رحلة استكشافية للغة البرمجة جافا

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class Fenetre extends JFrame {
  private JPanel container = new JPanel();
  private JComboBox combo = new JComboBox();
  private JLabel label = new JLabel("Une ComboBox");

  public Fenetre(){
    this.setTitle("Animation");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLocationRelativeTo(null);
    container.setBackground(Color.white);
    container.setLayout(new BorderLayout());
    combo.setPreferredSize(new Dimension(100, 20));

    JPanel top = new JPanel();
    top.add(label);
    top.add(combo);
    container.add(top, BorderLayout.NORTH);
    this.setContentPane(container);
    this.setVisible(true);
  }
}
```

الإصدار الثالث

تأليف: معتز عبد العظيم الطاهر

كود لبرمجيات الكمبيوتر

رحلة استكشافية للغة البرمجة جافا

الإصدار الثالث

أول إصدار: ذي القعدة 1433 هجرية الموافق أكتوبر 2012 ميلادية
الإصدار الحالي: رجب 1444 هجرية الموافق فبراير 2023 ميلادية

مقدمة

بسم الله الرحمن الرحيم والصلاة والسلام على أشرف الأنبياء والمرسلين، نبينا محمد وعلى آله وصحبه أجمعين. أما بعد. الهدف من هذا الكتاب هو تعريف المبرمج في فترة وجيزة وكمدخل سريع للغة البرمجة جافا باستخدام أداة التطوير NetBeans. ثم التعمّق تدريجياً في المفاهيم الأساسية في اللغة والبرمجة عموماً. وبهذا يكون هذا الكتاب موجه لمن لديه خبرة في لغة برمجة أخرى حتى لو كانت قليلة. كذلك يُمكن الإستفادة من هذا الكتاب كمقدمة لتعلم برمجة الموبايل بإستخدام جافا، مثل نظام أندرويد أو جافا موبايل وذلك لأن أساس اللغة واحد.
وقد كُتب هذا الكتاب بخلفية عملية وليس أكاديمية، والطريقة العملية تتطلب التركيز على ما يحتاج إليه المبرمج فعلياً أثناء تطويره للبرامج الموجهة للاستخدام في الواقع العملي.

لغة جافا

ظهرت لغة البرمجة جافا في عام 1995 م. وهي لغة متعددة الأغراض ومتعددة المنصات تصلح لعدد كبير من التطبيقات. ومترجم جافا يُنتج ملفات في شكل Byte code وهو يختلف عن الملفات التفيذية التي تنتج عن لغات البرمجة الأخرى مثل سي وباسكال.
وتحتاج البرامج المُنتجة بواسطة مترجم جافا إلى منصة في أنظمة التشغيل المختلفة لتتمكن برامجها من العمل في هذه الأنظمة.
وهذه المنصة تُسمى آلة جافا الإفتراضية Java Virtual Machine أو إختصاراً بـ JVM و تُسمى أحياناً Java Run-time.

آلة جافا الإفتراضية JVM

تتوفر هذه المنصة في عدد كبير من أنظمة التشغيل، ولابد من التأكد من وجود هذه المنصة أو الآلة الإفتراضية قبل تشغيل برنامج جافا. وكل نظام تشغيل يحتاج لآلة إفتراضية خاصة به. مثلاً نظام وندوز 32 بت يحتاج لآلة إفتراضية مخصصة لوندوز 32 بت، أما نُسخة وندوز 64 بت فهي تحتاج لآلة إفتراضية 64 بت. وهذا مثال لإسم ملف لتثبيت آلة جافا الإفتراضية لنظام وندوز 64 بت:
jre-8u51-windows-x64.exe
وهو يُمثل نسخة جافا 1.8 أو مايُسمى جافا 8.

واسم الملف التالي يُمثل حزمة تحتوي على الآلة الإفتراضية لجافا 8 لنظام أوبونتو :

```
openjdk-8-jre
```

وتختلف معماريتها حسب معمارية نظام أوبونتو، فإذا كان النظام هو 32 بت تكون حزمة جافا 32 بت، وإذا كان 64 بت تكون حزمة جافا 64 بت. لكن يمكن تثبيت جافا 32 بت في نظام أوبونتو 64 بت – كذلك في نظام وندوز– وذلك لأن بعض البرامج تتطلب جافا 32 بت، لكن لا يمكن تثبيت جافا 64 بت في نظام تشغيل 32 بت.

عند إنتاج برامج جافا يُمكن تشغيلها في أي نظام تشغيل مباشرة عند وجود الآلة الإفتراضية المناسبة، ولا يحتاج البرنامج لإعادة ترجمة حتى يعمل في أنظمة غير النظام الذي طُور البرنامج فيه. مثلاً يُمكن تطوير برنامج جافا في بيئة لينكس لإنتاج برامج تُنقل

وتُشغل مباشرة في وندوز أو ماكنتوش. وتختلف عنها لغة سي وأوبجكت باسكال في أنها تحتاج لإعادة ترجمة البرامج مرة أخرى في كل نظام تشغيل على حده قبل تشغيل تلك البرامج. لكن برامج لغة سي وأوبجكت باسكال لاتحتاج لآلة إفتراضية في أنظمة التشغيل بل تتعامل مع نظام التشغيل ومكتباته مباشرة.

أدوات تطوير جافا Java SDK

آلة جافا الإفتراضية السابقة تُمكّن برامج جافا من العمل في نظام التشغيل، لكنها لا تحتوي على مترجم، لذلك لا يمكن كتابة برامج جافا وتطويرها بها، ولتطوير وترجمة وتنقيح برامج جافا وتحويلها إلى byte code لابد من الحصول على الـ (Software Development Kit (SDK الخاص بالجافا، أي مايُعرف بـالـ Java SDK. وهو يأتي في شكل برنامج للتثبيت به مترجم جافا (compiler)، ومنقح (debugger)، وآلة جافا الإفتراضية، أي لانحتاج لتثبيت آلة جافا الإفتراضية لوحدها عند تثبيت Java SDK. واسم الملف التالي يُمثل الـ Java SDK لبيئة وندوز:

```
jdk-7u51-windows-x64.exe
```

وهو مخصص لنظام وندوز 64 بت ويُمثّل جافا 7. ونُلاحظ أنه يبدأ بالإسم jdk وهو إختصار لـ Java Development Kit.
والملف التالي يُمثل حزمةJava SDK لنظام التشغيل أوبونتو:

```
openjdk-8-jdk
```

ترخيص جافا

حدث تغيير مهم في ترخيص جافا، فبعد أن كانت أدوات التطوير وآلة جافا الإفتراضية مجانية منذ أطلقتها شركة صن ميكروسيستمز، تغيير ذلك في عام 2019 من شركة أوراكل التي اشترت شركة صن ومعها جافا، حيث اصبح استخدام جافا يتطلب اشتراك وترخيص شهري، هذا بالنسبة لـ Oracle Java SE أما OpenJDK فلم تتأثر و مازال استخدامها مجانياً ولا يتطلب ترخيص أو دفع اشتراكات شهرية. لذلك يجب مراجعة آخر المعلومات المتوفرة عن ترخيص جافا قبل البداية في دراستها أو أخذ القرار لاستخدامها في مؤسسة ما.

بيئة التطوير NetBeans

وهي من أفضل بيئات التطوير للغة جافا، وقد كُتبت باستخدام لغة جافا نفسها بواسطة شركة أوراكل صاحبة تلك اللغة، بعد ذلك سُلّمت إلى مؤسسة أباتشي Apache.org للاستمرار بتطويرها وإصدار نُسخ جديدة منها.
يُمكن استخدام هذه الأداة لتطوير برامج بلغات برمجة أخرى غير الجافا مثل برامج PHP و سي + +.

توجد أدوات تطوير أخرى مشهورة و هي Eclipse وهي مستخدمة من قبل مبرمجين كُثر، و أخرى تسمى IntelliJ والتي بُنيت عليها بيئة تطوير أندرويد.
جميع بيئات التطوير هذه تحتاج إلى تثبيت Java SDK أولاً قبل تثبيتها

المؤلف: معتز عبدالعظيم

أعمل مطور برامج ومعماري أنظمة، وقد كُنت استخدم فقط لغة أوبجكت باسكال كلغة برمجة أساسية في الماضي، متمثلة في دلفي وفري باسكال، لكن منذ عام 2011 بدأت تعلم لغة جافا وكتبت بها عدد من البرامج. وكان سبب تعلمي لها واعتمادي لها في تطوير كثير من البرامج هو:

1. أنه يوجد عدد كبير من المبرمجين يستخدمون لغة جافا، بل أن معظمهم درسها في الجامعة. لذلك يُمكن أن تكون لغة مشتركة بين عدد كبير من المبرمجين.

2. يوجد دعم كبير لها من حيث المكتبات ومن حيث حل المشكلات، وذلك بسبب أنها أعتمدت لوقت طويل كلغة موجهة للأعمال الكبيرة والمؤسسات Business and Enterprise

3. أنها كانت مجانية ويتوفر لها أدوات تطوير متكاملة ذات إمكانات عالية في عدد من المنصات. ماعلى المبرمج إلا إختيار المنصة المناسبة له (نرجو مراجعة فقرة ترخيص جافا)

4. تدعم البرمجة الكائنية بصورة قوية، والبرمجة الكائنية تشكل أداة أساسية لتطوير البرامج بطريقة مثالية.

5. أن البرامج الناتجة عنها متعددة المنصات والمعماريات بمعنى الكلمة، ولايحتاج المُبرمج إنتاج عدد من الملفات التنفيذية لكل معمارية على حده. بل يحتاج لإنتاج ملف byte code واحد يكفي لمعظم المعماريات وأنظمة تشغيل الكمبيوتر المعروفة.

6. أداة التطوير Netbeans وطريقة تقسيم الحزم packages مناسبة للبرامج الكبيرة والتي تحتاج تقسيماً منطقياً والوصول لتلك الأقسام بسرعة وسهولة وتُسهل كذلك التشارك في كتابة البرامج.

ثم في بداية عام 2017 بدأنا التحول إلى لغة Go

ترخيص الكتاب

هذا الكتاب مجاني تحت ترخيص

creative commons

CC BY-SA 3.0

ملحوظة هامة:

لا يُفضّل نسخ ثم اللصق في بيئة NetBeans من هذا الكتاب لأنه تُنقل أحياناً رموز غير مرئية تتسب في تعثر ترجمة البرامج. لذلك من الأفضل كتابة الأمثلة يدوياً، لكن يجب كتابة الكود مطابقاً للمثال مثلاً حالة الأحرف الإنجليزية (الأحرف الكبيرة والصغيرة)

المحتويات

مقدمة.....2
لغة جافا.....2
آلة جافا الإفتراضية JVM.....2
أدوات تطوير جافا Java SDK.....3
ترخيص جافا.....3
بيئة التطوير NetBeans.....4
المؤلف: معتز عبدالعظيم.....4
ترخيص الكتاب.....5
تثبيت جافا SDK.....8
كتابة أول برنامج.....10
تثبيت أداة التطوير NetBeans.....12
البرنامج الأول بواسطة NetBeans.....13
الملفات Files.....19
كتابة نص في ملف.....23
القراءة من ملف نصي.....25
استعراض أسماء الملفات.....28
سلسلة البيانات Streams.....30
نسخ الملفات.....31
الخصائص Properties.....34
المصفوفات arrays.....37
السلاسل ArrayList.....38
تعريف الكائنات والذاكرة.....41
برنامج الواجهة رسومية GUI.....44
الفورم الثاني.....47
برنامج اختيار الملف.....49
كتابة فئة كائن جديد New Class.....51
المتغيرات والإجراءات الساكنة (static).....54
قاعدة البيانات SQLite.....56
برنامج لقراءة قاعدة بيانات SQLite.....57
الوراثة inheritance.....64
تكرار حدث بواسطة مؤقت.....68
برمجة الويب باستخدام جافا.....71
تثبيت مخدم الويب.....71

أول برنامج ويب..75
تثبيت برامج الويب...80
تقنية JSP..81
تضمين ملف jsp داخل Servlet...86
خدمات الويب Web services..88
برنامج خدمة ويب للكتابة في ملف..90
برنامج عميل خدمة ويب Web service client..97
القراءة من مخدم ويب بواسطة HTTP..100
خدمات ويب الـ RESTFull...102
استخدام نسق JSON...110

تثبيت جافا SDK

قبل بداية تطوير البرامج باستخدام لغة جافا لابد من عمل إختبار لوجودها في الحاسوب الذي نستخدمه، فإن لم توجد يجب تثبيت الـ SDK المناسبة لنظام التشغيل الذي نستخدمه.
لاختبار هل آلة جافا الإفتراضية موجودة أم لا، نُشّغل برنامج سطر الأوامر عن طريق برنامج cmd في نظام وندوز أو terminal في نظام لينكس، ثم نكتب الأمر java -version فإذا كانت النتيجة مشابهة للتالي:

```
java -version
java version "1.8.0_111"
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
Java HotSpot(TM) Server VM (build 25.111-b14, mixed mode)
```

فهذا يعني أن Java runtime مثبتة أو ما يُعرف بـ Java JRE ، وكما يظهر أعلاه توجد النسخة رقم 1.8 من جافا، أو جافا 8. وهي تعني إمكانية تشغيل برامج جافا، لكن لا نستطيع تطوير برامج جافا بها.
أما إذا كانت النتيجة بأن الأمر java غير موجود فهي تعني أن آلة جافا الإفتراضية غير موجودة.

الإختبار الثاني هو التأكد من وجود Java JDK/SDK والتي نحتاج إليها لتطوير برامج جافا، أي يحتاجها المبرمج، أما المستخدم العادي الذي نستهدفه لتشغيل برامج جافا فيكفيه آلة جافا الإفتراضية JRE فقط.
نكتب javac -version في سطر الأوامر و javac يعني مترجم لغة جافا، فإذا كانت النتيجة مشابهة لأدناه فتعني أن مترجم جافا موجود والذي هو جزء من Java JDK:

```
javac -version
javac 1.8.0_181
```

أما إذا كانت النتيجة أن الأمر غير صحيح أو غير موجود فهي تعني أن Java JDK غير مثبتة، في هذه الحال يمكن تثبيتها حسب نظام التشغيل في الخطوات المذكورة في الفقرة التالية.

ملحوظة:
يمكن أن تكون جافا run-time JRE أو JDK مثبتة أو موجودة في نظام التشغيل، لكن غير معّرفة في مسار التشغيل path لذلك عند تشغيل الأوامر java أو javac تكون النتيجة أن هذه الأوامر غير موجودة. يمكن البحث في نظام وندوز في الدليل:

C:\Program Files\Java\

وهذا مثال لدليل جافا JRE 1.8 في نظام وندوز:

C:\Program Files\Java\jre1.8.0_31\bin

في حال أن هذا الدليل غير معرف في المسار يمكن التوجه إليه أولاً ثم تشغيل الأمر java لكن الأفضل إضافة هذا الدليل للمسار للتمكن من تشغيل الأمر java من أي دليل في نظام التشغيل.
كذلك يمكن أن تكون هُناك أكثر من نُسخة لجافا مثلاً JRE بالإضافة لـ JDK، كذلك يمكن أن تكون هُناك أكثر من إصدار، مثلاً جافا 8 بالإضافة إلى 9، ويمكن الإختيار بينها لجعل واحدة إفتراضية.

تثبيت جافا SDK في نظام لينكس (أبونتو، أو دبيان ومشتقاتها):

لتثبيت جافا SDK نكتب الأمر التالي في نافذة سطر الأوامر terminal :

```
sudo apt-get install openjdk-9-jdk
```

بهذه الطريقة تُثبت حزمة جافا 9 في نظام أوبونتو أو دبيان، و لابد أن ننبه بأن نتحصل على آخر نُسخة متوفرة لنظام التشغيل من جافا، فإن لم توجد نرجع لنُسخة سابقة، مثلاً النُسخة رقم 8:

```
sudo apt-get install openjdk-8-jdk
```

ثم نتأكد من التثبيت بإعادة كتابة الأوامر السابقة في سطر الأوامر:

```
java -version
```

```
javac -version
```

بعض توزيعات لينكس تأتي معها آلة جافا الإفتراضية مثبتة مسبقاً، مثل لينكس مينت، و راسبيان، لكن ربما آلة جافا الإفتراضية JRE فقط وليست أدوات التطوير JDK/SDK

تثبيت جافا SDK في نظام وندوز:

يمكننا الحصول عليها من موقع جافا www.java.com أو من موقع شركة أوراكل. كذلك يمكن البحث بدلالة هذه الجملة:

download java sdk for windows

ثم نتأكد أننا اخترنا ملف يُمثل نظام وندوز الذي نستخدمه، فإذا كان وندوز 64 بت فلابد من اختيار جافا 64 بت، وهكذا، ونتأكد أن الملف يحتوي على المقطع (sdk) لأن (jre) تحتوي فقط على الآلة الإفتراضية ولا تحتوي على المترجم. مثلاً الملف أدناه يُمثل جافا SDK 8 لنظام وندوز 64 بت:

```
jdk-8u101-windows-x64.exe
```

بعد ذلك نُثبت البرنامج ثم نختبره بواسطة سطر الأوامر كما سبق تفصيله.

كتابة أول برنامج

بعد تثبيت مترجم جافا يمكننا كتابة برنامج مبسطة للغة جافا باستخدام أي محرر للنصوص، مثلاً notepad في وندوز أو gedit في نظام لينكس.
نكتُب البرنامج التالي

```
public class First {

  public static void main(String args[]){

     System.out.println("Hello Java");

  }

}
```

الكلمة: *public* تعني أنه يمكن الوصول إلى المُعرّف من الخارج أو من حزمة أخرى.
و *class* تعني فئة، وقد أسميناها *First* في هذا المثال، حيث أن لغة جافا لغة كائنية، لابد من تعريف الوحدات كفئات أو كائنات. فهذه هي الفئة الرئيسة.
أما بالنسبة للدالة *main* فهي الدالة الرئيسة والتي لابد من توفرها بنفس شكلها المكتوبة والمعرفة به من حيث أنها public و static و void والأخيرة تعني أن الدالة لا ترجع شيئاً أي لا ترجع قيمة، فقط يُنفّذ الكود الذي بداخلها.
المُدخلات String []args فهي مصفوفة من المدخلات أو البراميترات التي يمكن إدخالها عند تشغيل البرنامج من سطر الأوامر، مثلاً:

```
First param1 param2
```

نحفظ الملف تحت إسم First.java، مطابقاً للإسم المستخدم بعد الكلمة *class* من حيث الحروف الكبيرة والصغيرة، في هذا المثال كلمة *First* تبدأ بحرف F كبير، وباقي الأحرف صغيرة، و لابد من التأكد من ذلك وإلا حدث خطأ.

بعد حفظه، نذهب لسطر الأوامر ثم نُترجمه بواسطة javac بالطريقة التالية في الدليل الذي حفظنا فيه الملف:

```
javac First.java
```

إذا لم يحدث خطأ سوف يظهر ملف جديد في نفس الدليل اسمه First.class وهو يُمثل ملف جافا المترجم إلى صيغة byte code والذي يمكن نقله وتشغيله في أي نظام تشغيل بدون إعادة ترجمته،
نُشغله بواسطة آلة جافا الإفتراضية باستخدام الأمر التالي:

```
java First
```

فيكون الناتج:

```
Hello Java
```

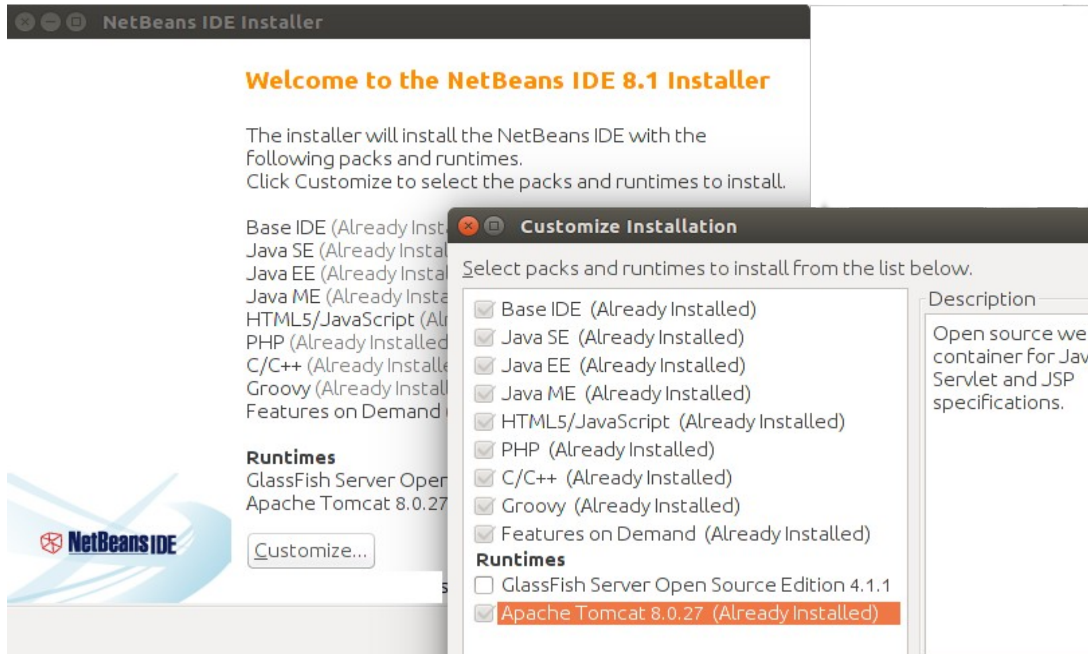
بهذه الطريقة كتبنا أول برنامج جافا وترجمناه إلى byte code ثم شغلناه بواسطة آلة جافا الإفتراضية JRE ، و إذا كان لدينا أكثر من نظام تشغيل يمكن نقل ملف الناتج من مترجم جافا First.class إلى النظام الآخر ثم تشغليه حتى لو لم يكن مترجم جافا موجود javac فقط يكفي وجود الـ JRE.

لكتابة برامج أكثر تعقيداً لابد من استخدام محرر وأداة تطوير متقدمة، مثل NetBeans التي استخدمناها مع أمثلة هذا الكتاب.

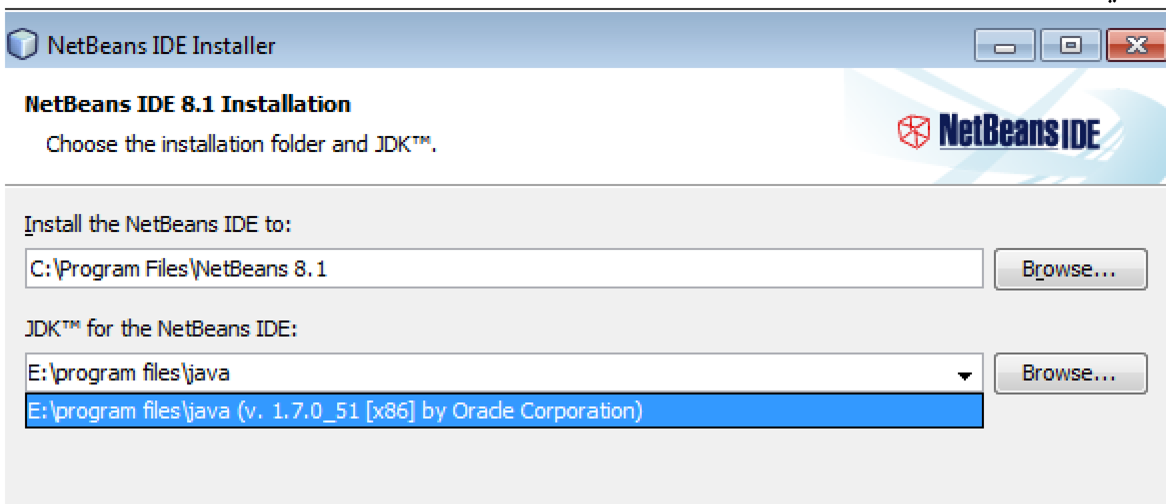
تثبيت أداة التطوير NetBeans

لتثبيت أداة التطوير NetBeans نُحملها أولاً من موقع netbeans.org. ولا بد من أن نختار النسخة المناسبة لنظام التشغيل، واختيار النسخة التي تحوي على كافة الميزات، منها برمجة الويب (مكتوب أمامها Java EE).

في بداية التثبيت لابد من اختيار Customize ثم اختيار Apache Tomcat بدلاً من Glassfish حيث سوف نستخدم Apache Tomcat كمخدم ويب لاحقاً:



ثم بعد ذلك في الشاشة التالية لابد من التأكد من اختيار جافا SDK:

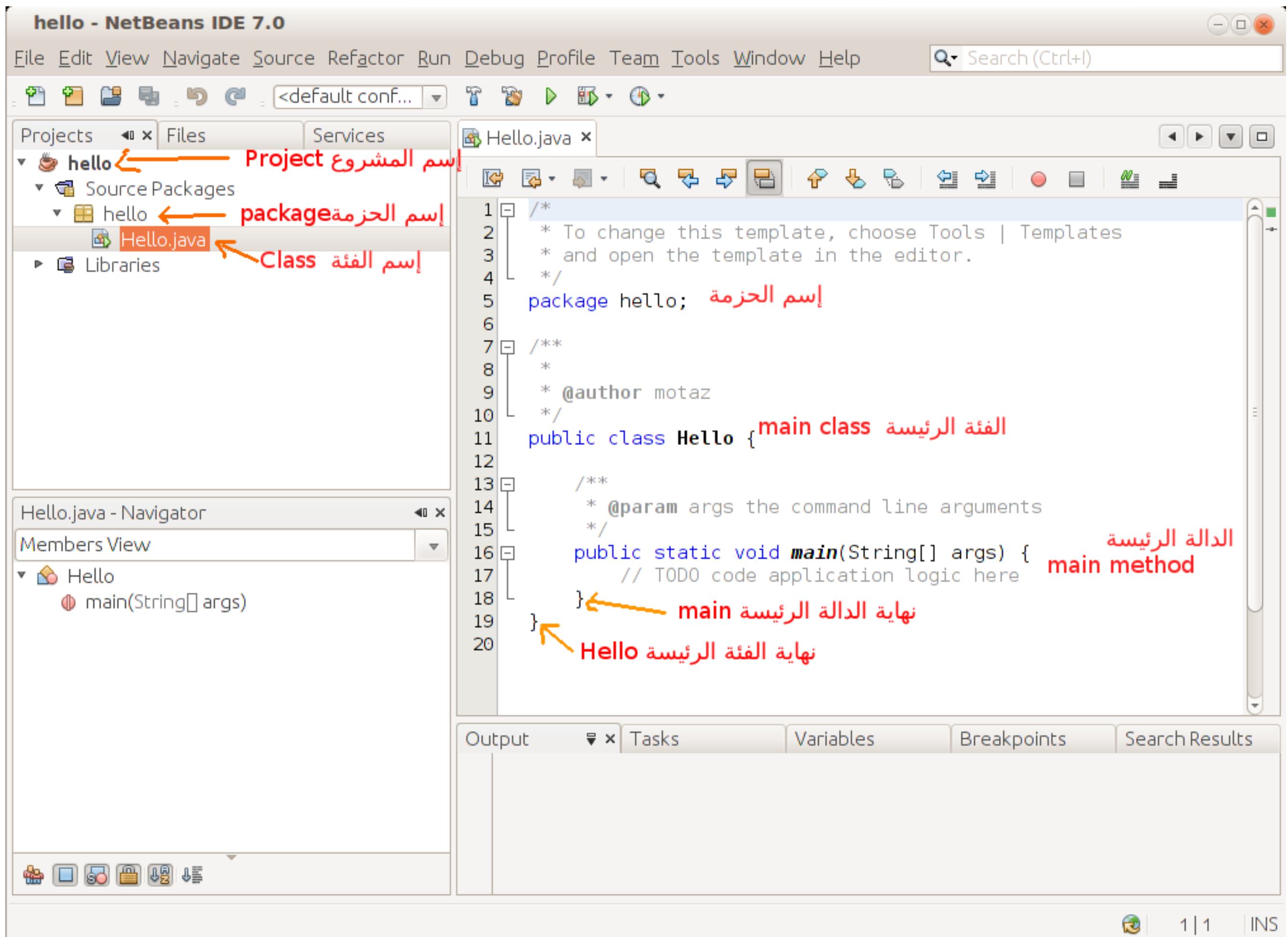


البرنامج الأول بواسطة NetBeans

بعد تثبيت آلة جافا الإفتراضية وأداة التطوير NetBeans نختار New/Project ثم Java/Java Application. ثم نُسمي البرنامج *hello* ليظهر لنا الكود التالي:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package hello;
/*
 * @author motaz
 */
public class Hello {
    /*
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}
```

فإذا لم يظهر الكود نفتح الملف *hello.java* بواسطة شاشة المشروع التي تظهر يسار شاشة NetBeans كما في الشكل التالي، نرجو تأملها مطوّلاً:



يوجد في الصورة أعلاه شرح باللون الأحمر لأجزاء برنامج جافا، و لابد من دراسة هذا الهيكل جيداً لأننا سوف نستخدمه في كثير من البرامج بإذن الله.

نلاحظ أن القوسين يمثلان حدود تلك الدالة { }، حيث أن القوس المفتوح } يعني بداية الدالة، والقوس المغلق { يعني نهاية الدالة

بعد ذلك نكتب السطر التالي داخل الدالة الرئيسة main

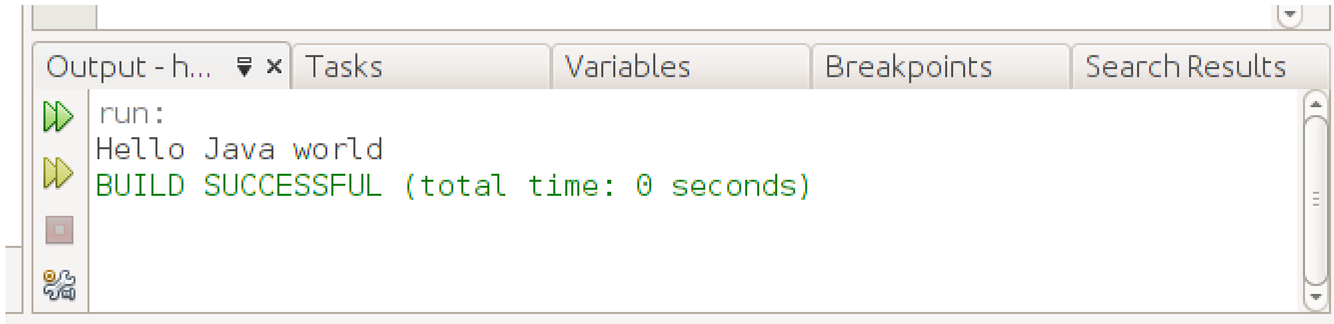
```
        System.out.print("Hello Java world\n");
```

ليصبح الكود كالتالي:

```
package hello;

public class Hello {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.print("Hello Java world\n");
    }
```

```
}
```

يُشغّل البرنامج عن طريق المفتاح F6 أو بواسطة الضغط على زر السهم الأخضر في أعلى وسط الشاشة لتظهر لنا المُخرجات في أسفل شاشة NetBeans



ملحوظة:

لغة جافا لغة حساسة للحروف الكبيرة والصغيرة case sensitive وهذا بالنسبة لأسماء المتغيرات، وأسماء الإجراءات، و أسماء الفئات، والكلمات المفتاحية، مثلاً في المثال السابق إذا كتبنا system بإستخدام الحرف s الصغير بدلاً من System بإستخدام حرف **S** الكبير، فإن البرنامج لا يعمل، وسوف يحدث خطأ في الترجمة أن كلمة system غير معروفة، كذلك النوع String لابد من كتابته بإستخدام حرف S كبير

كذلك لابد من عدم نسيان الفاصلة المنقوطة في نهاية كل عبارة ;

الأمر *System.out.print* يكتُب نص أو متغير في شاشة الطرفية.

الرمز n\ مهمته هو الإنتقال للسطر الجديد في الطرفية، يمكن استخدام *println* والذي ينقل مؤشر الكتابة للسطر التالي دون الحاجة لإستخدام رمز السطر الجديد n\ ليصبح الأمر كالتالي:

```
System.out.println("Hello Java world");
```

لتشغيل البرنامج الناتج خارج بيئة التطوير، نُنتج الملف التفنيذي بواسطة Build وذلك بالضغط على المفاتيح Shift + F11 . بعدها نبحث عن الدليل الذي يحتوي على برامج NetBeans ويكون اسمه في الغالب *NetBeansProjects* ثم داخل الدليل *hello* نجد دليل اسمه dist يحتوي على الملف التنفيذي. في هذه الحالة يكون اسمه *hello.jar*

يُمكن تنفيذ هذا البرنامج في سطر الأوامر في نظام التشغيل بواسطة كتابة الأمر التالي:

```
java -jar hello.jar
```

يُمكن نقل هذا الملف التنفيذي من نوع Byte code إلى أي نظام تشغيل آخر يحتوي على آلة جافا الإفتراضية ثم تنفيذه بهذه الطريقة. ونُلاحظ أن حجم الملف التنفيذي صغير نسبياً (حوالي كيلو ونصف) وذلك لأننا لم نستخدم مكتبات إضافية.

بعد ذلك نُغيير الكود إلى التالي:

```
        int num = 9;
        System.out.println(num + " * 2 = " + num * 2);
```

وهذه طريقة لتعريف متغير صحيح أسميناه *num* وأسندنا له قيمة إبتدائية هي 9

وفي السطر الذي يليه كتبنا قيمة المتغير، ثم قيمته مضروبة في الرقم 2. وهذا هو ناتج تشغيل البرنامج:

```
9 * 2 = 18
```

لطباعة التاريخ والساعة الحاليين نكتب هذه الأسطر:

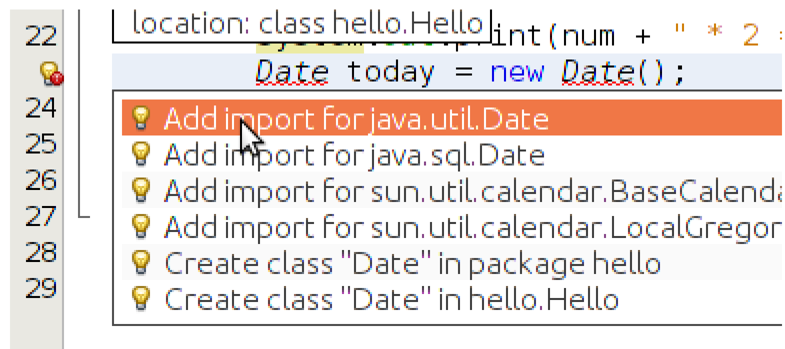
```
        Date today = new Date();
        System.out.println("Today is: " + today.toString());
```

ولابد من إضافة المكتبة المحتوية على الفئة Date في بداية البرنامج، بعد سطر package،

```
import java.util.Date;
```

أو يمكن الإستعانة بمحرر NetBeans لإضافة إسم المكتبة تلقائياً:

عند ظهور العلامة الصفراء شمال السطر الموجودة فيه الفئة Class التي تحتاج لتلك المكتبة كما تظهر في هذه الصورة:



ثم اختيار *Add import for java.util.Date*

وهذه ميزة مهمة في أي أداة تطوير تغني عن حفظ أسماء المكتبات المختلفة أو إضافتها يدوياً.

فيصبح شكل كود البرنامج الكلي هو:

```
package hello;

import java.util.Date;
```

```java
public class Hello {

    /*
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int num = 9;

        System.out.println(num + " * 2 = " + num * 2);
        Date today = new Date();
        System.out.println("Today is: " + today.toString());

    }
}
```

وهذا مثال لناتج تشغيل البرنامج :

```
9 * 2 = 18
Today is: Fri Jul 31 11:59:46 EAT 2015
```

يمكن تغيير نسق التاريخ والساعة وذلك باستخدام الكائن SimpleDateTime كما في المثال التالي:

```java
    SimpleDateFormat simpleFormat = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
    Date today = new Date();
    System.out.println("Today is: " + simpleFormat.format(today));
```

والناتج هو:

```
Today is: 31.07.2015 12:03
```

ويمكن تغيير النسق بتغيير موضع الرموز التي ترمز لمكونات التاريخ وهي:

dd يُمثل اليوم

MM يُمثل رقم الشهر

yyyy يُمثل السنة كاملة، يمكن اختصارها في yy لتصبح رقمين فقط، مثلاً 15 والتي تعني 2015

HH: الساعة بنسق 24 ساعة

mm: الدقائق

ss: الثواني

وهذا مثال آخر لنسق مختلف:

```java
  SimpleDateFormat simpleFormat = new SimpleDateFormat("E dd.MMMM.yyyy hh:mm:ss a");
  Date today = new Date();
  System.out.println("Today is: " + simpleFormat.format(today));
```

وهذا هو الناتج:

```
Today is: Fri 31.July.2015 12:10:21 PM
```

استخدمنا E لكتابة اليوم من الإسبوع، و MMMM لكتابة اسم الشهر كاملاً، ويمكن استخدام MMM لكتابة اسم الشهر بطريقة مختصرة، واستخدمنا h لكتابة الساعة بنسة 12 ساعة، ولا بد من استخدام a معها لتوضيح هل هو مساءً أم صباحاً am/pm

يمكن استخدام SimpleDateFormat لتحويل التاريخ من نص String إلى تاريخ Date وذلك باستخدام الدالة parse لكن لابد من مطابقة النسق وإلا حدث خطأ:

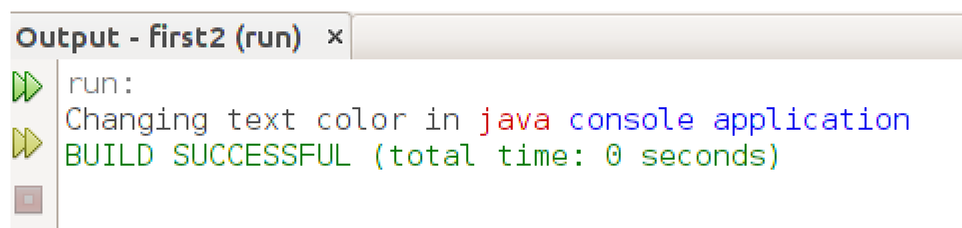
```
    public static void main(String[] args) throws ParseException {

        String todayStr = "15.10.2012";
        SimpleDateFormat simpleFormat = new SimpleDateFormat("dd.mm.yyyy");
        Date today = simpleFormat.parse(todayStr);
        System.out.println("Today is: " + simpleFormat.format(today));
    }
```

نلاحظ أننا أضفنا *throws ParseException* في بداية الدالة *main* وذلك بعد أن أقترحت علينا بيئة التطوير هذه الإضافة وذلك لأن عملية التحويل هذه ربما ينتج عنها خطأ إذا كانت القيمة المدخلة غير صحيحة أو تحتوي على أحرف مثلاً أو قيم تاريخ غير صحيحة مثلاً أدخل الرقم 15 في خانة الشهر أو 32 في خانة الأيام، أو ربما كُتب تاريخ بغير النسق، مثلاً 15/10/2012. وسوف نتكلم لاحقاً على معالجة الإستثناءات في لغة جافا في هذا الكتاب بإذن الله.

