2017

java

تعلم لغة جافا من الصفر الى الاحتراف

AHMED IBRAHIM

فهرس المحتويات

المقدمة

كتبت هذا الكتاب ليكون كمرجع مختصر للرجوع اليه وقت الحاجة لأوامر وتعليمات لغة جافا واعتمدت بشكل اساسي على كتاب Java a Beginner Guide للكاتب Herbert Schildt ، يبدأ الكتاب في لغة جافا من الصفر الى مستوى متوسط او متقدم، ربما لن تجد شرح تفصيلي عن اللغة وانما فقط توضيح لمبدأ عمل كل جزء من اجزاء اللغة وذلك لكي لا يكون حجم الكتاب كبير وممل عند القراءة، سيستفاد من هذا الكتاب من لديه معلومات (ولو قليلة) عن لغة جافا اكثر مِن مَن لم يستبق له التعامل مع هذه اللغة او اي لغة برمجة من قبل.

لم الكتب الكتاب بهدف نشره وانما كتبته لنفسي لارجع للغة عند نسيان شيء منها، لذلك فان اغلب الامثلة والشروح في الكتاب هي للتذكير بما نعرفه مسبقا عن اللغة (ويمكن ان يتعلم منها من لم يكن يعرف مسبقا) ولكني نشرت الكتاب ليساعد ولو قليلا من يريد تعلم هذه اللغة خصوصا وان الكتب العربية قليلة، ومن لديه سؤال او استفسار يمكنه التواصل معي عبر حسابي على الفيسبوك :

https://www.facebook.com/ah.ib.93

هذه هي الطبعة الاولى من الكتاب نشرت سنة 2017 وربما ساضيف تفاصيل واوضح نقاط اخرى وانشر الكتاب كطبعة ثانية.

هذا الكتاب مجاني اي يمكن لاي شخص قراءته واعادة نشره، وبنفس مبدأ هذا الكتاب كتبت بعض الكتب لتعليم لغات برمجية اخرى مثل (HTML, CSS, JavaScript, jQuery, PHP, Quick Basic) يمكنكم قراءة وتحميل كل هذه الكتب من هذا الرابط :

https://drive.google.com/open?id=0B2aI_a6mphOUQi1jdzlYSFhITWs

احمد ابراهيم

الفصل الأول (المتغيرات ومعلومات عامة)

لغة جافا تبدأ بالكلمة المحجوزة class ثم يليها اسم البرنامج الذي اختاره المبرمج ويجب ان يحفظ الملف بنفس الاسم , ويحتوي الـ class على الدالة الرئيسية main وتكتب هكذا :

```
public static void main (String args[ ])
```

ويبدأ تنفيذ البرنامج من هذه الجملة، مثال :

```
class NAME {
public static void main(String args[ ]) {
// البرنامج
} }
```

انواع البيانات في جافا

byte	يأخذ عدد صحيح بين موجب 127 وسالب 128	ياخذ مساحة تخزين byte 1
short	ياخذ عدد صحيح بين موجب 32,767 وسالب 32,768	ياخذ مساحة تخزين byte 2
int	ياخذ عدد صحيح بين موجب 2,147,483,647 وسالب نفس الرقم	ياخذ مساحة تخزين byte 4
long	ياخذ عدد صحيح بين موجب 9,223,372,036,854,775,807 وسالب نفس الرقم	ياخذ مساحة تخزين byte 8
float	ياخذ كسر عشري صغير	ياخذ مساحة تخزين byte 4
double	ياخذ كسر عشري كبير	ياخذ مساحة تخزين byte 8
boolean	ياخذ قيم منطقية اما true او false	
char	ياخذ حرف واحد ويجب ان يوضع بين علامة تنصيص مفردة	
String	ياخذ نصوص ويجب وضعها بين علامة تنصيص مزدوجة , وهو في الحقيقة عبارة عن class معرف مسبقا في لغة جافا	

المتغيرات

* يمكن ان يبدأ اسم المتغير بفاصلة سفلية او علامة الدولار او حرف لاكن لايمكن ان يبدأ برقم .

لتعريف المتغير نكتب نوع المتغير ثم اسمه ويمكن ان نضيف له قيمة بعد المساواة او نتركه بدون قيمة وتضاف القيمة في وقت اخر , مثال

```
int name = 5 ;
```

يمكن تعريف عدة متغيرات في جملة واحدة اذا كانت من نفس النوع عن طريق الفصل بينها بالفارزة , مثال

```
int x , y , z ;
int X = 4 , B = 7 ;
```

* المتغير من نوع char ياخذ كود ASCII لذا يمكن ان نضع فيه حرف او رقم اسكي الذي يمثل الحرف او يمكن زيادته او انقاصه ليأخذ الحرف الذي قبله او بعده، وكذلك يمكن اجراء العمليات الاخرى مثل اكبر او اصغر من للمقارنة بين حرفين، مثال :

```
char ch = 'x';
ch++          // قيمة المتغير ستكون y
ch = 90       // قيمة المتغير ستكون z
```

* لطباعة قيمة نستخدم الكود التالي :

```
System.out.print ("Hello world");
```

حيث ان الدالة print تقوم بالطباعة بنفس السطر والدالة println تقوم بالطباعة بسطر منفصل .

* تستخدم اشارة الزائد + لربط قيم المتغيرات سواء كانت نصية او رقمية , مثال :

```
String  X = "Hello" + "World";
System.out.print ("Hello world" + x );
```

التحويل بين انواع المتغيرات

يمكن تحويل المتغيرات بشكل اوتوماتيكي من نوع الى اخر اذا اسندناها لبعض وكان النوعين متوافقين من ناحية القيمة ومن ناحية الحجم (حيث يمكن تحويل اصغير الى كبير ولا يمكن العكس) مثال :

```
int X = 10 ;
float Y ;
Y = X ;
```

هنا سيتم تحويل X من int الى float بشكل اوتوماتيكي

* كذلك يمكن التحويل بشكل يدوي بين انواع المتغيرات حتى لو كانت غير متوافقة مثل الاول حروف والاخر ارقام (والتي لا يمكن تحويلها اوتوماتيكيا) وذلك من خلال وضع اسم النوع الذي نريد التحويل اليه بين قوسين، مثال :

```
int i = 88 ;
char ch ;
ch = ( char ) i ;
```

هنا ستكون قيمة ch هي الحرف X لان الرقم 88 هو رمز الاسكي للحرف X .

اذا حولنا متغير من القيمة float الى int فانه سيحذف كل الارقام التي بعد الفارزة

العمليات الرياضية

الجمع	+
الطرح	-
الضرب	*
القسمة	/
باقي القسمة	%

* يمكن استخدام الاضافة بواحد ++ او الانقاص بوحد -- مع اي متغير وكذلك يمكن اضافتها قبل او بعد المتغير، مثال :

```
X = X++
```

```
X = ++X
```

وفي كلا الحالتين سيضيف واحد الى قيمة X الاولية، لكن الفرق هو ان البرنامج في الحالة الاولى سيضع قيمة X ثم يضيف اما في الحالة الثانية سيضيف القيمة الى قيمة X ، المثال التالي سيوضح الفرق :

```
X = 10 ;
Y = X++ ;
```

هنا ستكون قيمة X 11 وقيمة Y 10 ، اما في هذا المثال :

```
X = 10 ;
Y = ++x ;
```

هنا ستكون قيمة X 11 وقيمة Y 11 ايضا .

* يمكن استخدام علامة المساوات مع العمليات الرياضية، مثال :

```
X = X + 10 ;
```

او يمكن كتابتها بـهذا الشكل :

```
X += 10 ;
```

وبنفس الطريقة يمكن ان تحدث مع جميع انواع العمليات الحسابية والمنطقية

+=	-=	*=	/=
%=	&=	|=	^=

العبارات المنطقية

Short and	&&
Short or	||
Not	!
Xor	^
And	&
Or	|

الـ xor تعود بالنتيجة true اذا كان طرفي المقارنة مختلفين احدهم صح والاخر خطأ اما اذا كان الطرفين متشابهين سواء صح ام خطأ ستعود بالنتيجة false .

الـ && و || هما تماما نفس عمل الـ & و | لكن الفرق الوحيد هو ان الـ && و || هما يختصران الوقت في تنفيذ البرنامج لانهما لا يفحصان القيمة الاخرى في المقارنة إلا اذا تطلب الامر ذلك لانه يمكن معرفة النتيجة احيانا من خلال معرفة عامل المقارنه الاول فمثلا في الـ And اذا كان الاول خطأ ستكون النتيجة خطأ بغض النظر عن المعامل الثاني، وفي الـ Or اذا كان المعامل الاول صح ستكون النتيجة صح بغض النظر عن المعامل الثاني .

متعاقبات الهروب Escape Sequences

تستخدم للتعبير عن اشارات غير قابلة للاظهار

“\n”	تستخدم لطباعة سطر فارغ
“\r”	تستخدم للعودة الى بداية السطر والطباعة من هناك
“\t”	تطبع مسافة فارغة بمقدار 8 خانات افقيا
“\b”	ترجع بمقدار خانة واحدة
“\f”	صفحة جديدة
“\\”	تطبع خط مائل
“\’”	تطبع علامة تنصيص مفردة
“\””	تطبع علامة تنصيص مزدوجة

الفصل الثاني (جمل التحكم والتكرار)

عبارات المقارنة

اكبر من	>
اصغر من	<
يساوي	==
لا يساوي	!=
اكبر من او يساوي	>=
اصغر من او يساوي	<=

* يمكن استخدام عبارات المقارنة في جمل الطباعة لتعطينا قيمة من نوع true او false ، مثال :

```
System.out.println("10 > 9 is " + (10 > 9));    // true ستطبع القمية
```

عبارة الشرط if و if else

والصيغة العامة لها بهذا الشكل

```
if ( Condetion ) {
Statement1 ;
} else {
Statement2 ; }
```

* يمكن التعبير عن if else بطريقة اخرى والصيغة العامة لها تكون بهذا الشكل :

```
variable = ( Condition ) ? Number1 : Number2 ;
```

مثال :

```
int y = 2  ;
int X = ( y > 8 ) ? 9 : 4 ;
```

حيث انه في هذا المثال اذا كان الشرط صحيح ستكون قيمة المتغير X هي 9 اما اذا كان خاطئ تكون قيمته 4

ويمكن عمل شروط بشكل متداخل بعدد ما نريد , مثال :

```
int y = 2  ;
int X = ( y > 8 ) ? ( ( y == 6 ) ? 2 : 1 ) : 4 ;
```

عبارة الشرط switch

والصيغة العامة لها :

```
switch (variable){
case value1 :
statement ;
break ;
case value2 :
statement ;
break ;
default :
statement ; }
```

حيث ان variable يمثل اسم المتغير المطلوب اجراء الاختبار على قيمته , ويشترط ان يكون من النوع int او char, value1 و value2 عبارة عن قيم يتم مقارنتها مع قيم المتغير فاذا تطابقت قيمة المتغير مع قيمة احد هذه القيم ستنفذ الجمل التي بعدها الى حد عبارة break , اما في حال عدم تطابق قيمة المتغير مع اي من القيم الموضوعة ستنفذ الجمل بعد عبارة default , مثال :

```
int X = 4 ;
switch(X){
case 5 :
System.out.println(X);
break;
case 4 :
 System.out.println(X);
```

```
break;
default :
 System.out.println("None");  }
```

* عبارة break هي اختيارية على الرغم من انها دائما ما توضع، لكن في حال عدم وضعها عندما يتطابق ما نفحصه مع عبارة case فانه سينتقل للجملة التي بعدها وينفذها الى نهاية الـ switch او الى ان يجد break في مكان اخر.

* يمكن وضع case فارغة بدون جملة، مثال :

```
switch(i) {
case 1:
case 2:
case 3: System.out.println("i is 1, 2 or 3");
break;
case 4: System.out.println("i is 4");
break;
}
```

هنا في هذا المثال سيطبع الجملة الاولى اذا كانت i 1 او 2 او 3 اما اذا كانت i 4 فسيطبع الجملة الثانية .
يمكن ان تكون switch في داخل switch اخرى عند كتابتها في الجملة case ، مثال :

```
switch(ch1) {
case 'A': System.out.println(" This A is part of outer switch. ");
switch(ch2) {
case 'A':
System.out.println(" This A is part of inner switch ");
break;
case 'B': // ...
} // end of inner switch
break;
case 'B': // ...
```

حلقة التكرار for

مثال :

```
for( int i = 1;  i < 6;  i++ ) {
System.out.println("The Value is : " + i);    }
```

* يمكن وضع اكثر من عداد في حلقة التكرار ونفصل بينهم بفارزة، مثال :

```
int i, j;
for( i=0,  j=10;  i < j;  i++,  j-- )
System.out.println("i and j: " + i + " " + j);
```

* يمكن للشرط ان يكون غير مرتبط بالعداد في حلقة التكرار، لاحظ حلقة التكرار هذه التي ستبقى تتكرر الى ان يدخل المستخدم الحرف S :

```
class ForTest {
public static void main(String args[]) throws java.io.IOException {
int i;
System.out.println("Press S to stop.");
for( i = 0;  (char) System.in.read( ) != 'S';  i++ )
System.out.println("Pass #" + i);
} }
```

* يمكن عدم وضع القيمة الاولية للعداد او عدم وضع تكرار العداد او كلاهما في داخل قوسي الحلقة ، مثال بعدم وضع كلاهما :

```
int i = 0;
for( ; i < 10; ) {
System.out.println("Pass #" + i);
i++; }
```

* يمكن لحلقة التكرار ان تكون غير نهائية بعدم وضع شرط في داخلها وغالبا هذا النوع من الحلقات يتم الخروج منه من خلال عبارة break ، مثال :

```
for( ; ; ) {
//...
}
```

* يمكن ان تكون حلقة التكرار for بدون جسد اي انها لا تنفذ اي جملة فقط تتكرر بعدد من المرات ، وهذه قد تكون مفيدة اذا اردنا ان نضاعف عدد معين مثلا، مثال :

```
int i;
int sum = 0;
for( i = 1; i <= 5; sum += i++ ) ;
System.out.println( sum );
```

سيكون ناتج البرنامج 15

* هناك نوع اخر من حلقة التكرار for وهو يستخدم فقط مع المصفوفات والغرض منه هو تسهيل العمل مع المصفوفات عندما نريد ان نجمع او نعرف قيمة كل عنصر في المصفوفة، والصيغة العامة له هي :

```
for(type itr-var : ArrayName) {
// statements
}
```

ويجب ان يكون نوع المتغير itr-var نفس نوع المصفوفة لانه سيحمل في كل حلقة تكرار عنصر من عناصر المصفوفة يبدأ من العنصر الاول انتهاءا بالعنصر الاخير، مثال لجمع عناصر المصفوفة :

```
int nums[ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

وبواسطة حلقتين متداخلتين يمكن ان نتعامل مع مصفوفة ثنائية البعد، مثال :

```
int num[ ] [ ] = {{1, 2, 3}, {4, 5, 6}};
int sum = 0;
for(int x[ ] : num) {
for(int y : x) {
sum += y;
} }
```

عبارة التكرار while

الصيغة العامة لها هي :

```
while ( condition ) {
statement ;
}
```

مثال لطباعة الحروف الابجدية :

```
char ch = 'a';
while(ch <= 'z') {
System.out.print(ch);
ch++; }
```

حلقة التكرار do while

الصيغة العامة لها هي:

```
do {
statement ;
} while ( condition ) ;
```

العبارة break

تستخدم هذه العبارة للخروج من حلقة التكرار او من البلوك { الاقواس المعقوفه }، مثال :

```
int t = 0;
while( t < 100 ) {
if(t == 10) break;
System.out.println( t );
t++; }
```

كذلك يمكن استخدام break للخروج من بلوك معين (اعطيناه اسم مسبقا) ، مثال :

```
int i;
for(i=1; i<4; i++) {
one: {
two: {
three: {
```

```
System.out.println("\ni is " + i);
if(i==1) break one;
if(i==2) break two;
if(i==3) break three;
// this is never reached
System.out.println("won't print");
}
System.out.println("After block three.");
}
System.out.println("After block two.");
}
System.out.println("After block one.");
}
System.out.println("After for.");
```

ستكون نتيجة البرنامج كالتالي :

```
i is 1
After block one.
i is 2
After block two.
After block one.
i is 3
After block three.
After block two.
After block one.
After for.
```

من المهم ان ننتبه للمكان الذي نضع فيه التسمية، مثال :

```
int x=0, y=0;
// here, put label before for statement.
stop1: for(x=0; x < 5; x++) {
for(y = 0; y < 5; y++) {
```

```
if(y == 2) break stop1;
System.out.println("x and y: " + x + " " + y);
} }
System.out.println();
// now, put label immediately before {
for(x=0; x < 5; x++)
stop2: {
for(y = 0; y < 5; y++) {
if(y == 2) break stop2;
System.out.println("x and y: " + x + " " + y);
}  }
```

هنا في الـ stop1 عنما يخرج سيخرج من حلقة التكرار نهائيا ولن يعود لها اما في الـ stop2 عندما يخرج من حلقة التكرار سيعود لها ليتحقق من شرط التكرار اذا متحقق سيعمل تكرار اخر للحلقة لذلك ستكون مخرجات البرنامج بهذا الشكل :

```
x and y: 0 0
x and y: 0 1
x and y: 0 0
x and y: 0 1
x and y: 1 0
x and y: 1 1
x and y: 2 0
x and y: 2 1
x and y: 3 0
x and y: 3 1
x and y: 4 0
x and y: 4 1
```

العبارة continue

تستخدم هذه العبارة للخروج من تكرار معين وليس من حلقة التكرار بالكامل، مثال :

```
int i;
for(i = 0; i<=100; i++) {
if((i%2) != 0) continue;
System.out.println(i); }
```

في هذا المثال سيطبع فقط الارقام الزوجية

كذلك يمكن استخدام العبارة continue للقفز الى بلوك مسمى مسبقا، مثال :

```
outerloop: for(int i=1; i < 10; i++) {
System.out.print("\nOuter loop pass " + i + ", Inner loop: ");
for(int j = 1; j < 10; j++) {
if(j == 5) continue outerloop;
System.out.print(j); }
}
```

الفصل الثالث (المصفوفات)

المصفوفة هي عبارة عن اوبجكت، وهناك عدة انواع من المصفوفات :

المصفوفات احادية البعد

الصيغة العامة لتعريف مصفوفة احادية البعد هي :

```
DataType[ ] ArrayName = new DataType[ number ] ;
```

او

```
DataType  ArrayName[ ] = new DataType[ number ] ;
```

مثال :

```
int array [ ] = new int [ 5 ] ;
```

ويمكن وضع قيم ابتدائية لعناصر المصفوفة بهذا الشكل :

```
array[ 0 ] = 1 ;
array[ 1 ] = 2 ;
array[ 2 ] = 3 ;
array[ 3 ] = 4 ;
array[ 4 ] = 5 ;
```

او يمكن وضع القيم للمصفوفة عند بداية التعريف بهذا الشكل :

```
int array[ ] = { 1, 2, 3, 4, 5 };
```

لطباعة عنصر من المصفوفة

```
System.out.print( array[ 3 ] );
```

المصفوفات ثنائية البعد

يمكن تعريف مصفوفة ثنائية البعد بهذا الشكل

```
int arr2[ ] [ ] = new int [ 2 ] [ 3 ] ;
```

يمكن تعريف ووضع قيمة اولية لمصفوفة ثنائية البعد بهذا الشكل :

```
int arr2[ ][ ] = new int [ ] [ ] { { 1,2,3} , {4,5,6} } ;
```

يمكن طباعة عنصر محدد من مصفوفة ثنائية البعد بهذا الشكل :

```
System.out.print( arr2[0][2] );
```

* يمكن ان تكون المصفوفة ثنائية البعد متعددة الحجوم بالبعد الثاني، حيث يمكننا ان نسند الحجم فقط للبعد الاول اما الثاني فنسد له الاحجام في وقت اخر، مثال :

```
int table[ ][ ] = new int[3][ ];
table[0] = new int[4];
table[1] = new int[2];
table[2] = new int[9];
```

المصفوفات متعددة الابعاد

الصيغة العامة لتعريفها هي :

```
type ArrayName[ ][ ]...[ ] = new type[number 1][ number 2]...[ numberN];
```

ويمكن وضع القيم لهذه المصفوفة بهذا الشكل :

```
type-specifier array_name[ ] [ ] = {
{ val, val, val, ..., val },
{ val, val, val, ..., val },
...
{ val, val, val, ..., val }
};
```

* يمكن ان نسند مصفوفة الى اخرى من خلال المساواة وهنا ستنسخ عناصر المصفوفة الاولى للثانية وسيكون اسم كلا المصفوفتين يشر الى نفس المصفوفة اي وكأنه تولدت لدينا مصفوفة تحمل اسمين واي تغير يحدث على احداهما سيطبق على الاخرى .

