم باضافة \ مكان وضعھا |

###### **تعریف متغیر من نوع Number**

یُعرّف المتغیر من نوع numbe كتعریف المتغیر العادي، ویُسند إلیه قیمة رقم. لاحظ معي المثال التالي:

int valueType = 2;

###### **تعریف متغیر من نوع Double**

یُعرّف المتغیر من نوع Double كتعریف المتغیر العادي، ویُسند إلیه قیمة رقم عشري. لاحظ معي المثال التالي:

double valueType = 2.0;

###### **تعریف متغیر من نوع Boolean**

یُعرّف المتغیر من نوع boolean كسائر المتغیرات، ولكن یتمیز النوع boolean عن غیره من بقیة الأنواع أنهیحتويعلى قیمتین فقط، أي أن أي متغیر یكون نوعه boolean فستكون قیمته إما true أو false. لتوضیح الفكرة، لاحظ معي المثال التالي:

boolean value = true;

**2.5 العمليات Oparators**

ھناك عدد من العملیات المختلفة التي یمكنك استخدامھا أثناء البرمجة، مثل العملیات الریاضیة وعملیات المقارنة والعملیات المنطقیة وغیرھا من العملیات المُختلفة. سنتحدث في ھذا الجزء عن مجموعة من أھم العملیات التي توفرھا لغة java.

**العمليات الرياضية Arithmetic Operators**

ببساطة، يمكنك تنفيذ العمليات الرياضية المختلفة باستخدام الصيغة التالية:

result = left op right

حيث يمثل op نوع العملية الرياضية المراد استخدامها، ويمثل كل من left و right القيمتين (او المتغيرين) اللذان سيتم تنفيذ العملية op عليهما. يوضح الجدول التالي أنواع العمليات الحسابية:

|  |  |  |
| --- | --- | --- |
| **اسم العملیة** | **رمز العملیة** | **وظیفتھا** |
| 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 الاخرى. يوضح السطر التالي هذا الامر:

int result = 5 + 2;

في المثال أعلاه، قمنا بتنفیذ عملیة الجمع باستخدام + وسیتم تخزین ناتج العملیة وهو في ھذه الحالة 7 في المتغیر result.

**عمليات المقارنة Comparison Operators**

یمكنك تنفیذ عملیات المقارنة المختلفة باستخدام الصیغة التالیة (مع التنبیه على أنه یمكنك استخدامھا في سیاقات برمجیة أخرى دون إسنادھا إلى قیمة، مثل استخدامھا كشرط مع جملة if كما سنرى لاحقًا):

result = left op right

یمثل op نوع عملیة المقارنة المُراد استخدامھا ویمثل كل من left و right القیمتین (أو المتغیرین) اللذان سیتم تنفیذ العملیة op علیھما. وستكون نتیجة عملیات المقارنة ھي قیمة من نوع boolean ، أي أن ناتج المقارنة سیكون إما true أو false.

یوضح الجدول التالي ھذا الأمر:

|  |  |  |
| --- | --- | --- |
| **الوصف** | **استخدامها** | **العملية** |
| x اكبر من y | x > y | > |
| x اكبر من او يساوي y | x >= y | >= |
| x اصغرمن y | x < y | < |
| x اصغرمن او يساوي y | x <= y | <= |
| x يساوي y | x == y | == |
| x لا يساوي y | x != y | != |

**Logical Operators العملیات المنطقیة**

ھناك ثلاث عملیات منطقیة، اثنتان منھما تكتب بالصیغة التالیة (مع التنبیه على أنه یمكنك استخدامھا في سیاقات برمجیة أخرى دون إسنادھا إلى قیمة، مثل استخدامھا كشرط مع جملة if كما سنرى لاحقًا):

result = left op right

یمثل op نوع العملیة المنطقیة المُراد استخدامھا ویمثل كل من left و right القیمتین (أو المتغیرین) اللذان سیتم تنفیذ العملیة op علیھما. وستكون نتیجة العملیات المنطقیة ھي قیمة من نوع boolean ، أي أن ناتج المقارنة سیكون إما true or false .

