2010/11/26

المادة : برمجة -3-

لغة البرمجة : Java

My -1- Lecture :

Introduction to Java

By: Mohammed Abdullah Najeeb

أهلاً بكم في مادة البرمجة3 ، بعدما تم الانتهاء من برمجة1 التي تختص بإعطاء فكرة البرمجة بلغة C++ و تطبيقات سريعة و بسيطة ، و مادة البرمجة2 التي كانت أكثر اختصاصية في I/O Files و struct و Pointer .
تأتي مادة البرمجة3 تأكيداً لمبدأ البرمجة الغرضية التوجه OOP ، و لا يُذكر هذا المبدأ إلا و قد أُلحق بلغة Java التي بُنيت على أساسه .
في سلسلة الدروس القادمة (و بما أن لغتنا الأم –برمجياً – هي الـC++) فسأقوم بوضع كل كلمة محجوزة بما يقابلها في الجافا ، و سنتجه إلى تعميق الفكر الغرضي التوجه أكثر من الـSyntax للغة (بما أن جميع اللغات تتشارك في نفس الجذع البرمجي)

Why OOP ?

لماذا ؟ لأنه يسهل على المبرمج فهمة ألية البرنامج ، و يقربه إلى مبدأ تبسيط يُقارب ما نعيش فيه –بشرياً - ،و يقلل من حجم الكود البرمجي و من تكرار التوابع methods ، أضف إلى ذلك أن تقنيات كثيرة ارتكزت عليه ، مثل التغليف و الوراثة و الواجهات وتعدد الأشكال .
من اللغات البرمجية المشهورة في هذا المبدأ هي لغة البرمجة Java فلقد بنيت على أساس الغرضية التوجه و ستلاحظ ذلك من خلال البدأ بأول كلمة من البرنامج و هي Class ، ستقول لي : حسناً لقد مررنا على هذا المصطلح في C++ فلماذا Java . في الحقيقة إن C++ هي لغة هجينة دمجت بين البرمجة الغرضية التوجه و البرمجة الـFunctional ، فمثلاً يحق لك إنشاء تابع من غير أن يكون ضمن Class إلا أن هذا الأمر في جافا ممنوع . لذا فإن C# " لغة برمجية أنشأت بعد C++ " تشابه Javaكثيراً نظراً لأنها أتت كي تواكب التطور البرمجي الذي حصل .
ملاحظة "فيما بعد" : سيتم التركيز بشكل أكبر على مفاهيم الـOOP خلال المحاضرات القادمة.

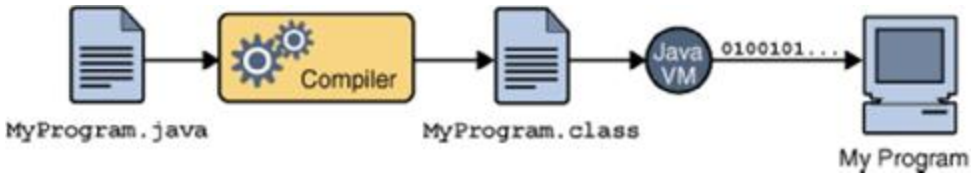
Java ! Power ! : Write once, run anywhere

JVM (Java Virtual Machine) الآلة التخيلية
+
GC (Garbage Collector) جامع نفايات البرنامج
+
Security of the code أمان الكود البرمجي

Java buzzwords
سيتم شرحها في هذه المحاضرة على شكل ملاحظات

- Simple
- Architecture neutral
- Object oriented
- Portable
- Distributed
- High performance
- Multithreaded
- Robust
- Dynamic
- Secure

الـJWM : و هي طبقة وسيطة بين البرنامج و الـhardware تقوم بقراءة الملف .class و الذي يحوي Byte Code إلى لغة الآلة و تنفذه .



تتميز بأنها تمنح الجافا Portability التوافقية و إمكانية التنفيذ على أكثر من نظام تشغيل ، و كذلك الأمان Security الذي سنتحدث عنهلاحقاً . إلا أنها تحويعيباً و هو البطء في التشغيل و ذلك بسبب تعدد تلك الطبقات

بالنسبة للـGC فوظيفته تظنيف الـHeap من كل حجز لم يعد مستخدماً في البرنامج من الـHeap (تذكر JVW المفسرة) و تستطيع بتحديد بطرق خوارزمية معينة تحديد هذه الخلية الذاكرية و حذفها free it

من المصطلحات التي قد تسمعها في جافا أيضاً :

JRE (Java Runtime Environment)
يتكون من الـJVW و هو الذي ينفذ الـByte-Code

JDK (Java Development Kit)
تؤمن للمبرمج ما يحتاجه لكي يبرمج و تتشبع إلى ثلاثة أقسام :

Java SE (Standard Edition)
Java EE (Enterprise Edition)
Java ME (Micro Edition)

سنختص فقط في النوع الأول SE في مادة البرمجة3 و هو النوع المخصص للتطبيقات المكتبية

Hello World !

ستسمع هذه الجملة كثيراً أثناء تعلمك للغة برمجية جديدة "خاصةً بعدما تكون لديك فهم برمجي للبرمجة في معناها العام" و لأنك لن تحتاج في كل مرة إلى أن تفهم ماهية البرمجة بقدر ما يهمك كيف أن تُعبر للغة بكلماتها المحجوزة الخاصة .