```
int num1 = {1, 2, 3};
int num2 = {4, 5, 6};
num2 = num1;
for(int i=0; i < 3; i++)
System.out.print(num2[i] + "  ");
System.out.println();
num2[2] = 9;
for(int i=0; i < 3; i++)
System.out.print(num1[i] + "  ");
```

هنا سيكون الناتج كالتالي :

```
1  2  3
1  2  9
```

* يمكن ان نكون مصفوفة من الاوبجكتات حيث سيكون كل عنصر من المصفوفة عبارة عن اوبجكت، بهذا الشكل :

```
ClassName  Ob[ ] = new ClassName[3] ;
Ob[0] = new ClassName( );
Ob[1] = new ClassName(3, 19 );
Ob[0].x = 43;
Ob[1].Method( );
```

او يمكن كتابة مصفوفة من الاوبجكتات واسناد القيم اليها مباشرتا كما في هذا المثال :

```
private ClassName Ob[ ] = new ClassName[ ] {
new ClassName(2),
new ClassName(9),
new ClassName( ),
};
int X = Ob[0].Method( ) ;
```

الدالة length

تستخدم هذه الدالة مع المصفوفات لتعود لنا برقم يمثل عدد عناصر المصفوفة , مثال :

```
int X = array.length ;
```

ولحساب عدد عناصر مصفوفة داخلية في مصفوفة متعددة الابعاد نضع رقم المصفوفة بهذا الشكل :

```
int X = array[2].length ;
```

الفصل الرابع (The Classes)

يتم عمل الكلاس بهذه الصيغة العامة :

```
class ClassName {
// the variables
// the methods
}
```

وللاستفادة من الكلاس لابد من استخدام الاوبجكت object للوصول الى محتويات الكلاس، ويتم عمل الاوبجكت بهذه الصيغة العامة :

```
ClassName ObjectName = new ClassNeme ( ) ;
```

ويمكن انشاء اكثر من اوبجكت للكلاس الواحد ونتعامل مع كل منهم على انفراد.

مثال :

```
class Vehicle {
int passengers ;
int fuelcap ;
 int mpg ;
}
```

في هذا المثال عملنا كلاس اسمه Vehicle ووضعنا فيه ثلاثة متغيرات، والان سنعمل اوجكت اسمه minivan لهذا الكلاس

```
Vehicle minivan = new Vehicle ( ) ;
```

وللوصول الى متغيرات الكلاس من الاوجكت نكتب اسم الاوبجكت بعده نقطة بعده اسم المتغير وهنا يمكن ان نعطيه القيمة التي نريد :

```
minivan.fuelcap = 16;
```

والان سنكتب المثال السابق لكن بشكل متكامل لتتضح الصورة وفي نفس الوقت سنعمل 2 objects للكلاس :

```
class Vehicle {
int passengers ;
int fuelcap ;
int mpg ;
}
class TwoVehicles {
public static void main(String args[]) {
Vehicle minivan = new Vehicle();
Vehicle sportscar = new Vehicle();
int range1, range2;
// assign values to fields in minivan
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;
// assign values to fields in sportscar
sportscar.passengers = 2;
sportscar.fuelcap = 14;
sportscar.mpg = 12;
// compute the ranges
range1 = minivan.fuelcap * minivan.mpg;
range2 = sportscar.fuelcap * sportscar.mpg;
System.out.println("Minivan can carry " + minivan.passengers +
" with a range of " + range1);
System.out.println("Sportscar can carry " + sportscar.passengers +
" with a range of " + range2);
} }
```

ستكون مخرجات البرنامج بهذا الشكل :

```
Minivan can carry 7 with a range of 336
Sportscar can carry 2 with a range of 168
```

Methods

الصيغة العامة لها هي :

```
type Name ( parameters ) {
// body of method
}
```

type يمثل نوع البيانات التي تعيدها هذه الدالة واذا لم تكن تعيد اي بيانات يكون النوع void ، والبارامترات (وهي عبارة عن متغيرات او مصفوفة او اوبجكت) التي نكتبها بين قوسي الدالة نفصل بينها بفارزة اذا كانت اكثر من واحدة او يمكن ان لا تحتوي الدالة على اي باراميتر فيكون ما بين القوسين فارغ.
ويمكن الوصول الى الدالة من كلاس اخر عن طريق الاوبجكت كأنها متغير لكن فقط نضع قوسين امامها، مثال :

```
class Vehicle {
int p ;
void  range ( int f ) {
System.out.println( p * f );
} }
class AddMeth {
public static void main ( String args[ ] ) {
Vehicle minivan = new Vehicle( );
minivan.p = 5 ;
minivan.range( 6 );
} }
```

عند تنفيذ البرنامج سيطبع القيمة 30

* يمكن للكلاس ان يحتوي على اكثر من دالة تحمل نفس الاسم لكن بشرط ان تحمل بارامترات مختلفة من حيث العدد او النوع وعند استدعائها مفسر لغة جافا سيميز بينها من خلال البارامترات الممررة لها .

* مثال لدالة بارامتراتها عبارة عن اوبجكت :

```
class Block {
```

```
int a, b, c;
int volume;
Block(int i, int j, int k) {
a = i;
b = j;
c = k;
volume = a * b * c;
}
boolean sameBlock(Block ob) {
if((ob.a == a) & (ob.b == b) & (ob.c == c) & (ob.volume == volume)) return true;
else return false;
} }
class PassOb {
public static void main(String args[]) {
Block ob1 = new Block(10, 2, 5);
Block ob2 = new Block(10, 2, 5);
System.out.println("ob1 same as ob2: " + ob1.sameBlock(ob2));
} }
```

* يمكن استدعاء الدالة من داخل نفس الدالة وبذلك ستعيد نفسها وتتكرر الى ان تكتمل العملية، مثال :

```
int factR(int n) {
int result;
if(n==1) return 1;
result = factR(n-1) * n;
return result;
}
```

يمكن كتابة نفس المثال السابق لكن بطريقة اخرى (اي نستخدم حلقة تكرار بدلا من استدعاء الدالة من داخل الدالة :

```
int factI(int n) {
int t, result;
result = 1;
```

```
for(t=1; t <= n; t++) result *= t;
return result;
}
```

العبارة return

تستخدم هذه العبارة في داخل الدالة لسببين الاول ان تعود لنا بقيمة (يمكن ان تكون القيمة متغير او مصفوفة او اوبجكت) عند استدعائها وعندها سيكون نوع الدالة نفس نوع القيمة التي تعيدها لنا هذه العبارة والصيغة العامة لها هي :

```
return value;
```

اما السبب الثاني لاستخدامها هو للخروج من الدالة عند الوصول اليها وغالبا ما تكتب داخل شرط معين وهي تكتب لوحدها وبعدها فارزة منقوطة، وفي هذه الحالة يمكن استخدام اكثر من عبارة return في داخل الدالة الواحدة وهنا اذا لم تكن الدالة تعيد اي قيمة ستبقى من النوع void .

مثال لدالة تعيد اوبجكت :

```
class Err {
String msg;
int severity;
Err(String m, int s) {
msg = m;
severity = s;
} }
class ErrorInfo {
String msgs[] = {"Output Error", "Input Error", "Disk Full", "Index Out-Of-Bounds"};
int howbad[] = { 3, 3, 2, 4 };
Err getErrorInfo(int i) {
if(i >= 0 & i < msgs.length)
return new Err(msgs[i], howbad[i]);
else
return new Err("Invalid Error Code", 0);
```

```
} }
class ErrInfo {
public static void main(String args[]) {
ErrorInfo err = new ErrorInfo();
Err e;
e = err.getErrorInfo(2);
System.out.println(e.msg + " severity: " + e.severity);
} }
```

ستكون نتيجة البرنامج Disk Full severity: 2

مثال لدالة تعيد مصفوفة :

```
class array {
int[ ] getNextArray(int n) {
int[ ] vals = new int[n];
for(int i=0; i < n; i++) vals[i] = i;
return vals;
} }
```

Constructors

هي عبارة عن دالة Method ، عندما ننشأ اوبجكت ستولد عندنا Constructor بشكل اوتوماتيكي للكلاس ولكن لنستفاد منه علينا ان ننشأه بداخل الكلاس ويجب ان يحمل نفس اسم الكلاس الموجود داخله، وهو لا يعيد اي قيمة ، الصيغة العامة له هي :

```
ClassName ( parameters ){
// the body of Constructor
}
```

يمكن وضع بارامترات بين قوسيه او عدم وضعها وهذه البارامترات هي عبارة عن متغيرات او مصفوفة او اوبجكت، الفائدة من الـ Constructor هو انه يسهل علينا العمل عندما نستدعي الكلاس من الاوبجكت، مثال :

```
class MyClass {
```

```
int x;
MyClass(int i) {
x = i;
} }
class ParmConsDemo {
public static void main(String args[]) {
MyClass  t1 = new MyClass(10);
MyClass  t2 = new MyClass(88);
System.out.println(t1.x + " " + t2.x);
} }
```

سيكون ناتج البرنامج هو : 88 10

* يمكن للكلاس ان يحتوي على اكثر من Constructor يحمل نفس الاسم لكل بشرط ان يحمل بارامترات مختلفة من حيث العدد او النوع وعند استدعائه مفسر لغة جافا سيميز بينهم من خلال البارامترات الممررة لهم .

كلاسات متداخلة

يمكن عمل كلاس داخل كلاس اخر وعندها سيكون نطاق الكلاس الداخلي هو داخل الكلاس الخارجي اي لايمكن التعرف عليه من كلاس اخر، مثال :

```
class Outer {
int nums[];
Outer(int n[]) {
nums = n;
}
void analyze() {
Inner inOb = new Inner();
System.out.println("Minimum: " + inOb.min());
}
class Inner {
int min() {
```

```
int m = nums[0];
for(int i=1; i < nums.length; i++)
if(nums[i] < m) m = nums[i];
return m;
} } }
class NestedClassDemo {
public static void main(String args[]) {
int x[] = { 3, 2, 1, 5, 6, 9, 7, 8 };
Outer outOb = new Outer(x);
outOb.analyze();
} }
```

Varargs

هذه عبارة عن باراميتر يمرر للدالة وهو يمثل مصفوفة من المتغرات، فائدته تكمن في انه يتيح لنا الامكانية في وضع باراميتر واحد او اكثر او عدم وضع اي باراميتر بين قوسي الدالة عند استدعائها، ويكتب بهذا الشكل :

```
Type … name
int … x
```

مثال :

```
class VarArgs {
static void vaTest(int ... v) {
for(int i=0; i < v.length; i++)
System.out.println(" arg " + i + ": " + v[i]);
}
public static void main(String args[]){
vaTest(10);
vaTest(1, 2, 3);
vaTest();
} }
```

* يمكن ان نضع في الدالة بارمترات عادية بجانب الباراميتر vararg لكن يشترط ان الباراميتر vararg يكون في الاخير ، مثال :

```
int doIt(int a, int b, double c, int ... vals) {
// the statements
}
```

فهنا عند استدعاء الدالة الثلاث بارامترات الاولى يجب وضعها اما الباراميتر الاخير فنحن مخيرون بوضعه.
* لاحظ ان الدالة لا يمكن ان تحتوي على اكثر من باراميتر واحد من النوع vararg .

this

تستخدم this للاشارة الى المتغيرات او الدوال الموجودة في داخل الكلاس المستخدمة فيه وذلك للتمييز بينها وبين المتغيرات من خارج الكلاس ان وجدت، وفي كثير من الاحيان كتابتها وعدم كتابتها لا يفرق شيء لكن في بعض الاحيان تكون مهمة، ونفصل بينها وبين اسم المتغير او الدالة بنقطة :

```
this.X = 10 ;
```

انواع البيانات الخاصة والعامة

عند تعريف اي متغير او دالة في داخل كلاس يجب ان نبين نوعه (نسبقه بكلمة النوع) هل هو public اي عام او protected اي محمي او افتراضي اي لا نسبقة باي كلمة او private اي خاص، العام نقصد به انه يمكن الوصول له والتعامل معه من اي مكان في البرنامج سواء خارج الكلاس او حتى خارج الحزمة المعرف داخلهما، اما المحمي فيمكن الوصول اليه من اي مكان داخل الحزمة بالاضافة الى الكلاسات الوارثة المرتبطة به حتى من خارج الحزمة، اما الافراضي فيمكن الوصول اليه من خارج الكلاس لكن داخل الحزمة وليس خارجها ولا يمكن الوصول اليه من الكلاسات الوارثة من خارج الحزمة، اما اخاص فلا يمكن الوصول اليه الا من داخل نفس الكلاس فقط، مثال :

```
private  void  Meth( ){
// The body of Method
}
```

static

يمكن للمتغيرات او الدوال في داخل الكلاس ان تكون من انوع static ، هذه الكلمة المفتاحية ستتيح الامكانية للوصول الى الدالة او المتغير من خارج الكلاس بدون اوبجكت ، كل ما نحتاجه هو ان نذكر اسم الكلاس بعده نقطة ثم اسم المتغير او الدالة، بهذا الشكل :

```
ClassName.Variable  = Value ;
ClassName.Method( ) ;
```

مثال :

```
class StaticDemo {
int x;
static int y;
int sum() {
return x + y;
} }
class SDemo {
public static void main(String args[]) {
StaticDemo ob1 = new StaticDemo();
ob1.x = 10;
StaticDemo.y = 19;
System.out.println("ob1.sum(): " + ob1.sum());
} }
```

اذا عرفنا دالة من النوع static يجب ان تكون المتغيرات في داخلها من النوع static ايضا، مثال خاطئ لاستخدامه متغير عادي في داخل دالة static :

```
class StaticError {
int denom = 3;
static int val = 1024;
static  int  valDivDenom( ) {
return val/denom
} }
```

* يمكن ان ننشأ بلوك من النوع static ، وهذا البلوك سينفذ كأول جزء من الكلاس قبل اي شئ اخر في الكلاس بمجرد تحميل الكلاس .

```
static { // the statements }
```

الفصل الخامس (الكلاس Strings والكلاس Math)

الكلاس Strings

وهو عبارة عن كلاس من خلاله نكتب النصوص، وهو ينشأ بشكل اوتوماتيكي في اللغة بمجرد ان نكتب نص بين علامتي تنصيص ويمكن ان ننشأ منه اوبجكت بهذا الشكل :

```
String  str = new String("Hello");
```

او ننشأه بالطريقة العادية لكتابة المتغيرات (والاوبجكت سينشأ بشكل آلي) هكذا :

```
String str = “Hello” ;
```

وهناك عدة دوال يمكن تطبيقها على الـ String

العبارة length

تستخدم هذه العبارة لاعادة رقم يمثل عدد الاحرف (يشمل الفراغات والرموز)، مثال :

```
Int x = str.length( );
```

العبارة charAt( )

تستخدم هذه العبارة لاعادة حرف من وقع معين من النص وتأخذ بين قوسيها رقم يمثل موقع الحرف المراد، مثال يضع الحرف e في المتغير x :

```
String str = “Hello”;
char x = str.charAt(1);
```

العبارة equals

تستخدم هذه العبارة للتحقق من تطابق متغيرين نصيين فإذا كانا متطابقين تعيد القيمة true وبخلاق ذلك تعيد القيمة false ، مثال :

```
String x = “Hello”;
String y = “Hello”;
if( x.equals(y) ) System.out.println("x equals y");
```

العبارة compareTo( )

تستخدم هذه العبارة للمقارنة بين متغيرين نصيين حيث انها ستعيد القيمة 0 اذا كانا متساويين في الطول او ستعيد قيمة اكبر من 0 اذا كان المتغير الذي بين قوسيها هو الاصغر او قيمة اقل من 0 اذا كان المتغير الذي بين قوسيها هو الاكبر، مثال :

```
String x = “Hello”;
String y = “Welcome”;
int result ;
result = y.compareTo(x);
if(result == 0) System.out.print("y and x are equal");
else if(result < 0) System.out.print ("y is less than x");
else System.out.print ("y is greater than x");
```

هذا المثال سيطبع القيمة y is greater than x

العبارة indexOf( )

تستخدم هذه العبارة للبحث في نص معين عن ما موجود بين قوسيها وتعيد رقم يمثل موقع الحرف الاول من اول نتيجة تطابق تجدها او انها ستعيد الرقم -1 اذا لم تجد اي تطابق، مثال :

```
Sting  str = "One Two Three One";
int x = str.indexOf("One");
```

قيمة x ستكون 0

العبارة lastIndexOf( )

تستخدم هذه العبارة للبحث في نص معين عن ما موجود بين قوسيها وتعيد رقم يمثل موقع الحرف الاول من اخر نتيجة تطابق تجدها او انها ستعيد الرقم -1 اذا لم تجد اي تطابق، مثال :

```
Sting  str = "One Two Three One";
int x = str. lastIndexOf ("One");
```

قيمة x ستكون 14

العبارة substring( )

تستخدم هذه العبارة لنسخ جزء من نص ووضعه في متغير نصي اخر دون التأثير على المتغير الاصلي، وهي تأخذ بين قوسيها رقمين الاول يمثل موقع بداية النسخ والرقم الاخر يمثل موقع نهاية النسخ، مثال :

```
String str = "Java makes the Web move.";
String str2 = str.substring(5, 18);
System.out.println( str2 );
```

هذا المثال سيطبع makes the Web

الكلاس Math

هذا الكلاس موجود في الحزمة java.lang اي يمكن الوصول الى دوالة مباشرتا لان لغة جافا تستدعي هذه الحزمة بشكل الي، ويحتوي هذا الكلاس على عدة دوال حسابية منها :

الدالة sqrt( )

تستخدم لاخذ الجذر التربيعي للعدد، مثال :

```
double z = Math.sqrt(25);
```

الدالة pow( )

تستخدم هذه الدالة لتعيد قيمة عدد مرفوع الى اس معين، تأخذ بين قوسيها عددين الاول يمثل العدد والثاني هو الاس، مثال :

```
double z = Math. pow(5, 2);
```

الدالة abs( )

تستخدم هذه الدالة لاعادة القيمة المطلقة للرقم الذي يوضع بين قوسيها.

الدالة random( )

تستخدم هذه الدالة لاعادة رقم عشوائي .

الفصل السادس (الوراثة)

الوراثة هي ان نضيف كل محتويات كلاس في كلاس اخر وهذا الكلاس بدوره يمكن ان يضاف في كلاس اخر حيث سيمكننا الوصول الى كل ما موجود في الكلاس الاول وكأنه مكتوب في الكلاس الثاني، لكن العكس غير صحيح اي ان الكلاس الاول لا يحتوي اي معلومات عن الكلاس الثاني، ولوراثة اي كلاس نستخدم العبارة extends ، مثال :

```
class Shape {
double w;
void show() {
System.out.println("W is " + w);
} }
class Tria extends Shape {
double area() {
return w / 2;
} }
class SDem {
public static void main(String args[]) {
Tria t1 = new Tria();
t1.w = 4.0;
t1.show();
System.out.println("Area is " + t1.area());
} }
```

* العناصر التي نعلن عنها انها خاصة private فانها لا يمكننا الوصول اليها عند توريث الكلاس بشكل مباشر لانها ستكون خاصة في كلاسها المكتوبه فيه فقط ، لكن سيمكننا الوصول للمتغيرات الخاصة عن طريق الدوال بهذا الشكل، مثال :

```
class Shape {
private double w;
double getW() { return w; }
```

```
void setW(double x) { w = x; }
void show() {
System.out.println("W is " + w);
} }
class Tria extends Shape {
double area() {
return getW() / 2;
} }
class SDem {
public static void main(String args[]) {
Tria t1 = new Tria();
t1.setW(4.0);
t1.show();
System.out.println("Area is " + t1.area());
} }
```

* يمكن ان يحتوي الكلاس الوارث او الموروث على constructor ، اذا احتوى الكلاس الثاني (الوارث) على الـ constructor فهنا سيمكننا الوصول اليه بالطريقة الاعتيادة من الاوبجكت عند استدعائه ، لكن اذا احتوى الكلاس الاول (الموروث) على الـ constructor فانه يجب استخدامها في الكلاس الثاني (الوارث) وهنا نحتاج الى استخدام العبارة super التي تشير الى constructor الكلاس الاول وبين قوسيها البارامترات التي وضعت في constructor الكلاس الاول، ولاحظ انه يجب ان تكتب العبارة super كاول عبارة في داخل constructor الكلاس الثاني ، لتوضيح الكلام لاحظ هذا المثال :

```
class Shape {
double w;
Shape(double x) {
w = x;
} }
class Tria extends Shape {
private String style;
Tria(String s, double x) {
super(x);
```

```
style = s;
}
double area() {
return w / 2;
} }
class SDem {
public static void main(String args[]) {
Tria t1 = new Tria("filled", 4.0);
} }
```

* اذا احتوى الكلاس الموروث (الاول) على اكثر من constructor فانه يجب ان يحتوي الكلاس الوارث (الثاني) على نفس عدد الـ constructor ليتم استدعاء كل واحدة منها بالعبارة super بنفس الطريقة في المثال السابق .

* هناك استخدام اخر للعبارة super وهو للوصول الى متغيرات او دوال الكلاس الموروث (الاول) من الكلاس الوارث (الثاني) وهو سيكون بمثابة this لكن يشير الى الكلاس الموروث وليس الموجود فيه ، عندما يحتوي الكلاس الوارث (الثاني) على اسم متغير او دالة ويكون هذا الاسم مستخدم في الكلاس الموروث (الاول) هنا سيلغي الاسم الجديد الاسم القديم (اذا كانت دالة فيجب ان تحتوي على نفس البارامترات ايضا وليس فقط نفس الاسم) فعندما نستخدم الاسم في الكلاس الثاني فان هذا يشير الى المتغير الموجود في هذا الكلاس وليس القادم من الكلاس الاول وللوصول الى المتغير القادم من الكلاس الاول نستخدم العبارة super بعدها نقطة وبعدها اسم المتغير او الدالة ، مثال :

```
class A {
int i;
}
class B extends A {
int i;
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
} }
```

final

هناك عدة استخدامات للعبارة final ، اولها هو انها اذا كتبت قبل اسم الدالة او المتغير فانها تمنع استخدام هذا الاسم في الكلاس الاخر الذي سيرث هذا الكلاس ، وهي مجر تسبق اسم الدالة او الكلاس الذي نريد عدم استخدام اسمه بشكل متكرر (اقصد بالمتكرر هو ان يكون عندما دالتين بنفس الاسم) ، مثال :

```
final void Method( ) { // the statements }
```

الاستخدام الثاني للعبارة هو لمنع الكلاس من ان تتم وراثته من قبل اي كلاس اخر ، وبالتاكيد اذا كان الكلاس final فان كل الدوال التي فيه تكون كأنها مسبوقة بـ final ، وتكتب مع الكلاس بهذا الشكل :

```
final class A { // the body of Class A }
```

الاستخدام الاخر للعبارة هو لجعل متغير ما ذو قيمة ثابتة ولا تتغير هذه القيمة ابدا طوال ابرنامج ، وتستخدم بهذا الشكل :