بالنسبة للعملیة المتبقیة، أي العملیة الثالثة، فھي تكتب بالصیغة التالیة :

result = left op right

:توضح الجداول التالية العملیات المنطقیة

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A&&B** |
| FALSE | FALSE | FALSE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | TRUE |

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A||B** |
| FALSE | FALSE | FALSE |
| FALSE | TRUE | TURE |
| TRUE | FALSE | TRUE |
| TRUE | TRUE | TRUE |

توضح الأسطر التالیة استخدام العملیات السابقة برمجیاً:

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

###### **Decrement و Increment نظرة على**

من العملیات المتكررة أثناء البرمجة، عملیة زیادة واحد على قیمة المتغیر الحالیة أو إنقاص واحد منھا. تسمى عملیة الزیادة في ھذه الحالة Increment وتُسمى عملیة الإنقاص Decrement. لتوضیح الفكرة العامة، لاحظ معي الأسطر التالیة:

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

في السطر الثاني، قمنا بزیادة واحد على قیمة number، لتصبح القیمة 6 ، وھذا ھو المقصود بمفھوم Increment. وقمنا في السطر الثالث بإنقاص واحد من قیمة number ، لتصبح 5 ، وھذا ھو المقصود بمفھوم Decrement.

توفر لغة java طریقة مُختصرة لتنفیذ كلتا العملیتین السابقتین، وذلك من خلال استخدام معامل الزیادة ++ لزیادة واحد على قیمة المتغیر، ومعامل الإنقاص -- لإنقاص واحد من قیمة المتغیر. لتوضیح الفكرة، دعنا نقوم بإعادة كتابة المثال السابق بالطریقة المُختصرة في المثال التالیة:

int number = 5;  
number++; // number = number + 1 (increment)

number--; // number = number - 1 (decrement)

## **3. الشروط Condittions**

**3.1 تعريف الشروط**

الشروط (conditions) تستخدم لتحدید طریقة عمل البرنامج نسبةً للمتغیرات التي تطرأ على الكود.

كمثال بسیط، یمكنك بناء برنامج لمشاھدة المسلسلات، عند الدخول إلیه یطلب من المستخدم في البدایة أن یدخل عمره لكي یقوم بعرض أفلام تناسب عمره.

یمكنك وضع العدد الذي تریده من الشروط في البرنامج الواحد ، و تستطیع وضع الشروط بداخل بعضھا البعض أیضاً.

**3.2 أنواع جمل الشروط**

|  |  |
| --- | --- |
| **اسم الجملة** | **وظیفتھا** |
| if - else - else if statements | نستخدمھم إذا كنا نرید تنفیذ كود معین في حال تحقق الشرط أو مجموعة من الشروط التي وضعناھا. |
| switch statement | نستخدمھا إذا كنا نرید إختبار قیمة متغیر معین مع لائحة من الإحتمالات نقوم نحن بوضعھا, و إذا تساوت ھذه القیمة مع أي إحتمال وضعناه ستتنفذ الأوامر التي وضعناھا في ھذا الإحتمال فقط. |

**طریقة كتابة الشرط:**

if (condition)  
 {  
 // إذا كان الشرط صحیحا نفذ ھذا الكود  
 }  
  
else if (condition)  
 {  
 // إذا كان الشرط صحیحا نفذ ھذا الكود  
 }  
  
else  
 {  
 // نفذ ھذا الكود في حال لم یتم التعرف على الكود في أي شرط  
 }

**3.2.1 جملة الشرط if**

تعني باللغة العربیة “اذا”. وھي تستخدم فقط في حال كنت ترید تنفیذ كود معین حسب شرط معین.

int number = 1;  
  
if( number < 6 )  
{  
 System.*out*.print("number is smaller than 6");  
}

المخرجات

number is smaller than 6

**3.2.2 الشرط else**

في اللغة العربیة تعني "أي شيء آخر". وھي تستخدم فقط في حال كنا نرید تنفیذ كود معین في حال كانت نتیجة جمیع الشروط التي قبلھا تساوي false.

یجب وضعھا دائماً في الأخیر، لأنھا تستخدم في حال لم یتم تنفیذ أي جملة شرطیة قبلھا.

إذاً، إذا نفذ البرنامج الجملة if أو else if فإنه سیتجاھل الجملة else. وإذا لم ینفذ أي جملة من الجمل if و else if فإنه سینفذ 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

**3.2.3 جملة الشرط else if**

جملة else if تستخدم اذا كنت ترید وضع اكثر من احتمال (شرط). جملة او جمل الelse if یوضعون في الوسط بین else و if

int number = 1;  
  
if( number == 0 ) {  
 System.*out*.print("Zero");  
}  
else if( number == 1 ) {  
 System.*out*.print("One");  
}  
 else {  
 System.*out*.print("Negative Number");  
}

المخرجات

One

**3.3 تعريف Switch**

نستخدمھا إذا كنا نرید إختبار قیمة متغیر معین مع لائحة من الإحتمالات نقوم نحن بوضعھا فیھا ، و إذا تساوت ھذه القیمة مع أي إحتمال وضعناه ستتنفذ الأوامر التي وضعناھا في ھذا الإحتمال فقط.