ملاحظة "فيما بعد" : سنتسخدم بشكل افتراضي للتنفيذ الـ IDE " NetBeans " إلا في حال ذكر غير ذلك .

لطباعة هذه الجملة في برنامج Java سيكون الشكل كالتالي :

Java Code

```
public class Test {
   public static void main(String[] args) {
      System.out.print("Hello Wolrd!");
   }
}
```

السطر1 : public class Test

كما أسلف ذاكراً . هذه اللغة مبنية على أساس OOP لذا فإن البرنامج بأكمله ما هو إلا جزءٌ من class و يسمى الـclass باسم البرنامج (في مثالنا سميته Test)

السطر2: الدالة main

تعود لنا من جديد و هي كما في C++ التي يبدأ منها التنفيذ – الدالة الرئيسية – يجب أن تكون public و إلا سيعطي المفسر خطأً كي تكون مرئية لبقية الـclasses و كذلك يجب أن تكون static و هنا معنى الـstatic أن الـclass يمكن أن ينشء دون الحاجة إلى إنشاء Object " سيتم شرحها لاحقاً في هذه المحاضرة "

السطر3:

الـmethod الخاص بطباعة جملة Hello World يستقبل بارمتر واحد فقط و هو من النوع Object " أي أنه يقبل كافة الأنواع الأساسية ..... int , float , String , Char , " سواء كان ثابت أو متغير و يوجد دالة تشابهها لكنها تترك سطراً في النهاية و هي println و توجد أيضاً الدالة printf التي سيتم مناقشتها في فصل الإخراج و التي تستقبل عدداً لا نهائياً من الـobjects

ملاحظة "Java" : عند استعمال println افصل بين المتغيرات و الـStrings (الذي بين إشارتي Quotation " ") بإشارة الجمع +

الإدخال و الإخراج :

سنناقش طريقة الإدخال و الإخراج "Keyboard" و التي تماثل الـcout و cin في C++

- **الإخراج**

نلاحظ أننا استعملنا في المثال الأول system.out.printf للطباعة على الشاشة و لها عدة أمثلة للطباعة مثلاً :

Java Code

```
int x=10;
int y=2;
system.out.printf("Your number is %d and the other is %d \n",x,y);
```

نعبر عن المتغير الصحيح int بالإشارة **%d** داخل الـstring "...." المراد طباعته

Java Code

```
String name="AleppoSoft";
int id=123;
system.out.printf("I love this website : \" %s \" and my id = %d ",name,id);
```

نعبر عن المتغير المحرفي **%s** (لاحظ الترتيب في التسلسل)

و يوجد **%f** لطباعة الـfloat

ملاحظة "C++" : في Java لا يوجد نوع ~~unsigned~~ مثل الذي كان موجود في C++

ملاحظة "C++" : عوضاً عن ~~const~~ التي كانت في C++ فإن ما يقابلها في جافا هي **Final** للثوابت و تستخدم بنفس الطريقة مع العلم أن الكلة const هي كلمة محجوزة في الـJava إلاّ أنها ليس لها أي استخدام

ملاحظة "C++" : في حال طباعة متحول غير مصرح عنه " لا قيمة له " no initialize فإن المفسر سيعطي خطأ بعكس C++ التي كانت تطبع قيمة عشوائية .

ملاحظة "Java" : في بعض الأحيان لا تقبل بعض المترجمات رقم عشري للـfloat بغرض تحويله النوع إلى double لضمان دقة رقمية أكبر (بعرض الأرقام العشرية لا يمكن تحويلها إلى رقم ثنائي ذو عدد خانات محددة) ، لذا يوضع الحرف f في نهاية الرقم عشري كي يُوضع في المتحول float .

ملاحظة "C++" : كل ما كان ينفذ من تعليمات و تنسيقات الخرج في C++تُطبق هنا مثل : /t , /" , // , /n

- **الإدخال**

توجد طرق عديدة للإدخال ، و إحداها طريقة الماسح Scanner

Java Code

```
Scanner input=new Scanner(System.in);
int x=input.nextInt();
string y=input.nextLine();
```

يمكنك أن ترى بقية الـmethods الموجودة داخل الـclass فلكل نوع متحول إدخال خاص به .. , float , double

لا تنسى يجب إضافة المكتبة java.util.Scanner و إضافة المكتبات في جافا يتم بالطريقة التالية :

في C++ كان الكود :

include<iostream.h>

في Java أصبح الكود :

import java.util.Scanner;

لاحظ أن المكتبات في Java تعتمد على الـClass و تسمى كل مكتبة بـpackage و أحياناً يوضع *. للدلالة على جميع الـclasses

ملاحظة "فيما بعد": توجد طرق أخرى للإدخال و الإخراج على شكل نوافذ تعتمد على الـpackage " JOption " قد نتحدثعنها لاحقاً و توجد طريقة Console باستخدام الـBufferقد نتحدث عنها لاحقاً

التعبيرات و المعاملات الرياضية

+ الجمع	- الطرح	* الضرب	/ القسمة	% باقي القسمة
= يعادل	== يساوي	!= لا يساوي	> أكبر	< أصغر
=< أكبر من أو يساوي	<= أصغر من أو يساوي	Bitwise & , | , ~ , ^ , >>	|| أو (عملية منطقية)	&& و (عملية منطقية)