```
final int VAR = 42 ;
```

abstract

نستخدم عبارة abstract لانشاء كلاس لكن هذا الكلاس لايمكن ان ننشأ منه اوبجكت وانما يمكن توريثه لكلاسات اخرى فقط ، وفي داخل هذا الكلاس يمكن انشاء دوال من النوع abstract ولاحظ ان الدالة اذا كانت من النوع abstract فانها يجب ان تكون ضمن كلاس من النوع abstract ايضا ، والدالة التي تكون من النوع abstract لا يمكن ان تحتوي على جسم او كتلة الدالة وانما فقط اعلان عن اسمها وعند توريث الكلاس الخاص بها هناك سيتم كتابة جسمها (محتوياتها) ، ولاحظ ايضا ان الدالة التي تكون من النوع abstract يجب ان تذكر وتكتب في الكلاس الذي سيرثها واذا كانت اكثر من دالة فيجب كتابتهن كلهن، والفائدة من ذلك انه احيانا نحتاج ان ننشأ دالة ضمن كلاس ونورثه لكلاسات لكن كل كلاس من هذه الكلاسات سينفذ هذه الدالة بطريقة مختلفة .

* الدالة التي تكون من النوع abstract لا يمكن ان تكون static ولا يمكن ان تكون construct .

* كما ذكرنا قبل قليل الكلاس الذي يكون من النوع abstract لا يمكن ان ننشأ منه اوبجكت لكن يمكن ان ننسخ فيه اوبجكت احد الكلاسات التي ورثته وكذلك يمكن ان يمرر كاوبجكت للدالة على شكل باراميتر او في الارجاع return وهنا ايضا يقصد به اوبجكت لاحد الكلاسات التي ورثته ، مثال :

```
public abstract class shape {
```

```
public abstract void Draw( ) ;
}
public class cir extends shape{
public void Draw( ) {
// the body
} }
public class Dem {
cir x = new cir( );
shape s = x ;
public void drawshape( shape ob) {
ob.Draw( );
}
public shape getshape( ){
cir c = new cir( );
return c;
} }
```

The Object Class

في لغة جافا هناك كلاس خاص معرف باسم Object وهذا الكلاس يكون بشكل افتراضي موروث (الاول) لكل الكلاسات الاخرى التي نستخدمها، فبالتالي كل الدوال المعرفة في هذا الكلاس يمكن استخدامها والوصول اليها مبارشرتا، هناك اربع دوال معرفة في هذا الكلاس وهي من النوع final فبالتالي لا يمكن استخدام اسماءها كاسماء للدوال التي ننشأها وهذه الدوال getClass( ), notify( ), notifyAll( ), wait( ) ، اما بقية الدوال فيمكن اعادة استخدام اسماءها ، بقية الدوال هي clone( ), equals(), finalize( ), hashCode( ), toString( )

الفصل السابع (Package و interface)

Package

في لغة جافا كل كلاس يتم تعريفة هو في الحقيقة جزء من حزمة package معينة ، الكلاسات التي لا نكتبها ضمن حزمة ستكون بشكل افتراضي ضمن الحزمة العامة global ، هناك عدة فوائد من تضمين الكلاسات في حزم حيث الكلاسات المعرفة ضمن حزمة يمكن ان تكون خاصة اي لايمكن الوصول اليها من خارج الحزمة المعرفة ضمنها بالاضافة الى انه في البرامج الكبيرة فانه يكون من الصعب تسمية كل كلاس في البرنامج باسم فريد اما باستخدام الحزم فانه يمكن ان نسمي كلاسين بنفس الاسم اذا كانا ينتميان لحزمتين محتلفتين ، بالاضافة الى ان الحزمة الواحدة يمكن ان تكون ضمن ملف واحد او ضمن اكثر من ملف لكن يجب ان يكون اسم الملف بنفس اسم الحزمة ، الحزم يمكن ان تورث اي تكون حزمة بداخل حزمة اخرى، لعمل حزمة نستخدم الكلمة المفتاحية package بعدها اسم الحزمة ، مثال :

```
package pak;
class A {
// body the class
}
class B {
public static void main(String args[]) {
// body the class
}
```

لاحظ في هذا المثال الكلاسين A و B هم من ضمن الحزمة pak ، هذا الملف سنسميه B.java ونضعه في الدليل الذي يدعى pak .

* اذا اردنا الوصول لكلاس معين من كلاس اخر وكلاهما في حزمة مختلفة فيجب ان يكون الكلاس الذي نريد الوصول اليه public عام او الـ constructor تبعه تكون public عامة او احد دواله عامة للوصول الى هذه الدالة، مثال :

```
package pak;
public  class A {
// body the class
```

```
}
```

الان نعمل حزمة اخرى باسم pak2 ومنها نصل للكلاس A والذي هو عام وسنعمل منه object لكن لاحظ انه يجب ان نسبق اسم الكلاس باسم الحزمة الموجود فيها وإلا فلن يتعرف البرنامج على مكان الكلاس :

```
package pak2;
class B {
public static void main(String args[]) {
pak.A  ob = new pak.A( ) ;
// body the class
}
```

* كذلك الامر اذا اردنا ان نورث كلاس لكلاس اخر وكلاهما في حزمة مختلفة، يجب ان نسبق اسم الكلاس باسم الحزمة التي تحتويه ليحدد البرنامج مكان هذا الكلاس، مثال :

```
package pak;
class A {
// body the class
}
```

فاذا اردنا توريث الكلاس A الى الكلاس B من حزمة اخرى تدعى pak2 نستخدم اسم حزمة الكلاس A والتي هي pak لاحظ :

```
package pak2;
class B extends pak.A {
// body the class
}
```

Import

تستخدم هذه العبارة لجلب كل محتويات حزمة او جزء منها الى حزمة اخرى وعند جلبها تكون البيانات الواردة وكأنها جزء من الحزمة الجديدة والصيغة العامة لها هي :

```
Import  mypack.MyClass
```

هنا سيتم جلب فقط الكلاس MyClass الموجود في الحزمة mypack ويمكن كتابة المسار الكامل للحزمة اذا كانت في مكان مختلف ، اما اذا اردنا ان نجلب كل محتويات الحزمة نستخدم هذا الكود :

```
import  mypack.*;
```

* لاحظ عند جلب حزمة او كلاس فانه عندما نريد الوصول لهذا الكلاس فاننا لانحتاج الى ان نذكر اسم الحزمة عندما نورث الكلاس او نعمل object منه ، مثال :

```
package pak;
class A {
// body the class
}
```

الان نعمل حزمة اخرى باسم pak2 ونجلب الحزمة pak وسنعمل object من الكلاس A ولاحظ انه لا نسبق اسم الكلاس باسم الحزمة الموجود فيها :

```
package pak2;
import  mypack.*;
class B {
A  ob = new A( ) ;
// body the class
}
```

* يمكن ان نستخدم import مع static لجلب عناصر كلاس او انترفيس معين اي اننا سوف لن نحتاج الى ذكر اسم الكلاس للوصول اليهم، مثلا اذا اردنا ان نأخذ جذر تربيعي لعدد فإننا نحتاج ان نكتب Math.sqrt او اذا اردنا ان نرفع عدد الى اس نكتب Math.pow لكن يمكن ان نتخلص من اسم الكلاس Math بهذا الشكل :

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
class Quadratic {
public static void main(String args[ ]) {
double a, b;
a = sqrt(9);
b = pow(3, 2);
System.out.println("a is " + a + " and b is " + b);
} }
```

اذا اردنا ان نصل الى كل عناصر كلاس او انترفيس معين يمكن ان نستخدم علامة النجمة * ، مثلا يمكن ان نكتب العبارة التالية لتشمل كل دوال الكلاس Math :

```
import static java.lang.Math.*;
```

interface

هو عبارة عن كلاس لكن الفرق هو انه يوصف ماذا ستفعل الدوال وليس كيف ستفعلها (اي ان الدوال تكون بدون جسد) ونعرف كلاس اخر (واحد او اكثر) ونربطه بالانترفيس وهذا الكلاس يجب ان يحتوي على كل الدوال التي ذكرت في الانترفيس وطريقة تنفيذها (اي جسدها) ويجب ان تكون من نفس النوع تماما من حيث نوع القيمة التي ستعيدها او من حيث خصوصيتها ان كانت عامة اوخاصة، بالاضافة الى ان الكلاس يمكن ان يحتوي محتواه الخاص به بالاضافة الى محتوى الانترفيس، مثال :

```
public interface Series {
int getNext();
void setStart(int x);
}
class ByTwos implements Series {
int start, val;
ByTwos() {
start = 0;
val = 0;
}
public int getNext() {
val += 2;
return val;
}
public void setStart(int x) {
start = x;
val = x;
} }
class SeriesDemo {
```

```
public static void main(String args[]) {
ByTwos ob = new ByTwos();
int x = ob.getNext();
Series obs ;
obs = ob;
int y = obs.getNext();
}
```

في هذا المثال عرفنا انترفيس اسمه Series ووضعنا فيه الدوال وبعدها ربطناه بكلاس اخر ينفذ دواله اسمه ByTwos من خلال العبارة implements ، وكما تلاحظ فان الدوال التي نفذت دوال الانترفيس كانت من النوع public وذلك لان الدوال في الانترفيس كانت عامة لان الانترفيس باكمله قد تم تعريفه عام، وكما تلاحظ في الدالة الرئيسية main فانه تم تعريف اوبجكت من النوع Series ومن ثم تم مساواته مع اوبجكت من النوع ByTwos .

* يمكن ان يحتوي الانترفيس على متغيرات لكن هذه المتغيرات تكون ثوابت وهي ستكون بشكل ضمني من النوع public و static و final .

* يمكن توريث انترفيس الى انترفيس اخر باستخدام العبارة extend وذلك كاي كلاس عادي يتم توريثه، ولكن عندما نريد ان نربط كلاس مع انترفيس وهذا الانترفيس يرث من انترفيس اخر فانه يجب تطبيق كل الدوال الموجودة في هذا الانترفيس وكذلك الدوال الموجودة في الانترفيس الموروث ايضا .

```
interface A {
void meth1();
void meth2();
}
interface B extends A {
void meth3();
}
```

في JDK8 اصبح من الممكن ان يتم وضع دالة معرفة في الانترفيس كاي دالة اعتيادية (اي ان لها جسد) ولكن هذه الدالة يجب ان تسبق بالعبارة default ، وهذه الدالة الافتراضية ليس بالضرورة ان يتم ذكرها في الكلاس المرتبط بهذا الانترفيس، مثال :

```
public interface MyIF {
int getUserID();
```

```
default int getAdminID() {
return 1;
} }
class MyIFImp implements MyIF {
public int getUserID() {
return 100;
} }
class DefaultMethodDemo {
public static void main(String args[]) {
MyIFImp obj = new MyIFImp();
System.out.println("User ID is " + obj.getUserID());
System.out.println("Administrator ID is " + obj.getAdminID());
} }
```

كما تلاحظ فان الكلاس MyIFImp لم يذكر الدالة getAdminID لكن مع ذلك البرنامج صحيح وتمكنا من الوصول الى هذه الدالة من خلال الاوبجكت وتنفيذها، ولاحظ ايضا انه يمكن كذلك الوصول الى الدالة الافتراضية من الكلاس المرتبط بالانترفيس والتعديل عليها وتغيير قيمها لتناسب هذا الكلاس، لاحظ هذا الكلاس المرتبط بالانترفيس MyIF المذكور في المثال السابق كيف سيضع التنفيذ الخاص به للدالة الافتراضية :

```
class MyIFImp2 implements MyIF {
public int getUserID() {
return 100;
}
public int getAdminID() {
return 42;
} }
```

* يمكن للكلاس ان يكون مرتبط باكثر من انترفيس وهنا نفصل بين اسمائهم بفارزة وسيتحتم على الكلاس ان يحتوي على جميع دوال كل انترفيس، مثال لكلاس يرتبط بثلاثة انترفيسز اسمائهم هي A, B, C واسم الكلاس هو Cla :

```
class Cla implements A, B, C {
// the body
```

```
}
```

* يمكن للانترفيس ان يحتوي على دالة من النوع static وهذه الدالة يمكن ان تحتوي على جسد كاي دالة اعتيادية في اي كلاس، لكن لاحظ ان هذه الدالة لا يمكن وراثتها بواسطة كلاس او انترفيس اخر.

الفصل الثامن (معالجة الاخطاء)

في بعض الاحيان تحدث اخطاء عند تنفيذ الكود المكتوب، هذه الاخطاء تسبب توقف البرنامج، لغة جافا تتعامل مع هذه الاخطاء بارسالها الى كلاس رئيسي اسمه Throwable ومن هذا الكلاس يتم توريث كلاسين الاول هو الكلاس Error وهو مختص بالاخطاء التي تحدث في الآلة الافتراضية وهذه الاخطاء تعالج بشكل افتراضي وليس للمبرمج دخل فيها، اما الكلاس الثاني والذي هو ما يهمنا اسمه Exception وهذا الكلاس مختص بالاخطاء التي تحدث عند الاستخدام كأن يتم القسمة على صفر او ادخال قيمة نصية من قبل المستخدم في متغير معرف انه int او محاولة الوصول الى ملف غير موجود إلخ من الاخطاء، من الكلاس Exception تم توريث العديد من الكلاسات حيث ان كل كلاس يتعامل مع خطأ معين من هذه الاخطاء .

* هناك نوعين من الاستثناءات :

الاول يحدث وقت التشغيل وهي مشتقة من الكلاس RuntimeException وتسمى هذه الكلاسات unchecked لانه لا يتم التحقق منها الا وقت تنفيذ البرنامج وهي موضحة في هذا الجدول :

اسم الكلاس	الخطأ الذي يتعامل معه
RuntimeException	مسؤل عن مجموعة من الاخطاء الاكثر شيوعيا في الحدوث، وهذه الاخطاء تحدث اثناء تشغيل البرنامج
ArithmeticException	اذا حدث خطأ حسابي كالتقسيم على صفر
InputMismatchException	اذا ادخل المستخدم قيمة غيرمتطابقة مع المتغير المعرف كان يدخل قيمة نصية في متغير رقمي
ArrayIndexOutOfBoundsException	اذا اردنا الوصول الى عنصر في المصفوفة وهذا العنصر خارج حدود العناصر المعرفة في المصفوفة
ArrayStoreException	تعيين عنصر لمصفوفة ليست من نفس نوع القيمة
ClassCastException	
EnumConstantNotPresentException	
IllegalArgumentException	
IllegalMonitorStateException	
IllegalStateException	
IllegalThreadStateException	
IndexOutOfBoundsException	
NegativeArraySizeException	انشاء مصفوفة وعدد عناصرها سالب
NullPointerException	

NullPointerException	
SecurityException	محاولة لانتهاك الامن
StringIndexOutOfBoundsException	
TypeNotPresentException	
UnsupportedOperationException	

الثاني يحدث اثناء كتابة او تفسير الكود (قبل تنفيذ البرنامج) وتسمى هذه الكلاسات checked :

اسم الكلاس	الخطأ الذي يتعامل معه
ClassNotFoundException	اذا لم يجد الكلاس
CloneNotSupportedException	
IllegalAccessException	رفض الوصول الى كلاس معين
InstantiationException	محاولة انشاء اوبجكت من كلاس او انترفيس من النوع abstract
InterruptedException	
NoSuchFieldException	طلب حقل غير موجود
NoSuchMethodException	طلب دالة غير موجودة
ReflectiveOperationException	

العبارة try و catch

عندما نكتب كود ومن المحتمل ان يحدث خطأ في داخل هذا الكود فاننا نضع الكود في داخل بلوك العبارة try فاذا حدث خطأ فان البرنامج سينتقل مباشرتا الى الكود الموجود في داخل العبارة catch وان لم يكن هناك خطأ فانه لن يدخل في كود العبارة catch ويكمل البرنامج من بعد العبارة، والصيغة العامة لها هي :

```
try {
// block of code to monitor for errors
}
catch (ExcepType1 exOb) {
// handler for ExcepType1
}
catch (ExcepType2 exOb) {
```

```
// handler for ExcepType2
}
.
```

كما تلاحظ من الصيغة العامه فانه يمكن ان يكون عندنا اكثر من عبارة catch واحدة حيث ان كل عبارة من هذه العبارات يمكن ان تتعامل مع خطأ محدد ونوع هذا الخطأ هو ExcepType والذي يمثل اسم الكلاس الذي يتعامل مع هذا الخطأ و exOb يمثل اسم اوبجكت نعرفه ليحمل المعلومات القادمة من الكلاس ExcepType ، مثال :

```
class ExcDemo1 {
public static void main(String args[ ]) {
int nums[ ] = new int[4];
try {
System.out.println("Before exception is generated.");
nums[7] = 10;
System.out.println("this won't be displayed");
}
catch (ArrayIndexOutOfBoundsException exc) {
System.out.println("Index out-of-bounds!");
}
System.out.println("After catch statement.");
} }
```

هنا حدث خطأ لان المصفوفة nums تتكون من اربع عناصر فقط وليس فيها عنصر من الاندكس 7 وهذا النوع من الخطأ يخزن في الكلاس ArrayIndexOutOfBoundsException ومن هذا الكلاس عرفنا اوبجكت اسمه exc ، وستكون مخرجات البرنامج بهذا الشكل :

```
Before exception is generated.
Index out-of-bounds!
After catch statement.
```

* اذا كان اسم الكلاس الذي يتعامل مع الخطأ الموجود بين قوسي الدالة catch غير متطابق مع الخطأ الذي حدث فانه لن يتم تنفيذ الدالة catch واذا لم يجد catch اخرى تتعامل مع الخطأ فانه سيتوقف البنامج .

* يمكن كتابة الكلاس Exception بين قوسي الدالة catch لان هذا الكلاس يحمل جميع الاخطاء بكل انواعها لكن اذا اردنا ان نحدد خطأ معين اذا حدث نعمل كذا فاننا نكتب اسم الكلاس الفرعي الذي يحمل هذا الخطأ.

* يمكن ان لا نكتب اي شيء في بلوك الدالة catch وهنا عند حدوث خطأ لن يحدث شيء وسيكمل البرنامج.

* يمكن للعبارة try و catch ان تكون متداخلة مع عبارات try و catch اخرى .

* اذا اردنا ان ننفذ نفس الكود مع اكثر من خطأ يمكن ان نستخدم عبارة catch واحدة وفيها اكثر من خطأ ونفصل بينهم بـ OR مثال :

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
// the code
}
```

العبارة throw

في بعض الاحيان نحتاج الى ان نرمي خطأ بشكل يدوي ويمكن ذلك من خلال العبارة throw وبعدها نضع اوبجكت، ولاحظ انه اذا نفذت العبارة throw فانه لن يتم تنفيذ الكود الذي بعدها، مثال :

```
class ThrowDemo {
try {
System.out.println("Before throw.");
throw new ArithmeticException();
}
catch (ArithmeticException exc) {
System.out.println("Exception caught.");
}
System.out.println("After try/catch block.");
}
```

* يجب ان يكون اسم الكلاس (الاوبجكت) المرمي بالعبارة throw هو نفس اسم الكلاس (الاوبجكت) الموجود بين قوسي العبارة catch التي ستعالج الخطأ، حيث في هذا المثال هو ArithmeticException .

العبارة throws

هناك نوعين من الدوال اثناء التعامل مع الاستثناء (الخطأ) الاولى تتعامل مع الخطأ وتعالجه في داخلها والثانية ترمي الخطأ ولا تعالجه ويتم معالجته بالعبارة try في المكان الذي تستدعى فيه الدالة وفي هذا النوع من الدوال تكون اهمية العبارة throws ، مثال لدالة من النوع الاول (ترمي الخطأ وتعالجه) :

```
public class A{
public void set( int val ){
try{
if( val > 100 ) throw new Exception("value overflow");
System.out.println(val);
}
catch( Exception e ){
System.out.println(e.getMessage() );
}
} }
class Main{
public static void main(String args[]){
A x = new A( );
x.set(1000);
} }
```

مثال لدالة من النوع الثالثي (ترمي الخطأ ولكنها لا تعالجه) وهنا يجب استخدام العبارة throws بعد قوسي الدالة وبعدها نكتب اسم كلاس الخطأ (الاستثناء) الذي سترميه واذا كان اكثر من خطأ نفصل بين اسمائهم بفارزة :

```
public class A{
public void set( int val ) throws Exception {
if( val > 100 ) throw new Exception("value overflow");
System.out.println(val);
} }
class Main{
public static void main(String args[ ]){
```

```
A x = new A( );
try{
x.set(1000);
}
catch( Exception e ){
System.out.println(e.getMessage() );
}
} }
```

* الكلاسات المشتقة من الكلاس RuntimeException ستكون معرفة بشكل افتراضي في لغة جافا لذا يمكن عدم كتابتها مع throws ولن يكون هناك اي خطأ .

دوال الاستثناء

بما ان ما بين قوسي العبارة catch هو اوبجكت (اسم كلاس الاستثناء وبعده اسم الاوبجكت) فهذا يعني ان هناك مجموعة من الدوال التي يمكن تطبيقها على هذا الاوبجكت، هذه الدوال هي :

اسم الدالة	عملها
fillInStackTrace( )	تعيد اوبجكت من النوع throwable وهذا الاوبجكت يمكن ان يرمى بالعبارة throw
getLocalizedMessage( )	تعيد لموقع الخطأ من النوع String
getMessage( )	تعيد وصف الخطأ من النوع String
printStackTrace( )	تعرض الـ stack trace
printStackTrace(PrintStream stream)	ترسل stack trace الى stream محدد
printStackTrace(PrintWriter stream)	ترسل stack trace الى stream محدد
toString( )	تعيد اوبجكت من النوع String يحمل وصف كامل للخطأ

مثال :

```
class ExcTest {
static void genException() {
int nums[] = new int[4];
System.out.println("Before exception is generated.");
```

```
nums[7] = 10;
System.out.println("this won't be displayed");
} }
class UseThrowableMethods {
public static void main(String args[]) {
try {
ExcTest.genException();
}
catch (ArrayIndexOutOfBoundsException exc) {
System.out.println("Standard message is: ");
System.out.println(exc);
System.out.println("\nStack trace: ");
exc.printStackTrace();
}
System.out.println("After catch statement.");
} }
```

ستكون مخرجات البرنامج بهذا الشكل :