في المثال التالي غيرنا لون جزء من النص بالطريقة التالية:

```
        System.out.print("Changing text color in ");
        System.out.print("\033[31m"); // Change color to red
        System.out.print("java ");
        System.out.print("\033[34m"); // Change to blue
        System.out.print("console application");
        System.out.println("\033[0m"); // change to default color
```

فتظهر النتيجة بالشكل التالي في بيئة NetBeans:



وتظهر بالشكل التالي عند تنفيذ البرنامج من الطرفية:

```
otaz@motazt400:~/NetBeansProjects/first2$ java -jar dist/first2.jar
hanging text color in java console application
otaz@motazt400:~/NetBeansProjects/first2$
```

الملفات Files

التعامل مع الملفات من أساسيات أنظمة التشغيل، لذلك فإن لغات البرمجة توفر كافة الإمكانات والعمليات المطلوبة للتعامل مع الملفات بأنواعها. وتشمل العمليات على الملفات والمجلدات الأمثلة التالية:

- إنشاء ملف جديد
- الكتابة في ملف
- قراءة محتويات ملف
- حذف ملف
- التأكد من وجود ملف في مسار معين.
- عرض أسماء الملفات في مسار معين
- إنشاء مجلد جديد

في المثال التالي نُريد اختبار وجود الملف *myfile.txt* فإذا لم يكن موجود فسوف يُنشيء البرنامج ملف جديد بهذا الإسم: المثال التالي لنظام وندوز:

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

        File file = new File("c:\\Users\\Motaz\\myfile.txt");
        if (file.exists()) {
            System.out.println("File exists");
        }
        else {
            System.out.println("File does not exist");
            file.createNewFile();
        }
    }
```

نلاحظ عند كتابة الكود السابق أنه يظهر خطأ في النوع *File* أنه غير معروف، ويظهر باللون أصفر يساره في المحرر، وعند الضغط عليه بالزر اليسار بالماوس يظهر إقتراح إضافة المكتبة ، فنضغط عليها لإضافة المكتبة تلقائياً كما في الصورة أدناه:

```
17      public static void main(String[] args) {
18
            File file = new File("c:\\Users\\Motaz\\myfile.txt");
20   Add import for java.io.File
21   Create class "File" in package filesample (Source Packages)
22   Create class "File" with constructor "File(java.lang.String)" in package filesample (Source Packages)
23   Create class "File" in filesample.FileSample
24   Create class "File" in filesample.FileSample
25   Split into declaration and assignment
26
```

فسوف تُضاف المكتبة تلقائياً في بداية البرنامج:

```java
import java.io.File;
```

كذلك يمكن إضافتها يدوياً إذا كنا نعرف إسم المكتبة التي نستخدمها

بعد ذلك يظهر خطأ آخر في سطر إنشاء ملف جديد، و السبب هو أنه يمكن أن تحدث مشكلة عند تنفيذ هذا الجزء من الكود، مثلاً إذا كان المسار غير صحيح، أو القرص ليس فيه سماحية كتابة أو كان ممتليء. فيكون الحل بإضافة إمكانية إصدار خطأ أثناء التشغيل لهذا الإجراء:

```
25          else {
26              System.out.println("File does not exist");
                file.createNewFile();
28     Add throws clause for java.io.IOException
29     Surround Statement with try-catch
30     Surround Block with try-catch
31     Assign Return Value To New Variable
32
```

وعند الضغط على الخيار Add throws clause ... تُضاف عبارة throws IOException في تعريف الدالة الرئيسة main، ويوجد بديل لهذه الطريقة سوف نستخدمها لنفس المثال لاحقاً بإذن الله:

```java
public static void main(String[] args) throws IOException {
```

بعد هذه الخطوات، يجب تغيير مسار الملف حسب إسم الدخول في نظام وندوز، مثلاً إذا كان اسم المستخدم الحالي هو mohammed يجب تغييره إلى التالي:

```java
File file = new File("c:\\Users\\Mohammed\\myfile.txt");
```

كذلك نلاحظ أننا استخدمنا الفاصل backslash مكرر \\ بدلا من مرة واحدة، وذلك لأن الفاصلة تعني القيام بعملية خاصة في المقاطع في لغة جافا، مثلاً

```
\n
```

تعني سطر جديد، و الرمز:

```
\t
```

تعني إظهار مسافة بين الكلمات، أما استخدام \\ فهي تعني أنا نقصد إظهار الفاصلة نفسها \ وليس إجراء خاص

هذا نفس المثال في نظام لينكس، نلاحظ أنه فقط تغير مسار الملف، ولم نحتاج لكتابة فاصلة لينكس /
(slash) وهي نفسها فاصلة عناوين الإنترنت، وهي عكس فاصلة وندوز \، والفاصلة / ليس لديها معني خاص في جافا، لذلك تُكتب بطريقة عادية:

```
    public static void main(String[] args) throws IOException {

       File file = new File("/home/motaz/myfile.txt");
       if (file.exists()) {
           System.out.println("File exists");
       }
       else {
           System.out.println("File does not exist");
           file.createNewFile();
       }
    }
```

كذلك استخدمنا النوع *File* و عرّفنا كائن منه هو *file* وذلك لغرض ربط البرنامج بالملف الخارجي على القرص. ويمكن عمل عدة عمليات للملف مثل الحذف *file.delete* او الإنشاء *file.createNewFile* أو التأكد من وجود الملف *file.exists*

بدلاً من استخدام عبارة throws IOException كان من الممكن عمل معالجة للأخطاء وذلك بالطريقة التالية:

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

       try {
           File file = new File("/home/motaz/myfile.txt");
           if (file.exists()) {
               System.out.println("File exists");
           }
           else {
               System.out.println("File does not exist");
               file.createNewFile();
           }
       }
       catch (Exception ex){
               System.err.println("Unable to create file: " +
                      ex.toString());
       }
    }
```

فإذا حدث أي خطأ بعد عبارة *try* يتحول التنفيذ إلى جزء الـ*catch*. وهي طريقة أفضل لإظهار المشكلة كما يريدها المبرمج

للمستخدم، بدلاً من ترك المترجم يكتب رسالة الخطأ مباشرة للمستخدم.

هذه هي طريقة حماية أي جزء من الكود والذي يمكن أن يكون عرضة للأخطاء أثناء التشغيل:

```
     try{
       // الكود المعرض لأخطاء التشغيل

         return (true);

      }
      catch (Exception e)
      {
        System.err.println("Error: " + e.getMessage());
        return (false); // fail
      }
```

كتابة نص في ملف

توجد عدة طرق للكتابة أو لقراءة ملف نصي text file، اخترنا في هذه الأمثلة أحد هذه الطرق، وهو باستخدام الفئة *FileWriter* وهي مخصصة لكتابة نص في ملف.

في المثال التالي نُريد الكتابة في ملف نصي باستخدام برنامج بدون واجهة رسومية (console application) وذلك بإنشاء مشروع جديد ثم اختيار Java/Java Application.

هذه المرة نُريد كتابة إجراء جديد نعطيه إسم الملف المُراد إنشاءه والكتابة فيه والنص الذي نُريد كتابته في هذا الملف.

المشروع أسميناه *files*، وكتبنا الإجراء الجديد أسفل الإجراء *main* الموجود مسبقاً. وأسمينا الإجراء الجديد *writeToTextFile* وعرفناه بهذه الطريقة:

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

}

private static boolean writeToTextFile(String aFileName, String text) {

}
```

نلاحظ أننا عرّفنا مُدخلين لهذا الإجراء وهما *aFileName* وهو من النوع النصي String ليستقبل إسم الملف المراد كتابته، والآخر *text* وهو من النوع النصي و الذي يُمثل المحتويات المُراد كتابتها في الملف.

ثم نكتب الكود التالي داخل هذا الإجراء:

```
private static boolean writeToTextFile(String aFileName, String text) {
   try {

     File file = new File(aFileName);
     FileWriter writer = new FileWriter(file);

     writer.write(text + "\r\n");
     writer.close();
     return (true); // success

    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
      return (false); // fail
    }
}
```

نلاحظ أننا أرجعنا القيمة *true* في حال أن الكتابة في الملف حدثت بدون أخطاء. أما في حالة حدوث خطأ فئرجع القيمة *false* وذلك ليعرف من يُنادي هذا الإجراء أن العملية نجحت أم لا.

بالنسبة لتعريف الملف وتعريف طريقة الكتابة عليه كتبنا هذين السطرين:

```
    File file = new File(aFileName);
    FileWriter writer = new FileWriter(file);
```

في العبارة الأولى عرّفنا الكائن *file* من نوع الفئة *File* وهو كائن للربط مع الملف الخارجي. وقد أعطيناه إسم الملف في المدخلات. وفي العبارة الثانية عرّفنا الكائن *writer* من النوع *FileWriter* المتخصص في الكتابة النصية كما سبق ذكره، ومُدخلاته هو الكائن *file* الذي رُبط بالملف الفعلي في القرص.

بعد ذلك كتبنا النص المُرسل داخل الملف باستخدام الكائن *writer* بالطريقة التالية:

```
    writer.write(text + "\r\n");
```

في النهاية أغلقنا الملف باستخدام عبارة *writer.close* وهي من الأهمية بمكان بحيث أنه يمنع برنامج آخر بالكتابة على هذا الملف الذي لم يُغلق، وكذلك فإن الملف غير المغلق يمكن أن يتسبب في إهدار للموارد، حيث أن نظام التشغيل يسمح بفتح عدد معين من الملفات في آن واحد، فتكرار عملية فتح الملف دون أن يكون هناك إغلاق له يمكن أن يمنع فتح ملفات جديدة أثناء تشغيل البرنامج ويتسبب بإنهيار بعض البرامج.

ولنداء إجراء الكتابة في ملف نصي يجب إستدعاءه من الدلة الرئيسة main بالطريقة التالي:

```
     writeToTextFile("myfile.txt", "my text");
```

ويُمكن تحديد المسار أو الدليل الذي نُريد كتابة الملف عليه كما فعلنا في المثال التالي لنداء هذا الإجراء. وقد أضفنا التاريخ والوقت الذي كُتب فيه الملف:

```
  public static void main(String[] args) {
      // TODO code application logic here
      Date now = new Date();
      boolean result;
      result = writeToTextFile("/home/motaz/java.txt",
              "This file has been written using Java\r\n" + now.toString());
      if (result){
          System.out.print("File has been written successfully\n");
      }
      else{
          System.out.print("Error has occurred while writing in the file\n");
      }
  }
```

كذلك فقد عرّفنا المتغير *result* من النوع المنطقي *boolean* والذي يحتمل فقط القيم *true/false* وذلك لإرجاع نتيجة العملية هل نجحت أم لا.

وقد فحصنا قيمة المتغير *result* لعرض رسالة تُفيد بأن العلمية نجحت، أو فشلت في حالة أن قيمته *false*. و العبارة الشرطية هي *if*

```
 if (result)
```

معناها أن قيمة *result* إذا كانت تحمل القيمة *true* نفّذ العبارة التالية، أما إذا لم تكن تحمل تلك القيمة فنفّذ العبارة بعد الكلمة

else

لتنفيذ هذا البرنامج نحتاج لإضافة المكتبات التالية، والتي تساعد أداة التطوير في إضافتها تلقائياً:

```
import java.io.File;
import java.io.FileWriter;
import java.util.Date;
```

بدلاً من حذف محتويات الملف في كل مرة، يمكن الإضافة فقط في النهاية بما يعرف بمصطلح *append* وهو يعني الإضافة في نهاية الملف. لعمل ذلك نُغيّر طريقة تهيئة الكائن *writer* وذلك بإضافة المُدخل *true* كالتالي:

```
      FileWriter writer = new FileWriter(file, true);
```

فعند تشغيله أكثر من مرة، نلاحظ أن المحتويات القديمة موجودة وأن الإضافة تحدث في النهاية.

القراءة من ملف نصي

للقراءة من ملف يُمكن استخدام النوع *FileReader* لقراءة محتويات الملفات النصية، كما في المثال التالي:

```
  private static boolean readTextFile(String aFileName) {
     try {

       File file = new File(aFileName);
       FileReader reader = new FileReader(file);

       char buf[] = new char[10];
       int numread;
       while ((numread=reader.read(buf)) > 0) {

           String text = new String(buf, 0, numread);
           System.out.print(text);
       }
       reader.close();
       return (true); // success

      } catch (Exception e) {
        System.err.println("Error: " + e.getMessage());
        return (false); // fail
      }

  }
```

نلاحظ أننا استخدمنا سلسلة من النوع char وهو يُخزّن رمز، والنصوص هي مجموعة من الرموز.

```
char buf[] = new char[10];
```

لقراءة كل محتويات الملف، لابد من قراءة جميع الأحرف، في كل مرة نقرأ 10 أحرف على الأكثر إلى أن تنتهي محتويات الملف. استخدمنا العبارة التالية لقراءة جزء من الملف ثم اختباره هل وصل الملف إلى نهايته أم لا:

```
while ((numread=reader.read(buf)) > 0) {
```

في هذ الجزء يقرأ البرنامج محتويات الملف ثم يُخزنها في السلسلة *buf* وبما أن حجمها هو 10 بايت فتُقرأ 10 رموز أو أحرف من الملف كحد أقصى في المرة الواحدة، ثم يُرجع العدد الذي قُريء فعلياً في المتغير *numread*، وفي نهاية الملف يمكن أن يتبقى جزء أقل من 10 أحرف، فبدلاً من الحصول على القيمة 10، نتحصل على ما تبقى مثلاً 5 أحرف. كذلك فإن البرنامج في نفس السطر يُقارن قيمة *numread* هل هي أكبر من الرقم 0 والتي تعني أنه نجح في قراءة بايت على الأقل، أما إذا كانت النتيجة 1– فهي تعني أنه لم يتبقى مقطع للقراء في الملف فيخرج تنفيذ البرنامج من حلقة while.

بعد ذلك حولنا سلسلة الأحرف إلى مقطع لسهولة التعامل معه وكتابته في الشاشة:

```
String text = new String(buf, 0, numread);
```

في معظم الأحوال فإن طول السلسة buf هو 10 بايت، لكن ربما قرأ البرنامج عدداً أقل من الأحرف في نهاية الملف، لذلك ننسخ الجزء الذي قُريء فعلياً لذلك حددنا المقطع المراد قراءته بواسطة المُدخلات *0, numread* حتى لا نتحصل على أحرف أو كلمات إضافية من القراءة السابقة، لأننا استخدمنا المصفوفة *buf* عدة مرات فكل مرة يكون فيه أحرف من قراءة سابقة.

نفرض أن الملف يحتوي على 25 رمزاً، فتكون القراءة كالتالي: في الدورة الأولى تُقرأ 10 رموز، ثم في الدورة الثانية 10 رموز ثم 5 رموز. هذه الرموز تُمثل أحرف و رمز السطر الجديد المعروف بالـ new line/line feed في وندوز يُستخدم رمزين للدلالة على نهاية السطر، أما في نظام لينكس فيُستخدم رمز واحد فقط وهو new line. كتبنا رمز السطر الجديد في المثال السابق (الكتابة في ملف نصي) وذلك باستخدام

```
\n
```

لهذا السبب استخدمنا *print* بدلاً من *println* وذلك لأن النص المقروء من الملف يحتوي على رمز السطر الجديد بعد نهاية كل سطر، أما إذا استخدمنا *println* فسوف ينتقل المؤشر إلى سطر جديد بعد كتابة كل 10 أحرف فتصبح الجُمل مقطعة كالتالي:

```
This file
has been w
ritten usi
ng Java
Fr
i Aug 28 0
9:20:47 EA
T 2015
```

لكن عند استخدام *print* يظهر النص واضحاً كالتالي:

```
This file has been written using Java
Fri Aug 28 09:20:47 EAT 2015
```

يُمكن تحويل كود القراءة في هذا الإجراء بأن تُقرأ محتويات الملف سطراً سطراً بدلاً من قراءة عدد من الرموز ثم تحويلها إلى مقطع *String*:

هذه المرة استخدمنا الفئات: *File* و *FileReader* و *BufferedReader* وذلك لقراءة سطر كامل في كل مرة كالتالي:

```
private static boolean readTextFile(String aFileName) {
  try{
    File file = new File(aFileName);

    FileReader fileReader = new FileReader(file);
    BufferedReader bufferedReader = new BufferedReader(fileReader);

    System.out.print("Reading " + aFileName + "\n-------------\n");

    String line;

    while ((line = bufferedReader.readLine()) != null){
       System.out.println (line);
    }
    bufferedReader.close();
    fileReader.close();
    return (true); // success

  } catch (Exception ex) {
     System.err.println("Error in readTextFile: " + ex.getMessage());
     return (false); // fail
  }

}
```

نادينا الإجراء الجديد من داخل main. ليصبح الإجراء كاملاً هو:

```
public static void main(String[] args) {
    // TODO code application logic here
    Date now = new Date();
    boolean result;
    result = writeToTextFile("/home/motaz/java.txt",
            "This file has been written\r\n using Java\r\n" + now.toString());
    if (result){
        System.out.print("File has been written successfully\n");
    }
    else {
        System.out.print("Error has occured while writing in the file\n");
    }

    readTextFile("/home/motaz/java.txt");
}
```

استعراض أسماء الملفات

احياناً نحتاج لأن نستعرض أسماء الملفات الموجودة في مسار معين، ويمكن أن يحتوي هذا المسار على ملفات ومسارات فرعية أخرى.

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

             File folder = new File("/etc/");

             // retrieve all files in taht directory
             File files[] = folder.listFiles();

             for(File afile: files) {
                 System.out.print(afile.getName());

                 // Check if it is normal file or directory
                 if (afile.isDirectory()) {
                    System.out.println(" <DIR>");
                 } else {
                     System.out.println("");
                 }
             }

           } catch (Exception ex){

               System.out.println("Error reading directory: " + ex.toString());
           }
    }
```

لابد من إضافة مكتبة File يدوياً أو عن طريق إضافتها عند اقتراحها بواسطة NetBeans:

```
import java.io.File;
```

عدّلنا البرنامج السابق لإظهار معلومات إضافية، وهي حجم الملف بالبايت وتاريخ آخر تعديل باستخدام الخصائص *length* و *lastModified* بالنسبة للكائن *afile* المُستخدم للحصول على معلومات الملف الحالي:

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

              File folder = new File("/etc/");

              // retrieve all files in taht directory
              File files[] = folder.listFiles();

              for(File afile: files) {

                  System.out.print(afile.getName());
```

```
                // Check if it is normal file or directory
                if (afile.isDirectory()) {
                   System.out.print(" <DIR>");
                } else {
                    System.out.print(" size " +  afile.length() + " Bytes ");
                }
                Date lastModifiedTime = new Date(afile.lastModified());
                System.out.println(" " + lastModifiedTime.toString());
            }