تماماً كما هي في الـC++ فكل ما تم تطبيقه من معاملات في C++ له نفس الاستخدام في Java " بشكل عام هذه المعايير تكون مشتركة في معظم اللغات البرمجية عالية المستوى "

ملاحظة "C++" : لا ننسى كذلك الـ += ، *= ، و ما شابهها و كذلك ++ و -- فهي موجودة و مستعملة في الـJava

خطأ شائع :
المقارنة بين String عن طريق الأداة == و هذا الخطأ يؤدي إلى مقارنة العنوان لكلا السلسلتين المحرفتين و الأوجب استخدام الـequals method كما في المثال التالي :
ملاحظة "تطبيق": قم بتطبيق التعليقات عوضاً عن الكود لن يطبع شيئاً و ذلك لأن العنوان اختلف عن طريق new

Java Code

```
public class Test {
   public static void main(String[] args) {
      String name1="Ali";//String name1=new String("Ali");
      String name2="Ali";//String name2=new String("Ali");

      if(name1.equals(name2))//if(name1==name)
      {
         System.out.println("name1 Equal name2");
      }
   }
}
```

للاطلاع : توفر جافا في الـpackage java.math أنواع متغيرات مثل : BigIntger , BigDemical وظيفتها استعياب الأعداد الضخمة جداً و لا يستطيع int و long تحملها ، لكن للأسف و على عكس C++ لا تتوفر خاصية التحميل الزائد للعمليات Overloading Operators في الـJava لذا في حال جمع عددين Big يتم الاستعانة بالـmethod add(Object).

```
BigInteger x=BigInteger.valueOf(912332312);
x.add(BigInteger.valueOf(3232));
```

الشرط Condition :

تتشابه جافا مع C++ في الشرط if statement و كذلك في else و Switch و حتى إشارة الاستفهام

لكن هنالك **بعض الملاحظات** و منها :

في C++ كانت القيمة العددية مهما ما كانت تشير إلى القيمة البولانية true أما في جافا فلا يصح ذلك بل يجب أن يكون هنالك شرط منطقي ، مثال :

كود C++ :

```
int x=5;
if(x)
cout<<"hi";
```

سيكون الناتج hi

أما في جافا فسيعطي خطأ في زمن الترجمة و ذلك لأنها لا تقبل إلا عمليات منطقية أو بولانية مثل :

```
if(x==5)
if(x!=3)
```

سبب ذلك أن لغة جافا تعتمد على مبدأ البرمجة الأمنة Safe Program. قد تظن الأمر عائقاً لكن توجد طرق بديلة و يجنبك هذا الأمر الكثير من المخاطر التي قد لا تتوقعها .

الحلقات Loops :

تتشابه جافا مع C++ في الحلقات for , while , do while و لا يوجد أي اختلاف

ملاحظة : اعتمدت جافا على أن تشابه C++ في معظم الأمور كي تتضمن انتقال المبرمجين إليها ففي وقتها عام 95 كان معظم المبرمجين معتمدين على C++ و VB6 و هذه من عناصر قوة الجافا كذلك

و يوجد في جافا كذلك الأمرين break , continue بنفس طريقة الاستخدام

أضف إلى ذلك **حلقة for المحسنة** ، " كانت موجودة في C++ لكن لم نتطرق إليها في منهاج برمجة1و 2 و هي " for each "

```
for(type name : arrayname)
{
....
}
```

مثال :

Java Code

```
public static void main(String[] args) {
    int sum=0,array[]={1,2,3,4};
    for(int a:array) // type name : name_of_array
    {
      sum+=a; //a=array[index]
    }
    System.out.println(sum); //sum=10
  }
```

التوابع Function :

من أبرز الأخطاء التي يُصادفها من انتقل من C++ إلى Java هي عدم وضع كلمة Static أمام التوابع \ أو المتغيرات العامة Global فكما أسلف في فصل " hello World " أن الدالة main وضعنا قبلها كلمة static لأنه لا حاجة إلى إنشاء Object جديد لها عن طريق new ، يوجد طريقتين إذا للتوابع \ و المتغيرات الـGlobal :

- الطريقة الأولى باستعمال Static :

Java Code

```
public class Test {
  static int var1 = 10;
  static public void f()
  {
    System.out.print("Function");
  }
  public static void main(String[] args) {
    System.out.println(var1);
    f();
  }
}
```

- الطريقة الثانية دون استعمال Static :

Java Code

```
public class Test {
 int var1 = 10;
  public void f()
  {
     System.out.print("Function");
  }

  public static void main(String[] args) {
     Test Object_1 = new Test();
     System.out.print(Object_1.var1);
     Object_1.f();
  }
}
```

المصفوفات و المؤشرات Arrays & Pointers :

مبدأ العمل تطابق طريقة الـc++ و بالإضافة إلى ذلك أمكنت الجافا طريقتين للتعريف عن مصفوفة :

```
int []a={1,3,2};
int b[]=new int[2];
```

أي يجوز وضع القوسين الكبيرين قبل أو بعد اسم المتحول.