```
Before exception is generated.
Standard message is:
java.lang.ArrayIndexOutOfBoundsException: 7
Stack trace:
java.lang.ArrayIndexOutOfBoundsException: 7
at ExcTest.genException(UseThrowableMethods.java:10)
at UseThrowableMethods.main(UseThrowableMethods.java:19)
After catch statement.
```

العبارة finally

تستخدم هذه العبارة ليتم تنفيذها بعد عبارة catch وفائدتها انه اذا حدث خطأ واستخدمنا try و catch للتعامل معه وهذا الخطأ قد يتسبب في الخروج من الكلاس او الدالة قبل نهاية تنفيذها ولكننا نريد ان ننفذ شيء مهم قبل الخروج كأن نغلق ملفات قد فتحناها مسبقا او نغلق اتصال بالانترنت اجريناه، والصيغة العامة لها هي :

```
finally {
// finally code
}
```

* الكود المكتوب في بلوك العبارة finally سيتم تنفيذه بعد الخروج من بلوك العبارة try و catch بغض النظر اذا كان هناك خطأ او لا .

My Exception

يمكن انشاء كلاس استثناء خاص بنا مثل الكلاسات الجاهزة الموجودة في لغة جافا، ولانشاء كلاس استثناء كل ما علينا فعله هو انشاء كلاس اعتيادي ونجعله يرث من الكلاس الرئيسي Exception ، مثال :

```
class NonIntResultException extends Exception {
int n, d;
NonIntResultException(int i, int j) {
n = i;
d = j;
}
public String toString() {
return "Result of " + n + " / " + d +" is non-integer.";
} }
class CustomExceptDemo {
public static void main(String args[]) {
int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };
int denom[] = { 2, 0, 4, 4, 0, 8 };
for(int i=0; i<numer.length; i++) {
```

```
try {
if((numer[i]%2) != 0) throw new NonIntResultException(numer[i], denom[i]);
System.out.println(numer[i] + " / " + denom[i] + " is " + numer[i]/denom[i]);
}
catch (ArithmeticException exc) {
System.out.println("Can't divide by Zero!");
}
catch (ArrayIndexOutOfBoundsException exc) {
System.out.println("No matching element found.");
}
catch (NonIntResultException exc) {
System.out.println(exc);
} } } }
```

في هذا المثال انشأنا كلاس سميناه NonIntResultException وجعلناه Exception وظيفته هو ان يرمي خطأ اذا تم قسمة عدد int على عدد int اخر وكان الناتج عدد فيه كسور، وسيكون ناتج البرنامج بهذا الشكل:

```
4 / 2 is 2
Can't divide by Zero!
Result of 15 / 4 is non-integer.
32 / 4 is 8
Can't divide by Zero!
Result of 127 / 8 is non-integer.
No matching element found.
No matching element found.
```

الفصل التاسع (I/O)

في لغة جافا تم تعريف حزمة java.io للتعامل مع البيانات المدخلة والمخرجة على شكل مجرى متدفق من البايتات او المحارف، واذا اردنا ان نتعامل مع كلاسات هذه الحزمة يجب عمل استدعاء لهذه الحزمة باستخدام عبارة import :

```
import  java.io.*;
```

علامة النجمة تشير الى كل كلاسات الحزمة وكذلك يمكن جلب فقط كلاس محدد اذا اردنا ان نتعامل فقط مع هذا الكلاس.

اولاً : مجرى البايتات

كلاسات مجرى البايتات

كلاسات مجرى البيانات (البايتات) كلها تشتق من الكلاسين الرئيسيين المستخدمين للقراءة (الادخال) والكتابة (الاخراج) وهما InputStream و OutputStream والكلاسات المشتقة هي :

اسم الكلاس	وظيفته
BufferedInputStream	يخزن المدخلات بشكل مؤقت
BufferedOutputStream	يخزن المخرجات بشكل مؤقت
ByteArrayInputStream	قراءة المدخلات من مصفوفة بايتات
ByteArrayOutputStream	كتابة المخرجات على مصفوفة بايتات
DataInputStream	مجرى ادخال يحتوي على مجموعة دوال لقراءة انواع البيانات الاساسية في جافا
DataOutputStream	مجرى اخراج يحتوي على مجموعة دوال لكتابة انواع البيانات الاساسية في جافا
FileInputStream	مجرى ادخال يقراء من الملف
FileOutputStream	مجرى اخراج يكتب على الملف
FilterInputStream	تطبيقات للكلاس InputStream
FilterOutputStream	تطبيقات للكلاس OutputStream
InputStream	كلاس من النوع abstract يصف مجرى الادخال
ObjectInputStream	مجرى ادخال للاوبجكتات
ObjectOutputStream	مجرى اخراج للاوبجكتات
OutputStream	كلاس من النوع abstract يصف مجرى الاخراج

PipedInputStream	انبوب ادخال
PipedOutputStream	انبوب اخراج
PrintStream	مجرى اخراج يحتوي على ( )print و ( )println
PushbackInputStream	مجرى ادخال يسمح للبايتات بان تعاد للمجرى
SequenceInputStream	مجرى ادخال يكون عبارة عن مجريين ادخال او اكثر والتي سوف تقراء بالتعاقب الواحد تلو الاخر

دوال الكلاس InputStream و OutputStream

كل دوال هذين الكلاسين ترمي الاخطاء في الاستثناء IOException ، الجدول الاول يبين دوال الكلاس InputStream والجدول الثالني يبين دوال الكلاس OutputStream

الدالة	المعنى
int available( )	تعيد عدد البايتات الحالية المتاحة للقراءة
void close( )	تغلق مصدر الادخال بالاضافة الى انها تقراء المحاولات التي تولد للاستثناء IOException
void mark(int numBytes)	تضع علامة على مجرى الادخال الحالي وهذا يبقى متاح الى ان يقراء بايتات الـ numBytes
boolean markSupported( )	تعيد true اذا كانت الدالة ( )mark او ( )reset مدعمة بواسطة مجرى الـ ...
int read( )	تقراء باينت واحد من مجرى الادخال ويكون على شكل عدد صحيح وتعيد -1 اذا وصلت الى نهاية المجرى
int read(byte buffer[ ])	تحاول تقراء مافي الذاكرة المؤقته. طول البايتات في المخزن المؤقت وتعيد الرقم الحقيقي للبايتات التي قرأت بنجاح. تعيد -1 اذا وصلت الى نهاية المجرى
int read(byte buffer[ ], int offset,int numBytes)	تحاول تقراء بايتات numBytes في الذاكرة المؤقتة وتبدأ من buffer[offset] ، تعيد رقم البايتات التي تمت قراءتها بنجاح ، تعيد -1 اذا وصلت الى نهاية المجرى
void reset( )	تعيد وضع مؤشر الادخال الى الموضع السابق
long skip(long numBytes)	تتجاهل بايتات numBytes وتعيد العدد الحقيقي للبايتات التي تم تجاهلها

المعنى	الدالة
تغلق مجرى الخروج بالاضافة الى انها تولد محاولة الكتابة على الاستثناء IOException	void close( )
	void flush( )
تكتب بايت مفرد على مجرى الخروج	void write(int b)
تكتب مصفوفة بايتات على مجرى خروج	void write(byte buffer[ ])
تكتب جزء من بايتات numBytes من المصفوفة buffer وتبدأ من buffer[offset]	void write(byte buffer[ ], int offset,int numBytes)

القراءة والكتابة على مجرى الادخال والاخراج

لغة جافا تجلب الحزمة java.lang بشكل افتراضي وهذه الحزمة تعرف كلاس اسمه System يحتوي هذا الكلاس على مجرى متغيرات معرفة مسبقا، تدعى in و out و err ، وهي مصرح عنها على انها public و final و static في داخل الكلاس System ، هذا يعني انه يمكننا استخدامها مباشرتا مع System بدون الحاجة الى اوبجكت معين.

System.out تشير الى مجرى الاخراج القياسي وهي بشكل افتراضي شاشة العرض، System.in تشير الى مجرى الادخال القياسي وهو بشكل افتراضي لوحة المفاتيح، System.err وهي تشير الى مجرى الخطأ وهو بشكل افتراضي شاشة العرض، هذه المجاري يمكن اعادة توجيهها الى اي اداة I/O متوافقة.

System.in هو اوبجكت من انوع InputStream ، System.out و System.err هي اوبجكتات للنوع PrintStream ، هذه مجاري بايتات ، لكنها تستخدم بشكل افتراضي لقراءة ولكتابة المحارف من وعلى شاشة العرض.

بما ان System.in هي مثال من InputStream فهذا يعني انها يمكن ان تستخدم كل دوال InputStream هناك ثلاثة اشكال من الدالة read وهي :

int read( ) throws IOException
int read(byte data[ ]) throws IOException
int read(byte data[ ], int start, int max) throws IOException

الاولى تستخدم لقراءة بايت واحد وتعيد -1 عندما تصل الى نهاية المجرى، الثانية تقراء البايتات من مجرى الادخال وتضعهم في المصفوفة data حتى تمتلئ المصفوفة او تصل الى نهاية المجرى او يحدث خطأ وهي تعيد البايتات التي قرأتها او تعيد -1 عندما تصادف نهاية المجرى، الثالثة تقراء البايتات من مجرى الادخال

وتضعهم في المصفوفة data وتبدأ في المصفوفة من الموقع المحدد start حتى max وهي تعيد البايتات التي قرأتها او تعيد -1 عندما تصادف نهاية المجرى، وكلها ترمي استثناء IOException اذا حدث خطأ، عند القراءة من System.in وبعدها الضغط على انتر سيتولد انتهاء للمجرى.

مثال:

```
import java.io.*;
class ReadBytes {
public static void main(String args[ ]) throws IOException {
byte data[ ] = new byte[10];
System.out.println("Enter some characters.");
System.in.read(data);
System.out.print("You entered: ");
for(int i=0; i < data.length; i++)
System.out.print((char) data[i]);
} }
```

وبنفس الشكل يمكن استخدام System.out مع دوال PrintStream والتي هي print( ) و println( ) او مع دوال OutputStream ، هناك ثلاثة اشكال من الدالة write وهي :

```
void write (int byte) throws IOException
void write (byte data[ ]) throws IOException
void write (byte data[ ], int start, int max) throws IOException
```

الاولى تستخدم لكتابة البايت الموجود بين قوسيها (ويجب ان يكون بايت واحد فقط) ، الثانية تكتب البايتات في المصفوفة data ، الثالثة تكتب البايتات في المصفوفة data وتبدأ في المصفوفة من الموقع المحدد start الى max ، وكلها ترمي استثناء IOException اذا حدث خطأ.

مثال لطباعة الحرف X :

```
class WriteDemo {
public static void main(String args[ ]) {
int b;
b = 'X';
System.out.write(b);
```

```
} }
```

كما تلاحظ المتغير b من النوع int وذلك لان الدالة write تاخذ فقط قيم من النوع int او التي تكون بايت واحد اي انها 8 بت .

القراءة والكتابة على الملفات باستخدام مجاري البايتات

مجرى البايتات تكون في اللغة بشكل افتراضي لذا يمكن ان تستخدم هذه المجاري لانشاء اوبجكتات محارف والتعامل مع الملفات، لانشاء مجرى ادخال يتعامل مع الملفات نعمل اوبجكت من احد هذه الكلاسات FileInputStream او FileOutputStream ، وحالما يفتح الملف يمكن القراءة والكتابة منه.

القراءة من الملفات

لفتح ملف ننشأ اوبجكت من الكلاس FileInputStream ، وهذا هو الشكل الغالب في استخدامه :

```
FileInputStream(String  fileName) throws FileNotFoundException
```

حيث ان filename يمثل اسم الملف المراد فتحه فاذا لم يجد الملف سيرمي استثناء في الكلاس FileNotFoundException والذي هو كلاس فرعي من الكلاس IOException ، وللقراءة من الملف يمكن استخدام ( )read بهذا الشكل :

```
int read( ) throws IOException
```

وبعد النتهاء من التعامل مع الملف يجب غلق الملف والصيغة العامة هي :

```
void close( ) throws IOException
```

مثال :

```
import  java.io.*;
class ShowFile {
public static void main(String args[ ]) {
int i;
if(args.length != 1) {
System.out.println("Usage: ShowFile File");
return; }
```

```
try {
FileInputStream fin = new FileInputStream(args[0]);
}  catch(FileNotFoundException exc) {
System.out.println("File Not Found");
return; }
try {
do {
i = fin.read( );
if(i != -1) System.out.print((char) i);
} while(i != -1);
} catch(IOException exc) {
System.out.println("Error reading file."); }
try {
fin.close( );
} catch(IOException exc) {
System.out.println("Error closing file.");
} } }
```

اسم الملف يكون بين علامتي تنصيص بين قوسي الاوبجكت الذي ننشئه من الكلاس FileInputStream وفي هذا المثال اسم الملف هو args[0] ، لتوضيح الفكرة يمكن ان يكون سطر الوبجكت بهذا الشكل :

```
FileInputStream fin = new FileInputStream(“Name.txt”);
```

الكتابة على الملفات

للكتابة على ملف ننشأ اوبجكت من الكلاس FileOutputStream ، والطريقتين الاكثر شيوعيا في الاستخدام هما :

```
FileOutputStream(String  fileName) throws FileNotFoundException
FileOutputStream(String  fileName, boolean append) throws FileNotFoundException
```

في هذه الصيغ اذا لم يتمكن من انشاء الملف فان الاستثناء سيرمى في الكلاس FileNotFoundException والذي هو كلاس فرعي من الكلاس IOException ، في الصيغة الاولى عندما يفتح ملف فانه سيحذف كل ما موجود فيه ويكتب فوقه اما في الطريقة الثانية اذا كانت true append فانه سيضيف الكتابة الى نهاية

الكتابة الموجودة في الملف وإلا فانه سيستبدل بالكتابة الجديدة ، للكتابة على الملف نستخدم الدالة write والصيغة العامة لها هي :

```
void write(int byteval) throws IOException
```

وعند الانتهاء من التعامل مع الملف يجب غلق الملف بهذا الشكل :

```
void close( ) throws IOException
```

مثال :

```
import  java.io.*;
class test {
public static void main(String args[ ]) {
try {
FileInputStream fis = new FileInputStream("E:\\test\\abc.jpg");
FileOutputStream fos = new FileInputStream("abc2.jpg");
int a;
while((a = fis.read()) != -1){fos.write(a);}
fis.close();
fos.close();
} catch (Exception ex){ex.printStackTrace();}
} }
```

في هذا المثال نسخنا الملف abc.jpg والذي هو عبارة عن صورة الموجود في المسار المحدد الى ملف اخر (اي انشأنا صورة اخرى) وذلك من خلال قراءة بايتات الصورة الاولى وكتابتها في بايتات الصورة الثانية abc2.jpg ، المثال السابق يأخذ وقت طويل في عملية النسخ وذلك لانه يقراء وينسخ بايت واحد كل مرة ولكتابة نفس المثال لكن ينفذ بشكل اسرع نعمل مصفوفة ونقرأها بالكامل ومن ثم تنسخ بالكامل :

```
import  java.io.*;
class test {
public static void main(String args[ ]) {
try {
FileInputStream fis = new FileInputStream("E:\\test\\abc.jpg");
FileOutputStream fos = new FileInputStream("abc2.jpg");
```

```
byte[] m = new byte[1000];
int a;
while((a = fis.read(m)) != -1){ fos.write(m, 0, a); }
fis.close( );
fos.close( );
} catch (Exception ex){ex.printStackTrace( );}
} }
```

اغلاق الملف بشكل الي

على الرغم من انه يمكننا ان نغلق الملف بشكل يدوي باستخدام الدالة close الا ان لغة جافا توفر امكانية لغلق الملف بشكل اوتوماتيكي باستخدام شكل خاص من عبارة try ، مثال :

```
import java.io.*;
class ShowFile {
public static void main(String args[ ]) {
int i;
if(args.length != 1) {
System.out.println("Usage: ShowFile filename");
return; }
try(FileInputStream fin = new FileInputStream(args[0])) {
do {
i = fin.read( );
if(i != -1) System.out.print((char) i);
} while(i != -1);
} catch(IOException exc) {
System.out.println("I/O Error: " + exc);
} } }
```

وكذلك يمكن التعامل مع اكثر من ملف وغلقها بشكل الي بستخدام عبارة try واحدة وذلك بالفصل بينها بفارزة منقوطة، مثال :

```
try (FileInputStream fin = new FileInputStream(args[0]);
```

```
FileOutputStream fout = new FileOutputStream(args[1]) ) {
// the code
}
```

قراءة وكتابة البايتات الثنائية

للقراءة او الكتابة على الملفات التي تحتوي على بيانات ثنائية كأن تكون int او double او short اي انها ليست ASCII نستخدم الكلاسين DataInputStream و DataOutputStream ، الصيغة العامة للكلاس DataOutputStream هي :

DataOutputStream(OutputStream OS)

حيث ان OS هو مجرى لاي بيانات تكتب من هذا النوع، لكتابة بيانات على ملف يمكن استخدام اوبجكت تم انشائه من الكلاس FileOutputStream .

والصيغة العامة للكلاس DataInputStream هي :

DataInputStream(InputStream OS)

حيث ان OS هو مجرى لاي بيانات تقراء من هذا النوع، لقارءة بيانات من ملف يمكن استخدام اوبجكت تم انشائه من الكلاس FileInputStream .

الكلاس DataOutputStream ينفذ انترفيس DataOutput والكلاس DataInputStream ينفذ انترفيس DataInput وهذه الانترفيسز تعرف دوال لكتابة وقراءة كل انواع البيانات على الملف، دوال الكتابة موضحة في الجدول الاول اما دوال القراءة موضحة في الجدول الثاني، وكل الدوال في كلا الجدولين ترميان استثناء من النوع IOException :

المعنى	دوال الاخراج
تكتب قيمة من النوع boolean محدد بواسطة val	void writeBoolean(boolean val)
تكتب قيمة من النوع low-order byte محدد بواسطة val	void writeByte(int val)
تكتب قيمة محدد بواسطة val من النوع char	void writeChar(int val)
تكتب قيمة من النوع double محددة بواسطة val	void writeDouble(double val)
تكتب قيمة من النوع float محددة بواسطة val	void writeFloat(float val)
تكتب قيمة من النوع int محددة بواسطة val	void writeInt(int val)
تكتب قيمة من النوع long محددة بواسطة val	void writeLong(long val)
تكتب قيمة من النوع short محددة بواسطة val	void writeShort(int val)

دوال الادخال	المعنى
boolean readBoolean( )	تقراء قيمة من النوع boolean
byte readByte( )	تقراء قيمة من النوع byte
char readChar( )	تقراء قيمة من النوع char
double readDouble( )	تقراء قيمة من النوع double
float readFloat( )	تقراء قيمة من النوع float
int readInt( )	تقراء قيمة من النوع int
long readLong( )	تقراء قيمة من النوع long
short readShort( )	تقراء قيمة من النوع short

مثال :

```
import java.io.*;
class RWData {
public static void main(String args[ ]) {
double d = 1023.56;
try (DataOutputStream dataOut =
 new DataOutputStream(new FileOutputStream("testdata"))) {
dataOut.writeDouble(d);
} catch(IOException exc) {
System.out.println("Write error.");
return; }
try (DataInputStream dataIn =
 new DataInputStream(new FileInputStream("testdata"))) {
d = dataIn.readDouble();
System.out.println("Reading " + d);
} catch(IOException exc) {
System.out.println("Read error.");
} } }
```

RandomAccessFile

يمكن الوصول الى الملف بشكل عشوائي اي ليس بشكل متسلسل كما كنا نفعل في السابق وكذلك يمكن الوصول الى جزء محدد من الملف اي وضع مؤشر الكتابة في مكان معين وذلك من خلال الكلاس RandomAccessFile وهذا الكلاس غير مشتق من InputStream او من OutputStream لكنه ينفذ الانترفيس DataInput و DataOutput والتي تعرف دوال I/O الاساسية ، الصيغة العامة له هي :

RandomAccessFile(String fileName, String access) throws FileNotFoundException

حيث ان fileName يمثل اسم الملف و access تحدد نوع الوصول للملف اذا كانت "r" فهذا يعني ان الملف يستخدم للقراءة فقط اما اذا كانت "rw" فيمكن استخدام الملف للقراءة والكتابة.

* الدالة seek( )

تستخدم هذه الدالة لتحدد موضع المؤشر داخل الملف، والصيغة العامة لها هي :

void seek(long newPos) throws IOException

حيث ان newPos يمثل موضع المؤشر اين سيكون وهو يأخذ قيمة بالبايت تحدد من بداية الملف الى الموضع المراد، بعد استخدام الدالة seek فان عملية القراءة والكتابة ستكون من موضع المؤشر المحدد.

يمكن للكلاس RandomAccessFile ان يستخدم الدوال ( )read و ( )write وبما انه ينفذ الانترفيسز DataInput و DataOutput فانه يمكن ان يستخدم كل دوالهم، الموضحه في الجدول في الاعلى.

مثال :

```
import java.io.*;
class RandomAccessDemo {
public static void main(String args[ ]) {
double data[ ] = { 19.4, 10.1, 123.54, 33.0, 87.9, 74.25 };
double d;
try (RandomAccessFile raf = new RandomAccessFile("random.dat", "rw")) {
for(int i=0; i < data.length; i++) {
raf.writeDouble(data[i]);
}
raf.seek(0);
```