كل إحتمال نضعه یسمى case.

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

**بعض القواعد المھمة Switch statements:**

* غیر مسموح بقیم Case المكررة
* یجب ان تكون قیمة Case من نفس نوع المتغیر
* یجب ان تكون قیمة Case ثابتة او حرفیة
* مسموح أن یتم استخدام Break داخل switch لانھاء تسلسل العبارة
* عبارة break اختیاریة اذا تم حذفھا، فسیستمر التنفیذ في الحالة التالیة
* العبارة الافتراضیة و یمكن ان تظھر في اي مكان داخل Switch block في حالة اذا لم تكن في النھایة فیجب الاحتفاظ بتعلیمة break بعد العبارة الافتراضیة لحذف تنفیذ case التالیة

**4.التكرار Loop**

**4.1 تكرار العملیة باستخدام for**

قد نحتاج في بعض الحالات إلى تكـرار مجموعـة من التعلیمات البرمجیة عدد من المرات 10، 20 ، أو حتى 100 مرة وأكثر، فبـدلاً مـن تكرار كتابة نفـس التعلیمات البرمجیة یمكنـنا اسـتخدام for .

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

المخرجات

|  |
| --- |
| 0 |
| 1 |
|  |
| 2  3  4 |

**4.2 تكرار العملیة باستخدام while**

* یمكننا تكرار تنفیذ أمر برمجي وفقًا لشرط معین باستخدام while على النحو التالي:
* كتابة المتغیر الذي سیتم قیاس تحقق الشرط على قیمته.
* استخدام while لكتابة الشر ط الذي سیتم تكرار تنفیذ الأمر حال تحققه وھو أن تكون قیمة المتغیر number أكبر من أو تساوي 1.

الأوامر التي سیتم تكرار تنفیذھا حال تحقق الشرط وھي كالتالي:

* طباعة قیمة المتغیر number.
* طرح 1 من المتغیر number.

int number = 5;   
while(number >= 1({  
 System.*out*.println(number);

number--;  
 }

المخرجات

|  |
| --- |
| 5 |
| 4  3 |
| 2  1 |

**4.3 تكرار العملیة باستخدام do-while**

یمكننا تكرار تنفیذ أمر برمجي وفقًا لشرط معین باستخدام do-while على النحو التالي:

تختلف do-while عن while بحیث أنها سوف تنفذ الأمر الموجود بداخلھا مرة واحدة بغض النظر عن الشرط.

في المثال التالي سوف یكون الشرط ان number >=1 ولكن قیمة number تساوي -1 حیث انه لن یتم تنفیذ الشرط:

int number = -1;   
do{  
 System.*out*.println(number);

number--;  
 }while(number >= 1);

المخرجات

-1

**4.4 إیقاف التكرار واستخدام break**

تقوم break بإنھاء عملیـة التكرار بشـكل كامـل. یوضح المثال التالي كیفیة الخـروج مـن for عندما نحصل على الرقم الزوجي الأول:

for(int i = 1; i < 10; i++){  
 if(i % 2 ==0){

break;   
 }  
 System.*out*.println(i);  
 }

**4.5 تجاوز خطوة من التكرار واستخدام continue**

تقوم continue بايقاف التعلیمات البرمجیة والانتقال لتكـرار التعلیمـات البرمجیة للعنصرالتالي. یوضـح المثـال التالي كیف یمكننا اسـتخدام عبـارة continue لطباعة الأرقام الفردیة فقط:

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

**5. المصفوفات Arrays**

فكر في المصفوفات على أنھا متغیر أو ثابت یتكون من مجموعة من القیم، ویمكن الوصول لكل خانة أو قیمة من تلك القیم من خلال رقم یدعى index وھو ترتیب القیمة بین القیم.

لتعریف مصفوفة في java سنستخدم الأقواس المربعة [ ] بجانب اسم النوع الخاص بالمصفوفة ثم نقوم بإسناد مجموع قیم بداخل اقواس متعرجة { }، ونفصل بین كل قیمة والأخرى بفاصلة , . لتوضیح الفكرة، دعنا نقوم بتعریف مصفوفة تحتوي على ثلاثة ألوان، الأحمر red والأخضر green والأزر ق blue كما ھو موضح في السطر التالي:

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

**5.1 الوصول لقیمة من خلال رقم Index**

ذكرنا سابقا أن المصفوفة تحتوي على أكثر كل قیمة مُرتبطة برقم یسُمى index، والذي یساعدنا على

الوصول إلى تلك القیمة سواء لجلبھا أو لتغییرھا إلى قیمة أخرى. یبدأ ترقیم خانات وقیم المصفوفة من الیسار للیمین، ویبدأ العد من index رقم 0.

