Encapsulationالتغليف

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

التغليف هو أحد مفاهيم OOP الأساسية الأربعة. الثلاثة الآخرون هم الميراث وتعدد الأشكال والتجريد.

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as **data hiding**.

التغليف هو تقنية جعل الحقول في الفصل خاصة وإتاحة الوصول إلى الحقول عبر الطرق العامة. إذا تم الإعلان عن حقل خاص ، فلن يتمكن أي شخص خارج الفصل من الوصول إليه ، وبالتالي إخفاء الحقول داخل الفصل. لهذا السبب ، يُشار إلى التغليف أيضًا بإخفاء البيانات.

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

يمكن وصف التغليف بأنه حاجز وقائي يمنع الوصول العشوائي إلى الشفرة والبيانات بواسطة كود آخر محدد خارج الفصل. يتم التحكم في الوصول إلى البيانات والرمز بإحكام بواسطة واجهة.

The **main benefit** of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

تتمثل الفائدة الرئيسية للتغليف في القدرة على تعديل الكود الذي تم تنفيذه دون كسر الكود الخاص بالآخرين الذين يستخدمون الكود الخاص بنا. باستخدام هذه الميزة ، يوفر التغليف إمكانية الصيانة والمرونة والتوسعة في التعليمات البرمجية الخاصة بنا.

Benefits of Encapsulation:فوائد التغليف :

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

- يمكن جعل حقول الفصل للقراءة فقط أو للكتابة فقط.
- يمكن للفصل أن يكون له سيطرة كاملة على ما يتم تخزينه في مجالاته.
- لا يعرف مستخدمو الفصل كيف يخزن الفصل بياناته. يمكن للفصل تغيير نوع بيانات الحقل ولا يحتاج مستخدمو الفصل إلى تغيير أي من التعليمات البرمجية الخاصة بهم.

Pass Objects to Methods:تمرير الكائنات إلى الأساليب :

Up to this point, the examples in passed lectures have been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, the following program defines a class called **Block** that stores the dimensions of a three-dimensional block:

حتى هذه النقطة ، كانت الأمثلة في المحاضرات التي تم اجتيازها تستخدم أنواعًا بسيطة كمعلمات للطرق. ومع ذلك ، فمن الصحيح والشائع تمرير الكائنات إلى الأساليب. على سبيل المثال ، يحدد البرنامج التالي فئة تسمى Block تخزن أبعاد كتلة ثلاثية الأبعاد:

```
// Objects can be passed to methods
```

// يمكن تمرير الكائنات إلى الأساليب.

```
.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;
}

// Return true if ob defines same block
```

// إرجاع صحيحًا إذا كان ob يعرف نفس الكتلة.

.

```
boolean sameBlock(Block ob) {
if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
else return false;
}
// Return true if ob has same volume
```

// إرجاع صحيح إذا كان ob له نفس الحجم.

```
boolean sameVolume(Block ob) {
if(ob.volume == volume) return true;
else return false;
}
}
class Program1 {
public static void main(String args[]) {
Block ob1 = new Block(10, 2, 5);
Block ob2 = new Block(10, 2, 5);
Block ob3 = new Block(4, 5, 5);
System.out.println("ob1 same dimensions as ob2: " +
ob1.sameBlock(ob2));
System.out.println("ob1 same dimensions as ob3: " +
ob1.sameBlock(ob3));
System.out.println("ob1 same volume as ob3: " +
ob1.sameVolume(ob3));
}
}
```

This program generates the following output:يولد هذا البرنامج المخرجات التالية :

```
ob1 same dimensions as ob2: true
ob1 same dimensions as ob3: false
ob1 same volume as ob3: true
```

The **sameBlock( )** and **sameVolume( )** methods compare the **Block** object passed as a parameter to the invoking object. For **sameBlock( )**, the dimensions of the objects are compared and **true** is returned only if the two blocks are the same. For **sameVolume( )**, the two blocks are compared only to determine whether they have the same volume. In both cases, notice that the parameter **ob** specifies **Block** as its type. Although **Block** is a class type created by the program, it is used in the same way as Java's built-in types.

تقارن أساليب sameBlock () و sameVolume () كائن Block الذي تم تمريره كمعامل إلى كائن استدعاء. بالنسبة لـ sameBlock () ، تتم مقارنة أبعاد الكائنات ويتم إرجاع true فقط إذا كانت الكتلتان متماثلتان. بالنسبة إلى sameVolume () ، تتم مقارنة الكتلتين فقط لتحديد ما إذا كان لهما نفس الحجم. في كلتا الحالتين ، لاحظ أن المعلمة ob تحدد Block كنوعها. على الرغم من أن Block هو نوع فئة تم إنشاؤه بواسطة البرنامج ، إلا أنه يتم استخدامه بنفس طريقة استخدام الأنواع المضمنة في Java.