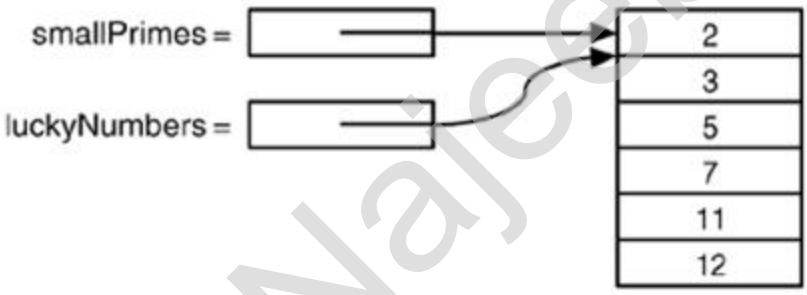
أتتذكر عندما تكلمنا عن GC و عن الـsafe program ؟ لن تتفاجئ أنه لا وجود للمؤشرات في جافا و ذلك حفاظاً على الحجر الذاكرية في الـheap . لكن كما قلت لك عزيزي المبرمج كل شي غير آمن يقابله شيء آمن ، و قد لجئت جافا إلى المصفوفات كبديل للمؤشرات ، انظر المثال الثاني و توقع ما هو الخرج !

Java Code

```
public static void main(String[] args) {
   int []smallPrimes=new int[]{2,3,5,7,11,12};
   int []luckyNumbers;
   luckyNumbers = smallPrimes; //Copy Arrays with reference
   luckyNumbers [0]=20;
   System.out.print(smallPrimes [0]);
}
```

أجل لقد حزرتها (أمل ذلك :D) فالخرج هو : 20 و ليس2 ، و ذلك لأن array2 و array1 أصبحتا تشيران إلى نفس الحجرة و الصورة التالية توضح المقصود :

Figure 3-15. Copying an array variable



ملاحظة "فكرة برمجية" : ستسفيد من هذا الأمر بجعل مصفوفة طولها 1 ، فسيشابه عملها عمل المؤشر مع بض التقييد الأمني.

الكلمات المحجوزة في جافا Key Words :

لاحظ أن الكثير من الكلمات الملونة بالأزرق Bold كانت موجودة في C++

strictfp	transient	throw	extends	abstract
volatile	catch	byte	**int**	**continue**
Const*	**goto***	**else**	**short**	**for**
float	**private**	import	try	**new**
native	**this**	**public**	**char**	**switch**
super	**break**	throws	final	assert
while	**double**	**case**	interface	**default**
instanceof	implements	**enum**	**static**	**if**
return	**protected**	**class**	**void**	package
do	boolean	finally	**long**	synchronized

* : كلمة محجوزة لكنها غير مستخدمة (ليس لها استعمال Not used)

ملاحظة "Java" : الكلمات (null ,True ,False) لا تعتبر من الكلمات المحجوزة بل هي محارف literals

شرح لكل كلمة محجوزة على الرابط التالي :

http://www.oracle.com/technetwork/java/glossary-135216.html

المراجع المستخدمة :

- **Site: A White Paper** - May 1996 - James Gosling - Henry McGilton http://java.sun.com/docs/white/langenv/
- **Tutorial**: http://download.oracle.com/javase/tutorial/java/
- **Book: Core Java™ 2 Volume I - Fundamentals, Seventh Edition** - By Cay S. Horstmann, Gary Cornell

ملاحظة : المحاضرة جهد شخصي بالاستعانة بالمراجع أعلاه و ليست من محاضرات دكتور المادة

انتهت المحاضرة

M.Najeeb

My -2- Lecture :

Your are now in JAVA

المواضيع التي سيتم التطرق إليها في هذه المحاضرة كالتالي :

1- التعامل مع الأصناف Classes و الكائنات Objects
- مقدمة
- محددات الوصول Modifiers
- إنشاء حزمة package
- instanceof

2- الوراثة Inheritance و الواجهات Interfaces
3- ملاحظات برمجية عامة
- Reference
- .length()
- القصر Casting

4- الفرق بين inner classes و static nested classes (للاطلاع)

M.Najeeb

Classes and Objects

الأمر ليس مختلفاً عن C++كثيراً ، فالكلمات المحجوزة المستخدمة متشابهة ، و بعض المفاهيم الجديدة التي سأشرحها و ستشعر أنك أصبحت في قلب الجافا J .

extends : **كلمة محجوزة** و هي للوارثة و هي بديل عن إشارة النقطتين الرأسيتين في C++
fields : **مصطلح** هي المتغيرات التي يتم تعريفها داخل الكلاس و يمكن رؤيتها في الكلاس كله (أي أنها ليس بارمترات أو متغيرات معرفة داخلmethods)

Super Class :**مصطلح** هو الكلاس الأب كما كنا نقول عنه من قبل في C++
Sub Class : **مصطلح** الكلاس الابن الموروث من الأب

سيتم شرح بقية المفاهيم أو الكلمات المحجوزة في حال استخدامها .