الان ، لو اردنا الوصول لقيمة Green لطباعتها مثلاً، فسنجد ان رقم index الخاص بها هو 2، لذلك، سنستخدم اسم المصفوفة array مع الرقم 2 للوصول اليها، على النحو التالي:

System.out.println(array[2]);

أعلاه، قمنا بطباعة قیمة المتغیر array الموجود في index رقم 2.

المخرجات:

Green

لاحظ أنھا تمت طباعة القیمة Green وذلك لأن رقم index المرتبط بقیمة Green یساوي 2.

**5.2 تعدیل قیمة من قیم المصفوفة**

عند رغبتنا في تعدیل أو تحدیث قیمة معینة من قیم المصفوفة سنقوم باستخدام رقم index بنفس الطریقة السابقة للوصول إلى المكان الذي نرُید وضع القیمة فیه على سبیل المثال، لو أننا أردنا تعدیل قیمة اللون blue لتصبح black في مصفوفة array السابقة، عندھا سنقوم بكتابة السطر التالي:

**5.3 عدد عناصر المصفوفة وإستخدام length**

یمكننا معرفة عدد العناصر الموجودة داخل مصفوفة معینة عن طریق استخدام خاصیة length. لتوضیح الفكرة، لاحظ معي المثال التالي:

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

المخرجات:

3

**5.4 قراءة عناصر المصفوفة**

یمكنك قراءة جمیع العناصر الموجودة بداخل المصفوفة عن طریق إستخدام for-loop على النحو الموضح في السطر التالي:

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

المخرجات:

Red

Blue

Green

**6. الدوال Methods**

**6.1 تعريف دالة Methods**

لتعریف دالة یمكننا استخدام كلمة Static ثم نوع الدالة – مثل Void وهي دالة لا تعيد لنا أي قيمة – متبوعة باسم الدالة، وفي حالتنا هنا اسم الدالة printInMessage. لتوضيح الفكرة، لاحظ المثال التالي:

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

قمنا ببناء دالة باسم printlnMessage ، والتي تحتوي على أمر طباعة واحد حیث ان اي سطر برمجي یكتب داخل {} یكون تابع للدالة، وعند تنفیذ ھذه الدالة، ستتم طباعة الرسالة الظاھرة بداخل الدالة.

**6.2 استدعاء دالة Function call**

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

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

**6.****3 الدالة والمدخلات ومفھوم Parameters**

إنشاء الدالة مع parameter نقوم بتحدید نوع واسم المدخل الذي سوف یستقبل القیمة بین الأقواس (parameter)

، ویصبح بإمكاننا استخدامه داخل الدالة. كما ھو موضح بالشكل التالي مدخل من نوع String واسمه:

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

**6.4 الدالة والمدخلات ومفھوم Arguments**

لاستدعاء الدالة السابقة نقوم بكتابة اسم الدالة ثم بین القوسین نكتب القیمة المطلوب إسنادھا، كما ھو موضح بالشكل التالي:

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

ملاحظة: یجب ان یكون ترتیب قیم Arguments مبني على .Parameters

**6.5 الفرق بين Parameter و Argument**

تسمى المدخلات التي یتم كتابتھا عند تعریف الدالة parameter بینما تسمى القیم الممررة للدالة عند الاستدعاء argument.

**6.6 إنشاء الدالة مع Return**

لإنشاء دالة تعود بقیمة نقوم بالبدایة بتعریف الدالة بنوع القیمة المرجعة ثم نستخدم عبارة return متبوعةً بالقیمة التي سوف تعود بھا الدالة. یوضح المثال التالي كیفیة إنشاء دالة تعود بناتج جمع عددین:

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

**7. الاستثناءات Exceptions**

الإستثناء في البرمجة عبارة عن حدث يقوم بتعطيل السير الطبيعي للبرنامج. وهو كائن يتم إلقاؤه في وقت التشغيل. يمكن أن يحدث استثناء لأسباب عديدة ومختلفة منها الخطأ النحوي والخطأ وقت التشغيل والخطأ في المنطق.