```
d = raf.readDouble();
System.out.println("First value is " + d);
raf.seek(8 * 3);
d = raf.readDouble();
System.out.println("Fourth value is " + d);
} catch(IOException exc) {
System.out.println("I/O Error: " + exc);
} } }
```

لاحظ ان كل قيمة double تساوي 8 بايت لذا للوصول الى القيمة الرابعة كتبنا 3 * 8 و 0 تمثل القيمة الاولى.

ثانياً : مجرى المحارف

كلاسات مجرى المحارف

وهي مماثلة (موازية) لكلاسات البايتات ولكن وظيفتها قراءة المحارف وليس البايتات وهي كذلك تشتق من كلاسين رئيسيين هما Reader للقراءة (الادخال) و Writer للكتابة (الاخراج) والكلاسات المشتقة هي :

اسم الكلاس	وظيفته
BufferedReader	تخزين مؤقت لمجرى ادخال المحارف
BufferedWriter	تخزين مؤقت لمجرى اخراج المحارف
CharArrayReader	مجرى ادخال للقراءة من مصفوفة محارف
CharArrayWriter	مجرى اخراج للكتابة على مصفوفة محارف
FileReader	مجرى ادخال يقراء من الملف
FileWriter	مجرى اخراج يكتب على الملف

FilterReader	فلتر قراءة
FilterWriter	فلتر كتابة
InputStreamReader	مجرى ادخال يحول البايتات الى محارف
LineNumberReader	مجرى ادخال يعد الاسطر
OutputStreamWriter	مجرى اخراج يحول المحارف الى بايتات
PipedReader	انبوب ادخال
PipedWriter	انبوب اخراج
PrintWriter	مجرى اخراج يحتوي على ( )print و ( )println
PushbackReader	مجرى ادخال يسمح ان تعاد المحارف الى مجرى الادخال
Reader	كلاس من النوع abstract يصف مجرى الادخال
StringReader	مجرى ادخال يقراء من النصوص
StringWriter	مجرى اخراج يكتب على النصوص
Writer	كلاس من النوع abstract يصف مجرى الاخراج

دوال الكلاس Reader و Writer

اغلب دوال هذين الكلاسين ترمي الاخطاء في الاستثناء IOException ، الجدول الاول يبين دوال الكلاس Reader والجدول الثالني يبين دوال الكلاس Writer

الدالة	المعنى
abstract void close( )	تغلق مصدر الادخال، اضافة الى تولد محاولات القراءة على IOException
void mark(int numChars)	تضع علامة على النقطة الحالية لمجرى الادخال والتي سوف تبقيها متاحة حتى تقراء حروف numChars
boolean markSupported( )	تعيد true اذا كانت ( )mark/( )reset كانت مدعومة على هذا المجرى
int read( )	تعيد قيمة انتجر تمثل الحرف التالي المتاح على مجرى الدخال، وتعيد -1 اذا وصلت الى نهاية المجرى

int read(char buffer[ ])	محاولات لقراءة حروف buffer.length في المخزن المؤقت وتعيد عدد الحروف الحقيقية التي تم قراءتها بنجاح، وتعيد القيمة -1 اذا وصلت الى نهاية المجرى
abstract int read(char buffer[ ], int offset, int numChars)	تحاول قراءة حروف numChars في المخزن المؤقت وتبدأ من buffer[offset] ، وتعيد القيمة -1 اذا وصلت الى نهياة المجرى
int read(CharBuffer buffer)	تحاول ملئ المخزن المؤقت بواسطة buffer وتعيد عدد الحروف التي تمت قرائتها بنجاح ، تعيد -1 اذا وصلت الى نهاية المجرى
boolean ready( )	تعيد القيمة true اذا لم ينتظر طلب الادخال القادم وتعيد fulse بخلاف ذلك
void reset( )	تعيد ضبط مؤثر الادخال الى العلامة السابقة
long skip(long numChars)	تتخطى محارف الادخال من numChars وتعيد العدد الحقيقي للحروف التي تخطتها

المعنى	الدالة
تلحق ch الى نهاية مجرى الاخراج	Writer append(char ch)
تلحق chars الى نهاية مجرى الاخراج ، CharSequence هو انترفيس يعرف عمليات القراءة فقط على محارف متتابعة	Writer append(CharSequence chars)
تلحق متتالي من chars تبدأ عند begin وتتوقف عند end الى نهاية مجرى الاخراج CharSequence هو انترفيس يعرف عمليات القراءة فقط على محارف متتابعة	Writer append(CharSequence chars, int begin, int end)
تغلق مجرى الاخراج تكتب المحاولات التي تولد على IOException	abstract void close( )
تسبب اي مخرج الذي يخزن بشكل مؤقت ليرسل الى مكان معين	abstract void flush( )
تكتب حرف مفرد على مجرى الاخراج ولاحظ ان البراميتر من النوع int والذي يمثل اي محرف اقل من 8 بت	void write(int ch)
يكتب مصفوفة كاملة من الحروف الى مجرى الاخراج	void write(char buffer[ ])
يكتب نطاق فرعي من حروف numChars من مصفوفة buffer ويبدأ عند buffer[offset] الى مجرى الاخراج	abstract void write(char buffer[ ], int offset, int numChars)
يكتب str على مجرى الاخراج	void write(String str)
يكتب نطاق فرعي من حروف numChars من مصفوفة str ويبدأ عند offset	void write(String str, int offset, int numChars)

الادخال بستخدام مجرى المحارف

كما لاحظنا مسبقا فانه يمكن ادخال البيانات بستخدام مجرى البايتات لكن لادخال المحارف بشكل اكثر كفاءة يفضل استخدام الكلاس BufferedReader لكن بما ان System.in من مجرى البايتات فانه يجب تحويلها الى محارف بستخدام الكلاس InputStreamReader ، الصيغة العامة للكلاس InputStreamReader هي :

InputStreamReader(InputStream IS)

حيث ان IS تمثل اوبجكت من النوع InputStream وبما ان System.in فانه يمكن استخدامها، الصيغة العامة للكلاس BufferedReader هي:

BufferedReader(Reader inputReader)

وبربط الصيغتين مع بعض يمكن ان نكون الشكل التالي :

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

بعد هذه الجملة سيكون br مرتبط مجرى محارف مع System.in .

قراءة الحروف

يمكن استخدام الدالة read مع System.in لقراءة الحروف بنفس الطريقة التي كانت تقراء بها في مجرى البايتات، وهناك ثلاثة انواع من الدالة read وهي :

int read( ) throws IOException
int read(char data[ ]) throws IOException
int read(char data[ ], int start, int max) throws IOException

الاولى تستخدم لقراءة حرف Unicode واحد وتعيد -1 عندما تصل الى نهاية المجرى، الثانية تقراء الحروف من مجرى الادخال وتضعهم في المصفوفة data حتى تمتلئ المصفوفة او تصل الى نهاية المجرى او يحدث خطأ وهي تعيد عدد الحروف التي قرأتها او تعيد -1 عندما تصادف نهاية المجرى، الثالثة تقراء الحروف من مجرى الادخال وتضعهم في المصفوفة data وتبدأ في المصفوفة من الموقع المحدد start حتى max وهي تعيد عدد الحروف التي قرأتها او تعيد -1 عندما تصادف نهاية المجرى، وكلها ترمي استثناء IOException اذا حدث خطأ، عند القراءة من System.in وبعدها الضغط على انتر سيتولد انتهاء للمجرى.

مثال :

```
import java.io.*;
class ReadChars {
public static void main(String args[ ]) throws IOException {
char c;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, period to quit.");
do {
c = (char) br.read( );
System.out.println(c);
} while(c != '.');
} }
```

هنا سينتهي البرنامج عندما يدخل المستخدم نقطة.

قراءة النصوص Strings

لقراءة النصوص نستخدم الدالة readLine والتي هي من ضمن دوال الكلاس BufferedReader ، والصيغة العامة لها هي :

```
String readLine( ) throws IOException
```

تعيد اوبجكت من النوع String يحتوي على الحروف المقروءة وتعيد القيمة null عند محاولة القراءة في نهاية المجرى، لاحظ هذا المثال الذي سيقرأ اسطر من النص الى ان تكتب كلمة stop :

```
import java.io.*;
class ReadLines {
public static void main(String args[ ]) throws IOException {
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter text. then Enter 'stop' to quit.");
do {
str = br.readLine( );
```

```
System.out.println(str);
} while(!str.equals("stop"));
} }
```

الاخراج بستخدام مجرى المحارف

يمكن استخدام الكلاس PrintWriter للاخراج وله عدة صيغ في الاستخدام منها :

```
PrintWriter(OutputStream OS, boolean flushingOn)
```

حيث هنا OS هو اوبجكت من النوع OutputStream و flushingOn تتحكم فيما اذا كانت جافا تورد مجرى اخراج كل مرة الدالة println( ) ، اذا كانت flushingOn صحيحة true التوريد سياخذ مكان بشكل اوتوماتيكي اما اذا كانت fulse فانه لن يكون اوتوماتيكي.

الكلاس PrintWriter يدعم الدوال print( ) و println( ) ايضا ويمكن استخدامها كما تستخدم مع System.out ، مثال :

```
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[ ]) {
PrintWriter pw = new PrintWriter(System.out, true);
int i = 10;
double d = 123.65;
pw.println("Using a PrintWriter.");
pw.println(i + " + " + d + " is " + (i+d));
} }
```

القرائة والكتابة على الملف بستخدام مجرى المحارف

كما رأينا سابقا فانه يمكن ان نكتب او نقراء من الملف بستخدام مجاري البايتات ولكن اذا اردنا ان نتعامل مع محارف من نوع Unicode فانه يجب ان نستخدم مجاري المحارف، وافضل كلاسين للقيام بذلك هما FileReader و FileWriter .

استخدام FileWriter

FileWriter ينشأ Writer يمكن استخدامها للكتابة على الملف، والصيغتين الاشهر لاستخدام الكلاس هي :

```
FileWriter(String fileName) throws IOException
FileWriter(String fileName, boolean append) throws IOException
```

حيث ان filename هو اسم ومسار الملف، اذا كانت append قيمتها true فانه سيضيف الى نهاية محتويات الملف اما في خلاف ذلك فانه سيتم استبدال النص بالنص الجديد، وفي حالة الخطأ يرمي استثناء IOException ، الكلاس FileWriter مشتق من الكلاس OutputStreamWriter ومن Writer لذا فانه يصل الى دوال هذه الكلاسات ، مثال :

```
import java.io.*;
class KtoD {
public static void main(String args[ ]) {
String str;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter text ('stop' to quit).");
try (FileWriter fw = new FileWriter("test.txt")) {
do {
System.out.print(": ");
str = br.readLine( );
if(str.compareTo("stop") == 0) break;
str = str + "\r\n"; // add newline
```

```
fw.write(str);
} while(str.compareTo("stop") != 0);
} catch(IOException exc) {
System.out.println("I/O Error: " + exc);
} } }
```

استخدام FileReader

الكلاس FileReader ينشأ Reader لقراءة الملف، والصيغة العامة له هي :

```
FileReader(String fileName) throws FileNotFoundException
```

حيث ان filename هو اسم ومسار الملف، وفي حالة لم يجد الملف يرمي استثناء FileNotFoundException ، الكلاس FileReader مشتق من الكلاس InputStreamReader ومن Reader لذا فانه يصل الى دوال هذه الكلاسات ، مثال :

```
import java.io.*;
class DtoS {
public static void main(String args[ ]) {
String s;
try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {
while((s = br.readLine( )) != null) {
System.out.println(s);
} } catch(IOException exc) {
System.out.println("I/O Error: " + exc);
} } }
```

في هذا المثال لاحظ ان FileReader مغلف بـ BufferedReader وهذا يعطيه الصلاحية للوصول الى readLine

تحويل الارقام النصية الى عددية

كما نعلم فانه يمكن ان نكتب في الدالة println( ) اي نوع من البيانات لطباعتها سواء كانت ارقام او نصوص لكن هذا لا يحدث مع كل الدوال مثلا الدالة read( ) لايمكنها ان تقرأ الارقام النصية مثلا "100" لذلك توفر لغة جافا مجموعة من الكلاسات لتغليف الاعداد النصية للتعامل معها او تحويلها الى شكلها الثنائي، وهذه الدوال هي :

كلاس التغليف	دالة التحويل
Double	static double parseDouble(String str) throws NumberFormatException
Float	static float parseFloat(String str) throws NumberFormatException
Long	static long parseLong(String str) throws NumberFormatException
Integer	static int parseInt(String str) throws NumberFormatException
Short	static short parseShort(String str) throws NumberFormatException
Byte	static byte parseByte(String str) throws NumberFormatException

مثال :

```
import java.io.*;
class AvgNums {
public static void main(String args[ ]) throws IOException {
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str;
int n;
double sum = 0.0;
double avg, t;
System.out.print("How many numbers will you enter: ");
str = br.readLine( );
try {
n = Integer.parseInt(str);
} catch(NumberFormatException exc) {
System.out.println("Invalid format");
n = 0; }
System.out.println("Enter " + n + " values.");
```

```
for(int i=0; i < n ; i++) {
System.out.print(": ");
str = br.readLine( );
try {
t = Double.parseDouble(str);
} catch(NumberFormatException exc) {
System.out.println("Invalid format");
t = 0.0; }
sum += t; }
avg = sum / n;
System.out.println("Average is " + avg); } }
```

الفصل العاشر (Multithreading)

في لغة جافا يمكن ان ينفذ البرنامج اكثر من جزء في نفس الوقت كأن يقوم بارسال البيانات عبر الانترنت وفي نفس الوقت (وقت فراغ المعالج) يقراء من ملف او يقوم بالطباعة طالما ان كل عملية من هذه العمليات في thread منفصل.

نظام الـ Multithreading في جافا بني على اساس الكلاس Thread والانترفيس Runnable وكلاهما موجودين في الحزمة java.lang ، الكلاس Thread يعرف دوال للتعامل وادارة تعدد الوظائف في البرنامج من هذه الدوال :

المعنى	الدالة
تعيد اسم الثريد	final String getName( )
تضع اسم للثريد	final void setName(String threadName)
تعيد اولوية (افضلية) الثريد	final int getPriority( )
تحدد اولوية ثريد على اخر اي تحدد مستوى اولوية كل ثريد	final void setPriority(int level)
تحدد فيما اذا كان الثريد قيد التنفيذ حاليا (لايزال يعمل)	final boolean isAlive( )
تنتظر الثريد الى ان ينتهي و اذا حدث خطأ ترمي الاستثناء InterruptedException	final void join( )
نقطة الدخول الى الثريد	void run( )
يوقف بشكل مؤقت عمل الثريد لفترة تحدد بالملي ثانية واذا حدث خطأ ترمي استثناء من النوع InterruptedException	static void sleep(long milliseconds)
يبدأ الثريد بواسطة استدعاء دالة الـ run الخاصة بها	void start( )

* كل العمليات التي تنفذ في لغة جافا لديها على الاقل ثريد واحد ينفذ وعادتا تدعى الثريد الرئيسي، وهذه الثريد تنفذ عند بداية تشغيل اي برنامج، وكل ثريد يتم انشائه يكون من هذا الثريد.

انشاء ثريد Thread

ننشأ الثريد بواسطة اوبجكت من النوع Thread ، في جافا هناك طريقتين لانشاء اوبجكت runnable ، الاولى عن طريق implements الانترفيس Runnable والثانية من خلال extend الكلاس Thread ، وكلا الطريقتين تؤديان نفس الوظيفة الفرق هو فقط كيف تنشأ الثريد.

الانترفيس Runnable يعرف دالة واحدة فقط وهي ( )run وهي معلن عنها بهذا الشكل :

```
public void run( )
```

في داخل الدالة run نعرف الكود الذي ينشأ ثريد جديدة، ولاحظ ان الدالة run يمكن ان تستدعي دوال اخرى ، كلاسات اخرى وتصرح عن متغيرات تماما مثل الثريد الرئيسي، الفرق الوحيد هو ان هذه الدالة تولد نقطة داخلية من اخرى، وتنفذ في وقت واحد في برنامجك، وهذه الثريد سوف تنتهي عندما يتم عمل returne للدالة run ، هناك اكثر من شكل للكلاس Thread وابسط شكل له هو :

```
Thread(Runnable threadOb)
```

حيث ان threadOb هو كلاس ينفذ الانترفيس Runnable ، وهذا التعريف سيكون بداية تنفيذ الثريد، ولكنها لن تعمل الا بعد استدعاء دالة start الخاصة بها ، مثال :

```
/* Objects of MyThread can be run in their own threads because MyThread
implements Runnable interface */
class MyThread implements Runnable {
String thrdName;
MyThread(String name) {
thrdName = name; }
// Entry point of thread.
public void run( ) {
System.out.println(thrdName + " starting.");
try {
for(int count=0; count < 10; count++) {
Thread.sleep(400);
System.out.println("In " + thrdName + ", count is " + count);
} } catch(InterruptedException exc) {
System.out.println(thrdName + " interrupted."); }
```

```
System.out.println(thrdName + " terminating.");
} }
class UseThreads {
public static void main(String args[ ]) {
System.out.println("Main thread starting.");
// First, construct a MyThread object.
MyThread mt = new MyThread("Child #1");
// Next, construct a thread from that object.
Thread newThrd = new Thread(mt);
// Finally, start execution of the thread.
newThrd.start( );
for(int i=0; i<50; i++) {
System.out.print(".");
try {
Thread.sleep(100);
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted."); } }
System.out.println("Main thread ending.");
} }
```

ناتج البرنامج سيكون بهذا الشكل :

```
Main thread starting.
.Child #1 starting.
...In Child #1, count is 0
....In Child #1, count is 1
....In Child #1, count is 2
...In Child #1, count is 3
....In Child #1, count is 4
....In Child #1, count is 5
....In Child #1, count is 6
...In Child #1, count is 7
```

```
....In Child #1, count is 8
....In Child #1, count is 9
Child #1 terminating.
.....Main thread ending.
```

في هذا المثال في البداية انشأنا كلاس اسمه MyThread وربطناه عن طريق عمل implements له مع الانترفيس Runnable وهذا يعني انه يمكن ان ننشأ اوبجكت من هذا الكلاس ونمرره للـ constructor للكلاس Thread ، عملنا الدالة sleep في داخل ادالة run لاجل ان نشاهد كيف يعمل البرنامج ببطئ، ونتيجة البرنامج قد تختلف عما موجود هنا لانها تعتمد على نوع الحاسوب الذي ينفذ البرنامج حيث يعتمد على سرعة معالج الحاسوب الذي سيعالج الامرين (الدالة main وادالة run) في آن واحد.

* هناك ملاحظة مهمه وهي انه يجب دائما ان لا نجعل الدالة main تنتهي قبل ان ينتهي تنفيذ كل الثريد يمكن عمل ذلك من خلال فرق الوقت كما في المثال السابق.

* يمكن ان نعمل تحسينات على المثال السابق وذلك عن طريق جعل كلاس الثريد يتم تنفيذه بمجرد ان ننشأه في الدالة main ونضع اسمه في داخله، اضافة الى انه نعمل اوبجكت الكلاس Thread في داخل كلاس الثريد الذي ننشأه، لاحظ :

```
class MyThread implements Runnable {
Thread thrd; //A reference to the thread is stored in thrd
MyThread(String name) {
thrd = new Thread(this, name); //The thread is named when it is created.
thrd.start( ); // start the thread
}
public void run( ) {
System.out.println(thrd.getName( ) + " starting.");
try {
for(int count=0; count<10; count++) {
Thread.sleep(400);
System.out.println("In " + thrd.getName( ) + ", count is " + count);
} } catch(InterruptedException exc) {
System.out.println(thrd.getName( ) + " interrupted.");
}
System.out.println(thrd.getName( ) + " terminating.");
```

```
} }
class UseThreadsImproved {
public static void main(String args[ ]) {
System.out.println("Main thread starting.");
MyThread mt = new MyThread("Child #1");//Now the thread starts when it is created
for(int i=0; i < 50; i++) {
System.out.print(".");
try {
Thread.sleep(100);
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted.");
} }
System.out.println("Main thread ending.");
} }
```

* كما ذكرنا في البداية فانه يمكن عمل كلاس الثريد بطريقتين اما من خلال عمل implements للانترفيس Runnable او عن طريق عمل extends للكلاس Thread، جربنا في المثالين السابقين عمل implements للانترفيس اما الان سنعمل extends للكلاس، لاحظ هذا المثال والذي هو نفس المثال السابق تماما عدا اختلاف الطريقة :

```
class MyThread extends Thread {
MyThread(String name) {
super(name); // name thread
start( ); // start the thread
}
public void run( ) {
System.out.println(getName( ) + " starting.");
try {
for(int count=0; count < 10; count++) {
Thread.sleep(400);
System.out.println("In " + getName( ) + ", count is " + count);
```

```
} } catch(InterruptedException exc) {
System.out.println(getName( ) + " interrupted."); }
System.out.println(getName( ) + " terminating.");
} }
class ExtendThread {
public static void main(String args[ ]) {
System.out.println("Main thread starting.");
MyThread mt = new MyThread("Child #1");
for(int i=0; i < 50; i++) {
System.out.print(".");
try {
Thread.sleep(100);
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted.");
} }
System.out.println("Main thread ending.");
} }
```

* يمكن عمل اكثر من ثريد لينفذ في نفس الوقت لعمل ثلاثة ثريد تنفذ في نفس الوقت من المثال السابق نستبدل السطر التالي في الدالة main :

```
MyThread mt = new MyThread("Child #1");
```

بالاسطر التالية :

```
MyThread mt = new MyThread("Child #1");
MyThread mt = new MyThread("Child #2");
MyThread mt = new MyThread("Child #2");
```

هنا سينفذ الكلاس MyThread اكثر من مرة حسب المعطيات في نفس الوقت اثناء تنفيذ الدالة main .

* كما ذكرنا فانه يجب دائما ان ينتهي تنفيذ الدالة main بعد ان ينتهي تنفيذ كل الثريد التي تعمل، في الامثلة السابقة استخدما عامل الوقت حيث ان الوقت الممرر للدالة sleep في الدالة main اكثر من الوقت الممرر للثريد لكن هناك طريقة اكثر فاعلية وهي عن طريق استخدام الدالة isAlive( ) حيث انها ستعيد القيمة true

اذا كان الثريد لا يزال يعمل وبخلاف ذلك ستعيد القيمة fulse ، سنكتب فقط الدالة main في المثال السابق لتوضيح طريقة عمل الدالة ( )isAlive :

```
class MoreThreads {
public static void main(String args[ ]) {
System.out.println("Main thread starting.");
MyThread mt1 = new MyThread("Child #1");
MyThread mt2 = new MyThread("Child #2");
MyThread mt3 = new MyThread("Child #3");
do {
System.out.print(".");
try {
Thread.sleep(100);
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted.");
} } while (mt1.thrd.isAlive() || mt2.thrd.isAlive() || mt3.thrd.isAlive());
//This waits until all threads terminate
System.out.println("Main thread ending.");
} }
```

* الطريقة الاخرى لانتظار كل الثريد الى ان تنتهي هو عن طريق الدالة ( )join ، مثال :