بعد إنشاء Class الخاص بالبرنامج و يكون عادةً Public
ملاحظة مهمة: يفضل أن يكون الكلاس الذي يحوي تابع الـmain هو الـpublic) و يجب أن يكون هنالك ملف java. لكل public class

Java Code

```
class Test {
   public void cout() {
      private String s="Hello Wolrd!";
      System.out.println(s);
   }
}

public class First_class {
   public static void main(String args[]) {
      Test t1 = new Test();
      t1.cout();
   }
```

ملاحظات:

- لا فرق سواء كان التصريح عن المتغيرات قبل أو بعد استخدامها ، لأن البرمجة الغرضية التوجه تبحث عن كل تصريح للمتغيرات قبل البدء بالاستعمال ، و نفس الأمر في الكلاس ، أي أنه يصح لو وضعنا Test أسفل First_class

- جميع الـmethods يتم تعريفها داخل الـClass و لا يجوز كتابة الـbody خارج الـClass كما في الـ++Cسابقاً

- التابع الباني Constructor بنفس الطريقة التي كانت في C++ ، أي نفس اسم الكلاس و بدون قيمة مرجعة و لكن لا حاجة للتابع الباني لإعطاء قيمة أولية للمتغيرات ، ففي JAVA إعطاء القيم الأولية داخل الكلاس مسموحٌ به

- قد تتوقع أن هذه الملاحظة ستتكلم عن التابع الهدام J .. لكن كما ذكرنا في المحاضرة 1 أنه في جافا هنالك GC مهمته حذف البيانات التي أصبحت ضائعة أو الغير مستخدمة ، لذلك فلا يوجد في جافا أمر delete أو free فلا وجود للتابع الهادم

إنشاء حزمة(مكتبة) package

قم بإنشاء ملف جديد .java

اكتب في بداية البرنامج :

package package_name;

هذه العبارة تدل على أن جميع الأصناف classes الموجودة داخل الملف ستتضمن في حزمة (مكتبة) جديدة اسمها package_name

بعد الانتهاء من كتابة Classes.. و لاستدعائها إلى ملف أخر نكتب

import package_name.*;

أو يمكننا تحديد الكلاسات في حال كان هنالك أكثر من واحدة و ذلك بكتابة اسم الكلاس عوضاً عن الـ*

import static

يوجد نوع ثانٍ من تضمين الحزم و هو الـstatic ، سأضرب مثالاً من الحزم القياسية في اللغة ، و هي الـjava.lang.Math ، عندما نقوم بعمل import لها ، نلاحظ أننا لا نستطيع استعمال المتغير الثابت و الستاتيكي PI ، و لاستعمال تلك المتغيرات لابد لنا من أن نستدعي المكتبة هكذا :

import static java.lang.Math.PI;

الآن يمكننا استعمال **PI** بمفردها دون الحاجة لكتابة Math.PI أو حتى absو غيرها من التوابع و ذلك بوضع * بدلاً من PI

System.out.print(**PI**);

instanceof

كلمة محجوزة وظيفتها أن تتحقق اذا كان الـObject ينتمي لـClass معين أو لا .

لاحظ أنه في حال كانت المقارنة بين Object لـSubClass و SuperClass فإنه يرجع قيمة بولانيةture و لكن العكس غير صحيح .

```
Class1 obj = new Class1;
if( obj instanceof Class1 )
        System.out.println("obj is an object from Class1 ");
```

تطبيق : هنالك ميزة في JAVA و هي استخدام Object (كلمة محجوزة) و هي تشابه الـ.... , String , int و لكنها تقبل كل تلك القيم و تستقبلها ، مثال :

```
Object x="Ahmed";
x=12;
x=32.121f;
```

عند الحاجة لمقارنتها مع int مثلاً ، لا نقوم بكتابة ~~x instanceof int~~ و ذلك لأنint ليس كلاس ، إنما كلمة محجوزة ، لذلك فقد وفرت جافا الكلاس Integer الذي يمكن المقارنة به ، و التطبيقات لهذا الأمر كثيرة و متعددة

الوراثة Inheritance

كما تم التطرق إليها في البرمجة2 ، نعود إليها هنا و ستكون مركزة بشكل أكبر،

للوراثة نكتب extends عند تعريف الـSubClass ، و بعدها نكتب الـSuperClass

```
class Father {
……
}
class Son extends Father  {
……
}
```

لا يمكنك الوراثة من أكثر من كلاس ، أقصد أن ما بعد extends كلاس واحد فقط (الوراثة المضاعفة) علماً أنها مسموحة في C++ .
تسمى العلاقة بين كلاس الأب و كلاس الابن بـ is-a

بقية الأمور مثلما كانت في C++ ، أقصد بمحددات الوصول (public , private , protected) ، فقط الـpublic + protected يتم الوراثة منهما إلى الكلاس الجديد.

ملاحظة : للوصول إلى التابع الباني للأب نستخدم الكلمة المحجوزة super و نمرر بداخلها البارمترات اللازمة في التابع الباني ، مثال:

```
class Father{
      int number;
      Father(int x)  //بافتراض أنه الباني
      {
          this.number = x + 5;
          // this كلمة محجوزة تدل على عنوان الغرض الذي نعمل عليه حاليا
      }
}
class Son extends Father{
     Son(int a) {
          super(a);
           System.out.println(number);
      }

}
```

عندما ننشء Object عن طريق new Son(5) مثلاً سيكون الخرج 10 ، و تستخدم super كذلك للوصول إلى المتغيرات و التوابع الموجودة في كلاس الأب ، مثال super.number;

الواجهات interfaces

تستخدم الواجهة عندما يكون هنالك توابع مشتركة بين عدة Classes ، فهي تحسن من فهم الكود أكثر . و من شروطها :