How Arguments Are Passed:كيف يتم تمرير الحجج :

As the preceding example demonstrated, passing an object to a method is a straightforward task. In certain cases, the effects of passing an object will be different from those experienced when passing non-object arguments. To see why, you need to understand the two ways in which an argument can be passed to a subroutine.
The first way is ***call-by-value***. This approach copies the *value* of an argument into the

كما أوضح المثال السابق ، يعد تمرير كائن إلى طريقة مهمة مباشرة. في بعض الحالات ، ستكون تأثيرات تمرير كائن مختلفة عن تلك التي يتم تجربتها عند تمرير وسيطات غير كائن. لمعرفة السبب ، تحتاج إلى فهم الطريقتين اللتين يمكن من خلالهما تمرير الحجة إلى روتين فرعي.
الطريقة الأولى هي الاتصال بالقيمة. هذا النهج ينسخ قيمة الحجة في

formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument in the call.

المعلمة الرسمية للروتين الفرعي. لذلك ، لا تؤثر التغييرات التي تم إجراؤها على معامل الإجراء الفرعي على الوسيطة في الاستدعاء.

The second way an argument can be passed is ***call-by-reference***. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter *will* affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed.

الطريقة الثانية لتمرير الوسيطة هي call-by-reference. في هذا النهج ، يتم تمرير مرجع إلى وسيطة (وليس قيمة الوسيطة) إلى المعلمة. داخل الروتين الفرعي ، يتم استخدام هذا المرجع للوصول إلى الوسيطة الفعلية المحددة في الاستدعاء. هذا يعني أن التغييرات التي يتم إجراؤها على المعلمة ستؤثر على الوسيطة المستخدمة لاستدعاء الروتين الفرعي. كما سترى ، تستخدم Java كلا الأسلوبين ، اعتمادًا على ما تم تمريره.

In Java, when you pass a primitive type, such as **int** or **double**, to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

في Java ، عندما تمرر نوعًا أوليًا ، مثل int أو double ، إلى طريقة ، يتم تمريرها بالقيمة. وبالتالي ، فإن ما يحدث للمعامل الذي يتلقى الوسيطة ليس له أي تأثير خارج الطريقة. على سبيل المثال ، ضع في اعتبارك البرنامج التالي:

```
// Primitive types are passed by value.
// يتم تمرير الأنواع الأولية بالقيمة.
class Test {
/* This method causes no change to the arguments used in the call.
*/
/ * لا تسبب هذه الطريقة أي تغيير في الوسائط المستخدمة في الاستدعاء.
/ *

void noChange(int i, int j) {
i = i + j;
j = -j;
}
}
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " + a + " " + b);
ob.noChange(a, b);
System.out.println("a and b after call: " + a + " " + b);
}}
```

The output from this program is shown here:يظهر ناتج هذا البرنامج هنا :

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **noChange( )** have no effect on the values of **a** and **b** used in the call.

كما ترى ، فإن العمليات التي تحدث داخل noChange () ليس لها أي تأثير على قيم a و b المستخدمة في المكالمة.

When you pass an object to a method, the situation changes dramatically, because objects are implicitly passed by reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

عندما تقوم بتمرير كائن إلى أسلوب ، يتغير الموقف بشكل كبير ، لأن الكائنات يتم تمريرها ضمنيًا عن طريق المرجع. ضع في اعتبارك أنه عند إنشاء متغير من نوع فئة ، فأنت تقوم فقط بإنشاء مرجع لكائن. وبالتالي ، عندما تقوم بتمرير هذا المرجع إلى طريقة ، فإن المعلمة التي تتلقاها ستشير إلى نفس الكائن الذي تشير إليه الوسيطة. هذا يعني بشكل فعال أن الكائنات يتم تمريرها إلى الأساليب باستخدام الاستدعاء من خلال المرجع. التغييرات التي يتم إجراؤها على الكائن داخل الطريقة تؤثر على الكائن المستخدم كوسيطة. على سبيل المثال ، ضع في اعتبارك البرنامج التالي:

```
// Objects are passed by reference
```

// يتم تمرير الكائنات عن طريق المرجع . .

```
class Test {
int a, b;
```

```
Test(int i, int j) {
a = i;
b = j;
}
/* Pass an object. Now, ob.a and ob.b in object
used in the call will be changed. */
```
/ * تمرير كائن. الآن ، سيتم تغيير ob.a و ob.b في الكائن المستخدم في الاستدعاء. * /

```
void change(Test ob) {
ob.a = ob.a + ob.b;
ob.b = -ob.b;
}
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.change(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```
This program generates the following output:يولد هذا البرنامج المخرجات التالية :

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 35 -20
```

As you can see, in this case, the actions inside **change( )** have affected the object used as an argument. As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object referred to by its corresponding argument.