```
class MyThread implements Runnable {
Thread thrd;
MyThread(String name) {
thrd = new Thread(this, name);
thrd.start( ); }
public void run( ) {
System.out.println(thrd.getName( ) + " starting.");
try {
for(int count=0; count < 10; count++) {
Thread.sleep(400);
System.out.println("In " + thrd.getName( ) + ", count is " + count);
```

```
} } catch(InterruptedException exc) {
System.out.println(thrd.getName( ) + " interrupted."); }
System.out.println(thrd.getName( ) + " terminating.");
} }
class JoinThreads {
public static void main(String args[ ]) {
System.out.println("Main thread starting.");
MyThread mt1 = new MyThread("Child #1");
MyThread mt2 = new MyThread("Child #2");
MyThread mt3 = new MyThread("Child #3");
try {
// Wait until the specified thread ends
mt1.thrd.join( );
System.out.println("Child #1 joined.");
mt2.thrd.join( );
System.out.println("Child #2 joined.");
mt3.thrd.join( );
System.out.println("Child #3 joined.");
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted."); }
System.out.println("Main thread ending.");
} }
```

اولوية الثريد

يمكن تحديد اولوية لكل ثريد حيث ان الثريد ذو الاوليوية العالية يأخذ وقت اكثر من معالج الحاسوب من الثريد ذو الاولوية القليلة، لكن هذا لا يعني بالضرورة ان يتم تنفيذ الثريد ذو الاولوية العالية في البداية وانما كلما في الامر ستكون فرصته اكبر في التعامل مع المعالج، ولتحديد الاولوية نستخدم الدالة setPriority وهي تأخذ قيم من 1 ويعبر عنها بـ MIN_PRIORITY الى 10 ويعبر عنها بـ MAX_PRIORITY حيث ان 1 تمثل اولوية قليلة و 10 تمثل اولوية عالية والاولوية الاعتيادية للثريد هي 5 ويعبر عنها بـ NORM_PRIORITY ، وللحصول على اولوية ثريد نستخدم الدالة getPriority ، مثال :

```
class Priority implements Runnable {
int count;
Thread thrd;
static boolean stop = false;
static String currentName;
Priority(String name) {
thrd = new Thread(this, name);
count = 0;
currentName = name;
}
public void run( ) {
System.out.println(thrd.getName( ) + " starting.");
do {
count++;
if(currentName.compareTo(thrd.getName( )) != 0) {
currentName = thrd.getName();
System.out.println("In " + currentName); }
} while(stop == false && count < 10000000);
stop = true;
System.out.println("\n" + thrd.getName( ) + " terminating.");
} }
class PriorityDemo {
public static void main(String args[ ]) {
Priority mt1 = new Priority("High Priority");
Priority mt2 = new Priority("Low Priority");
// set the priorities
mt1.thrd.setPriority(Thread.NORM_PRIORITY+2);
mt2.thrd.setPriority(Thread.NORM_PRIORITY-2);
// start the threads
mt1.thrd.start( );
mt2.thrd.start( );
```

```
try {
mt1.thrd.join( );
mt2.thrd.join( );
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted."); }
System.out.println("\nHigh priority thread counted to " + mt1.count);
System.out.println("Low priority thread counted to " + mt2.count);
} }
```

synchronized

في بعض الاحيان نريد لدالة معينة ان لا تنفذ من قبل اكثر من اوبجكت في نفس الوقت من قبل الثريد اي انه عندما يبدأ اوبجكت في تنفيذها فانها ستغلق وبقية الاوبجكتات التي تحاول الوصول اليها ستبقى قيد الانتظار الى ان ينتهي منها الاوبجكت الاول، وهناك طريقتين لعمل هكذا دالة :

استخدام synchronized مع الدوال

عند انشاء اي دالة يمكن ان نصرح عنها على انها synchronized وهذا يعني انه عند انشاء اوبجكت من كلاس هذه الدالة فانه لا يمكن ان نستخدم هذه الدالة وهي قيد التنفيذ كأن نصل لها من اوبجكت اخر الى ان ينتهي الاوبجكت الاول من العمل معها بعدها سيتمكن الاوبجكت الثاني من الوصول والتعامل معها، مثال :

```
class SumArray {
private int sum;
synchronized int sumArray(int nums[ ]) {
sum = 0; // reset sum
for(int i=0; i<nums.length; i++) {
sum += nums[i];
System.out.println("Running total for " + Thread.currentThread().getName() + "
is " + sum);
try {
```

```
Thread.sleep(10); // allow task-switch
} catch(InterruptedException exc) {
System.out.println("Thread interrupted."); } }
return sum;
} }
class MyThread implements Runnable {
Thread thrd;
static SumArray sa = new SumArray( );
int a[ ];
int answer;
MyThread(String name, int nums[ ]) {
thrd = new Thread(this, name);
a = nums;
thrd.start( );
}
public void run( ) {
int sum;
System.out.println(thrd.getName( ) + " starting.");
answer = sa.sumArray(a);
System.out.println("Sum for " + thrd.getName( ) + " is " + answer);
System.out.println(thrd.getName( ) + " terminating.");
} }
class Sync {
public static void main(String args[ ]) {
int a[ ] = {1, 2, 3, 4, 5};
MyThread mt1 = new MyThread("Child #1", a);
MyThread mt2 = new MyThread("Child #2", a);
try {
mt1.thrd.join( );
mt2.thrd.join( );
} catch(InterruptedException exc) {
```

```
System.out.println("Main thread interrupted.");
} } }
```

ستكون نتيجة المثال كالتالي :

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 1
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.
```

كما تلاحظ فانه تم تنفيذ الاوبجكت الاول وبعده نفذ الاوبجكت الثاني وهذا جعل هناك تنظيم في قيمة sum لانها عبارة عن مجموع اعداد المصفوفة، ولاحظ كيف ستكون نتيجة البرنامج اذا حذفنا منه عبارة synchronized ، ستكون بهذا الشكل :

```
Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #2 is 1
Running total for Child #1 is 3
Running total for Child #2 is 5
Running total for Child #2 is 8
```

```
Running total for Child #1 is 11
Running total for Child #2 is 15
Running total for Child #1 is 19
Running total for Child #2 is 24
Sum for Child #2 is 24
Child #2 terminating.
Running total for Child #1 is 29
Sum for Child #1 is 29
Child #1 terminating.
```

استخدام **synchronized** من خارج الدوال

في بعض الاحيان قد لا نتمكن من الوصول لكود الدالة التي نريد ان نصرح على انها synchronized لذلك وجدت هذه الطريقة لتحدد الدالة التي نريدها من خلال بلوك الـ synchronized ، وصيغته العامة هي :

```
synchronized(objref) {
// statements to be synchronized
}
```

حيث ان objref يمثل اوبجكت كلاس الدالة التي نريد تحديدها، لاحظ هذا المثال الذي هو نفس المثال السابق لكن سنحدد الدالة من خارج كلاسها :

```
class SumArray {
private int sum;
int sumArray(int nums[ ]) {
sum = 0; // reset sum
for(int i=0; i<nums.length; i++) {
sum += nums[i];
System.out.println("Running total for " + Thread.currentThread().getName() + "
is " + sum);
```

```
try {
Thread.sleep(10); // allow task-switch
} catch(InterruptedException exc) {
System.out.println("Thread interrupted."); } }
return sum;
} }
class MyThread implements Runnable {
Thread thrd;
static SumArray sa = new SumArray( );
int a[ ];
int answer;
MyThread(String name, int nums[ ]) {
thrd = new Thread(this, name);
a = nums;
thrd.start( ); }
public void run( ) {
int sum;
System.out.println(thrd.getName( ) + " starting.");
// synchronize calls to sumArray( )
synchronized(sa) {
answer = sa.sumArray(a);
}
System.out.println("Sum for " + thrd.getName() + " is " + answer);
System.out.println(thrd.getName() + " terminating.");
} }
class Sync {
public static void main(String args[]) {
int a[ ] = {1, 2, 3, 4, 5};
MyThread mt1 = new MyThread("Child #1", a);
MyThread mt2 = new MyThread("Child #2", a);
try {
```

```
mt1.thrd.join( );
mt2.thrd.join( );
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted.");
} } }
```

اتصالات الثريد بستخدام notify( ), wait( ), notifyAll( )

تخيل اننا عملنا دالة وكانت synchronized وهذه الدالة تتصل بمصدر للبيانات قد لا يكون متوفر في وقت من الاوقات فان اوبجكت الثريد الذي يصل للدالة ولم يجد المصدر سيتوقف لعدم توفر المصدر وسيمنع اي كود يستخدم هذا الاوبجكت لانه سيكون مقفل، جافا توفر حل لهذه المشكلة وهي ان تسمح للثريد الذي يحاول الوصول لدالة synchronized من ان يتنحى (ينام) عن العمل ليسمح لباقي الثريد بالعمل وسيعود للعمل حالما يتوفر مصدر البيانات، هناك دوال يمكن كتابتها في دالخل الـ synchronized للتعامل مع هذه الحالة، الدالة wait تستخدم لجعل الثريد ينتظر (ينام) وسيتم ايقاضه عندما يستخدم ثريد اخر نفس المونيتور ويستدعي الدالة notify او الدالة notifyAll ، ولها هذه الاشكال في الاستخدام :

```
final void wait( ) throws InterruptedException
final void wait(long millis) throws InterruptedException
final void wait(long millis, int nanos) throws InterruptedException
```

الاولى ستبقى تنتظر الى ان يتم ابلاغها، الثاني ستبقى تنتظر الى ان يتم ابلاغها او الفترة المحددة من الملي ثانية تنتهي، الثالثة تتيح امكانية وضع فترة الانتهاء بالنانو ثانية، الصيغة العامة للدوال notify و notifyAll هي :

```
final void notify( )
final void notifyAll( )
```

استدعاء الدالة notify يستأنف عمل ثريد واحدة كانت قيد الانتظار، اما استدعاء الدالة notifyAll يستأنف عمل كل الثريد وستكون الاولوية للثريد التي تحمل اولوية اعلى للوصول الى الاوبجكت، مثال :

```
class TickTock {
String state;
synchronized void tick(boolean running) {
if(!running) {
```

```
state = "ticked";
notify( ); // notify any waiting threads
return;
}
System.out.print("Tick ");
state = "ticked"; // set the current state to ticked
notify( ); // let tock( ) run
try {
while(!state.equals("tocked"))
wait( ); // wait for tock() to complete
}
catch(InterruptedException exc) {
System.out.println("Thread interrupted.");
} }
synchronized void tock(boolean running) {
if(!running) {
state = "tocked";
notify( ); // notify any waiting threads
return;
}
System.out.println("Tock");
state = "tocked"; // set the current state to tocked
notify( ); // let tick( ) run
try {
while(!state.equals("ticked"))
wait( ); // wait for tick to complete
}
catch(InterruptedException exc) {
System.out.println("Thread interrupted.");
} } }
class MyThread implements Runnable {
```

```
Thread thrd;
TickTock ttOb;
MyThread(String name, TickTock tt) {
thrd = new Thread(this, name);
ttOb = tt;
thrd.start();
}
public void run( ) {
if(thrd.getName( ).compareTo("Tick") == 0) {
for(int i=0; i<5; i++) ttOb.tick(true);
ttOb.tick(false);
} else {
for(int i=0; i<5; i++) ttOb.tock(true);
ttOb.tock(false);
} } }
class ThreadCom {
public static void main(String args[ ]) {
TickTock tt = new TickTock( );
MyThread mt1 = new MyThread("Tick", tt);
MyThread mt2 = new MyThread("Tock", tt);
try {
mt1.thrd.join( );
mt2.thrd.join( );
} catch(InterruptedException exc) {
System.out.println("Main thread interrupted.");
} } }
```

ستكون نتيجة المثال بهذا الشكل :

```
Tick Tock
Tick Tock
Tick Tock
```

```
Tick Tock
Tick Tock
```

* تعليق، استاناف وايقاف الثريد ، يمكن عمل ذلك بستخدام الدالة ( )wait والدالة ( )notify ، لاحظ هذا المثال :

```
class MyThread implements Runnable {
Thread thrd;
boolean suspended;
boolean stopped;
MyThread(String name) {
thrd = new Thread(this, name);
suspended = false;
stopped = false;
thrd.start( );
}
// This is the entry point for thread.
public void run( ) {
System.out.println(thrd.getName( ) + " starting.");
try {
for(int i = 1; i < 1000; i++) {
System.out.print(i + " ");
if((i%10)==0) {
System.out.println( );
Thread.sleep(250);
}
// Use synchronized block to check suspended and stopped.
synchronized(this) {
while(suspended) {
wait( );
}
if(stopped) break;
```

```
}
}
} catch (InterruptedException exc) {
System.out.println(thrd.getName( ) + " interrupted.");
}
System.out.println(thrd.getName() + " exiting.");
}
// Stop the thread.
synchronized void mystop( ) {
stopped = true;
// The following ensures that a suspended thread can be stopped.
suspended = false;
notify( );
}
// Suspend the thread.
synchronized void mysuspend() {
suspended = true;
}
// Resume the thread.
synchronized void myresume() {
suspended = false;
notify();
}
}
class Suspend {
public static void main(String args[ ]) {
MyThread ob1 = new MyThread("My Thread");
try {
Thread.sleep(1000); // let ob1 thread start executing
ob1.mysuspend( );
System.out.println("Suspending thread.");
```

```
Thread.sleep(1000);
ob1.myresume( );
System.out.println("Resuming thread.");
Thread.sleep(1000);
ob1.mysuspend( );
System.out.println("Suspending thread.");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread.");
Thread.sleep(1000);
ob1.mysuspend( );
System.out.println("Stopping thread.");
ob1.mystop();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for thread to finish
try {
ob1.thrd.join( );
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

نتيجة البرنامج ستكون :

```
My Thread starting.
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
```

```
31 32 33 34 35 36 37 38 39 40
Suspending thread.
Resuming thread.
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Suspending thread.
Resuming thread.
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Stopping thread.
My Thread exiting.
Main thread exiting.
```

الفصل الحادي عشر (Enumerations)

هي قائمة محددة من الثوابت تعرف نوع جديد من البيانات، وابجكت الـ Enumeration يمكن ان يحتوي على فقط القيم المعرفة بواسطة هذه القائمة، الـ Enumeration موجودة في كل حياتنا اليومية مثلا ايام الاسبوع تعتبر Enumeration وكذلك اشهر السنه إلخ، يتم انشائها من خلال العبارة enum ، لاحظ هذا شكل بسيط لها :

```
enum Transport {
CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

حيث هنا Transport هو اسمها وثوابت القائمة هي CAR, TRUCK, AIRPLANE, TRAIN, BOAT وكل واحدة منها معرفة بشكل ضمني على انها public و static ، ويمكن ان ننشأ متغيرات من هذه الـ Enumeration بهذا الشكل :

```
Transport tp;
```

هنا انشأنا متغير اسمه tp من القائمة Transport ، وبما ان المتغير tp عرف على انه من النوع Transport لذا فهو يحمل فقط القيم المعرفة في الـ Transport ، مثلا سنسند له القيمة AIRPLANE بهذا الشكل :

```
tp = Transport.AIRPLANE;
```

ولعمل تحقق من قيمة المتغير عبر العبارة if الشرطية نكتب كالتالي :

```
if(tp == Transport.TRAIN) // ...
```

وكذلك يمكن ان نعمل مقارنه بعبارة switch لكن عبارات case الخاصة بها كلها يجب ان تحمل القيم التي تم تعريفها في القائمة Enumeration لذلك لا يوجد داعي ان نسبق القيمة باسم الـ Enumeration مثال :

```
switch(tp) {
case CAR:
// ...
case TRUCK:
```

```
// ...
```

لاحظ انه بعد case لم نضع Transport.CAR بل وضعنا فقط CAR ، لكن اذا اردنا ان نعرض متغير على الشاشة نستخدم اسم الـ Enumeration قبل المتغير لاحظ :

```
System.out.println(Transport.BOAT);
```

هذه الجملة ستعرض لنا كلمة BOAT على الشاشة.

* الـ Enumeration في لغة جافا هي نوع من انواع الكلاسات اي بعبارة ثانية انه ينطبق عليها كل ما ينطبق على الكلاس اي يمكن ان نضع فيها constructors ودوال ومتغيرات وكذلك يمكن عمل implement للانترفيسز، لكن فرقها الوحيد عن الكلاس هو عند انشاء الاوبجكت منها (كما رأينا في الامثلة السابقة) فاننا لا نستخدم العبارة new في انشاء الاوبجكت هكذا :

```
Transport tp;
tp = Transport.AIRPLANE;
```

او هكذا :

```
Transport tp = Transport.AIRPLANE;
```

الدوال values( ) و valueOf( )

كل enumeration تعرف بشكل اوتوماتيكي دالتين هما values و values وصيغتهم العامة هي :

```
public static enum-type[ ] values( )
public static enum-type valueOf(String str)
```

الدالة values تعيد مصفوفة تحتوي على اسماء ثوابت القائمة enumeration ، اما الدالة valueOf تعيد الثابت في قائمة الـ enumeration الذي يتوافق مع str الموجود بين قوسيها ، وفي كلا الحالتين enum-type يمثل نوع (اسم) الـ enumeration ، مثال :

```
enum Transport {
CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
class EnumDemo2 {
```

```
public static void main(String args[ ]) {
Transport tp;
Transport allTransports[ ] = Transport.values( );
for(Transport t : allTransports)
System.out.println(t);
tp = Transport.valueOf("AIRPLANE");
System.out.println("tp contains " + tp);
} }
```

نتيجة البرنامج ستكون :

```
CAR
TRUCK
AIRPLANE
TRAIN
BOAT
tp contains AIRPLANE
```

في هذا المثال استخدمنا حلقة التكرار for بهذه الصيغة لاننا نريد ان نتعامل مع مصفوفة وقد تم شرح هذه الصيغة من for في فصل حلقات التكرار في بداية هذا الكتاب ، allTransports هو اسم المصفوفة التي انشأناها لتحمل القيم .

الدوال، المتغيرات، الـ Constructors والـ enumeration

من المهم ان نعرف ان كل ثابت في القائمة العددية enumeration هو عبارة عن اوبجكت من نوع هذا الـ enumeration لذلك يمكن وضع متغيرات ودوال وconstructor في داخل الـ enumeration ، مثال:

```
enum Transport {
CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70), BOAT(22);
private int speed;
Transport(int s) { speed = s; }
int getSpeed( ) { return speed; }
}
```

```
class EnumDemo3 {
public static void main(String args[ ]) {
Transport tp;
System.out.println("speed for an airplane is " + Transport.AIRPLANE.getSpeed( )
+
 " miles per hour.");
System.out.println("All Transport speeds: ");
for(Transport t : Transport.values( ))
System.out.println(t + " typical speed is " + t.getSpeed() + " miles per hour.");
} }
```

نتيجة البرنامج ستكون :

```
speed for an airplane is 600 miles per hour.
All Transport speeds:
CAR typical speed is 65 miles per hour.
TRUCK typical speed is 55 miles per hour.
AIRPLANE typical speed is 600 miles per hour.
TRAIN typical speed is 70 miles per hour.
BOAT typical speed is 22 miles per hour.
```

في هذا المثال لاحظ عدة امور قد طرأت اول شيء ان ثوابت القائمة انتهت بفارزة منقوطة ، ثم ان كل ثابت من هذه الثوابت وضعنا بين قوسيه قيمة هذه القيمة ستذهب مباشرتا الى الـ constructor وتمثل ما بين قوسيها والذي هو s وهذه ستسند الى المتغير speed ، لذلك سنتمكن من الوصول لقيمة كل ثابت من خلال الدالة getSpeed ، في هذا المثال كان هناك فقط constructor واحدة لكن يمكن ان نضع اكثرمن واحدة كأي كلاس عادي .

* هناك بعض القيود تطبق على الـ enumeration وهي انه لا يمكنها ان ترث كلاس او تورث كلاس.

الكلاس Enum

كل الكلاسات التي نعرفها على انها من النوع enumeration ترث بشكل اوتوماتيكي من الكلاس Enum والذي يعرف عدة دوال منها الدالة ordinal والتي تعيد رقم يمثل تسلسل الثابت في القائمة enumeration ولاحظ ان التسلسل يبدأ من الرقم صفر وصيغتها العامة هي :

```
final int ordinal( )
```

الدالة compareTo تستخدم للمقارنة بين الثوابت التي تنتمي لنفس القائمة enumeration ، صيغتها العامة هي :

```
final int compareTo(enum-type e)
```

حيث ان enum-type يمثل نوع الـ enumeration و e يمثل الثابت الذي نريد المقارنة به، اذا كان تسلسل الثابت الذي نقارنه اقل من الثابت e فان الدالة تعيد قيمة سالبة، اما اذا كان كلا الثابتين لهم نفس التسلسل تعيد صفر واذا كان تسلسل الثابت الذي نقارنه اكبر من الثابت e فان الدالة تعيد قيمة موجبة.