1- أن لا يتم تعريف التابع بداخلها ، فالهدف منها فقط التصريح عن التوابع فقط
2- يسمح بإضافة متغيرات خاصة بالواجهة و كل متغير داخل الواجهة يعتبر static final أي لا يمكن تغيير قيمته في الكلاسات التي سوف يتم الوراثة منهعند عمل implements منها يجب أن يتم تعريف كل التوابع المصرحة بداخلها في الكلاس

3- التوابع المصرحة بداخلها ليس لها محدد وصول

implements هي **الكلمة المحجوزة** التي تسمح لك باستخدام التوابع المصرحة في الواجهة و تعريفها و المثال التالي يوضح :

```
interface Money{
    void in_money(int amount);
    int out_money();
}
class Bank implements Money{
int total=0;
public void in_money(int a)
{
     total+=a;
}
public int out_money()
{
   return total;
}
}
```

قد تتساءل ما الفائدة اذاً منه ؟ أليست الوراثة أجمل منها ؟
الأمر يتعلق بتنظيم الذاكرة أولاً و من ثم تنظيم العمل البرمجي و استخدام مفاهيم الـOOP ، تستطيع تنفيذ كل شيء بدونها ، لكنها تسهل عليك أن تعدل على الكود لاحقاً و أن تفهم نمذجية العمل بطريقة أفضل ، و هي في النهاية فصل لطبقات الكود لعزلها عن بعضها البعض . " سيتم التوسع بها لاحقاً عندما يتم دراسة الـabstract "

Protected أجل ... لكن ليست كما تتخيلها !!

العنوان السابق ليس عنوان الفقرة D: لكنني أثرت محدد الوصول protected عن غيره لأنه جعلني أنتبه إلى هذا الأمر ، في C++ كان يسمح هذا المحدد للوصول فقط للكلاس الموروث منه . أما private كان فقط للكلاس الذي فيه هذا المتغير أو التابع ، و public يسمح للوصول من أي مكان . و مازال هكذا هي الأمور في جافا ، لكن في Java هناك كما قلنا تم إضافة مفهوم الـpackage ، الجدول التالي يشرح ما أقصده :

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Y= Yes
N= No

لاحظ أن الـPackage يمكن رؤيتها للـprotected ، ما أقصده هنا لو صار معك مثلما بدأت في الكلاسات .. أي هكذا:

Java Code

```
public class Program {
   public static void main(String[] args) {
      Test t1 = new Test();
      t1.s = 6.2f;
   }
}
class Test {
   private int x;
   protected float s;
   public int a;
   int ss;
}
```

الفكرة التي أريد إيضاحها أنني قمت بوضع كلاسين في نفس الملف ، اصطلاحاً اعتبر الملف الواحد أي نفس الـpackage و بالرجوع إلى الجدول نرى أن محدد الوصول الـprotected يُسمح الوصول إليه فيها ، و اذا أردت أن تُطبق بشكل أصح كان من الأوجب أن نضع كل class في ملف بمفرده مستقلاً عن بقية الكلاسات ، و ذلك حفاظاً على أمن البيانات في كل صنف .

ملاحظات برمجية عامة :

1- .length()

عندما تريد أن تعرف اذا كانت السلسلة المحرفية String تحوي كلاماً بداخلها أم لا ، لا تستخدم هذه الطريقة

```
String x="";
if(x.equals(""))
        .....
```

بل استخدم عوضاً عنها :

```
if(x.length()==0)
        .....
```

لأن equals تدخلك في حلقة تكرارية كي تتفحص كل محرف ، أما الطريقة الثانية لهذا الغرض فهي فقط تفحص واحد

ملاحظة : length. موجود في المصفوفات أيضاً و هو يعيد طول المصفوفة\السلسلة

لمعرفة طول مصفوفة ثنائية

```
int a[][]=new int[5][10];
int rows=a.length;
```

int cloumsn=a[0].length;
يجوز وضع أي رقم دليلي بشرط أن يكون أقل من عدد الأسطر ، ولو كان أكثر فإنه سوف يعطي خطأ وقت التنفيذ//

Refrence & JAVA -2

هذه الـ& ليست للعنونة **J** بل هي الـand ، فحتى المرجع ممنوع في جافا .. قد تظن أنها الكارثة و أنك لن تستطيع أن تنفذ كل مشاريعك السابقة المعتمدة على تغيير المعاملات عن طريق العناوين . إجابتي لك أن جافا قد حدّت لك من بعض الصلاحيات (لن أقول لك لا تستخدم التوابع أو لا تفكر بأي شيء مرجعي)، لكن لا يزال المعامل new موجوداً ، و هو صلة وصلك إلى العناوين بشكل نسبي . سأتناول مثال عملية التبديل SWAP .