          }
          catch (Exception ex){

              System.out.println("Error reading directory: " + ex.toString());
          }
      }
```

لاننسى إضافة المكتبة Date في قسم import:

```
import java.util.Date;
```

بعد الحصول على أسماء الملفات ومعلوماتها يمكن الإستفادة منها في عمليات أو برامج أخرى، مثل قراءة محتويتها، أو نقلها إلى مسار آخر. كذلك يمكن عمل برنامج لإدارة الملفات مثلاً.

سلسلة البيانات Streams

توجد طرق للتعامل مع البيانات بطريقة شبيهة بالملفات مثل كتابة بيانات بطريقة متسلسلة أو قراءة متسلسلة، لكن هذه البيانات لا تُمثل ملفات موجودة في أي من وسائل التخزين الدائمة، ومثال لذلك تخزين بيانات بنفس شكل الملف في الذاكرة، لغرض التخزين المؤقت أو لغرض عرضها بطريقة ما كما استخدمنا InputStreamReader في قراءة ملف نصي سطراً سطراً، حيث نجد أن هذه الفئة لا تتعامل مع ملف في القرص، إنما معلومات تُستقبل بالتتالي في شكل سلسلة ثم نُخرجها بطريقة أسطر متسلسلة.

مثال آخر لاستخدام سلسلة البيانات streams هو إرسال معلومات إلى مخدم ويب عن طريق بروتوكول الـ HTTP وقراءة الناتج، فهذه الطريقة لا تتضمن تخزين ملف في وسيط، إنما إرسال بيانات عن طريق socket وقراءتها منها.

تُستخدم سلسلة البيانات streams بكثرة في لغة جافا ومكتباتها، وقد استخدمناها في هذا الكتاب عدة مرات، منها لنقل الملفات، وقراءة ملف نصي بطريقة الأسطر، وفي إرسال البيانات واستقبالها من وإلى مخدمات الويب.

كمثال لسلسلة البيانات استخدمنا أحد الفئات المنتمية للفئة الرئيسة *ByteArrayOutputStream* : *OutputStream*، وهي تُمثل سلسلة بيانات من نوع بايت تسمح بالكتابة فيها بطريقة متسلسلة وفي الذاكرة، أي أنه لا تُخزّن تلك المعلومات في القرص مثلاً.

في المثال التالي عرّفنا كائن اسميناه *output* من النوع *ByteArrayOutputStream*،

بعد ذلك كتبنا فيه مقاطع لتُخزّن بالتتالي، ثم استخرجناها منه بواسطة قراءة كافة البيانات وتحويلها إلى مصفوفة array أسميناها *data* (سوف نُشرح المصفوفات في فقرة لاحقة بإذن الله):

```
    public static void main(String[] args) throws IOException {

        ByteArrayOutputStream output;
        output = new ByteArrayOutputStream();

        // Write data into stream
        String line = "Hello Java world\n";
        output.write(line.getBytes());

        line = "This is second line of stream\n";
        output.write(line.getBytes());

        byte[] data = output.toByteArray();

        for (byte b: data){
            System.out.print((char)b);
        }

    }
```

بعد ذلك قرأنا من المصفوفة *data* في حلقة for بحيث كل مرة نقرأ بايت واحد في المتغير *b* ثم نكتبه في شكل حرف وليس كرقم لذلك حولناه لحظة الكتابة في الشاشة بواسطة *char(b)*.

يوجد فرق آخر في استخدام أنواع الـ Streams أنها تتعامل مع البيانات كـ Byte وليس كحرف char ويظهر الفرق عند استخدام أحرف Unicode مثل ترميز UTF-16، مثلاً باللغة العربية الحرف (أ) عندما نتعامل معه كحرف char فهو حرف واحد ويُقرأ في متغير واحد عند استخدام FileReader أما عند التعامل معه كبايت فهو يحتوي على وحدتين، وإذا قرأناه باستخدام الـ stream

فسوف يحتاج لخانتين من نوع byte. وللتأكد من ذلك يمكننا فتح ملف نصي جديد وكتابة جملة باللغة العربية ثم حساب الأحرف، مثلاً جملة (بسم الله) ثم نحفظ الملف، فنجد أن حجمه 16 بايت، مع أن عدد الأحرف هي 8، فهذا يعني أن الحرف باللغة العربية يحتاج لخانتين من الذاكرة أو من القرص لتسجيله. أما عند كتابة جملة باللغة الأنجليزية والتي تعتمد ترميز ASCII فهو يحتاج بايت واحد لتخزين حرف واحد، فإذا كتبنا جملة من 10 أحرف مثلاً في ملف ثم حفظناه فسوف نجد أن حجم الملف 10 بايت أو 11 بايت في حال يوجد رمز نهاية السطر.

نسخ الملفات

يوجد عدد من أنواع الملفات، منها النصية ومنها غير النصي، مثل الصور وملفات الصوت والفديو، وغيرها من الملفات التي تحتوي على بيانات أو حتى الملفات التنفيذية. لكن تشترك كل هذه الملفات في أن أصغر عنصر فيها هو البايت، فإذا اردنا نسخ ملف أو نقله عبر الشبكة مثلاً يمكننا قرائته بايت بايت ثم كتابته أثناء ذلك، أي قراءة بايت من ملف مصدر ثم كتابة هذا البايت إلى الملف الجديد المنسوخ، ثم تكرار هذه العملية إلى نهاية الملف المصدر، فإذا كان حجم الملف هو1 ميقابايت فإننا نُكرر هذه العملية مليون مرة. لكن إذا قرأنا مصفوفة حجمها كيلوبايت في كل مرة ثم كتابتها في الملف الآخر فإننا نحتاج لتكرار تلك العملية ألف مرة، وإذا كان حجم المصفوفة 10 كيلوبايت فنحتاج إلى مائة مرة فقط لنقل كامل الملف.

هذه المرة سوف نستخدم الفئات: *FileInputStream* لقراءة الملف و *FileOutputStream* للكتابة في الملف الجديد، ونوعية الـ *InputStream* هذه متخصصة في قراءة وكتابة الملفات على شكل بايت وليس في شكل رموز كما استخدمنا مع نوع الملفات النصية.

```
    private static void copyFiles(String sourceFileName, String targetFileName)
     throws IOException, FileNotFoundException  {

        File source;
        source = new File(sourceFileName);

        File target;
        target = new File(targetFileName);

        FileInputStream input;
        input = new FileInputStream(source);

        FileOutputStream output;
        output = new FileOutputStream(target);

        byte bucket[] = new byte[1024];
        int numread;
        while ((numread = input.read(bucket)) != -1){
            output.write(bucket, 0, numread);
        }
        output.close();
        input.close();
    }
```

في هذه العبارات عرّفنا كائن *source* و *target* من نوع الفئة *File* المسؤولة عن كتابة أو قراءة الملف من القرص:

```
        File source;
        source = new File(sourceFileName);

        File target;
        target = new File(targetFileName);
```

ثم في العبارات التي تليها عرّفنا الكائنات *input* و *output* من نوع *FileInputStream* و *FileOutputStream* على التوالي، وهي مسؤولة عن قراءة وكتابة مصفوفة من نوع بايت:

```
        FileInputStream input;
        input = new FileInputStream(source);

        FileOutputStream output;
        output = new FileOutputStream(target);
```

بعد ذلك عرّفنا مصفوفة البايت بإسم *packet* حجمها كيلو بايت، ثم أدخلناها في حلقة قراءة من المصدر وكتابة في الملف المراد نسخة إلى نهاية القراءة من المصدر، ونتعرف على نهاية القراءة عندما إرجاع القيمة -1 لعدد البايت التي قُرأت:

```
        byte bucket[] = new byte[1024];
        int numread;
        while ((numread = input.read(bucket)) != -1){
            output.write(bucket, 0, numread);
        }
```

نلاحظ أننا لم نكتب كامل المصفوفة في الملف الهدف كالتالي:

```
  output.write(bucket);
```

حيث أن هذه العبارة سوف تكتب كامل المصفوفة (كيلوبايت) في الملف المنسوخ، فإذا كان حجم الملف هو كيلوبايت ونصف الكيلو فإن الكتابة بتلك الطريقة سوف تكتب 2 كيلو في الملف الهدف، وسوف يحتوى الكيلو الثاني على زيادة هي عبارة عن باقي محتويات القراءة الأولى. لذلك كتبنا الجزء المقروء فقط من المصفوفة، فلو قُريء 100 بايت في أي دورة سوف يُكتب 100 بايت فقط، وإذا قُريء كيلو بايت كامل سوف يُكتب كيلوبايت:

```
  output.write(bucket, 0, numread);
```

والمُدخل (0) يعني الكتابة من بداية المصفوفة، و *numread* هو المكان الذي سوف تتوقف الكتابة قبله، أي الموقع *1023* في حال أن *numread* بها القيمة *1024*، و المصفوفة ذات الـ 1024 بايت تبدأ في البايت 0 وتنتهي عن البايت 1023.

بعد ذلك نُغلق كلا الملفين:

```
        output.close();
        input.close();
```

بهذه الطريقة يمكن نسخ أي نوع من الملفات بغض النظر عن محتوياتها، و الناتج هو ملف منسوخ مماثل في المحتويات للملف الأصلي.

نُنادي إجراء نسخ الملفات من الدالة الرئيسة كالتالي من نظام لينكس:

(ملاحظة: يجب تغيير أسماء الملفات والدلائل بدليل وملف موجود في الجهاز المستخدم قبل تشغيل المثال أدناه)

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

       copyFiles("/home/motaz/fish.jpg", "/home/motaz/fish-copy.jpg");

   }
```

أو كالمثال التالي في نظام وندوز:

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

       copyFiles("c:\\Users\\Mohammed\\photo.png",
                    "c:\\Users\\Mohammed\\copy.png");

   }
```

يمكن الاستغناء عن كائنات التعامل مع الملف source, target وربط الملف مباشرة أثناء تهيئة input و output ليصبح البرنامج مختصراً كالتالي:

```
    private static void copyFiles(String sourceFileName, String targetFileName)
      throws IOException, FileNotFoundException {

        FileInputStream input;
        input = new FileInputStream(sourceFileName);

        FileOutputStream output;
        output = new FileOutputStream(targetFileName);

        byte bucket[] = new byte[1024];
        int num;
        while ((num = input.read(bucket)) != -1){
            output.write(bucket, 0, num);
        }
        output.close();
        input.close();
    }
```

الخصائص Properties

نقصد بها تخزين وقراءة المعلومات في شكل إسم المعلومة مع قيمتها (name value pair) كالتالي:

```
name=value
```

مثلاً:

```
myname=Mohamed Ali
age=30
address=Sudan,Khartoum
```

يُستفاد من هذه التقنية باستخدامها في عمل إعدادات للبرامج، مثلاً البرنامج عندما يعمل يقرأ من ملف خصائص أو ملف إعدادات تخبره بمعلومات عن البيئة التي حوله، مثلاً عن اسم مخدم قاعدة البيانات، وعن اسم الدخول وكلمة المرور، وغيرها من المعلومات التي يحتاج إليها البرنامج حتى يعمل، وإذا أدخلنا هذه المعلومات (مثل اسم مخدم قاعدة البيانات) فإذا اردنا أن يعمل البرنامج في بيئة مختلفة وقاعدة بيانات أخرى لا نستطيع، إلا إذا عدّلنا البرنامج ثم أعدنا ترجمته، وهذه طريقة غير صحيحة تُسمى بالـ hard-coding وهي أن تكون البيانات البيئية التشغيلية للبرنامج توجد داخله، الحل الأمثل ان تكون خارجه في ملف إعدادات يمكن تغييرها بكل سهولة ويسر دون إعادة ترجمة البرنامج، واحياناً دون الحاجة لإغلاق ثم إعادة تشغيل البرنامج.

في المثال التالي عرّفنا كائن من نوع فئة الخصائص *Properties* في الدالة الرئيسة main ثم خزّنا قيم فيها ثم قرأنا تلك القيم:

```
        // Writing
        Properties myproperty = new Properties();
        myproperty.setProperty("Name", "Mohammed Ali");
        myproperty.setProperty("Age", "30");
        myproperty.setProperty("Age", "32");

        // Reading
        System.out.println(myproperty.getProperty("Name"));
        System.out.println(myproperty.getProperty("Age"));
        System.out.println(myproperty.toString());
```

فتكون المخرجات كالتالي:

```
Mohammed Ali
32
{Name=Mohammed Ali, Age=32}
```

نكتب أولاً إسم القيمة مثلا *Name* ثم قيمتها *Mohammed Ali* باستخدام الدالة *setProperty* ونُلاحظ أن هناك فرق في الإسم بين الحرف الكبير الصغير case sensitive، كذلك أننا وضعنا القيمة 30 تحت الإسم *Age* ثم أعدنا وضع القيمة 32، في هذه الحالة تغيرت قيمة *Age* بالقيمة الأخيرة ولا تُكرر الأسماء وهذه ميزة مهمة حيث لا يوضع متغيرين بنفس الإسم لهما قيم مختلفة.

بعد ذلك قرأنا القيم باستخدام *getProperty*. كذلك يمكن إضافة قيمة افتراضية في حال عدم وجود القيمة، مثلاً:

```
        System.out.println(myproperty.getProperty("Adress", "Sudan,Khartoum"));
```

فإذا كانت القيمة *Address* غير موجودة أي لم تُدخل، فتوضع قيماً بديلة لها، أما إذا كانت موجودة فتُتجاهل القيمة الإفتراضية كالتالي:

```
        System.out.println(myproperty.getProperty("Age", "20"));
```

حيث يُكتب 32، هذه الميزة يُمكن الاستفادة منها مع ملف الإعدادات لتعيين قيم بيئية افتراضية في حال أنه لا توجد إعدادات سبق تخصيصها.
العبارة myproperty.toString تُرجع كافة القيم في كائن الخصائص.

ينقص كائن الخصائص هذا التخزين في ملف حتى لا تضيع معلوماته، ولعمل هذه المُهمة، سوف نستخدم ملف من نوع *FileWriter* او *FileOutputStream* بإضافة الكود التالي:

```
        FileWriter writer = new FileWriter("/home/motaz/testing/config.ini");
        myproperty.store(writer, "Configuration");
        writer.close();
```

فيصبح لدينا ملف خارجي بهذه المحتويات:

```
#Configuration
#Fri Aug 26 16:00:41 EAT 2016
Name=Mohammed Ali
Age=30
```

وفي المرة القادمة لابد من قراءة الملف بدلاً من كتابة ملف جديد كل مرة، وذلك بإضافة الكود التالي لبداية استخدام كائن الخصائص، لكن أولاً يجب التأكد من أن الملف موجود في القرص، لذلك استخدمنا الفئة File:

```
        // Read from file
        File file = new File("/home/motaz/testing/config.ini");
        Properties myproperty = new Properties();
        if (file.exists()){
          FileReader reader = new FileReader(file);
          myproperty.load(reader);
          reader.close();
        }
```

وهذا هو الكود كاملاً:

```
    public static void main(String[] args)
           throws FileNotFoundException, IOException {

        // Read from file
        File file = new File("/home/motaz/testing/config.ini");
        Properties myproperty = new Properties();
        if (file.exists()){
          FileReader reader = new FileReader(file);
          myproperty.load(reader);
          reader.close();
        }
```

```java
        // Writing
        myproperty.setProperty("Name", "Mohammed Ali");
        myproperty.setProperty("Age", "30");
        myproperty.setProperty("Address", "Sudan,Khartoum");

        // Reading
        System.out.println(myproperty.getProperty("Name"));
        System.out.println(myproperty.getProperty("Age"));
        System.out.println(myproperty.getProperty("Address", "Sudan"));



        FileWriter writer = new FileWriter(file);
        myproperty.store(writer, "Configuration");
        writer.close();
    }
```

في الغالب فإننا نحتاج فقط قراءة القيم الموجودة في ملف الإعدادات حيث أن الملف غالباً يُكتب خارجياً بواسطة المبرمج أو المسؤول عن النظام، حيث نكتُب القيم في شكل name=value مباشرة في الملف باستخدام أي محرر نصوص، ليقرأ برنامج جافا تلك القيم ثم نستخدمها. وهذا إجراء كتبناه لقراءة أي قيمة مستخدمة فيها هذه الطريقة من أي ملف:

```java
    private static String readConfig(String name, String filename)
      throws IOException {

        // Read from file
        File file = new File(filename);
        Properties myproperty = new Properties();
        if (file.exists()){
            FileReader reader = new FileReader(file);
            myproperty.load(reader);
            reader.close();
        }
        String avalue = myproperty.getProperty(name);
        return avalue;
    }
```

ويمكن نداءه بالطريقة التالية من داخل الدالة الرئيسة main:

```java
   public static void main(String[] args){
     String avalue = readConfig("Address", "/home/motaz/testing/config.ini");
        System.out.println(avalue);
   }
```

المصفوفات arrays

المصفوفة هي مجموعة من المتغييرات من نفس النوع تُخزن بإسم متغير واحد، ولتعريف مصفوفة في لغة جافا نضيف إلى الاسم أو إلى نوع المتغير الأقواس المربعة [ ] بدون مسافة داخلها بعدة أشكال وهي:

```
int[] arr;
int []arr2;
int arr3[];
```

والطرق الثلاث صحيحة لتعريف مصفوفة من النوع int.

بعد ذلك تهيأ المصفوفة بالطول المناسب:

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

بهذا نكون قد حجزنا خمس خانات في المصفوفة تبدأ من 0 وتنتهي بـ 4، ويمكن وضع قيم فيها بالطريقة التالية:

```
arr[0] = 10;
arr[1] = 20;
arr[2] = 6;
arr[3] = 3;
arr[4] = 9;
```

ثم طباعة قيم المصفوفة كاملة باستخدام حلقة for كالتالي:

```
for (int i=0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

وتوجد طريقة مختصرة لحلقة for بالنسبة للمصفوفات والسلاسل عموماً في لغة جافا، حيث يمكن تحويلها إلى الطريقة التالية:

```
for (int x: arr) {
    System.out.println(x);
}
```

حيث تُسند قيمة المتغير x من المصفوفة بالتتالي لاستخدامها لاحقاً داخل الحلقة إلى أن تتنهي عناصرها.