كما ترى ، في هذه الحالة ، أثرت الإجراءات داخل التغيير () على الكائن المستخدم كوسيطة. كنقطة اهتمام ، عندما يتم تمرير مرجع كائن إلى طريقة ، يتم تمرير المرجع نفسه باستخدام استدعاء حسب القيمة. ومع ذلك ، نظرًا لأن القيمة التي يتم تمريرها تشير إلى كائن ، فإن نسخة تلك القيمة ستظل تشير إلى نفس الكائن المشار إليه بواسطة الوسيطة المقابلة له.

Returning Objects:عودة الكائنات :

A method can return any type of data, including class types. For example, the class **ErrorMsg** shown here could be used to report errors. Its method, **getErrorMsg( )**, returns a **String** object that contains a description of an error based upon the error code that it is passed.

يمكن للطريقة إرجاع أي نوع من البيانات ، بما في ذلك أنواع الفئات. على سبيل المثال ، يمكن استخدام فئة ErrorMsg الموضحة هنا للإبلاغ عن الأخطاء. تقوم طريقتها getErrorMsg () ، بإرجاع كائن سلسلة يحتوي على وصف لخطأ بناءً على رمز الخطأ الذي تم تمريره.

```
// Return a String object
```

// إرجاع كائن سلسلة.

```
class ErrorMsg {
     String msgs[] = {"Output Error","Input Error","Disk
     Full","Index Out-Of-Bounds"};

// Return the error message
```

// إرجاع رسالة الخطأ.

```
String getErrorMsg(int i) {
  if(i >=0 & i < msgs.length)
   return msgs[i];
  else
   return "Invalid Error Code";
 }
```

```
}
class ErrMsg {
public static void main(String args[]) {
ErrorMsg err = new ErrorMsg();
System.out.println(err.getErrorMsg(2));
System.out.println(err.getErrorMsg(19));
}
}
```

Its output is shown here:

```
Disk Full
Invalid Error Code
```

You can, of course, also return objects of classes that you create. For example, here is a reworked version of the preceding program that creates two error classes. One is called **Err**, and it encapsulates an error message along with a severity code. The second is called **ErrorInfo**. It defines a method called **getErrorInfo( )**, which returns an **Err** object.

يمكنك ، بالطبع ، أيضًا إرجاع كائنات الفئات التي تقوم بإنشائها. على سبيل المثال ، يوجد هنا نسخة معدلة من البرنامج السابق تقوم بإنشاء فئتين من فئات الأخطاء. أحدهما يسمى Err ، وهو يضم رسالة خطأ مع رمز خطورة. والثاني يسمى ErrorInfo. يعرّف طريقة تسمى getErrorInfo () ، والتي تُرجع كائن Err.

```
// Return a programmer-defined object
```

// إرجاع كائن محدد بواسطة المبرمج.

```
class Err
 {
      String msg; // error message
      int severity; // code indicating severity of error

      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 program1
{
     public static void main(String args[])
     {
          ErrorInfo e_info = new ErrorInfo();
          Err e;
          e = e_info.getErrorInfo(2);
          System.out.println(e.msg + " severity: " + e.severity);
          e = e_info.getErrorInfo(19);
          System.out.println(e.msg + " severity: " + e.severity);
     }
}
```

Here is the output:

```
Disk Full severity: 2
Invalid Error Code severity: 0
```

Each time **getErrorInfo( )** is invoked, a new **Err** object is created, and a reference to it is returned to the calling routine. This object is then used within **main( )** to display the error message and severity code.

في كل مرة يتم فيها استدعاء getErrorInfo () ، يتم إنشاء كائن Err جديد ، ويتم إرجاع مرجع إليه إلى إجراء الاستدعاء. ثم يتم استخدام هذا الكائن داخل main () لعرض رسالة الخطأ ورمز الخطورة.

When an object is returned by a method, it remains in existence until there are no more references to it. At that point it is subject to garbage collection. Thus, an object won't be destroyed just because the method that created it terminates.

عندما يتم إرجاع كائن بواسطة طريقة ، فإنه يظل موجودًا حتى لا توجد مراجع أخرى إليه. في هذه المرحلة يخضع لجمع القمامة. وبالتالي ، لن يتم تدمير الكائن لمجرد أن الطريقة التي أنشأته تنتهي.