في C++ كان من الممكن أن ننفذ تابع SWAP باستعمال التمرير بالمرجع كالتالي :

```
void swap(int &x,int &y)
{
    int temp=x;
    x=y;
    y=temp;
}
```

Java لا تمرر بالمرجع ، لكن الكود التالي سيثير فضولك
الطريقة الأولى : (باستخدام الـobject)

```
class Swap_var {
  public int x;
}

public class Program {
  public static void swap(Swap_var a, Swap_var b) {
    Swap_var temp = new Swap_var();
    temp.x = a.x;
    a.x = b.x;
    b.x = temp.x;
  }

  static public void main(String args[]) {
    Swap_var obj1 = new Swap_var();
    Swap_var obj2 = new Swap_var();
    obj1.x = 5;
    obj2.x = 10;
    swap(obj1, obj2);     //if you print them : obj1.x=10 , obj2.x=5
  }
}
```

لاحظ أننا لم نمرر قيم ، بل مررنا عناوين ، و ذلك لأنها objects من الكلاس Swap_var و لاحظ أيضاً أننا في عملية الـSwap قمنا بتغير القيم ، و ليست العناوين ، هذه الملاحظة قد تتذكرها فلقد مررنا عليها في البرمجة2 (و بالتحديد في فحص العملي **J**) فتغير العناوين سيؤدي إلى إنشاء Object جديد و تغيير قيمه لن يؤثر على الـparameter Object

الطريقة الثانية : (المصفوفة) و قد سبق شرحها في المحاضرة السابقة

```
public class Program {
   public static void swap(int []a,int []b) {
         int temp=a[0];
         a[0]=b[0];
         b[0]=temp;
   }
   static public void main(String args[]) {
         int []a=new int[1];
         int []b=new int[1];
      swap( a , b );
   }
}
```

4- Casting القصر

الفرق بين التحويل Casting و استعمال method جاهز هو أن الـcast يحول بين الأنواع بغض النظر عن الخطأ الذي ينتج مثل تحويل chat إلى int

```
char ch='a';
int x=(int) ch;      //97
```

نلاحظ أننا لو طبعنا x لكانت القيمة 97 و هي قيمة الأسكي الخاصة بالمحرف a

أما لو استخدمنا الـmethod الخاص بتحويل من String إلى int

```
Integer.parseInt(String parameter )      //return int
```

فإنه سيطبع لدينا المحرف نفسه " لكنه تحول إلى عدد int و يمكننا التعامل معهعددياً "

لاحظ أننا استخدمنا كلاس Integer و تأكد من أن القيمة المرجعة هي من النوع int و ليس من الكلاس Integer

ملاحظة : من الممكن أن نحول char إلى String عن طريق Character.toString(char parameter)

للاطلاع : الفرق بين inner classes و static nested classes

معروف أن الـClass لا يمكن أن يكون static في العادة ، لكن يجوز ذلك في حالة خاصة و هي nested class ، قبل أن أبدأ بها أريد أن أضع مثالاً للـinner class

مثال : للوصول إلى in نحتاج إلى الوصول إلى out

```
class out {
    class in {
   }
}
```

يجوز كتابة داخل الـout Class :

```
in obj = new in();
```

ملاحظة : في هذه الحالة تسمى العلاقة بين in و out علاقة has-a

(في هذه الحالة كافة المتغيرات في الصنف out حتى الـprivate يمكن الوصول إليها و ذلك لآن الصنف ضمني)
و اذا أردنا كتابته خارج الصنف out (هنا الـprivate لا يمكن الوصول إليه) يجب أولاً التصريح عن object من الصنف out ، و من ثم التصريح عن object للصنف in لكن بشكل ضمني و ها هو الكود :

```
out obj_out=new out();
out.in obj_in=obj_out.new in();
```

لكن يوجد نوع أخر يسمى Static nested class
يشابه السابق إلا أن الـclass الداخلي من الدرجة الأولى (محتواه في كلاس فوقي واحد) يكون معرفاً static

```
public class program {
   static class S_in {
  static class w{  } // ERORR!!! لآنه صنف داخلي من الدرجة الثانية
   }
   static class e {
   }
}
```

للتصريح عن Object من الصنف S_in:

خارج الـS_in Class يمكننا أن ننشء object للصنف ذلك بكتابة :

```
program.S_in obj=new program.S_in();
```

إعادة التعريف override

هي إعادة تعريف تابع method لتغيير تعريفه القديم و إنشاء تعريف جديد، و سأضرب مثال طباعة الـobject

Java Code

```
class work
{
   private String name="worker n.";
   private static int  Totalid=1;
   worker()
   {
      ++Totalid;
   }
   private int id=Totalid;
   public String toString()      //Here is the override !
   {
      return name+id;
   }
}
public class First_class
{
   public static void main(String args[])
   {
      work a1=new work();
      work a2=new work();
      System.out.println(a1);  //it will print : worker1
      System.out.println(a2);  //it will print: worker2
   }
}
```

في الحالة الطبيعية تكون طباعة أي object بطباعة اسم الكلاس @ العنوان ، لكننا هنا قمنا بإعادة تعريف التابع toString المخصص للتحويل إلى String عند استدعاء الـobject إلى ما نريده.