السلاسل ArrayList

طريقة المصفوفة السابقة تُستخدم عندما نعلم الطول أثناء كتابة البرنامج أو أثناء التشغيل، لكن احياناً لا يمكن معرفة الطول المطلوب إلا عند الإنتهاء من إدخال العناصر، مثلاً إذا طلبنا من المستخدم أن يُدخل أسماء، ثم بعد آخر إسم يضغط مفتاح الإدخال دون الكتابة لئعلم البرنامج بإنتهاء الإدخال، في هذه الحالة لا يمكن معرفة عدد الأسماء المراد إدخالها، وهنا لا يصلح استخدام المصفوفة، لكن نستخدم السلسلة ArrayList كما في المثال التالي:

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

       // Declare array list
       ArrayList<String> nameList;

       // Intialize arraylist
       nameList = new ArrayList<>();

       String name = "";

       // Read user input
       System.out.println("Please input names, and press enter");
       Scanner sc = new Scanner(System.in);
       do {
          name = sc.nextLine();

          // Add name to list
          nameList.add(name);
       } while (!name.isEmpty());

       // display list
       for (String aname: nameList){
           System.out.println(aname);
       }
}
```

في هذا المثال أعلنا عن السلسلة بالطريقة التالية:

```
       ArrayList<String> nameList;
```

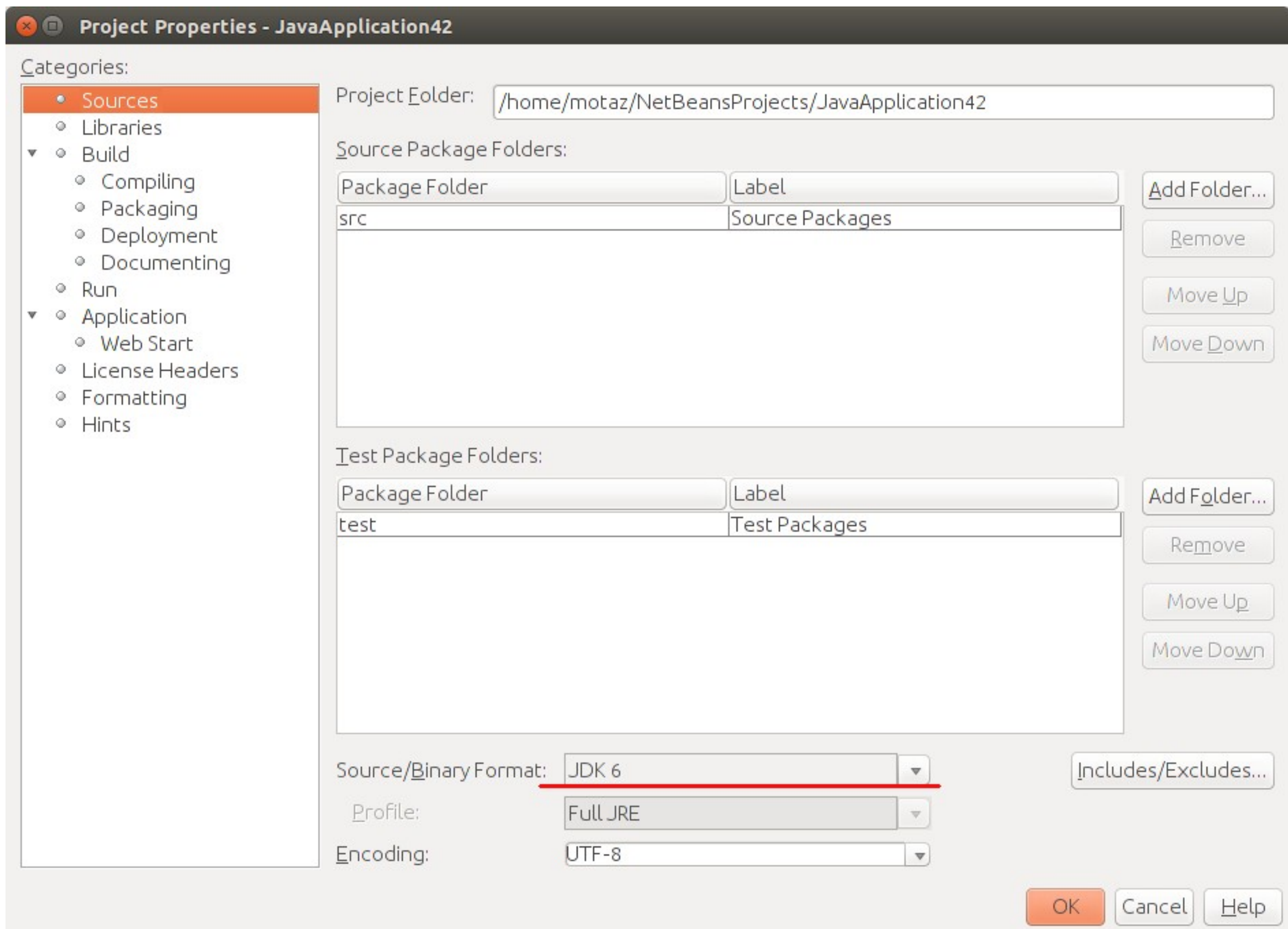
ونوع المتغير أو العنصر الواحد في السلسلة هو مقطع String، وثهيأ بالطريقة التالية:

```
       nameList = new ArrayList<>();
```

قديماً في النسخة السادسة من الجافا كان لابد من كتابة نوع العنصر مرة أخرى كالتالي:

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

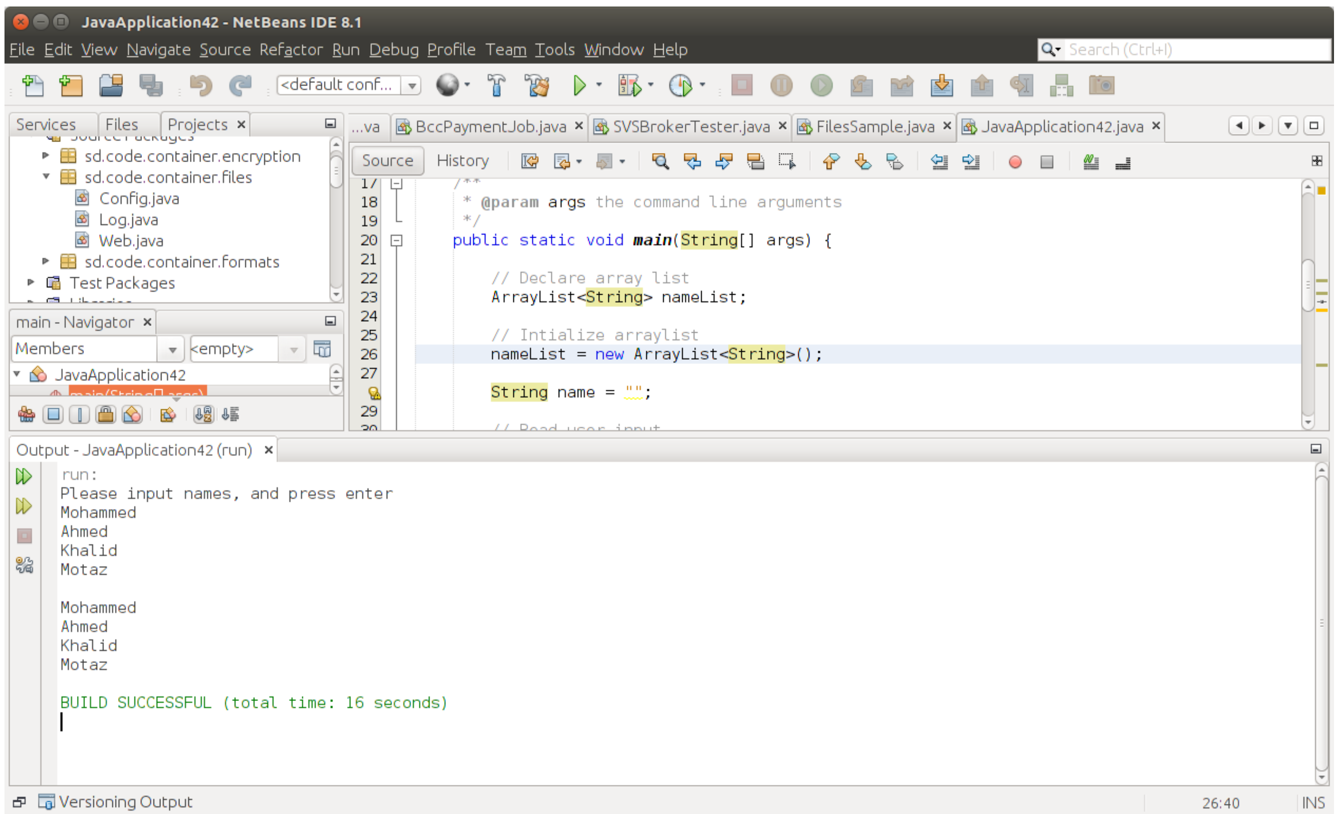
لكن منذ جافا 7 أختصرت طريقة تهيأة المصفوفة باختصار نوع العنصر بما يُسمى بعلامة الماسة Diamond <>
يمكن تحويل نسخة البرنامج إلى جافا 6 أو 5 بواسطة خصائص البرنامج properties ثم تغييرها كالتالي:



فنجد أن المترجم أعطى خطأ أمام التهيئة بواسطة <> وعند تحويلها إلى <String> يعدُها صحيحة.

طريقة تحويل مصدر برنامج جافا إلى نُسخة أقدم من جافا تُستخدم احياناً حينما نريد تشغيل برنامج جافا في آلة افتراضية قديمة في مخدم مثلاً. حيث أن بعض المخدمات يصعب تحديثها إلى نُسخة جديدة من جافا.

عند تشغيل البرنامج لابد من وضع المؤشر في الجزء الأسفل من الشاشة حتى نتمكن من إدخال الأسماء بالطريقة التالية:


JavaApplication42 - NetBeans IDE 8.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Services Files Projects
sd.code.container.encryption
sd.code.container.files
Config.java
Log.java
Web.java
sd.code.container.formats
Test Packages
main - Navigator
Members
<empty>
JavaApplication42
BccPaymentJob.java
SVSBrokerTester.java
FilesSample.java
JavaApplication42.java
Source History
* @param args the command line arguments
*/
public static void main(String[] args) {
// Declare array list
ArrayList<String> nameList;
// Intialize arraylist
nameList = new ArrayList<String>();
String name = "";
Output - JavaApplication42 (run)
run:
Please input names, and press enter
Mohammed
Ahmed
Khalid
Motaz
Mohammed
Ahmed
Khalid
Motaz
BUILD SUCCESSFUL (total time: 16 seconds)
Versioning Output
26:40
INS

تعريف الكائنات والذاكرة

من الأمثلة السابقة نُلاحظ أننا استخدمنا البرمجة الكائنية في قراءة وكتابة الملفات والتاريخ. ونُلاحظ أن تعريف الكائن وتهيته يمكن أن تكون في عبارة واحدة، مثلاً لتعريف التاريخ ثم تهيئته بالوقت الحالي استخدمنا:

```
        Date today = new Date();
```

وكان يُمكن فصل التعريف للكائن الجديد من تهيئته بالطريقة التالية:

```
        Date today;
        today = new Date();
```

هذه المرة في العبارة الأولى عرّفنا الكائن *today* من نوع الفئة *Date*. لكن إلى الآن لا يُمكننا استخدام الكائن *today* فلم يُحجز موقع له في الذاكرة.

أما في العبارة الثانية فقد حجزنا موقع له في الذاكرة بإستخدام الكلمة *new* ثم تهيأة الكائن باستخدام الإجراء :

```
    Date();
```

والذي بدوره يقرأ التاريخ والوقت الحالي لإسناده للكائن الجديد *today*. وهذا الإجراء يُسمى في البرمجة الكائنية constructor.

في هذا المثال Date هي عبارة عن فئة لكائن أو تُسمى class في البرمجة الكائنية. و المتغير today يُسمى كائن object أو instance ويُمكن تعريف أكثر من كائن instance من نفس الفئة لاستخدامها. وتعريف كائن جديد من فئة ما و تهيأتها تُسمى object instantiation في البرمجة الكائنية.

بعد الفراغ من استخدام الكائن يمكننا تحريره من الذاكرة وذلك باستخدام الدالة التالية:

```
   today = null;
```

توجد في لغة جافا ما يُعرف بال garbage collector وهي آلية لحذف الكائنات الغير مستخدمة من الذاكرة تلقائياً عندما ينتهي تنفيذ الإجراء. تُحذف فقط الكائنات المعرفة في نطاق هذا الإجراء. في معظم الأحيان لا نحتاج لاستخدام هذه العبارة، فإذا انتهى استخدام المتغير الذي يؤشر لهذا الكائن فإنه يُحرر تلقائياً.

مفهوم – غير مستخدم– يعني أنه لا يوجد مؤشر له من المتغيرات، حيث يمكن أن يكون لكائن ما عدد من المؤشرات تؤشر له، فعندما تنتهي جميع هذه المؤشرات ويصبح عدد المتغيرات التي تؤشر لهذا الكائن في الذاكرة صفراً فيُحذف بواسطة الـ garbage collector من الذاكرة بعد مدة معينة. أما لغات البرمجة الأخرى مثل سي وأوبجكت باسكال فعند استخدامها لابد من تحرير الكائنات يدوياً في معظم الحالات.

نهاية المتغير يكون بنهاية تنفيذ الحيز الموجود فيه وهو المحاط بالقوسين {}

نأخذ هذا المثال لشرح مفهوم حيز أو نطاق تعريف المتغير:

```
1     public static void main(String[] args) {
2
3         String yourName = "Mohammed";
4          {
5            String myName = "Motaz";
6            System.out.println(myName);
7          }
```

```
8          System.out.println(yourName);
9
10      }
```

نجد أن المتغير *yourName* معرف داخل الإجراء main لذلك لا ينتهي إلا بانتهاء هذا الإجراء، أي عند السطر رقم 10.

أما المتغير *myName* والمعرف في نطاق أضيق، فينتهي عند السطر رقم 7، فإذا أردنا أن نجعله ذو عمر أطول يمكن تعريفه خارج هذا النطاق الضيق:

```
1     public static void main(String[] args) {
2
3        String yourName = "Mohammed";
4        String myName;
5        {
6           myName = "Motaz";
7           System.out.println(myName);
8        }
9        System.out.println(yourName);
10
11       readTextFile("/home/motaz/java.txt");
12    }
```

بهذه الطريقة يصبح عمر المتغير myName مرتبط بنهاية الإجراء main، ونلاحظ أننا نقلنا فقط التعريف إلى الخارج، لكن التهيأة ما زالت داخل ذلك الحيز، لكن هذا لم يؤثر على قيمته أو نطاق تعريفه.

يمكن تهيئة كائن جديد بواسطة إسناد مؤشر كائن قديم له، في هذه الحالة يكون كلا المتغيرين يؤشران لنفس الكائن في الذاكرة:

```
 public static void main(String[] args){
        Date today;
        Date today2;
        today = new Date();
        today2 = today;
        today = null;
        System.out.print("Today is: " + today2.toString() + "\n");
 }
```

نلاحظ أننا لم نُهيئ المتغير *today2* لكن بدلاً من ذلك جعلناه يؤشر لنفس الكائن *today* الذي هُيئ من قبل.

بعد ذلك حررنا المتغير *today*، إلا أن ذلك لم يؤثر على الكائن، حيث أن الكائن لا يزال مرتبط بالمتغير *today2*. ولا تُحرره آلية garage collector من الذاكرة إلا عندما تصبح عدد المتغيرات التي تؤشر له صفراً. فإذا حررنا المتغير today2 يُحرر الكائن، و تحدث مشكلة عند تنفيذ السطر الأخير، وذلك لأن المتغير today2 أصبح لا يؤشر إلى كائن موجود في الذاكرة، ومحاولة الوصول إليه بالقراءة أو الكتابة ينتج عنها خطأ.

ولمعرفة ماهو الخطأ الذي ينتج أحطنا الكود بعبارة *try catch* كما في المثال التالي:

```
 public static void main(String[] args){
     try {
        Date today;
        Date today2;
        today = new Date();
```

```
        today2 = today;
        today = null;
        today2 = null;
        System.out.print("Today is: " + today2.toString() + "\n");
      } catch (Exception e) {
          System.out.print("Error: " + e.toString() + "\n");
      }
  }
```

والخطأ الذي تحصلنا عليه هو:

```
java.lang.NullPointerException
```

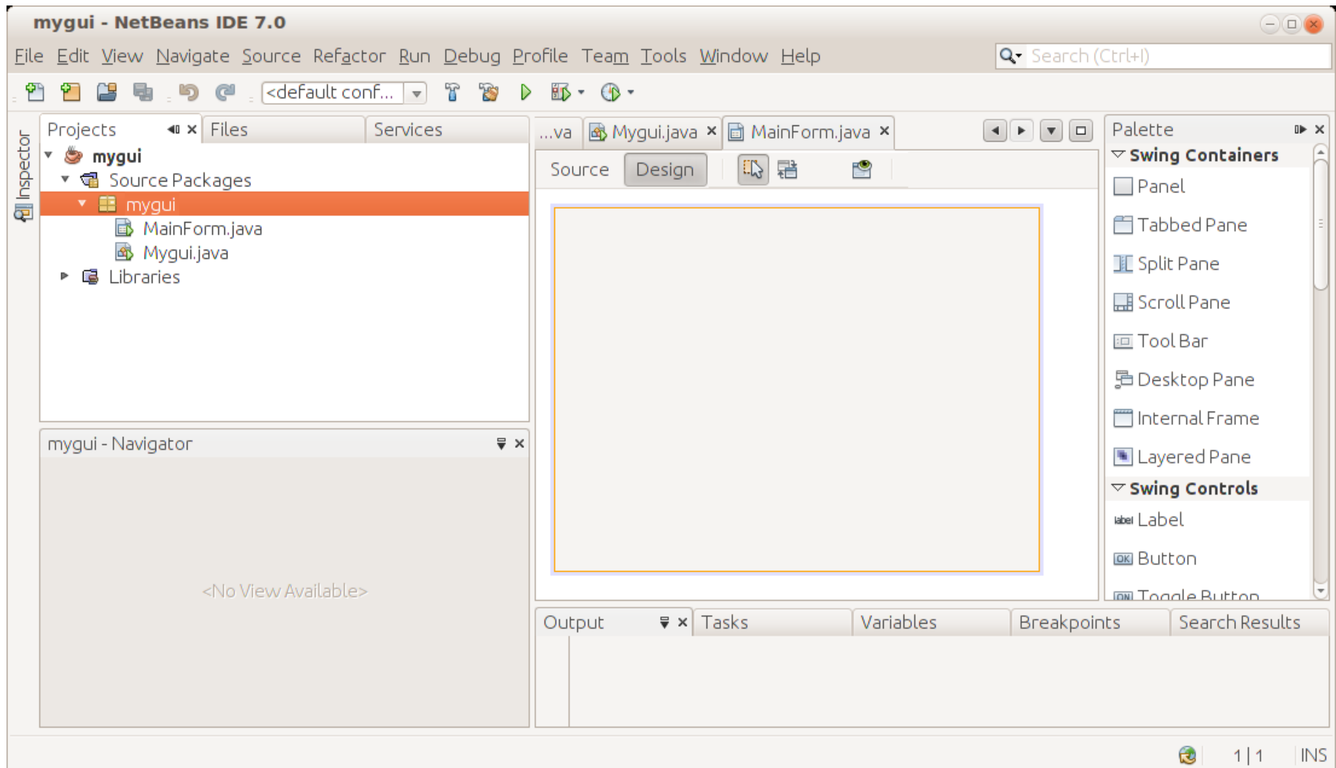
ملاحظة:

في لغة جافا أتفق على تسمية الفئات classes بطريقة أن يكون الحرف الأول كبير capital مثل Date, String, حتى الفئات التي يكتبها المبرمج. أما الكائنات objects/instances والتي تُمثل متغيرات فتبدأ بحرف صغير وذلك للتفرقة بين الفئة (class) والكائن (Object)، مثل today, today2, myName.

برنامج الواجهة رسومية GUI

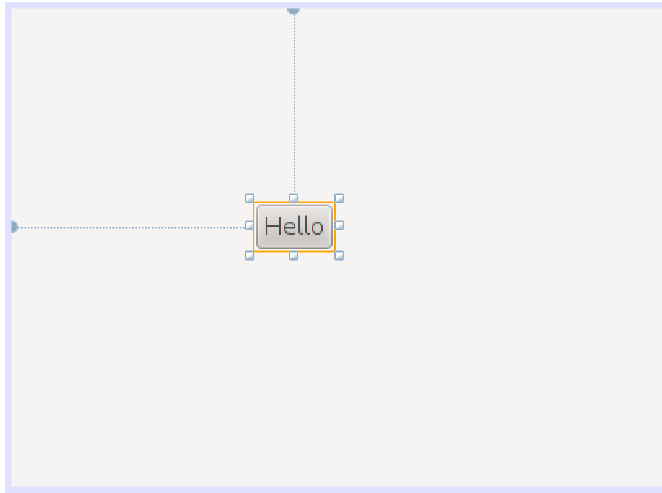
من الأشياء المهمة في أدوات التطوير و لغات البرمجة هو دعمها للواجهات الرسومية أو مايُسمى بالـ Widgets. كل نظام تشغيل يحتوي على مكتبة أو أكثر تُمثل واجهة رسومية، مثلاً يوجد في نظام لينكس واجهاتGTK و QT و في نظام وندوز توجد مكتبة وندوز الرسومية، وفي نظام ماكنتوش توجد مكتباتCarbon و Cocoa. أما جافا فلها مكتباتها الخاصة والتي تعمل في كل هذه الأنظمة ومنها واجهة Swing.

لكتابة أول برنامج ذو واجهة رسومية في جافا باستخدام NetBeans نختار File/New Project
ثم نختار Java/Java Application
ثم نُسميه مثلاً mygui.
في شاشة Projects نختار الحزمة mygui ثم بالزر اليمين للماوس نختار New/JFrame Form
نسمى هذا الفورم MainForm، ونلاحظ أن حرف M كبير، كذلك حرف F صغير، ولابد من استخدامه في باقي البرنامج بنفس الشكل، مثلاً إذا ناديناه على أنه Mainform باستخدام حرف f صغير فإن المترجم سوف لن يتعرف عليه، لذلك وجب التنبيه.
سوف يُضاف هذا الفورم للمشروع ويظهر بالشكل التالي:



يظهر الفورم الرئيسي المسمى MainForm.java في وسط الشاشة. وفي اليمين نلاحظ وجود عدد من المكونات في صفحة الـ Palette. نُدرج زر Button في وسط الفورم الرئيسي، ثم نُغيّر عنوانه إلى *Hello*، وذلك إما بالضغط على زر F2 ثم نُغيّر العنوان،

أو بالنقر على الزر اليمين في الماوس في هذا الزر ثم نختار Properties ثم Text



نرجع مرة أخرى للخصائص لنضيف حدث عند الضغط على الزر. هذه المرة نختار Events ثم في الخيار actionPerformed نختار الحدث jButton1ActionPerformed بعدها يظهر هذا الكود في شاشة الـ Source:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
}
```