**7.1 خطأ لغوي Syntax Error**

الخطأ اللغوي (Syntax Error) هو الخطأ في كتابة الأوامر البرمجية سواء في الكتابة بالأحرف الكبيرة أو نسيان الأقواس أو عدم استدعاء class وغيرها.

system.out.print;

**7.2 خطأ وقت التشغيل Runtime error**

يحدث الخطأ في وقت التشغيل عندما يكون البرنامج صحيحًا من الناحية اللغوية، ولكنه يحتوي على مشكلة يتم اكتشافها فقط أثناء تنفيذ البرنامج.

class DivByZero {  
 public static void main(String args[])  
 {  
 int var1 = 15;  
 int var2 = 5;  
 int var3 = 0;  
 int ans1 = var1 / var2;  
  
 // This statement causes a runtime error,  
 // as 15 is getting divided by 0 here  
 int ans2 = var1 / var3;  
  
 System.*out*.println(  
 "Division of va1"  
 + " by var2 is: "  
 + ans1);  
 System.*out*.println(  
 "Division of va1"  
 + " by var3 is: "  
 + ans2);  
 }  
}

**7.3 خطأ منطقي Logical error**

الأخطاء المنطقية هي الأخطاء التي يرتكبها المبرمجون. تعمل هذه البرامج التي بها هذه الأخطاء ولكنها لا تعطي النتائج المتوقعة.

class IncorrectMessage {  
 public static void main(String args[])  
 {  
 System.*out*.println(*sum*(6,3));  
 }  
 public static int sum(int num1,int num2){  
 return num2-num1;  
 }  
}

**7.4 حل استثناءات وقت التشغيل Solving Runtime Exceptions**

**try-catch 7.4.1**

يتم استخدام try-catch في جافا لإحاطة التعليمات البرمجية التي قد تؤدي إلى إستثناء. يجب استخدامه ضمن الميثود. إذا حدث استثناء في جملة معينة في try block، فلن يتم تنفيذ باقي الكود. لذلك، يوصى بعدم الاحتفاظ بالكود في try-block التي لن تؤدي إلى استثناء. يجب أن يتبع Java try block إما catch أو block.

try {  
 int[] myNumbers = {1, 2, 3};  
 System.out.println(myNumbers[10]);  
 } catch (Exception e) {  
 System.out.println("Something went wrong.");  
 }

**throw & throws 7.4.2**

**تعريف throws**

إذا قمت بتعريف دالة وأردت لهذه الدالة أن ترمي إستثناء إذا حدث شيء معين فعليك وضع الكلمة throws بعد أقواس الباراميترات ثم تحديد نوع الإستثناء الذي قد ترميه الدالة، وإذا قمت مسبقاً بتعريف إستثناء يمكنك جعل الدالة تقوم برميه.

public class Main {  
 static void checkAge(int age) throws ArithmeticException {  
 if (age < 18) {  
 throw new ArithmeticException("Access denied - You must be at least 18 years old.");  
 }  
 else {  
 System.*out*.println("Access granted - You are old enough!");  
 }  
 }

public static void main(String[] args) {  
 *checkAge*(15); // Set age to 15 (which is below 18...)  
 }  
}

**تعريف throw**

يتم استخدام الكلمة throw لإنشاء خطأ مخصص، يتم استخدام تعليمة throw مع نوع الاستثناء.

public class Main {  
 static void checkAge(int age) {  
 if (age < 18) {  
 throw new ArithmeticException("Access denied - You must be at least 18 years old.");  
 }  
 else {  
 System.*out*.println("Access granted - You are old enough!");  
 }  
 }  
  
 public static void main(String[] args) {  
 *checkAge*(15); // Set age to 15 (which is below 18...)  
 }  
}

**7.5 أنواع استثناءات وقت التشغيل - Types of Runtime Exceptions**

**Checked Exceptions 7.5.1**

هذه هي الاستثناءات التي يتم التحقق منها في وقت الترجمة. إذا ألقى بعض الكود ضمن طريقة استثناءً محددًا، فيجب إما معالجة الاستثناء أو تحديد الاستثناء باستخدام الكلمة الأساسية throws.

// Java Program to Illustrate Checked Exceptions  
// Where FileNotFoundException does not occur  
import java.io.\*;  
  
class GFG {  
  
 // Main driver method  
 public static void main(String[] args)  
 throws IOException  
 {  
 // Creating a file and reading from local repository  
 FileReader file = new FileReader("C:\\test\\a.txt");  
  
 // Reading content inside a file  
 BufferedReader fileInput = new BufferedReader(file);  
  
 // Printing first 3 lines of file "C:\test\a.txt"  
 for (int counter = 0; counter < 3; counter++)  
 System.*out*.println(fileInput.readLine());  
  
 // Closing all file connections  
 // using close() method  
 // Good practice to avoid any memory leakage  
 fileInput.close();  
 }  
}

**Unchecked Exceptions 7.5.2**

هذه هي الاستثناءات التي لم يتم التحقق منها في وقت compile. في استثناءات Java ضمن فئات Error وRuntimeException هي استثناءات لم يتم تحديدها، يتم التحقق من كل شيء آخر ضمن قابل للإلقاء.