مثال :

```
enum Transport {
CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
class EnumDemo4 {
public static void main(String args[ ]) {
Transport tp, tp2, tp3;
for(Transport t : Transport.values( ))
System.out.println(t + " " + t.ordinal( ));
tp = Transport.AIRPLANE;
tp2 = Transport.TRAIN;
tp3 = Transport.AIRPLANE;
if(tp.compareTo(tp2) < 0)
System.out.println(tp + " comes before " + tp2);
if(tp.compareTo(tp2) > 0)
System.out.println(tp2 + " comes before " + tp);
if(tp.compareTo(tp3) == 0)
```

```
System.out.println(tp + " equals " + tp3);
} }
```

ستكون مخرجات البرنامج بهذا الشكل :

```
CAR 0
TRUCK 1
AIRPLANE 2
TRAIN 3
BOAT 4
AIRPLANE comes before TRAIN
AIRPLANE equals AIRPLANE
```

انواع التغليف Wrappers

تحدثنا في فصل الـ I/O عن كيفية تحويل الارقام النصية الى عددية بستخدام دوال وكلاسات التغليف، والان سنتحدث عن التغليف بشكل عام اكثر، انواع كلاسات التغليف هي : Double, Float, Long, Integer, Short, Byte, Character, Boolean وهذه الكلاسات موجودة في الحزمة java.lang ، تحتوي على مجموعة واسعة من الدوال للتعامل مع انواع المتغيرات، ربما الكلاسات الاكثر استخداما هي التي تتعامل مع الاعداد والكلاسات العددية كلها ترث الكلاس Number والذي هو من النوع abstract ، هذا الكلاس يعرف مجموعة من الدوال التي تعيد من اوبجكت ما قيمة عددية من انواع مختلفة من الاعداد، وهذه الدوال هي :

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

مثلا الدالة doubleValue تعيد قيمة من اوبجكت كـ double ، وكذلك الامر بالنسبة للدالة floatValue تعيد قيمة من النوع float وهكذا مع بقية الدوال، كل كلاسات التغليف تعرف constructor يسمح بتمرير قيم لاوبجكتها وتكون القيمة عددية او اعداد نصية، لنأخذ مثال عن الـ Integer :

```
Integer(int num)
Integer(String str) throws NumberFormatException
```

اذا الـ str لم يكن يحتوي على عدد فانها سترمي استثناء من النوع NumberFormatException .

عملية تغليف القيمة في داخل الاوبجكت تسمى boxing ، مثال :

```
Integer iOb = new Integer(100);
```

هنا الاوبجكت iOb ستكون قيمته 100 ، وعملية استخراج القيمة منه الى متغير تسمى unboxing ، مثال :

```
int i = iOb.intValue( );
```

عملية الـ boxing والـ unboxing كانت تجرى بشكل يدوي في اصدارات جافا قبل JDK 5 ولكن ما بعدها اصبحت بشكل اوتوماتيكي.

Autoboxing و Auto-unboxing

كما ذكرت قبل قليل انه في السابق كانت عملية التغليف بشكل يدوي اما الان فان لغة جافا تجريها بشكل اوتوماتيكي، كل ماعلينا فعله هو تحديد كلاس التغليف المناسب، مثلا لتغليف قيمة في اوبجكت (autobox) نكتب :

```
Integer iOb = 100;
```

ولاستخراج قمية من اوبجكت (autobox) نكتب :

```
int i = iOb;
```

مثال :

```
class AutoBox2 {
static void m(Integer v) {
System.out.println("m( ) received " + v);
}
static int m2( ) {
return 10;
}
```

```
static Integer m3( ) {
return 99;
}
public static void main(String args[ ]) {
m(199);
Integer iOb = m2( );
System.out.println("Return value from m2( ) is " + iOb);
int i = m3( );
System.out.println("Return value from m3( ) is " + i);
iOb = 100;
System.out.println("Square root of iOb is " + Math.sqrt(iOb));
} }
```

ستكون مخرجات البرنامج بهذا الشكل :

```
m( ) received 199
Return value from m2( ) is 10
Return value from m3( ) is 99
Square root of iOb is 10.0
```

لاحظ ان عملية الـ Autoboxing و Auto-unboxing ستجعلنا نتعامل مع اوبجكت كلاسات التغليف وكأنها متغيرات اي يمكن ان نضعها في عمليات حسابية او نمررها الى دوال او الى حلقات تكرار او مقارنة وما الى ذلك .

Annotations

جافا تتيح لنا امكانية ان نضمن معلومات اضافية في الملف المصدر، هذه المعلومات تدعى annotation وهي لا تؤثر على مجرى البرنامج وفائدتها يمكن ان تكون في اعطاء معلومات عن مصمم البرنامج الخ، وهي تنشأ بالاعتماد على interface بهذا الشكل :

```
@interface MyAnno {
String str( );
int val( );
```

```
}
```

هنا اعلنا عن annotation اسمها MyAnno ، كل الـ annotation تحتوي فقط على دوال لكننا لا نكتب جسد الدوال لكن جافا تطبق هذه الدوال ، كل الـ annotation ترث بشكل اوتوماتيكي الانترفيس Annotation وهذا النترفيس موجود في الحزمة java.lang.annotation ، يمكن اعطاء قيم للـ annotation مثلا بهذا الشكل :

```
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth( ) { // ...
```

هذه الـ annotation مرتبطة مع الدالة myMeth( )

الفصل الثاني عشر (Generics)

Generics هي كلاسات تتيح لنا امكانية عمل انواع من البيانات على شكل اوبجكت لكن هذا الاوبجكت ممكن ان يحمل اكثر من نوع فمثلا يمكن ان يكون Integer ومرة اخرى يكون Double وهكذا، كل ما علينا فعله هو كتابة اسم الكلاس وبعده مباشرتا قوسين هكذا < > وفي داخلهما اسم مستعار للنوع الذي نريد عمله وبداخل الكلاس نعمل متغيرات او دوال، وعند انشاء اوبجكت من هذا الكلاس نمرر اسم النوع الذي نريده بدلا من الاسم المستعار الذي وضعناه، لاحظ هذا المثال :

```
class Gen<T> {
T ob; // declare an object of type T
Gen(T o) {
ob = o; }
T getob( ) {
return ob; }
void showType( ) {
System.out.println("Type of T is " + ob.getClass().getName());
} }
class GenDemo {
public static void main(String args[ ]) {
Gen<Integer> iOb = new Gen<Integer>(88);
iOb.showType( );
int v = iOb.getob();
System.out.println("value: " + v);
Gen<String> strOb;
strOb = new Gen<String>("Generics Test");
strOb.showType( );
String str = strOb.getob( );
System.out.println("value: " + str);
```

```
} }
```

مخرجات البرنامج ستكون كالتالي :

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

يمكن وضع اكثر من نوع متغير بين قوسي الكلاس والفصل بينهم بفاصلة، مثال :

```
class Gen<T, S> {
T ob;
S ob2;
void showType( ) {
System.out.println("Type of T is " + ob.getClass().getName());
System.out.println("Type of S is " + ob2.getClass().getName());
} }
class GenDemo {
public static void main(String args[ ]) {
Gen<Integer, String> iOb = new Gen<Integer, String>();
iob.T = 22;
iob.S = "Hello"
iOb.showType( );
} }
```

* لا يمكن تمرير الانواع int او char او الانواع من هذا الشكل الى قوسي الكلاس Generics انما يمكن تمرير اي نوع من انواع الكلاسات مثل الانواع التالية : Double, Float, Long, Integer, Short, Byte, Character, Boolean

* يمكن تحديد الانواع التي يمكن تمريرها الى الكلاس Generics وذلك من خلال العبارة extends وبعدها اسم الكلاس الذي نريد الانواع ان تكون من ضمنه، مثال:

```
class NumericFns<T extends Number> { //…
```

هنا الانواع التي يمكن تمريرها للاوبجكت الذي سينشأ من هذا الكلاس يجب ان تكون من الكلاس Number او من الكلاسات التي تشتق من الكلاس Number مثل Integer, Double الخ لكن لا يمكن ان تكون String مثلا، او يمكن ان نحدد النوع بهذا الشكل :

```
class Pair<T, V extends T> {
T first;
V second;
Pair(T a, V b) {
first = a;
second = b;
} }
```

هنا النوع V يجب ان يكون من نوع الكلاس T او من الكلاسات المشتقة منه، لذا يمكن عمل الاوبجكت بهذا الشكل :

```
Pair<Integer, Integer> x = new Pair<Integer, Integer>(1, 2);
Pair<Number, Integer> y = new Pair<Number, Integer>(10.4, 12);
```

لكن لا يمكن ان تكون بهذا الشكل مثلا :

```
Pair<Number, String> z = new Pair<Number, String>(10.4, "12");
```

Wildcard Arguments

عند تحديد الانواع التي نمررها الى الكلاس Generic قد تواجهنا مشكلة، اذا كان الكلاس مثلا بهذا الشكل :

```
class NumericFns<T extends Number> { //…
```

وعملنا بداخل هذا الكلاس constructor واردنا ان نضع فيه دالة اسمها absEqual وضيفتها تقارن بين القيم المطلقة لعددين وتعيد القيمة true اذا كانا متساويين قد يكون الاول من النوع Double والثاني من النوع Float ، الاوبجكتات التي نعملها من الكلاس NumericFns ستكون بهذا الشكل :

```
NumericFns<Double> dOb = new NumericFns<Double>(1.25); // Double
```

```
NumericFns<Float> fOb = new NumericFns<Float>(-1.25); // Float
if(dOb.absEqual(fOb))
System.out.println("Absolute values are the same.");
else
System.out.println("Absolute values differ.");
```

المشكله هنا كيف سكون شكل الدالة absEqual لانه لا يمكن ان نعملها بهذا الشكل :

```
boolean absEqual(NumericFns<T> ob) {
if(Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()) return
true;
return false;
}
```

لانه بهذا الشكل البارامیتر الممرر لها عند استدعائها يجب ان يكون من النوع T فاذا كان العدد الاول من النوع Integer فيجب ان يكون الثاني من نفس النوع، اي ان الاوبجكت dOb و الاوبجكت fOb من نفس النوع، ولحل هذه المشكل جائت لنا جافا بـ Wildcard والتي تكتب بشكل علامة الاستفهام اي ان الدالة absEqual سيكون شكلها بهذا الشكل :

```
boolean absEqual(NumericFns<?> ob) {  // …
```

هنا NumericFns<?> تطابق اي اوبجكت يمرر للكلاس NumericFns ، لتتضح الصورة اكثر لاحظ هذا المثال :

```
class NumericFns<T extends Number> {
T num;
NumericFns(T n) {
num = n; }
double reciprocal( ) {
return 1 / num.doubleValue( ); }
double fraction( ) {
return num.doubleValue( ) - num.intValue( ); }
boolean absEqual(NumericFns<?> ob) {
if(Math.abs(num.doubleValue( )) ==
```

```
        Math.abs(ob.num.doubleValue( ))) return true;
return false;
} }
class WildcardDemo {
public static void main(String args[ ]) {
NumericFns<Integer> iOb = new NumericFns<Integer>(6);
NumericFns<Double> dOb = new NumericFns<Double>(-6.0);
NumericFns<Long> lOb = new NumericFns<Long>(5L);
if(iOb.absEqual(dOb))
System.out.println("Absolute values are equal.");
else
System.out.println("Absolute values differ.");
if(iOb.absEqual(lOb))
System.out.println("Absolute values are equal.");
else
System.out.println("Absolute values differ.");
} }
```

حدود الـ Wildcard

يمكن تقييد نطاق الـ Wildcard في بارامتر الدالة، مثلا لنفترض ان لدينا الكلاس التالي :

```
class Gen<T> {  //…
```

هنا يمكن ان نمرر للكلاس اي اوبجكت من اي نوع، لكن اذا اردنا ان نعمل دالة في داخل هذا الكلاس تاخذ بارامتر اوبجكت من نوع محدد او من الكالسات المشتقة من هذا النوع وليس مثل الكلاس يمكن عمل ذلك بهذا الشكل :

```
void test(Gen<? extends Number > o) { /…
```

هنا الدالة يمكن ان يمرر لها بارامتر من النوع Number او الكلاسات المشتقة منه ولا يمكن تمرير غير ذلك على خلاف الكلاس Gen الذي يحتويها والذي يمكن ان يأخذ من النوع String مثلا، لذلك فالصيغة العامة لهذه الحالة هي :

<? extends superclass>

كذلك يمكن تحد كلاس معين او الكلاسات الاباء لهذا النوع اي عكس السابق بدلا من التكون الكلاسات المشتقة ستكون الكلاسات الذي هو يشتق منها وذلك بستخدام العبارة super والصيغة العامة لهذه الحالة هي:

<? super subclass>

الدوال من النوع Generic

كما لاحظنا سابقا فانه يمكن ان ننشأ دوال في داخل كلاس الـ Generic ويمرر لها بارامترات من نفس نوع كلاسها، لكن يمكننا ان ننشأ دوال تأخذ انواع غير كلاسها او انشاء دوال Generic في داخل كلاس عادي (ليس Generic) ، مثال :

```
class GenericMethodDemo {
static <T extends Comparable<T>, V extends T> boolean arraysEqual(T[] x, V[] y) {
if(x.length != y.length) return false;
for(int i=0; i < x.length; i++)
if(!x[i].equals(y[i])) return false;
return true;
}
public static void main(String args[]) {
Integer nums[] = { 1, 2, 3, 4, 5 };
Integer nums2[] = { 1, 2, 3, 4, 5 };
Integer nums3[] = { 1, 2, 7, 4, 5 };
Integer nums4[] = { 1, 2, 7, 4, 5, 6 };
if(arraysEqual(nums, nums)) System.out.println("nums equals nums");
if(arraysEqual(nums, nums2)) System.out.println("nums equals nums2");
if(arraysEqual(nums, nums3)) System.out.println("nums equals nums3");
if(arraysEqual(nums, nums4)) System.out.println("nums equals nums4");
Double dvals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
// This won't compile because nums and dvals are not of the same type.
// if(arraysEqual(nums, dvals)) System.out.println("nums equals dvals");
```

```
} }
```

نتيجة البرنامج ستكون :

```
nums equals nums
nums equals nums2
```

لاحظ العبارة Comparable<T> حيث ان Comparable هو اسم انترفيس موجود في الحزمة java.lang وضعناه هنا لاننا في مثالنا هذا نريد ان نقارن بين مصفوفات وهذا لضمان ان الاوبجكت الذي سيمرر يستخدم فقط للمقارنة، هذا الانترفيس هو من النوع generic ونوع بارامتراته يحدد نوع الاوبجكت الذي يقارن، ولاحظ ايضا ان الدالة arraysEqual عندما تم استدعائها في الدالة main استدعيت بشكل اعتيادي كأي دالة ليست من انوع generic ، وفي نهاية البرنامج وضعنا السطر الاخير في تعليق فاذا ازلنا عنه علامة التعليق ونفذناه فانه سيحدث خطأ لانه سيكون نوع الباراميتر الثاني الممرر للدالة ليس نفس نوع الباراميتر الاول لاننا صرحنا في الدالة arraysEqual ان الباراميتر الثاني يجب ان يكون نفس الاول او مشتق من الاول V extends T ، لاحظ ان الصيغة العامة للدالة من النوع generic تكون كالتالي :

```
<type-param-list> ret-type meth-name(param-list) { // ...
```

Constructor من النوع generic

الـ Constructor يمكن ان تكون من النوع generic حتى وان كان الكلاس الذي يحتويها ليس من هذا النوع ، مثال :

```
class Summation {
private int sum;
<T extends Number> Summation(T arg) {
sum = 0;
for(int i=0; i <= arg.intValue(); i++)
sum += i;
}
int getSum() {
return sum;
} }
class GenConsDemo {
```

```
public static void main(String args[]) {
Summation ob = new Summation(4.0);
System.out.println("Summation of 4.0 is " + ob.getSum());
} }
```

انترفيس من النوع generic

يمكن عمل انترفيس من النوع generic بنفس الطريقة التي نعمل بها كلاس من النوع generic وصيغته العامة بهذا الكشل :

```
interface interface-name<type-param-list> { // ...
```

مثال :

```
interface Containment<T> {
boolean contains(T o);
}
class MyClass<T> implements Containment<T> {
T[ ] arrayRef;
MyClass(T[ ] o) {
arrayRef = o; }
public boolean contains(T o) {
for(T x : arrayRef)
if(x.equals(o)) return true;
return false;
} }
class GenIFDemo {
public static void main(String args[ ]) {
Integer x[ ] = { 1, 2, 3 };
MyClass<Integer> ob = new MyClass<Integer>(x);
if(ob.contains(2))
System.out.println("2 is in ob");
```

```
else
System.out.println("2 is NOT in ob");
if(ob.contains(5))
System.out.println("5 is in ob");
else
System.out.println("5 is NOT in ob");
/* The following is illegal because ob is an Integer Containment and 9.25 is a
Double value. */
// if(ob.contains(9.25)) System.out.println("9.25 is in ob");
} }
```

نتيجة البرنامج ستكون :

```
2 is in ob
5 is NOT in ob
```

اي كلاس يعمل implements لانترفيس من نوع generic يجب ان يكون الكلاس نفسه ايضا من النوع generic او على الاقل يجب ان يكون الانترفيس يعرف بارامترات من نوع معروف، مثلا تعريف كلاس بهذا الكشل خطأ :

```
class MyClass implements Containment<T> { // Wrong!
```

لان T غير معروف ما هو لكن يمكن تعريف الكلاس بهذا الكشل :

```
class MyClass implements Containment<Double> { // OK
```

* اذا حددنا نوع البارامترات للانترفيس مثلا بهذا الشكل :

```
interface Containment<T extends Number> {
```

هنا فان الكلاس الذي يجلب الانترفيس يجب ان يكون من نفس نوع البارامترات بهذا الكشل :

```
class MyClass<T extends Number> implements Containment<T> {
```

لاحظ كيف كتبنا فقط T في الانترفيس وذلك لانها نفس الـ T التي حددناها في الكلاس يجب ان تكون مشتقة من Number ، لاحظ انه لا يمكن ان نكتب الكلاس بهذ الكشل :

```
class MyClass<T extends Number> implements Containment<T extends Number> {
```

النوع raw

يتم تعريف النوع raw من الكلاس generic كما في هذا المثال :

```
class Gen<T> {
T ob;Gen(T o) {
ob = o; }
T getob( ) {
return ob; }
}
class RawDemo {
public static void main(String args[ ]) {
Gen<Integer> iOb = new Gen<Integer>(88);
Gen<String> strOb = new Gen<String>("Generics Test");
Gen raw = new Gen(new Double(98.6));
double d = (Double) raw.getob( );
System.out.println("value: " + d);
// int i = (Integer) raw.getob(); // run-time error
// strOb = raw; // OK, but potentially wrong
// String str = strOb.getob(); // run-time error
// raw = iOb; // OK, but potentially wrong
// d = (Double) raw.getob(); // run-time error
} }
```

ملاحظات عامة عن الـ generic

* كنا نكتب في امثلة هذا الفصل عند تكوين اوبجكت من كلاس generic بهذا الشكل :

```
TwoGen<Integer, String> tgOb = new TwoGen<Integer, String>(42, "testing");
```

اما في JDK7 فما فوق اصبح بالامكان اختصار العملية بهذا الشكل :

```
TwoGen<Integer, String> tgOb = new TwoGen<>(42, "testing");
```

* عند عمل دالة overloaded في داخل كلاس generic وكان ياخذ اكثر من باراميتر لكن كلها غير محددة فانه لايمكن ان نمرر البارامترات للدوال وهي تأخذ نفس الاسم لان المفسر سوف لن يستطيع التمييز بينهما، مثال :

```
class MyGenClass<T, V> {
T ob1;
V ob2;
void set(T o) {
ob1 = o; }
void set(V o) {
ob2 = o;}
}
```

هذا المثال خطأ لان T و V على الرغم من انهم ليسو من نفس النوع لكنهم يمكن ان يكونوا من نفس النوع.

* لا يمكن ان نعمل بارامترات الكلاس generic من النوع static ، لاحظ هذا المثال خطأ :

```
class Wrong<T> {
static T ob; // illegal
static T getob( ) {  // illegal
return ob;
} }
```

لكن يمكن ان نعرف دالة من النوع generic كـ static هكذا :

```
static <T> int Meth(T) { //...
```

* هناك بعض القيود في التعامل مع المصفوفات في الـ generic ، هذا المثال يوضح ذلك :

```
class Gen<T extends Number> {
T ob;
T vals[ ]; // OK
Gen(T o, T[ ] nums) {
ob = o;
// vals = new T[10]; // illegal, can't create an array of T
vals = nums; // OK to assign reference to existent array
} }
class GenArrays {
public static void main(String args[ ]) {
Integer n[ ] = { 1, 2, 3, 4, 5 };
Gen<Integer> iOb = new Gen<Integer>(50, n);
// Can't create an array of type-specific generic references.
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
Gen<?> gens[] = new Gen<?>[10]; // This is OK.
} }
```

الفصل الثالث عشر (Lambda)

الـ Lambda هو موضوع اضيف في JDK8 ، وهو يتكون من جزئين الاول هو تعبير الـ Lambda وهو عبارة عن دالة مجهولة الاسم والثاني هو انترفيس وظيفي يحتوي على دالة abstract واحدة فقط وينفذ دالة الـ Lambda.