أو يُمكن إظهار هذا الكود بواسطة النقر المزدوج على الزر double click

فنكتُب كود لإظهار عبارة (السلام عليكم) عند الضغط على هذا الزر. فيصبح الكود الحدث كالتالي:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
      String msg = "السلام عليكم";
      JOptionPane.showMessageDialog(null, msg);
}
```

نلاحظ أننا عرّفنا المتغير *msg* من النوع المقطعي *String* ثم أسندنا قيمة إبتدائية له: *"السلام عليكم"*

بعد ذلك نرجع للحزمة الرئيسة Mygui.java ثم نكتب الكود التالي في الإجراء main:

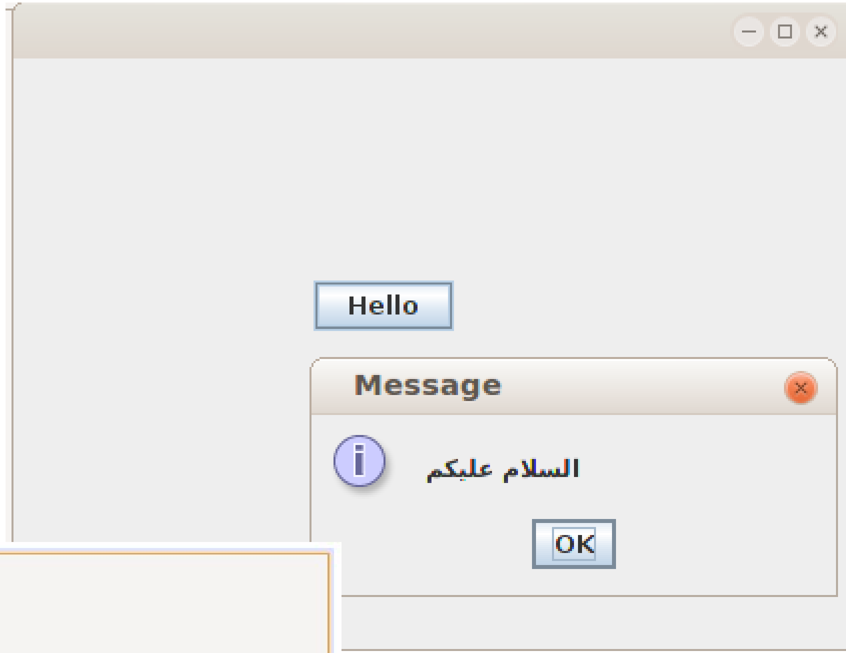
```
  public static void main(String[] args) {
      MainForm form = new MainForm();
      form.setVisible(true);
  }
```

في السطر الأول نُعرّف الكائن form من النوع MainForm الذي صممناه، ثم نُنشيء نسخة من هذا النوع و نُهيأه بواسطة:

```
new MainForm
```

وفي السطر الثاني نُظهر الفورم في الشاشة.

عند تنفيذ البرنامج يظهر بالشكل التالي عند الضغط على الزر:



نرجع مرة أخرى للفورم في شاشة التصميم (Design)

ثم نُدرج المكون TextField لنُدخل فيه إسم المستخدم، ثم مكون من نوع Label نكتب فيه كلمة (الإسم) ثم مكون آخر من نوع Label نغيّر إسمه إلى *jlName* وذلك في فورم الخصائص في صفحة Code في قيمة Variable Name

ثم نُدرج زر نكتب فيه كلمة (ترحيب) كما في الشكل التالي:

في الحدث ActionPerfomed لهذا الزر الجديد (ترحيب) نكتب الكود التالي لكتابة إسم المستخدم في المكون jLabel2:

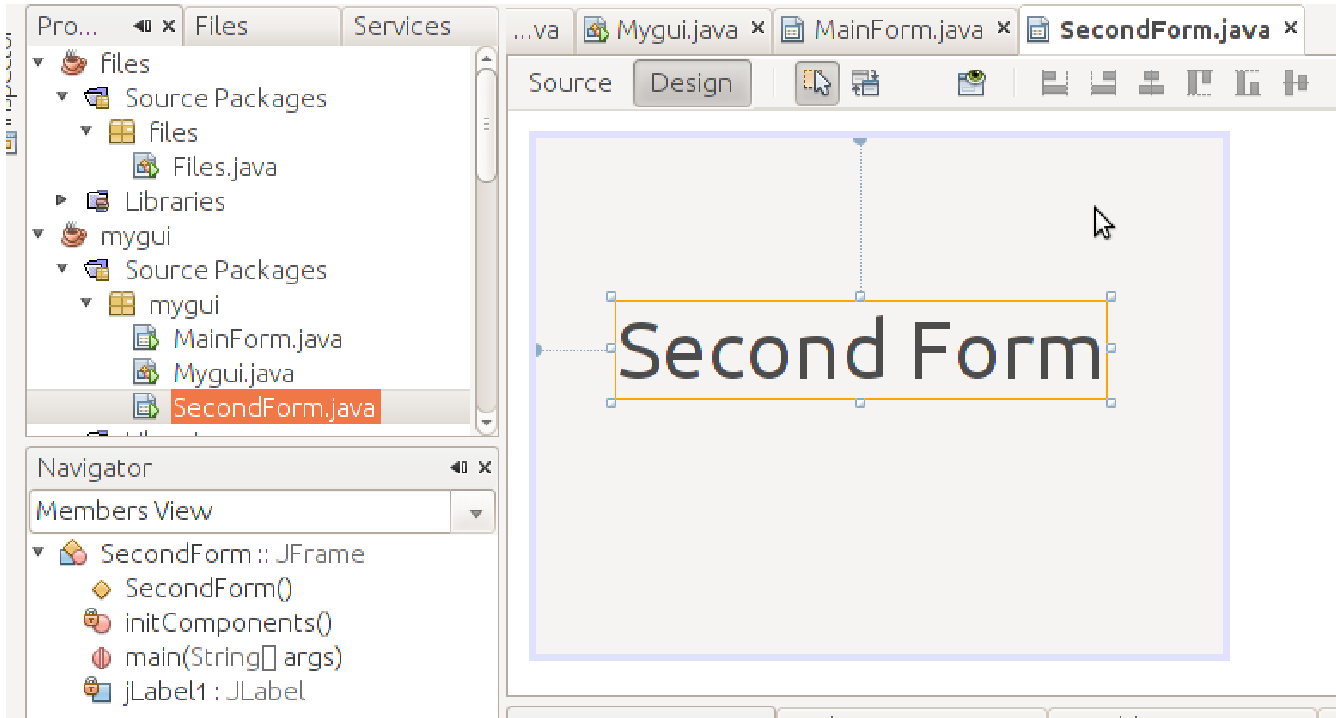
```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {

   jlName.setText("مرحباً بك " + jTextField1.getText());
}
```

نلاحظ أن الإجراء *getText* يُستخدم لقراءة محتويات الحقل النصي Text Field والإجراء *setText* يضع قيمة مقطعية أو عبارة نصّية في عنوان المكوّن Label أثناء التنفيذ.

الفورم الثاني

لإضافة وإظهار فورم ثاني في نفس البرنامج، نتبع الخطوات في المثال التالي:

نُضيف JFrame Form و نُسميه *SecondFrom* ونضع فيه Label نكتب فيه عبارة "Second Form" ونزيد حجم الخط في هذا العنوان بواسطة Properties/Font.



نضيف زر في الفورم الرئيسي MainForm ونكتب الكود التالي في الحدث ActionPerformed في هذا الزر الجديد لإظهار الفورم الثاني، أو يمكن كتابة هذا الكود في زر الترحيب.

```
SecondForm second = new SecondForm();
second.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
second.setVisible(true);
```

نلاحظ للسطر:

```
second.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

وهو يعني عند إغلاق هذا الفورم الثاني يُحذف من الذاكرة فقط دون إغلاق البرنامج، فإذا لم نضفه سوف ينغلق البرنامج عند إغلاق هذا الفورم. نحن نريد فقط أن ينغلق البرنامج عند إغلاق الفورم الرئيسي لذلك لم نضف هذا السطر في الفورم الرئيسي.

يُمكن إرسال كائن أو متغير للفورم الجديد. مثلاً نريد كتابة رسالة الترحيب في الفورم الثاني.

لعمل ذلك نحتاج لتغيير إجراء التهيئة constructor في الفورم الثاني والذي اسمه SecondForm ، نضيف إليه مدخلات:

```
public SecondForm(String atext) {
    initComponents();
    jLabel1.setText(atext);
}
```

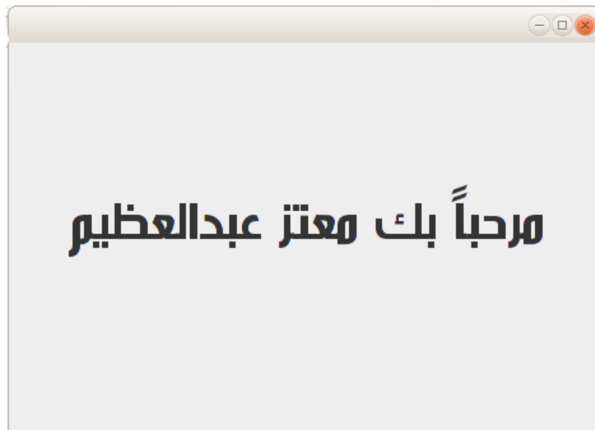
ثم نظهر هذه المدخلات – والتي هي عبارة عن رسالة الترحيب – في العنوان jLabel1

وعند تهيأة الفورم الثاني من الفورم الرئيسي نُعدّل الكود إلى التالي، وهذا الكود كتبناه في إجراء زر الترحيب:

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {

    jlName.setText("مرحباً بك " + jTextField1.getText());

    SecondForm second = new SecondForm(jlName.getText());
    second.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    second.setVisible(true);

}
```

عند التنفيذ يظهر

هذا الشكل:



ملحوظة:

في برامج جافا من نوع Java Application تنفذ فقط الدالة main عند تشغيل البرنامج، لذلك إذا كتبنا إجراء جديد ولم نناديه من داخل الدالة الرئيسة main فلن يُنفّذ ذلك الإجراء

برنامج اختيار الملف

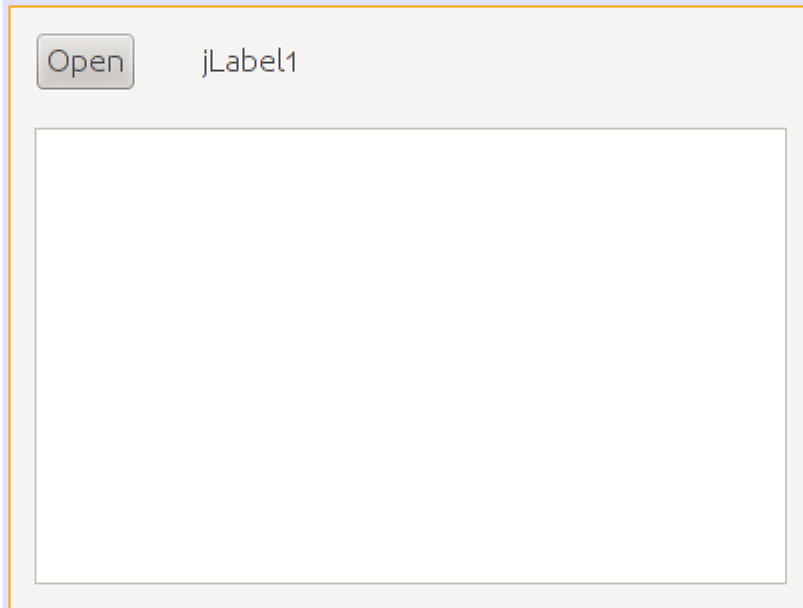
هذه المرة نُريد عمل برنامج ذو واجهة رسومية يسمح لنا بإختيار الملف بالماوس، ثم عرض محتوياته في صندوق نصي.

لعمل هذا البرنامج نفتح مشروع جديد بواسطة Java/Java Application. نسمي هذا المشروع *openfile*

نُضيف JFrame Form نسميه MainForm ونضع فيه المكونات التالية:

Button, Label, Text Area

كما في الشكل التالي:



بعد ذلك نكتب هذا الكود في الإجراء main في ملف البرنامج الرئيسي Openfile.java لإظهار الفورم فور تشغيل البرنامج:

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

        MainForm form = new MainForm();
        form.setVisible(true);
    }
```

نسخ الإجراء readTextFile من البرنامج السابق إلى كود البرنامج الحالي، ونعدله قليلاً، نضيف له مدخل جديد من نوع JTextArea وذلك لكتابة محتويات الملف في هذا المربع النصي بدلاً من شاشة سطر الأوامر Console. وهذا هو الإجراء المعدل:

```
  private static boolean readTextFile(String aFileName, JTextArea textArea) {
     try {
        FileInputStream fstream = new FileInputStream(aFileName);

        DataInputStream textReader = new DataInputStream(fstream);

        InputStreamReader isr = new InputStreamReader(textReader);
        BufferedReader lineReader = new BufferedReader(isr);

        String line;
        textArea.setText("");
```

```
      while ((line = lineReader.readLine()) != null){
        textArea.append(line + "\n");
      }

      fstream.close();
      return (true); // success

    } catch (Exception e) {
        textArea.append("Error in readTextFile: " + e.getMessage() + "\n");
        return (false); // fail
    }
}
```

وفي الحدث ActionPerformed في الزر نكتب الكود التالي:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

     final JFileChooser fc = new JFileChooser();
     int result = fc.showOpenDialog(null);
     if (result == JFileChooser.APPROVE_OPTION) {
            jLabel1.setText(fc.getSelectedFile().toString());
            readTextFile(fc.getSelectedFile().toString(), jTextArea1);
      }
    }
```

وقد عرّفنا كائن اختيار الملف في السطر التالي:

```
     final JFileChooser fc = new JFileChooser();
```

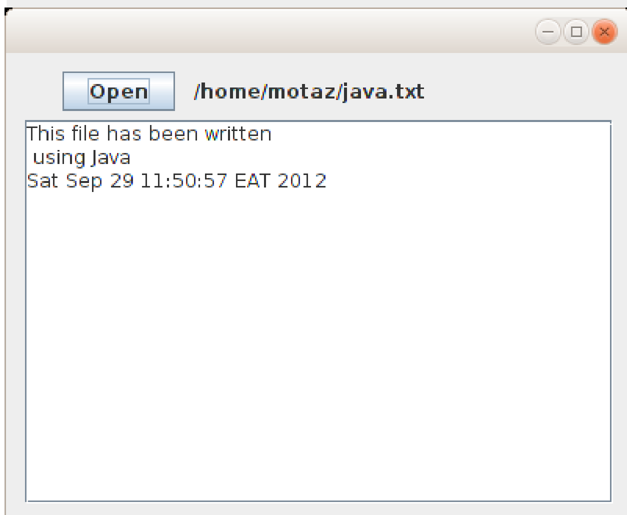
ثم أظهرناه ليختار المستخدم الملف في السطر التالي. ثم يُرجع النتيجة: هل اختار المستخدم ملف أم ضغط إلغاء:

```
     int result = fc.showOpenDialog(null);
```

فإذا اختار ملف نكتب اسمه في العنوان *jLabel1* ثم نظهر محتوياته داخل مربع النص:

```
     if (result == JFileChooser.APPROVE_OPTION) {
            jLabel1.setText(fc.getSelectedFile().toString());
            readTextFile(fc.getSelectedFile().toString(), jTextArea1);
                                                      }
```

وعند تشغيل البرنامج يظهر لنا بهذا الشكل بعد إختيار الملف:



كتابة فئة كائن جديد New Class

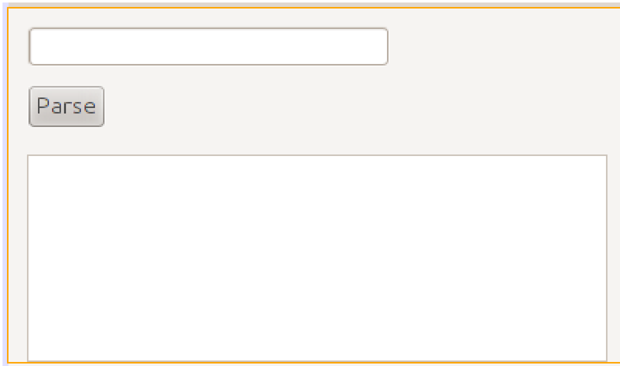
لغة جافا تعتمد فقط نموذج البرمجة الكائنية Object Oriented paradigm، وقد مر علينا في الأمثلة السابقة استخدام عدد من الكائنات، سواءً كانت لقراءة التاريخ أو للتعامل مع الملفات أو الكائنات الرسومية مثل Label والـ Text Area والفورم JFrameForm. لكن حتى تصبح البرمجة الكائنية أوضح لابد من إنشاء فئات classes جديدة بواسطة المبرمج لتعريف كائنات منها.

في هذا المثال سوف نُضيف فئة class جديدة نُدخل لها جملة نصية لإرجاع الكلمة الأولى والأخيرة من الجملة.

أنشأنا برنامج جديد من نوع Java Application، و أسميناه newclass،

بعد ذلك أضفنا MainForm من نوع JFrameForm

ثم أدرجنا Text Field و Button و Text Area بهذا الشكل في الفورم الرئيس:



ولا ننسى تعريف الفورم وتهيئته لإظهاره مع تشغيل البرنامج في الإجراء الرئيسي في الملف Newclass.java:

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

        MainForm form = new MainForm();
        form.setVisible(true);
    }
```