// Java Program to Illustrate Un-checked Exceptions  
  
// Main class  
class GFG {  
  
 // Main driver method  
 public static void main(String args[])  
 {  
 // Here we are dividing by 0  
 // which will not be caught at compile time  
 // as there is no mistake but caught at runtime  
 // because it is mathematically incorrect  
 int x = 0;  
 int y = 10;  
 int z = y / x;  
 }  
}

**8. نماذج البرمجة Programming Paradigms**

**Object-oriented programming 8.1**

في البرمجة الكائنية، يتم تمثيل المعلومات كفئة تصف مفاھيم مجال المشكلة ومنطق التطبيق. تحدد الفئات الطرق التي تحدد كيفية معالجة المعلومات.

**Procedural programming 8.2**

بينما في البرمجة الشيئية، يتم تشكيل ھيكل البرنامج من خلال البيانات التي يعالجھا، في البرمجة الإجرائية ، يتم تشكيل ھيكل البرنامج من خلال الوظيفة المطلوبة للبرنامج: يعمل البرنامج كدليل خطوة بخطوة للوظيفة المطلوب أداؤھا .

**Static keyword 8.3**

الفرق بين Non-Static vs. Static في الدروس السابقة قمنا بتعريف دوال باستخدام static كما في المثال التالي:

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

الفرق بين Static vs. Non-Static ھو انه في الدوال من نوع static يمكننا استخدامھا بدون انشاء object لھذا الكلاس بينما دوال Non-Static يجب علينا انشاء object لھذا الكلاس ثم استخدام ھذه الدالة كما في المثال التالي:

class Animal{  
 private String name;

private int age;  
  
  
 Animal(String name, int age) {

this.name = name;  
 this.age = age;  
 }  
  
 void hunt(){  
 System.*out*.println("We Are Hunting Now ");  
 }  
 static void staticHunt(){  
 System.*out*.println("StaticHunt: We Are Hunting Now ");  
  
 }  
}  
 Animal cat = new Animal("Cat",4);

cat.hunt();  
 Animal.staticHunt();

**Java Access Modifiers 8.4**

قبل البدء بشرح مفھوم Modifiers لاحظنا سابقا تكرار كلمة public كثيرا وھي احدى خيارات Modifiers في لغة Java، تنقسم Modifiers الى ثلاثة يمكن تعريفھا في الجدول التالي:

|  |  |
| --- | --- |
| Modifiers **نوع** | **التعریف** |
| public | يمكن الوصول الى العناصر الخاصة بالكلاس من اي مكان |
| private | لا يمكن الوصول الى العناصر الخاصة بالكلاس الا من داخل الكلاس |
| protected | يمكن الوصول الى العناصر الخاصة بالكلاس من الكلاسات الموجوة بنفس المجلد والكلاسات التي ترث منه |
| default | يمكن الوصول الى العناصر الخاصة بالكلاس من الكلاسات الموجوة بنفس المجلد |

لشرح الفكرة بشكل مبسط سوف نقوم بالتجربة على object من نوع Animal في الوقت الحالي جميع العناصر الخاصة بھذا class لم يتم تعريف Modifiers خاصة بھا اي انھا الان من نوع default ويتم الوصول لھا من الكلاسات الموجودة بنفس المجلد.

class Animal{  
 String name;  
 int age;

Animal(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
}  
  
public class Main {   
 public static void main(String[] args) {  
 Animal cat = new Animal("Cat",4);  
 System.*out*.println(cat.name);  
}  
}

في المثال الأعلى يمكننا طباعة خاصية name الخاصة ب cat لكن لو قمنا بتغيير نوع Modifiers الخاصة بعناصرھذا الكلاس فسوف يصبح من غير الممكن طباعتھا.

class Animal{  
 private String name;  
 private int age;  
  
 Animal(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
}  
  
  
public class Main {   
 public static void main(String[] args) {  
 Animal cat = new Animal("Cat",4);  
 System.*out*.println(cat.name);  
  
}  
}

وسوف نحصل على خطأ.

java: name has private access in Animal

ـــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــــ

**9.** **البرمجة الكائنية –** **Object Oriented Programming**

تنقسم OOP الى أربع مفاھیم أساسیة:

* التغليف Encapsulation
* الوراثة Inheritance
* تعدد الاشكال Polymorphism
* التجريد Abstraction

**9.1 التغليف Encapsulation**

معنى التغلیف ھو التأكد من إخفاء البیانات "الحساسة" عن المستخدمین. لتحقیق ذلك، یجب علیك:

* تعریف Attributes الخاصة بالكلاس على أنھا private
* توفیر طرق الحصول العامة وتعیینھا للوصول إلى قیمة المتغیر الخاص وتحدیثھا

تعرفنا في السابق على Modifiers ومن اجل تحقیق مفھوم Encapsulation سنحتاج الى جعل جمیع Attributes من نوع private كما في المثال التالي:

class Animal{  
 private String name;  
 private int age;  
  
 Animal(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
}

ثم سوف نقوم بتوفیر وسیلة للحصول على طریقة للوصول للعناصر الخاصة بھذا الكلاس وتسمى ھذه الطریقة ب Getters/Setters كما في المثال التالي:

class Animal{  
 private String name;  
 private int age;  
  
 Animal(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

* دوال getters
* قمنا بتعریف getName ووظیفتھا ھي الحصول على قیمة name
* قمنا بتعریف getAge ووظیفتھا ھي الحصول على قیمة age
* دوال setters
* قمنا بتعریف setName ووظیفتھا ھي تغییر قیمة name
* قمنا بتعریف setAge ووظیفتھا ھي تغییر قیمة age

من المھم ان یكون Modifiers الخاص بھذه الدوال من نوع public لنتمكن من الوصول لھا من اي موقع في مشروعنا.

**9.2 الوراثة Inheritance**

في Java، من الممكن أن ترث Attributes وmethods من كلاس إلى كلاس آخر.

مفھوم Inheritance یندرج تحته ثلاث مفاھیم اساسیة:

* الكلاس الفرعي subclass (الطفل) - الكلاس الذي یرث من كلاس اخر
* الكلاس الأساسي superclass (الأصل) - الكلاس الموروث منه
* لنرث صفات كلاس نستخدم كلمة extends

في المثال أدناه، ترث كلاس Car (الكلاس الفرعي) attributes وmethods من فئة Vehicle (الكلاس الأساسي):

class Vehicle {  
 void moving() {  
 System.*out*.println("The Vehicle is moving");  
 }  
}  
  
class Car extends Vehicle {  
}

كما في المثال الأعلى یمكننا استخدام جمیع methods الخاصة بكلاس Vehicle عند انشاء object من نوع Car:

public class Main {  
 public static void main(String[] args) {  
 Car car1 = new Car();  
 car1.moving();  
 }  
}

المخرجات:

The Vehicle is moving

رغم اننا لم نعرف اي method بإسم moving في الكلاس الخاص ب Car لكننا استطعنا استخدام دالة moving والسبب لأنھا موروثة من كلاس Vehicle ونستطیع استخدامھا.

**9.2.1 استخدام super**

عند وجود Constructor في الكلاس الأساسي وأردنا وراثة ھذا الكلاس فإننا سنحتاج الى استخدام كلمة super في Constructionالخاص بالكلاس الفرعي كما في المثال التالي:

class Vehicle{  
 String speed;  
 int yearOfManufacture;

public Vehicle(String speed, int yearOfManufacture) {  
 this.speed = speed;  
 this.yearOfManufacture = yearOfManufacture;  
 }  
  
 void moving(){  
 System.*out*.println("The Vehicle is moving");  
 }  
}  
  
class Car extends Vehicle{  
 String company;  
 public Car(String speed, int yearOfManufacture, String com pony) {  
 super(speed, yearOfManufacture);  
 this.company = compony;  
 }  
}

**9.3 تعدد الاشكال Polymorphism**

مفھوم تعدد الأشكال یحدث عندما یكون لدینا العدید من الكلاسات التي ترتبط ببعضھا البعض عن طریق الوراثة.

كما حددنا في الفصل السابق؛ تتیح لنا الوراثة وراثة Attributes وmethods من الكلاس الاساسي الى الكلاس الفرعي. یستخدم Polymorphism ھذه methods لأداء مھام مختلفة. ھذا یسمح لنا بأداء عمل واحد بطرق مختلفة.

على سبیل المثال، فكر في كلاس اساسي اسمه Animal و یحتوي method تسمىanimalSound(). یمكن أن تكون الكلاسات الفرعیة من كلاس Animal عبارة عن كلاس Cat و كلاس Dog ولدیھم أیضًا method خاصة بنفس الاسم تسمى :animalSound()

class Animal{  
 public void animalSound(){  
 System.*out*.println("The animal makes a sound");  
 }  
}  
  
class Dog extends Animal{  
 @Override  
 public void animalSound() {  
 System.*out*.println("The dog says: bow wow");  
 }  
}  
  
class Cat extends Animal{  
 @Override  
 public void animalSound() {  
 System.*out*.println("The cat says: meow meow");  
 }  
}

يتم استخدام @Override للإشارة ان هذه method موجودة مسبقاً في الكلاس الأساسي، ولكن تم تغييرها بداخل الكلاس الفرعي.