ملاحظة : يوجد الكثير من التوابع المسبقة التعريف في كل كلاس ، و منها ()toString المخصص لطباعة عنوان الغرض و ()finalize المخصص لاستدعاء الـGC و غيرها الكثير (سأضعها في أخر المحاضرة)

ملاحظة : بمجرد إعادة تعريف أي تابع سواء كان موروث أو ما شابهه فإن إعادة التعريف تُطبق على كل الكلاس التي تحوي التابع المعرف جديداً

سنستخدم مفهوم إعادة التعريف الـoverrideكثيراً عند تعاملنا مع التجريد abstract و الواجهات interfaces

التجريد abstract

عندما تريد لتابع ما أن لا يحوي أي تعريف له ، فإننا نسبقه بالكلمة المحجوزة abstract و هي تعني "مجرد" أي لا يحوي أي تعريف عنه ، و لا يمكننا استعمال أي تابع abstract ما لم يكن الكلاس الذي يحويه هو أيضاً abstract و لا يمكن استعمال هذا الكلاس إلا بعد الوراثة منه ، و ذلك كي لا نقوم باستخدامه خطأً . و بعد الوراثة يجب علينا أن نعيد تعريف كل تابع موجود في الـabstract class

Java Code

```
abstract class test {       //abstract class "Must" be all the methods inside it are abstract too
   abstract void test_method();
}

class A extends test
{
   public void test_method()      //Here is the override
   {
      System.out.println("hi test_method");
   }
}

public class First_class {
   public static void main(String args[])
   {
      A a1=new A();
      a1.test_method();
   }
}
```

حسناً .. و ماذا بعد .. يبدو أنك بدأت بالغضب D: .. فما فائدة التجريد مادام هنالك الواجهة !!

لكي أخلصك من دوامة الاستفهامات عليك بالجدول التالي:

Interface الواجهة	Abstract التجريد في الكلاس
كل ما فيه هو abstract methods فقط	لا يشترط أن تكون جميع التوابع abstract ، بل تابع واحد على الأقل
التوابع تكون public في حالة عدم imlementation	يمكن للتوابع أن تكون protected أو static
ٱمتغيرات تكون ثابتة دائماً	يمكن للمتغيرات أن تكون ثابتة أو غير ثابتة

تعدد الأشكال Polymorphism

أتى مفهموم تعدد الأشكال للتقليل من استعمال الشرط if أو switchو ذلك تسهيلاً للمبرمج و المطور و لقارئ الكود و للآلة لفهم الكود و القدرة على التعديل عليه مستقبلاً و تطويره ، و الأهم من ذلك أنها أتت لكي تتدعم العمومية أي الابتعاد عن تخصيص كلاس معين لغرض معين ، بل جعل كلاس الأب كلاس عام " ما يصح على الأب يصح على الابن " ، و تعتمد بشكل أساسي على الوراثة و التجريد.

بفرض أنك تريد إنشاء كلاس worker فيه توابع و متغيرة خاصة به ، سنهتم بتابع معين و ليكن على سبيل المثال تابع لزيادة الراتب add . نريد أن يكون في حال كان الـworker دكتور سيحصل على زيادة 10% أما في حال كان الـworker مدرس سيحصل على زيادة 5% .

كي تفكر بطريقة غرضية التوجه ينبغي عليك أن :

- **تحدد من الأب أولاً** ، ففي مثالنا واضح أنه الـworker
- بما أنه لدينا كلاس أب ، فإننا سنورث منه كلاسات أبناء ، هل يلزمنا إعادة تعريف لأحد التوابع ؟ كي نجعله abstract class ؟ ، ففي مثالنا أجل يلزمنا ، و ذلك لأننا نريد إعادة تعريف تابع زيادة الراتب add
- ثم تقوم بعملية الوراثة و تعرف وظيفة التابع في كل كلاس ابن "يمكن إضافة توابع و متغيرات خاصة لكل كلاس كما كنا نفعل في الوراثة من قبل"
- و عند إنشاء غرض للكلاس يفضل أن تتبع طريقة ما يصح على الأب يصح على الابن

Java Code

و هذا مثال للكود :

```
abstract class worker
{
   private String name;
   protected double salary;
   public worker()
   {
   };
   public worker(String name,double salary)
   {
      this.name=name;
      this.salary=salary;
   }
   public String toString()          //Here is an overriding
   {
      return(name + "\t" + salary);
   }
   public abstract void add();     //this is the abstract method
}
class Dr extends worker
{
   public Dr(String name,double salary)
   {
      super(name,salary);          //we need the constructer of worker
   }
   public void add()       //this is the override for the abstract method
   {
      salary=salary*1.1;
   }
}
```

```
class En extends worker
{
   public En(String name,double salary)
   {
      super(name,salary);          //we need the constructer of worker
   }
   public void add()       //this is the override for the abstract method
   {
      salary=salary*1.05;
   }
}
```

كلاس الفحص

```
public class test
{
   public static void main(String []args)
   {
      worker a[]=new worker[10];
   //هنا تم التصريح عن غرض من النوع الأب
   //نريد الأن أن نجعل فرضاً  أول 5 دكاترة و أخر  5 مهندسين
   for(int i=0;i<a.length;++i)
   {
   if(i<5)
      a[i]=new Dr("name",500d);
   else
      a[i]=new En("name",500d);
   }
      //هنا الفائدة من تعدد الأشكال .. مصفوفة لكن فيها أصناف أبناء مشكلة من نفس الأب
      a[3].add();   //here *1.1
      a[7].add();   //here *1.05
   }
}
```

أفكار برمجية:

1- كيف تنشئ غرض اذا كان التابع البناء Private

Java Code

```
class a
{
   private a()
   {
      System.out.println("Constructer");
   }
   public static a how()
   {
      return new a();
   }
   public void hi()
   {
      System.out.println("hhhi");
   }
}
public class test
{
   public static void main(String []args)
   {
     // a a1=new a();     //This is a Wrong , Because the constructer is private
      a a1=a.how();
   }
}
```

M.Najeeb