بعد ذلك أضفنا class جديدة وذلك بإختيار Source Packages/new class بالزر اليمين ثم اختيار New/Java Class من القائمة. ثم نسمي الفئة الجديدة Sentence فيظهر لنا هذا الكود:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package newclass;

/*
 *
 * @author motaz
 */
public class Sentence {

}
```

وفي داخل كود الفئة –بين القوسين المعكوفين {} – أضفنا متغير مقطعي اسميناه *mySentence* لنحفظ فيه الجملة التي يُراد

إدخالها ليكون الكائن محتفظ بها طوال فترة حياته.

ثم أضفنا الإجراء الذي يُستخدم في تهيأة الكائن، ولابد أن يكون اسمه مطابق لإسم الفئة:

```
    String mySentence;

    public Sentence (String atext){
       super();
       mySentence = atext;
    }
```

نلاحظ أنه في هذا الإجراء أسندت قيمة المُدخل *atext* إلى المتغير *mySentence* المُعرف على نطاق الكائن. حيث أن المتغير *atext* نطاقه فقط الإجراء *Sentence* وعند الإنتهاء من نداء هذا الإجراء يصبح غير معروف. لذلك إحتفظنا بالجملة المُدخلة في متغير في نطاق أعلى لتكون حياته أطول، حيث يُمكن استخدامه مادام الكائن لم يُحذف من الذاكرة.

بعد ذلك أضفنا إجراء جديد في نفس فئة الكائن اسمه *getFirst* وهو يُرجع الكلمة الأولى من الجملة:

```
    public String getFirst(){
        String first;
        int firstSpaceIndex;
        firstSpaceIndex = mySentence.indexOf(" ");

        if (firstSpaceIndex == -1) {
            first = mySentence;
        }
        else {
            first = mySentence.substring(0, firstSpaceIndex);
        }

        return (first);
    }
```

نلاحظ اننا استخدمنا الإجراء *indexOf* في المتغير أو الكائن المقطعي *mySentence* ثم أرسلنا مقطع يحتوي على مسافة. وهذا الإجراء أو الدالة مفترض به في هذه الحالة أن يُرجع موقع أول مسافة في الجملة، وبهذه الطريقة نعرف الكلمة الأولى، حيث أنها تقع إبتداءً من الحرف الأول إلى أول مسافة.

أما إذا لم تكن هُناك مسافة موجودة في الجملة فتكون نتيجة الدالة *indexOf* يساوي 1– وهذا يعني أن الجملة تتكون من كلمة واحدة فقط، في هذه الحالة نُرجع الجملة كاملة (الجملة = كلمة واحدة).

وإذا وُجدت المسافة فعندها ننسخ مقطع من الجملة بإستخدام الدالة substring والتي نُعطيها بداية ونهاية المقطع المُراد نسخه. ونتيجة النسخ ترجع في المتغير أو الكائن المقطعي *first*

الدالة أو الإجراء الآخر الذي أضفناه في الفئة *Sentence* هو *getLast* وهو يُرجع آخر كلمة في الجملة:

```
    public String getLast(){
        String last;
        int lastSpaceIndex;
        lastSpaceIndex = mySentence.lastIndexOf(" ");

        if (lastSpaceIndex == -1){
            last = mySentence;
        }
        else {
            last = mySentence.substring(lastSpaceIndex + 1, mySentence.length());
        }
        return (last);
    }
```

وهو مشابه للدالة الأخرى، ويختلف في أنه ينسخ من آخر مسافة موجودة في الجملة (*lastIndexOf*) إلى نهاية الجملة mySentence.length

والكود الكامل لهذه الفئة هو:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package newclass;

/*
 *
 * @author motaz
 */
public class Sentence {

    String mySentence;

    public Sentence (String atext){

       super();
       mySentence = atext;
    }

    public String getFirst(){

        String first;
        int firstSpaceIndex;
        firstSpaceIndex = mySentence.indexOf(" ");

        if (firstSpaceIndex == -1){
            first = mySentence;
        }
        else {
            first = mySentence.substring(0, firstSpaceIndex);
        }

        return (first);
    }

    public String getLast(){

        String last;
        int lastSpaceIndex;
        lastSpaceIndex = mySentence.lastIndexOf(" ");

        if (lastSpaceIndex == -1){
            last = mySentence;
        }
        else {
            last = mySentence.substring(lastSpaceIndex + 1, mySentence.length());
      }
```

```
        return (last);
    }
}
```

في كود الفورم الرئيسي للبرنامج MainForm.java عرّفنا وهيأنا ثم استخدمنا هذا الكائن، واستقبلنا الجملة في مربع النص Text Field. وهذا هو الكود الذي يُنفّذ عند الضغط على الزر:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

        Sentence mySent = null;
        mySent = new Sentence(jTextField1.getText());

        jTextArea1.append("First: " + mySent.getFirst() + "\n");
        jTextArea1.append("Last: " + mySent.getLast() + "\n");
    }
```

أنشأنا كائن جديد و هيأناه في هذا السطر:

```
        mySent = new Sentence(jTextField1.getText());
```

والجُملة المُدخلة أثناء التهيئة تحصلنا عليها من مربع النص jTextField1 بواسطة الإجراء *getText* الموجود في هذا الكائن.

المتغيرات والإجراءات الساكنة (static)

فيما سبق لنا من تعامل مع الفئات وجدنا أنه لابد من تعريف كائن من نوع الفئة قبل التعامل معها، فمثلاً لا نستطيع الوصول لإجراء الفئة بدون أن تصبح كائن. فنجد أن المثال التالي غير صحيح:

```
jTextArea1.append("First: " + Sentence.getFirst() + "\n");
```

لكن يُمكن استخدام إجراءات في فئات دون تعريف كائنات منها بتحويلها إلى إجراءات ساكنة static methods.

وهذا مثال لطريقة تعريف متغيرات وإجراءات ساكنة في لغة جافا:

```
public static class MyClass {
    public static int x;
    public static int getX(){
        return x;
    }
}
```

ويُمكن مناداتها مباشرة باستخدام إسم الفئة بدون تعريف كائن منها:

```
    MyClass.x = 10;
    System.out.println(MyClass.getX());
```

وبهذه الطريقة يُمكن أن يكون المتغير *x* مشتركاً في القيمة بين الكائنات المختلفة. لكن يجب الحذر والتقليل من استخدام متغيرات مشتركة Global variables حيث يصعب تتبع قيمتها ويصعب معرفة القيمة الحالية لها عند مراجعة الكود. والأفضل من ذلك هو استخدام إجراءات ثابتة تُرسل المتغيرات لها في شكل مُدخلات كما في المثال التالي والذي هو إجراء لتحويل الأحرف الأولى من الكلمات في جملة باللغة اللاتينية إلى حرف كبير Capital letter. وقد سمينا هذه الفئة *Cap*:

```
public class Cap {

   public static String Capitalize(String input) {

       input = input.toLowerCase();
       char[] chars = input.toCharArray();

       for (int i=0; i < chars.length; i++) {
         if (i==0 || chars[i -1] == ' ')  {

           chars[i] = Character.toUpperCase(chars[i]);

       }
   }
   String result = new String(chars);
   return(result);
}
```

نلاحظ أننا كتبنا إجراء من النوع الساكن static اسميناه *Capitalize* يستقبل متغير مقطعي اسمه *input* حيث يُرجع متغير مقطعي بعد تحويل بداية أحرفه إلى أحرف لاتينية كبيرة.
في البداية تُحوّل كل أحرف الجملة إلى حروف لاتنية صغيرة، ثم تُنسخ إلى مصفوفة من نوع الرموز char ثم تُحّول الأحرف التي تلى المسافة إلى حروف كبيرة ويُحوّل كذلك الحرف الأول في الجملة إلى حرف كبير. وفي النهاية تُنسخ تلك المصفوفة إلى متغير مقطعي جديد اسمه result ليُرجع في نداء الإجراء.
ويمكن مناداته مباشرة عن طريق إسم الفئة Cap بالطريقة التالية:

```
    String name = "motaz abdel azeem eltahir";
    System.out.println(Cap.Capitalize(name));
```

فتكون النتيجة كالتالي بعد التنفيذ:

```
Motaz Abdel Azeem Eltahir
```

يُمكن الإستفادة من الإجراءات الساكنة لكتابة مكتبة إجراءات مساعِدة عامة يُمكن استخدامها في عدد من البرامج. مثل إجراء لكتابة الأخطاء التي تحدث في ملف نصي والمعروف بالـ log file. أو تحويل التاريخ إلى شكل معين يُستخدم في نوعية معينة من البرامج، او غيرها من الإجراءات التي تُستخدم بكثرة لتوفير وقت للمبرمج.

قاعدة البيانات SQLite

قاعدة البيانات SQLite هي عبارة عن قاعدة بيانات في شكل مكتبة معتمدة على ذاتها *self-contained* للتعامل مع قاعدة SQLite. ويُمكن استخدام طريقة SQL للتعامل معها. ويمكن استخدامها في أنظمة التشغيل المختلفة بالإضافة إلى الموبايل، مثلاً في نظام أندرويد

يمكن الحصول على المكتبة الخاصة بها وبرنامج لإنشاء قواعد بيانات SQLite والتعامل مع بياناتها من هذا الرابط:

http://sqlite.org/download.html

لإستخدامها في نظام وندوز نبحث عن ملف يبدأ بالإسم sqlite-shell، أما في نظام لينكس يمكننا تثبيت تلك المكتبة وأدواتها بواسطة مثبت الحزم. فقط نبحث عن الحزمة sqlite3

بعد ذلك ننتقل إلى شاشة الطرفية terminal لتشغيل البرنامج وهو من نوع برامج سطر الأوامر، ثم نختار دليل معين لإنشاء قاعدة البيانات ثم نكتب هذا الأمر:

```
sqlite3 library.db
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

بهذه الطريقة نكون قد أنشأنا قاعدة بيانات في ملف إسمه library.db

والآن مازلنا نستخدم هذه الأداة للتعامل مع قاعدة البيانات. ثم أضفنا جدول جديد اسمه books بهذه الطريقة:

```
sqlite> create table books(BookId int, BookName varchar(100));
```

ثم أضفنا كتابين في هذا الجدول:

```
sqlite> insert into books values (1, "Introduction to Java 7");
sqlite> insert into books values (2, "One day trip with Java");
```

ثم عرضنا محتويات الجدول:

```
sqlite> select * from books;
1|Introduction to Java 7
2|One day trip with Java
sqlite>
```

للخروج من الشاشة السابقة نكتب:

```
.exit
```

الآن لدينا قاعدة بينات اسمها library.db وبها جدول اسمه books. يمكن الآن التعامل معها في برنامج جافا كما في المثال التالي.

برنامج لقراءة قاعدة بيانات SQLite

قبل بداية كتابة اي برنامج لقاعدة بيانات SQLite بواسطة جافا يجب أن نبحث عن مكتبة جافا الخاصة بها. وهي مكتبة إضافية غير موجودة في آلة جافا الإفتراضية. ويُمكن البحث عنها بدلالة هذه العبارة:

```
download sqlite-jdbc
```

وهذه إحدى النتائج للبحث يمكن التحميل منها:

http://www.java2s.com/Code/Jar/s/Downloadsqlitejdbc372jar.htm

وهذا مثال لأحد إصدارات هذه المكتبة يمكن تحميلها:

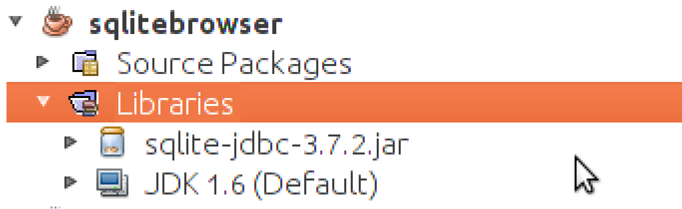
```
sqlite-jdbc-3.7.2.jar
```

يُفضل اختيار النسخة الأحدث.

وهذه المكتبة هي كُل مانحتاجه للتعامل مع قاعدة البيانات SQLite في برامج جافا، فهي لاتحتاج لمخدم لتثبيته حتى تعمل قاعدة البيانات كما قُلنا سابقاً.

أنشأنا مشروع جديد أسميناه sqlitebrowser لعرض قاعدة البيانات Library التي أنشأناها سابقاً.

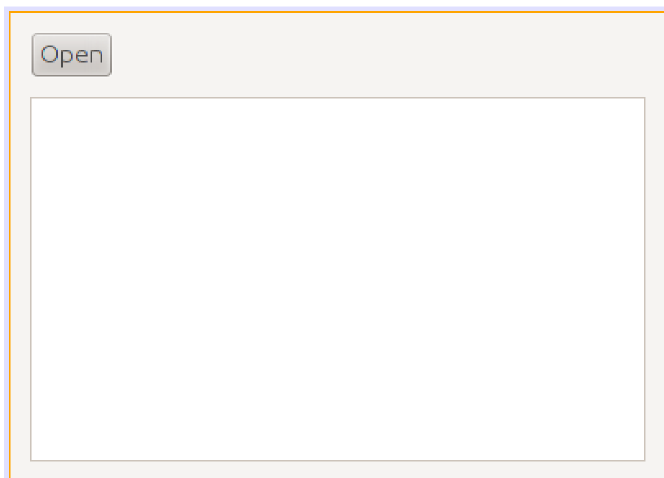
في شاشة المشروع يوجد فرع أسمه Libraries نقف عليه ثم نختار بالزر اليمين للماوس Add JAR/Folder ثم نختار الملف sqlite-jdbc-3.7.2.jar الذي حمّلناه من الإنترنت سابقاً.



أضفنا JFrame Form وأسميناه MainForm واستدعيناه من الملف الرئيسي Sqlitebrowser.java بالطريقة التالية:

```
    public static void main(String[] args) {
        MainForm form = new MainForm();
        form.setVisible(true);
    }
```

في الفورم أضفنا زر و مربع نص TextArea بالشكل التالي:



ثم أضفنا فئة كائن جديد New class أسميناه SqliteClient وهو الكائن الذي سوف يحتوي على إجراءات قراءة قاعدة بيانات *SQLite* والكتابة فيها.

عرّفنا الكائن *dbConnection* من نوع *Connection* داخل كود فئة الكائن *SqliteClient* لتعريف مسار قاعدة البيانات والإتصال بها للإستخدام لاحقاً في باقي إجراءات الكائن SqliteClient.

ثم كتبنا الكود لإستقبال إسم قاعدة البيانات ثم الإتصال بها في الإجراء الرئيسي *constructor* لهذا الكائن:

```
public class SqliteClient {
        Connection dbConnection = null;

    // Constructor
    public SqliteClient (String aDatabaseName) {
        super();
        try {
          Class.forName("org.sqlite.JDBC");
          dbConnection = DriverManager.getConnection(
            "jdbc:sqlite:" + aDatabaseName);
        }
        catch (Exception e){
            System.out.println("Error while connecting: " + e.toString());
        }

    }
```

نلاحظ أننا أحطنا الكود بواسطة *try .. catch* وذلك لأنه من المتوقع أن تحدث مشكلة أثناء التشغيل، مثلاً أن تكون قاعدة البيانات المُدخلة غير موجودة، أو أن مكتبة SQLite غير موجودة.

الإجراء الأول(Class.forName) يُحمّل مكتبة SQLite أثناء التشغيل لنتمكن من نداء الإجراءات الخاصة بهذه القاعدة من تلك المكتبة التي حمّلناها من الإنترنت، فإذا لم تكن موجودة سوف يحدث خطأ.

في السطر التالي هيأنا الكائن dbConnection و أعطيناه الملف التي أرسل اسمه عند تهيأة الكائن SqliteClient

بعد ذلك أضفنا الإجراء *showTable* إلى فئة الكائن SqliteClient لعرض محتويات الجدول المرسل لهذا الإجراء في مربع النص:

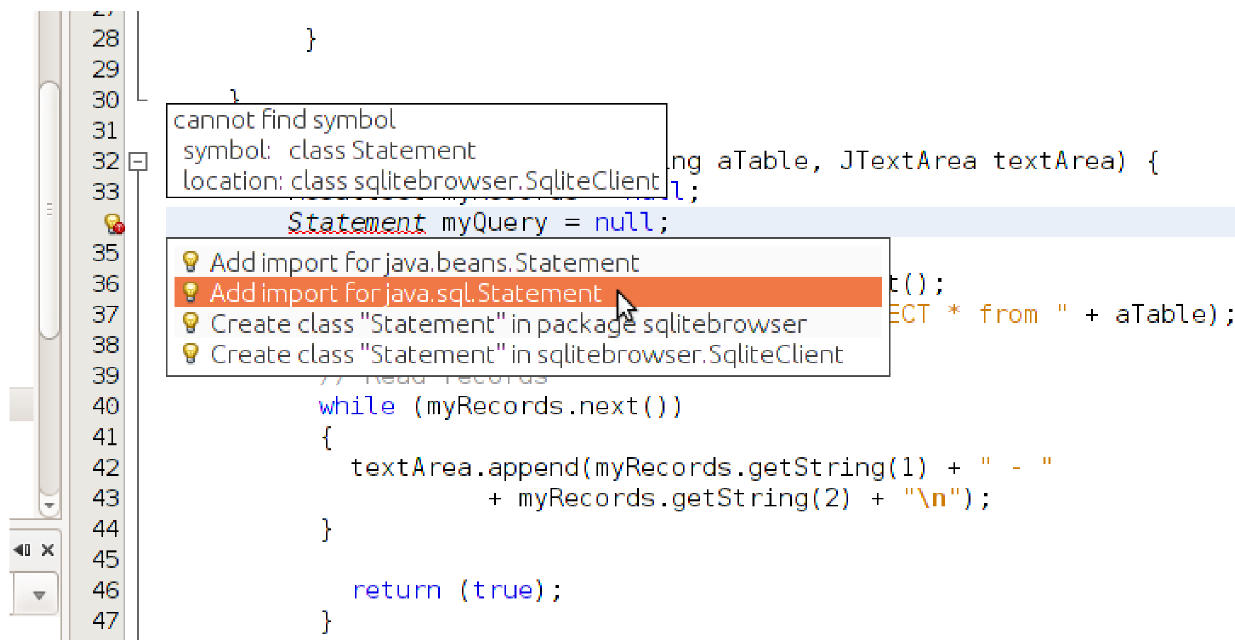
```
    public boolean showTable(String aTable, JTextArea textArea) {

        ResultSet myRecords = null;
        Statement myQuery = null;

        try {
          myQuery = dbConnection.createStatement();
          myRecords = myQuery.executeQuery("SELECT * from " + aTable);
          // Read records
          while (myRecords.next()) {

            textArea.append(myRecords.getString(1) + " - "
                     + myRecords.getString(2) + "\n");
          }
         } catch (Exception e){
```

```
            textArea.append("Error while reading table: " + e.toString() + "\n");
            return (false);
        }
    }
```

عرّفنا في هذا الإجراء كائن من نوع *Statement* أسميناه *myQuery* يسمح لنا بكتابة query بلغة SQL على قاعدة البيانات.