تعبير الـ Lambda

تعبير الـ Lambda يحتوي على lambda operator وهو عبارة عن سهم -> يقسم الدالة الى قسمين، قسم اليسار يحتوي على البارامترات التي تمرر الى الدالة (اذا كان باراميتر واحد يمكن ان لا يوضع بين اقواس وبخلاف ذلك يجب ان نضع الاقواس) وقسم اليمين يحتوي على جسد الدالة (اذا كان جسد الدالة سطر واحد يمكن ان لا يوضع في بلوك) مثال :

```
( ) -> 98.6
```

قوسي الدالة فارغين لانها لا تحتوي على اي بارامترات هذه الدالة هي مشابهه تماما لهذه الدالة :

```
double myMeth( ) { return 98.6; }
```

عدا ان دالة الـ Lambda لا تحتوي على اسم، دالة الـ Lambda يمكن ان تكون (تعيد) اي نوع على حسب القيم الموجودة بها، مثلا :

```
(n) -> (n % 2)==0
```

تعيد القيمة true اذا كانت n عدد زوجي اي ان الدالة من النوع Boolean وتم معرفه ذلك بشكل اوتوماتيكي من محتويات الدالة، وكذلك يمكن تحديد نوع بارامترات الدالة بشكل اوتوماتيكي، مثال :

```
(n) -> 1.0 / n;
```

هنا ان تكون من النوع Double ، ولاحظ ايضا انه يمكن ان نحدد نوع الباراميتير ايضا بهذا الشكل :

```
(double n) -> 1.0 / n;
```

على الرغم من انه ليس ضروري تحديد نوع البارامترات في الغالب، ومن المهم ان تلاحظ ايضا انه اذا كانت الدالة تحتوي على اكثر من باراميتر فانه اما ان تتم تحديد كل انواعهن او لا يتم تحديد الكل، هذا الشكل صحيح :

```
(int n, int d) -> (n % d) == 0
```

لكن هذه الشكل غير صحيح :

```
(int n, d) -> (n % d) == 0
```

Functional Interfaces

هو انترفيس وظيفي ينفذ دالة الـ Lambda يحتوي على دالة abstract واحدة فقط ، مثال لانترفيس وظيفي:

```
interface MyValue {
double getValue( );
}
```

لاحظ ان الدالة ستكون abstract بشك افتراضي لانه ليس لها جسد وهي الدالة الوحيدة لذا هذا الانترفيس وظيفي ووظيفته تحدد بواسطة هذه الدالة، من هذا الانترفيس نعمل مرجع بهذا الشكل :

```
MyValue myVal;
```

والان نعين دالة الـ Lambda في الانترفيس :

```
myVal = ( ) -> 98.6;
```

دالة الـ Lambda هذه متوافقة مع الدالة getValue لان كلاهما لا يحتوي على بارامترات وكلاهما يعيد النوع Double ، يجب ان تكون دالة الـ Lambda ودالة الـ abstract في الانترفيس الوظيفي متوافقات (اي يحتويان على نفس العدد من البارامترات وبارامتراتهم من نفس النوع وكذلك الدالتان تعيدان نفس النوع) وإلا فان خطأ سيحدث، لاحظ ان العبارتين السابقتين يمكن دمجهم معا بهذا الكشل :

```
MyValue myVal = ( ) -> 98.6;
```

ويمكن الوصول الى دالة الانترفيس بهذا الشكل :

```
System.out.println("A constant value: " + myVal.getValue( ));
```

مثال :

```
interface NumericTest {
boolean test(int n, int m);
```

```
}
class LambdaDemo2 {
public static void main(String args[ ]) {
NumericTest isFactor = (n, d) -> (n % d) == 0;
if(isFactor.test(10, 2))
System.out.println("2 is a factor of 10");
if(!isFactor.test(10, 3))
System.out.println("3 is not a factor of 10");
NumericTest lessThan = (n, m) -> (n < m);
if(lessThan.test(2, 10))
System.out.println("2 is less than 10");
if(!lessThan.test(10, 2))
System.out.println("10 is not less than 2");
NumericTest absEqual = (n, m) -> (n < 0 ? -n : n) == (m < 0 ? -m : m);
if(absEqual.test(4, -4))
System.out.println("Absolute values of 4 and -4 are equal.");
if(!lessThan.test(4, -5))
System.out.println("Absolute values of 4 and -5 are not equal.");
} }
```

ستكون مخرجات البرنامج بهذا الشكل :

```
2 is a factor of 10
3 is not a factor of 10
2 is less than 10
10 is not less than 2
Absolute values of 4 and -4 are equal.
Absolute values of 4 and -5 are not equal.
```

* كما ذكرنا في البداية فانه يمكن ان نضع جسد الدالة Lambda في بلوك لتنفيذ اكثر من جملة في الدالة، لاحظ انه يجب ان نضع في داخل البلوك العبارة return لتعود لنا بقيمة، مثال :

```
interface NumericFunc {
```

```
int func(int n);
}
class BlockLambdaDemo {
public static void main(String args[ ]) {
// This block lambda returns the smallest positive factor of a value.
NumericFunc smallestF = (n) -> {
int result = 1;
// Get absolute value of n.
n = n < 0 ? -n : n;
for(int i=2; i <= n/i; i++)
if((n % i) == 0) {
result = i;
break;
}
return result;
} ;
System.out.println("Smallest factor of 12 is " + smallestF.func(12));
System.out.println("Smallest factor of 11 is " + smallestF.func(11));
} }
```

لاحظ كيف وضعنا فارزة منقوطة بعد القوس المعقوف لبلوك دالة الـ Lambda ، ستكون مخرجات البرنامج بهذا الشكل :

```
Smallest factor of 12 is 2
Smallest factor of 11 is 1
```

Generic Functional Interfaces

لا يمكن عمل دالة Lambda من النوع Generic لكن يمكن عمل انترفيس وظيفي من النوع Generic ، مثال :

```
interface SomeTest<T> {
boolean test(T n, T m);
}
class GenericFunctionalInterfaceDemo {
public static void main(String args[ ]) {
SomeTest<Integer> isFactor = (n, d) -> (n % d) == 0;
if(isFactor.test(10, 2)) System.out.println("2 is a factor of 10");
SomeTest<Double> isFactorD = (n, d) -> (n % d) == 0;
if(isFactorD.test(212.0, 4.0)) System.out.println("4.0 is a factor of 212.0");
SomeTest<String> isIn = (a, b) -> a.indexOf(b) != -1;
String str = "Generic Functional Interface";
if(isIn.test(str, "face"))
System.out.println("'face' is found.");
else
System.out.println("'face' not found.");
} }
```

ستكون مخرجات البرنامج بهذا الشكل :

```
2 is a factor of 10
4.0 is a factor of 212.0
'face' is found.
```

* يمكن تمرير دالة Lambda كبارامتر لدالة اخرى، مثال :

```
interface StringFunc {
String func(String str);
}
class LambdaArgumentDemo {
```

```
static String changeStr(StringFunc sf, String s) {
return sf.func(s);
}
public static void main(String args[ ]) {
String inStr = "Lambda Expressions Expand Java";
String outStr;
System.out.println("Here is input string: " + inStr);
StringFunc reverse = (str) -> {
String result = "";
for(int i = str.length( )-1; i >= 0; i--)
result += str.charAt(i);
return result;
};
outStr = changeStr(reverse, inStr);
System.out.println("The string reversed: " + outStr);
outStr = changeStr((str) -> str.replace(' ', '-'), inStr);
System.out.println("The string with spaces replaced: " + outStr);
outStr = changeStr((str) -> {
String result = "";
char ch;
for(int i = 0; i < str.length( ); i++ ) {
ch = str.charAt(i);
if(Character.isUpperCase(ch))
result += Character.toLowerCase(ch);
else
result += Character.toUpperCase(ch);
}
return result;
}, inStr);
System.out.println("The string in reversed case: " + outStr);
} }
```

ستكون مخرجات البرنامج بهذا الشكل :

```
Here is input string: Lambda Expressions Expand Java
The string reversed: avaJ dnapxE snoisserpxE adbmaL
The string with spaces replaced: Lambda-Expressions-Expand-Java
The string in reversed case: lAMBDA eXPRESSIONS eXPAND jAVA
```

نطاق متغيرات Lambda

دالة Lambda يمكن ان تستخدم المتغيرات الموجودة في الكلاس الذي يحتويها لكنها لا تستطيع تغيير قيمة هذه المتغيرات، بعض المتغيرات تكون بشكل ضمني من النوع final حتى اذا لم نصرح عنها علانيتا لذا فلن يكون بالامكان تغيير قيمتها، مثال :

```
interface MyFunc {
int func(int n);
}
class VarCapture {
public static void main(String args[ ]) {
int num = 10;
MyFunc myLambda = (n) -> {
int v = num + n; // OK
// num++; // illegal
return v;
};
System.out.println(myLambda.func(8));
// num = 9; // error, because it would remove the effectively final status from num
} }
```

* يمكن تمرير مصفوفة كباراميتر لدالة Lambda لتوضح الفكرة لاحظ هذا الانترفيس الوظيفي من نوع generic :

```
interface MyTransform<T> {
void transform(T[ ] a); }
```

نكون منه دالة Lambda بهذا الشكل :

```
MyTransform<Double> sqrts = (v) -> {
for(int i=0; i < v.length; i++) v[i] = Math.sqrt(v[i]);
};
```

هنا اصبح باراميتر الدالة عبارة عن مصفوفة من النوع Double ، ولاحظ اننا لم نكتب الباراميتر بهذا الشكل v[ ] (هذا الشكل سيحدث خطأ) بل كتبنا فقط v ولكن ايضا يمكن كتابته عن طريق تحديده بهذا الشكل Double[ ] v .

رمي استثناء من داخل تعبير الـ Lambda

يمكن لدالة Lambda ان ترمي استثناء ولكن الاستثناء يجب ان يكون متوافق مع استثناء الانترفيس الوظيفي الذي يحدد من خلال العبارة throws في دالة الـ abstract في الانترفيس، على سبيل المثال اذا ترمي دالة الـ Lambda استثناء IOException فان دالة الـ abstract يجب ان ترمي استثناء IOException ايضا، مثال :

```
import java.io.*;
interface MyIOAction {
boolean ioAction(Reader rdr) throws IOException;
}
class LambdaExceptionDemo {
public static void main(String args[ ]) {
double[ ] values = { 1.0, 2.0, 3.0, 4.0 };
MyIOAction  myIO = (rdr) -> {
int ch = rdr.read( ); // could throw IOException
// ...
return true;
};
```

```
} }
```

لو لم نكتب استثناء بعد الدالة ioAction في الانترفيس فان البرنامج لن يتنفذ لان دالة Lambda محتمل ان ترمي استثناء وستعتبر غير متوافقة مع دالة الـ abstract .

Method References to static Methods

يكتب هذا النوع عن طريق تحديد اسم الدالة مسبوق باسم كلاسها ويفصل بينهم بنقطتين مزدوجتين، صيغتها العامة هي :

```
ClassName::methodName
```

وهذا النوع يمكن ان يستخدم في اي مكان بشرط ان تكون متوافقة مع دالة abstract في الانترفيس، مثال :

```
interface IntPredicate {
boolean test(int n);
}
class MyIntPredicates {
static boolean isPrime(int n) {
if(n < 2) return false;
for(int i=2; i <= n/i; i++) {
if((n % i) == 0)
return false;
}
return true;
}
static boolean isEven(int n) {
return (n % 2) == 0;
}
static boolean isPositive(int n) {
return n > 0;
} }
```

```
class MethodRefDemo {
static boolean numTest(IntPredicate p, int v) {
return p.test(v);
}
public static void main(String args[ ]) {
boolean result;
result = numTest(MyIntPredicates::isPrime, 17);
if(result) System.out.println("17 is prime.");
result = numTest(MyIntPredicates::isEven, 12);
if(result) System.out.println("12 is even.");
result = numTest(MyIntPredicates::isPositive, 11);
if(result) System.out.println("11 is positive.");
} }
```

نتيجة البرنامج ستكون :

```
17 is prime.
12 is even.
11 is positive.
```

Method References to Instance Methods

يتم عمل هذه الدوال من خلال اسم الدالة مسبوق باسم اوبجكت كلاسها ويفصل بينهم بنقطتين مزدوجتين، صيغتها العامة بهذا الشكل :

```
objRef::methodName
```

مثال :

```
interface IntPredicate {
boolean test(int n);
}
class MyIntNum {
private int v;
MyIntNum(int x) { v = x; }
int getNum() { return v; }
```

```
boolean isFactor(int n) {
return (v % n) == 0;
} }
class MethodRefDemo2 {
public static void main(String args[ ]) {
boolean result;
MyIntNum myNum = new MyIntNum(12);
MyIntNum myNum2 = new MyIntNum(16);
IntPredicate ip = myNum::isFactor;
result = ip.test(3);
if(result) System.out.println("3 is a factor of " + myNum.getNum());
ip = myNum2::isFactor;
result = ip.test(3);
if(!result) System.out.println("3 is not a factor of " + myNum2.getNum());
} }
```

مخرجات البرنامج ستكون :

```
3 is a factor of 12
3 is not a factor of 16
```

وهذا النوع يمكن ان تكون صيغته بهذا الشكل :

```
ClassName::instanceMethodName
```

اي نضع اسم الكلاس بدلا من اوبجكت، مثال :

```
interface MyIntNumPredicate {
boolean test(MyIntNum mv, int n);
}
class MyIntNum {
private int v;
MyIntNum(int x) { v = x; }
int getNum() { return v; }
boolean isFactor(int n) {
return (v % n) == 0;
```

```
} }
class MethodRefDemo3 {
public static void main(String args[ ]) {
boolean result;
MyIntNum myNum = new MyIntNum(12);
MyIntNum myNum2 = new MyIntNum(16);
// A method reference to any object of type MyIntNum
MyIntNumPredicate inp = MyIntNum::isFactor;
result = inp.test(myNum, 3);
if(result) System.out.println("3 is a factor of " + myNum.getNum());
result = inp.test(myNum2, 3);
if(!result) System.out.println("3 is a not a factor of " + myNum2.getNum());
} }
```

نتيجة البرنامج ستكون :

```
3 is a factor of 12
3 is a not a factor of 16
```

* يمكن ان تكون الدالة من النوع generic ، لو كان عندنا مثلا :

```
interface SomeTest<T> {
boolean test(T n, T m);
}
class MyClass {
static <T> boolean myGenMeth(T x, T y) {
boolean result = false;
// ...
return result;
} }
```

فان عبارة مثل هذه ستكون صحيحة :

```
SomeTest<Integer> mRef = MyClass::<Integer>myGenMeth;
```

Constructor References

بنفس الطريقة التي نعمل بها References للدالة فانه يمكن ان نعمل References للـ Constructor ، الصيغة العامة لها هي :

```
classname::new
```

هذا الـ reference يمكن ان يشير الى اي انترفيس وظيفي اذا كانت متوافقه مع دالة الـ abstract ، مثال :

```
interface MyFunc {
MyClass func(String s);
}
class MyClass {
private String str;
MyClass( ) { str = ""; }
// This constructor reference Because has compatible parameter with func( )
MyClass(String s) { str = s; }
String getStr( ) { return str; }
}
class ConstructorRefDemo {
public static void main(String args[ ]) {
// Create a reference to the MyClass constructor.
MyFunc myClassCons = MyClass::new;
MyClass mc = myClassCons.func("Testing");
System.out.println("str in mc is " + mc.getStr( ));
} }
```

ستكون نتيجة البرنامج :

```
str in mc is Testing
```

* يمكن عمل constructor reference تنشأ مصفوفة بهذه الصيغة :

```
type[ ]::new
```

هنا type يحدد نوع الاوبجكت الذي ينشأ، لتوضيح الفكرة افترض ان لدينا هذا الانترفيس :

```
interface MyClassArrayCreator {
MyClass[ ] func(int n);
}
```

في الدالة main يمكن انشائها بهذا الشكل :

```
MyClassArrayCreator mcArrayCons = MyClass[ ]::new;
MyClass[ ] a = mcArrayCons.func(3);
for(int i=0; i < 3; i++)
a[i] = new MyClass(i);
```

وكذلك يمكن عمل انترفيس وظيفي من النوع generic وانشاء constructor reference منه، بهذا الشكل :

```
MyGenClass<T> { // ...
```

وعند الانشاء نكتب :

```
MyGenClass<Integer>::new;
```

Predefined Functional Interfaces

لغة جافا تعرف مجموعة من الانترفيسز الوظيفية التي يمكن استخدامها بدلا من ان ننشأها نحن ووضعت هذا الانترفيسز في الحزمة java.util.function ، من هذه الانترفيسز :

الانترفيس	الغرض منه
UnaryOperator<T>	يطبق عملية واحدة على اوبجكت من النوع T ويعيد النتيجة وهي من النوع T واسم دالته ( )apply
BinaryOperator<T>	يطبق عملية على ثنين اوبجكت من النوع T ويعيد النتيجة وهي من النوع T واسم دالته ( )apply
Consumer<T>	ينفذ عملية على اوبجكت من النوع T واسم دالته ( )accept

Consumer<T>	يعيد اوبجكت من النوع T واسم دالته ( )get
Function<T, R>	يطبق عملية على اوبجكت من نوع T ويعيد قيمة من النوع R واسم دالته ( )apply
Predicate<T>	يحدد اذا كان اوبجكت ما يندز بعض القيود ويعيد قيمة من النوع boolean تشير الى المخرجات واسم دالته ( )test

مثال :

```
import java.util.function.Predicate;
class UsePredicateInterface {
public static void main(String args[ ]) {
// This lambda uses Predicate<Integer> to determine if a number is even.
Predicate<Integer> isEven = (n) -> (n %2) == 0;
if(isEven.test(4)) System.out.println("4 is even");
if(!isEven.test(5)) System.out.println("5 is odd");
} }
```

نتيجة البرنامج ستكون :

```
4 is even
5 is odd
```

الفصل الرابع عشر (Applets and Events)

الـ Applet هي برمجيات متكاملة تعمل مع تطبيقات الويب وتستعرض بواسطة متصفح الانترنت ولعرضها يمكن كتابتها في داخل كود HTML او من خلال البرنامج الافتراضي الذي نكتب فيه كود جافا، لنأخذ مثال ونشرحه لتتضح الصورة :

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("Java makes applets easy.", 20, 20);
} }
```

في بداية المثال جلبنا حزمتين الاولى تحتوي على AWT وهي اختصار لـ Abstract Window Toolkit وهذه الحزمة تحتوي على بعض اشكال النوافذ المستخدمة، اما الحزمة الثانية التي استدعيناها فانها تحتوي على الكلاس Applet ، كل كلاس نعمله للتعامل مع النوافذ AWT يجب ان يرث من الكلاس Applet ، في السطر الذي بعده انشأنا كلاس اسميناه SimpleApplet ويرث الكلاس Applet ولاحظ انه يجب ان يكون public لانه سيتم الوصول اليه من خارج الكود، وفي داخل هذا الكلاس دالة paint تحتوي على باراميتر واحد من النوع Graphics ، وفي داخل الدالة استخدمنا دالة ( )drawstring والتي هي موجودة في الكلاس Graphics والكلاس Graphics موجود في الحزمة java.awt ، هذه الدالة تظهر نص في الموقع X, y وصيغتها العامة بهذا الشكل :

```
void drawString(String message, int x, int y)
```

في لغة جافا الزاوية العلوية اليسرى تكون احداثياتها هي 0, 0 ، لاحظ في المثال انه لا توجد الدالة main وذلك لان تطبيقات الـ Applet لا تحتاج الدالة main لانها ستنفذ عند استدعائها .

* الـ Applet تنفذ وفقا للاحداث التي تحدث كضغط المستخدم على الكيبورد او الماوس، وهي لها دوال لدورة حياتها وهي ( )destroy, ( )stop, ( )start, ( )init بالاضافة الى الدالة ( )paint وهي كثيرا ما تستخدم على الرغم من انها ليست من ضمن دوال دورة الحياة وهذه الدوال الخمسة هي من ضمن دوال الكلاس Applet ويتم استخدام هذه الدوال بهذا الشكل :

```
import java.awt.*;
```

```
import java.applet.*;
// This code for html file.
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
// Called first.
public void init() {
// initialization
}
// Called second, after init(). Also called whenever the applet is restarted.
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
// Called when applet is terminated. This is the last method executed.
public void destroy() {
// perform shutdown activities
}
// Called when an AWT-based applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
} }
```

وعمل هذه الدوال كما هو موضح في التعليقات بداخل المثال، حيث من المهم معرفة انه يبدأ الـ applet في التنفيذ بشكل متسلسل بهذه الدوال :

1- init( )

2- start( )
3- paint( )

وعند انتهاء الـ applet فان الدوال التالية تنفذ بالترتيب :

1- stop( )
2- destroy( )

الدالة repaint( )

في بعض الاحيان عندما نريد اعادة اظهار النافذة والتي كتبنا كودها في الدالة paint نستخدم الدالة repaint وصيغتها العامة هي :

```
void repaint( )
```

هذه الصيغة ستعيد تنفيذ الدالة paint بالكامل لكن اذا اردنا ان نظهر جزء من النافذة نستخدم هذه الصيغة :

```
void repaint(int left, int top, int width, int height)
```

حيث ان left تمثل البعد عن الزاوية اليسرى و top تمثل البعد عن الاعلى اما width و height تمثل الطوال والعرض وكل هذه القياسات تكون بالبكسل .

الدالة update( )

هذه الدالة هي مشابهه للدالة repaint وتستخدم لنفس الغرض تقريبا وهي موجودة في الكلاس Component والذي يوجد بدوره في الحرزمة java.awt .

مثال لعرض رسالة نصية تمرر من اليمن لليسار :

```
import java.awt.*;
import java.applet.*;
// This code for html file.
/*
<applet code="Banner" width=300 height=50>
</applet>
```

```
*/
public class Banner extends Applet implements Runnable {
String msg = " Java Rules the Web ";
Thread t;
boolean stopFlag;
// Initialize t to null.
public void init( ) {
t = null;
}
// Start thread
public void start( ) {
t = new Thread(this);
stopFlag = false;
t.start( );
}
public void run() {
for( ; ; ) {
try {
repaint( );
Thread.sleep(250);
if(stopFlag)
break;
} catch(InterruptedException exc) {  }
} }
public void stop( ) {
stopFlag = true;
t = null;
}
public void paint(Graphics g) {
char ch;
ch = msg.charAt(0);
```

```
msg = msg.substring(1, msg.length());
msg += ch;
g.drawString(msg, 50, 30);
} }
```

ونتيجة البرنامج ستكون نافذه بهذا الشكل :

الدالة showStatus( )

تستخدم هذه الدالة لعرض رسالة لكن من خلال نافذة الحالة للمتصفح، وهذه الدالة موجودة في الكلاس Applet والصيغة العامة لها هي :

```
void showStatus(String msg)
```

حيث ان msg هي الرسالة التي سيتم عرضها، نافذة الحالة هي المكان الانسب لعرض معلومات او اظهار بعض الانواع من الاخطاء للمستخدم، مثال :

```
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
public void paint(Graphics g) {
g.drawString("This is in the applet window.", 10, 20);
```

```
showStatus("This is shown in the status window.");
} }
```

نتيجة البرنامج ستكون بهذا الكشل :

تمرير بارامترات للـ Applets

ملاحظات عامة

* يمكن كتابة التعليقات في جافا من خلال العلامة \\ وذلك لكتابة تعليق بسطر واحد او يمكن استخدام * The Comment *\ للتعليق باكثر من سطر.

* كل جملة في لغة جافا يجب ان تنتهي بفارزة منقوطة.

* لاي معلومات عن اللغة :

http://docs.oracle.com/javase/7/docs/api/java/lang/package-tree.html