تكمن قوة مفھوم Polymorphism عند تعریف object لھذه الأنواع كما في المثال التالي:

public class Main {  
 public static void main(String[] args) {  
 Animal animal = new Animal();  
 Animal cat = new Cat();  
 Animal dog = new Dog();  
 animal.animalSound();  
 cat.animalSound();  
 dog.animalSound();  
 }  
}

قمنا بتعريف 3 متغيرات من نوع Animal ولكن القيم لبمعطى لها تختلف:

* المتغیر الأول تم اسناد له قیمة من نوع Animal
* المتغير الثاني تم اسناد له قيمة من نوع Cat
* المتغير الثالث تم اسناد قيمة له من نوع Dog

وھنا تم استخدام مفھوم Polymorphism بحیث انه نوع Animal یقبل اشكال متعددة للقیم ولكن یجب ان تكون كل القیم عبارة عن كلاسات فرعیة لكلاس نوع المتغیر.

|  |
| --- |
| The animal makes a sound |
| The dog says: meow meow |
| The dog says: bow wow |

عند القیام بمناداة دالة animalSound() سوف نلاحظ اختلاف القیمة المطبوعة والسبب یرجع الى نوع القیمة المسندة لكل متغیر.

**9.4 التجريد Abstraction**

**9.4.1 مفھوم Abstract**

ھو اخفاء البیانات الخاصة بشكل كامل واظھار method فقط، یتم ذلك عن طریق استخدام احدى هذين المفهومين Interface او abstract classes.

مفھوم abstract class: ھو كلاس لا یمكن استخدامه لإنشاء object (للوصول إلیه، یجب أن یتم وراثته الى كلاسات فرعیة).

مفھوم abstract methods: ھي method فارغة یتم تعریفھا في abstract class ولكن یجب كتابة الكود الخاص فیھا في الكلاس الفرعي الذي یرث abstract class.

في المثال التالي تم تعریف abstract class بإسم Animal وتم تعریف abstract method بإسم

moving ونلاحظ انھا فارغة ولكن تم تعریف ھذه method في الكلاسات التي ترث من abstract class

Animl.

abstract class Animal {  
 public void animalSound() {  
 System.*out*.println("The animal makes a sound");  
 }  
 abstract void moving();  
}  
  
class Dog extends Animal {  
 @Override  
 void moving() {  
 System.*out*.println("The Dog is Moving");  
 }  
}  
class Cat extends Animal {  
 @Override  
 void moving() {  
 System.*out*.println("The Cat is Moving");  
 }  
}

یتم استخدام abstract class بنفس الطریقة الموجودة في Polymorphism ولكن یجب ان ننتبه انه لا یمكن

تعریفه بإستخدام نوع abstract class.

public class Main {  
 public static void main(String[] args) {  
 Animal cat = new Cat();  
 Animal dog = new Dog();  
 cat.moving();  
 dog.moving();  
 }  
}

المخرج

|  |
| --- |
| The Cat is Moving |
| The Dog is Moving |

**9.4.1 مفھوم interface**

الفرق الأساسي بين interface وabstract class هي أن الدوال في interface تكون مجردة كليًا abstract ولا يمكن أن تحوي على أي implementation أي أنها لا تقوم بتنفيذ شيء حتى يرثها الابن ويقوم بعمل override ،

في حين abstract class يمكن أن يحوي على دوال تقوم بتنفيذ سلوك افتراضي ثم يأتي الابن إما أن ينفذ هذا السلوك نفسه أو سلوك آخر أيضًا بعملية override.

interface Animal{  
 void moving();  
}  
  
class Dog implements Animal{  
 @Override  
 public void moving() {  
 System.*out*.println("The Dog is Moving");  
 }  
}  
  
class Cat implements Animal{  
 @Override  
 public void moving() {  
 System.*out*.println("The Cat is Moving");  
 }  
}

يمكن استخدام interface بنفس الطريقة الموجودة في Polymorphism ولكن يجب ان ننتبه ان لا يمكن تعريف كائن باستخدام نوع interface.

public class Main {  
 public static void main(String[] args) {  
// Animal animal = new Animal(); this Wrong You can't declare it using the interface class type  
 Animal cat = new Cat();  
 Animal dog = new Dog();  
 cat.moving();  
 dog.moving();  
 }  
}

المخرجات:

|  |
| --- |
| The Cat is Moving |
| The Dog is Moving |