وعند إضافة المكتبة المحتوية على الكائن *Statement* لابد من أن ننتبه لإختيار المكتبة java.sql.Statement ولا نختار الخيار الأول الذي يظهر عند الإضافة التلقائية java.beans.Statement التي تتسبب في أخطاء أثناء الترجمة.

الخيار الصحيح للمكتبة يظهر في الشكل التالي:



ثم هيأناه على النحو التالي:

```
        myQuery = dbConnection.createStatement();
```

ثم نادينا الإجراء executeQuery في الكائن myQuery وأعطيناه مقطع SQL والذي به أمر عرض محتويات الجدول. هذا الإجراء يُرجع كائن جديد من هو عبارة عن حزمة البيانات ResultSet. استقبلناه في الكائن *myRecords* والذي هو من نوع فئة الكائن ResultSet والذي عرّفناه في بداية الإجراء دون تهيأته.

بعد هذه الإجراءات مررنا على كل السجلات في هذا الجدول وعرضنا بعض الحقول في مربع النص الذي أرسل كُمدخل للإجراء *showTable*:

```
    // Read records
    while (myRecords.next())
    {
      textArea.append(myRecords.getString(1) + " - "
                  + myRecords.getString(2) + "\n");
    }
```

والإجراء *next* يُحرّك مؤشر القراءة لبداية الجدول أو حزمة البيانات ثم الإنتقال في كل مرة إلى السجل الذي يليه ويرجع القيمة *true*. وعندما تنتهي السجلات أو لايكون هُناك سجلات من البداية ترجع القيمة *false* وعندها تتوقف الحلقة.

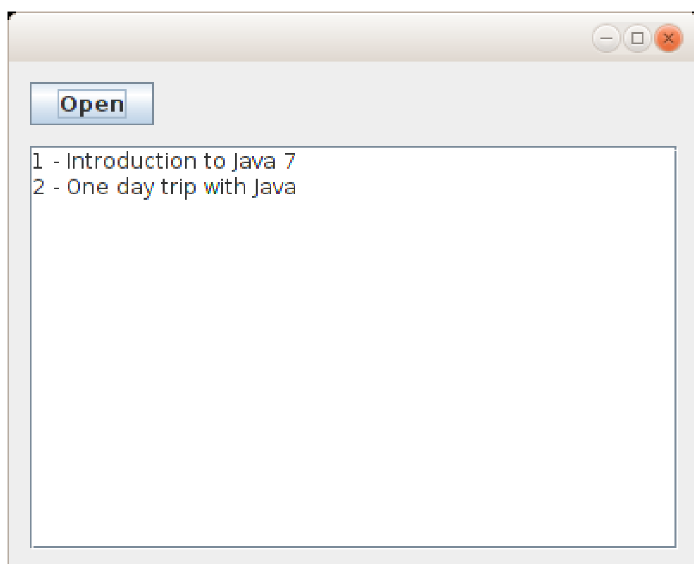
داخل الحلقة قرأنا الحقل الأول والثاني من الجدول المُرسل بواسطة الدالة *getString* وأعطيناها رقم الحقل Field/Column وهي تُرجع البيانات في شكل مقطع، ويُمكن استخدامها حتى مع الأنواع الأخرى مثل الأعداد الصحيحة مثلاً أو التاريخ، فكلها يُمكن تمثيلها في شكل مقاطع.

في الإجراء التابع للزر *Open* في الفورم الرئيس MainForm كتبنا هذا الكود لعرض سجلات الجدول books الموجود في قاعدة البيانات library.db

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

      SqliteClient sql = new SqliteClient("/home/motaz/library.db");

      jTextArea1.setText("");
      sql.showTable("books", jTextArea1);
}
```

في السطر الأول عرّفنا الكائن *sql* من نوع الفئة التي أنشأناها SqliteClient ثم هيأناها بإرسال إسم ملف قاعدة البيانات. وهذا المثال لبرنامج في بيئة لينكس.

ثم في السطر الثاني حذفنا محتويات مربع النص. ثم في السطر الثالث إستدعينا الإجراء showTable في هذا الكائن لعرض محتويات الجدول books وكانت النتيجة كالتالي:



لإضافة كتاب جديد في قاعدة البيانات في الجدول tables أولاً نُضيف إجراء جديد نسميه مثلاً *insertBook* في فئة الكائن SqliteClient بعد الإجراء showTable. وهذا هو الكود الذي كتبناه لإضافة كتاب جديد:

```
  public boolean insertBook(int bookID, String bookName) {

      try {
           PreparedStatement insertRecord = dbConnection.prepareStatement(
                "insert into books (BookID, BookName) values (?, ?)");

          insertRecord.setInt(1, bookID);
          insertRecord.setString(2, bookName);
          insertRecord.execute();
```

```
            return(true);

          } catch (Exception e){

            System.out.println("Error while reading table: " + e.toString());
            return (false);
        }
```

في العبارة الأولى لهذا الإجراء عرّفنا الكائن *insertRecord* من نوع الفئة PreparedStatement وهي تُستخدم لتنفيذ إجراء على البيانات DML مثل إضافة سجل، حذف سجل أو تعديل. ونلاحظ أننا وضعنا علامة إستفهام في مكان القيم التي نُريد إضافتها في الجزء values. وهذه تُسمى مدخلات parameters. سوف تُملأ لاحقاً.

في العبارة الثانية وضعنا رقم الكتاب في المُدخل الأول بواسطة *setInt*، ثم في العبارة الثالثة وضعنا إسم الكتاب في المُدخل الثاني بواسطة *setString*، ثم نفذنا هذا الإجراء بواسطة execute والتي تُرسل طلب الإضافة هذا إلى مكتبة SQLite والتي بدورها تُضيف تلك البيانات في ملف قاعدة البيانات library.db

بعد ذلك نضيف فورم ثاني من نوع JFrame Form ونسميه AddForm نضع فيه المكونات jLabel و jTextField بالشكل التالي:

Book ID

Book Name

Insert

وفي حدث الزر *Insert* نكتب فيه الكود التالي:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

        SqliteClient query = new SqliteClient("/home/motaz/library.db");

        int bookID;
        bookID = Integer.parseInt(jTextField1.getText().trim());
        query.insertBook(bookID, jTextField2.getText());
        setVisible(false);
    }
```

ننتبه لتغيير مسار الملف خصوصاً إذا كان في بيئة وندوز.

في السطر الأول عرّفنا الكائن query من نوع الفئة SqliteClient والتي تحتوي إجراء الإضافة الذي أضفناه مؤخراً.

في للسطر للثاني عرّفنا للمتغير bookID من نوع العدد الصحيح.

الحقل jTextField1 يُرجع محتوياته بواسطة الإجراء getText في شكل مقطع String. ونحن نريده أن يستقبل رقم الكتاب وهو من النوع الصحيح والمقطع يمكن أن يحتوي على عدد صحيح. ثم حولنا المقطع إلى عدد صحيح

بعد حذف أي مسافة غير مرغوب فيها –إن وجدت– بواسطة الدالة trim الموجودة في الكائن String، وذلك لأن العدد إذا كان يحتوي على حروف أو رموز أخرى أو مسافة فإن التحويل إلى رقم بواسطة الإجراء parseInt سوف ينتج عنها خطأ. والدالة trim تُرجع مقطع محذوفة منه المسافة من بداية ونهاية النص، لكنها لاتؤثر على الكائن الذي نُفذ فيه. مثلاً الكائن jTextField1 لا تُحذف المسافة منه. لتوضيح ذلك انظر المثال التالي:

```
        myText = aText.trim();
```

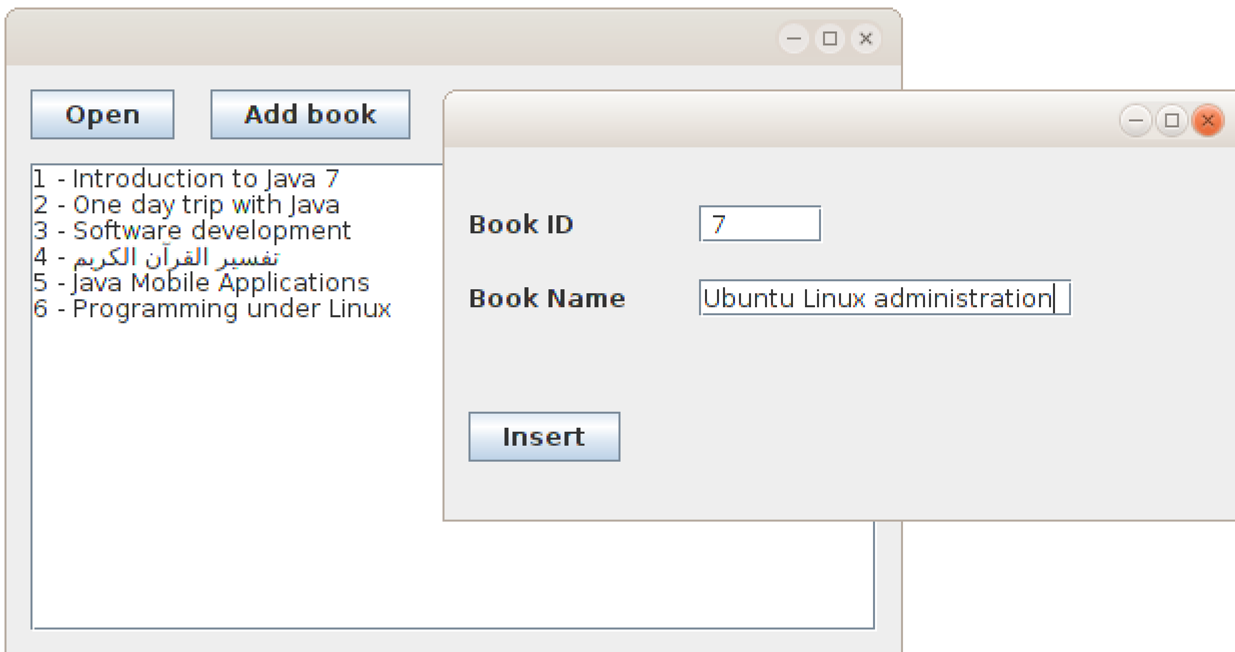
الكائن aText لايتثأر بالدالة trim أما المتغيّر myText فيُخزّن فيه نُسخة مقطع جديد من المقطع aText بدون مسافة.

في للسطر للرابع استدعينا الإجراء insertBook في الكائن query وأعطيناه رقم الكتاب في المدخل الأول ثم اسم للكتابفي للمُدخل للثاني. ثم أغلقنا للفورم في للسطر الأخير بواسطة setVisible وأعطيناها القيمة false.

في الفورم الرئيس أضفنا زر بعنوان (Add book) وكتبنا فيه الكود التالي لإظهار فورم الإضافة:

```
        AddForm add = new AddForm();
        add.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        add.setVisible(true);
```

وهذا هو شكل البرنامج بعد تنفيذه:



عند عمل build لهذا البرنامج بواسطة shift + F11 نلاحظ وجود دليل فرعي إسمه lib داخل الدليل dist وهو يحتوي على المكتبة التي استخدمناها والتي هي ليست جزء من آلة جافا الإفتراضية. وعند نقل البرنامج إلى أجهزة أخرىيجب نقل الدليل

lib مع البرنامج، وإلا تعذر تشغيل إجراءات قاعدة البيانات.

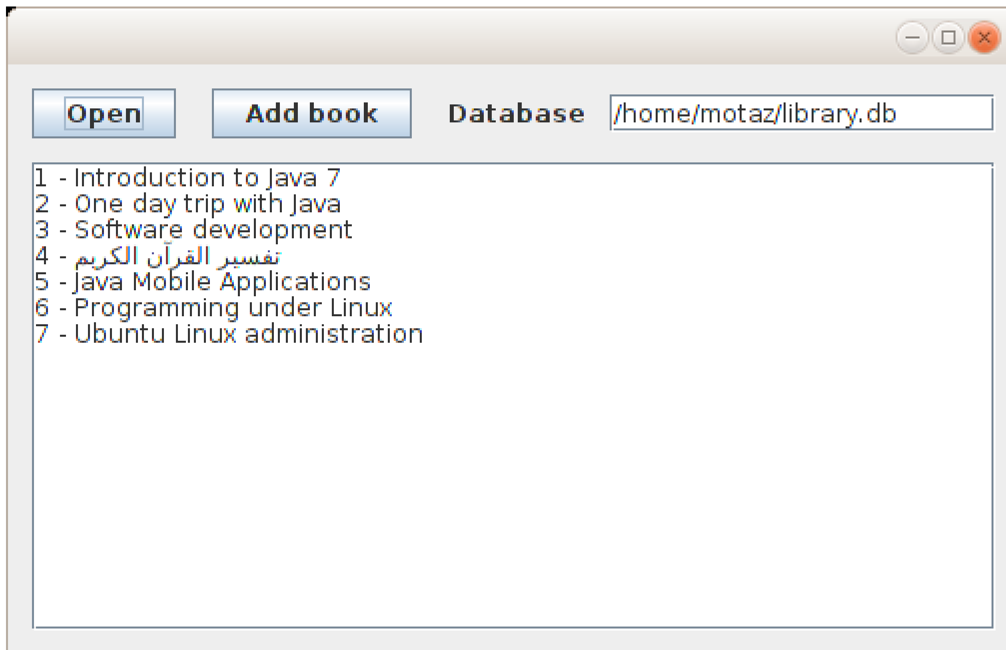
في هذا المثال نحتاج لنقل ثلاث ملفات:

- `sqlitebrowser.jar` وهو الملف التنفيذي للبرنامج في صيغة Byte code
- `sqlite-jdbc-3.7.2.jar` وهو ملف المكتبة داخل الدليل *lib* .

ولابد أن نضعه في هذا الدليل داخل الدليل الذي نضع فيه البرنامج

- library.db وهو ملف قاعدة البياناتوكان من الأفضل جعل مسار قاعدة البيانات خارج كود البرنامج، مثلاً

نضيف jTexField آخر ليكون البرنامج بالشكل التالي:



ونغير تهيئة كائن قاعدة البيانات بالشكل التالي:

```
        SqliteClient sql = new SqliteClient(jTextField1.getText());
```

بهذه الطريقة يكون البرنامج أكثر حرية في النقل portable و لايعتمد على ثوابت في نظام معين. وهي طريقة جيدةفي تطوير البرامج تزيد من إمكانية إستخدامه، خصوصا عند إختيار لغة برمجة متعددة المنصات مثل جافا.

الوراثة inheritance

الوراثة هي من أساسيات البرمجة الكائنية، وتُستخدم الوراثة لعدة أهداف منها التجريد abstraction وهي تقسيم الفئات إلى فئة عامة مجردة وفئة متخصصة تفصيلية، ومن تطبيقات التجريد هو إضافة إمكانات جديدة دون تغيير الفئة الأساسية.

في هذا المثال كتبنا فئة لإدارة الملفات (التأكد من وجود ملف، و إنشاء ملف جديد، و حذف ملف، و استعراض ملفات في دليل معين).

لعمل ذلك أنشأنا أولاً برنامج جافا جديد اسميناه *FilesManagement* ثم ا ضفنا فئة *Class* جديدة اسميناها *FileManager* وذلك عن طريق الوقوف على حزمة *filesmanagement* الموجودة في *source packages* ثم الضغط بالزر اليمين في الماوس ثم اختيار *New/Java Class*

ثم كتبنا هذا الكود داخل هذه الفئة الجديدة:

```
package filesmanagement;

import java.io.File;


public class FileManager {

    public boolean checkExistence(String filename){

        File file = new File(filename);
        return file.exists();
    }

    public boolean createFile(String filename){

        try {
            File file = new File(filename);
            file.createNewFile();
            return true;
        }
        catch (Exception ex){
            System.out.println("Unable to create file: " + ex.toString());
            return false;
        }

    }

    public boolean deleteFile(String filename){
        try {
            File file = new File(filename);
            file.delete();
            return true;
        }


        catch (Exception ex){
            System.out.println("Error while deleting file: " + ex.toString());
            return false;
        }
    }
```

```
    public String[] listFiles(String directory){

        File dir = new File(directory);

        File files[] = dir.listFiles();

        String fileNames[] = new String[files.length];

        // Fill file names array
        for (int i=0; i<files.length; i++){
            fileNames[i] = files[i].getName();
        }

        return fileNames;
    }
}
```

ثم عرّفنا و هيأنا ثم استخدمنا كائن من هذه الفئة في البرنامج الرئيس كالتالي:

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

        FileManager manager = new FileManager();

        if (! manager.checkExistence("/home/motaz/testing/first.txt")) {
            manager.createFile("/home/motaz/testing/first.txt");
        }
        String files[] = manager.listFiles("/home/motaz/testing");

        // List files
        for (String filename: files){
            System.out.println(filename);
        }
    }
```

نلاحظ أن هذه الفئة عامة في التعامل مع الملفات، حيث أن أي نوع ملف يمكن أن نتأكد من وجوده أو من عدم وجوده، كذلك إنشاءه وحذفه واستعراض أسماء ملفات في دليل معين هي ميزة مشتركة بين أنواع الملفات المختلفة، حيث تشترك فيها كل أنواع الملفات سواءً كانت نصية، أو ملفات صوتية أو صور أو فيديو أو حتى ملفات تنفيذية. هذا ما نُسمية بالتجريد بأن تكون هذه الفئة عامة الاستخدام وغير مخصصة لنوع معين من الملفات، فإذا أدخلنا إجراء مثلاً لقراءة ملف نصي داخل نفس الفئة فنكون قد كسرنا التجرد الذي تُمثله هذه الفئة، حيث لا يمكن استعراض ملف صورة مثلاً بهذه الإجراء الجديد، فاصبح هُناك إستثناءات في اجراءات هذه الدالة تعتمد على نوع الملف، وهذا لم يكن الهدف وراء إنشاء هذه الفئة من الأساس.
للمحافظة على تجريد الفئة السابقة دون تغيير قوانينها مع تحقيق هدف إضافة هذا الإجراء الجديد نُنشيء فئة جديدة نُسميها *TextFileManager* لإضافة هذا الإجراء الجديد، لكن حتى لا تضيع الإجراءات الموجودة أصلاً في الفئة *FileManager* نرث الفئة الأخيرة بما تحتويه من إجراءات في الفئة الجديدة.
أضفنا هذه الفئة عن طريق *New/Java Class* ثم أسميناها *TextFileManager* ثم أضفنا عبارة :

```
extends FileManager
```

ليصبح كود الفئة كالتالي:

```
public class TextFileManager extends FileManager {

}
```

ثم بعد ذلك نشرع في إضافة الإجراء الجديد *readTextFile*

```
    public void readTextFile(String fileName){

        try {

          FileReader reader = new FileReader(fileName);
          char buf[] = new char[1024];
          int numread;
          while ((numread=reader.read(buf)) != -1){
              String text = new String(buf, 0, numread);
              System.out.print(text);
          }
          reader.close();

        } catch (Exception ex){
            System.out.println("Error while reading file: " + ex.toString());
        }

    }
```

كذلك يمكننا إضافة إجراء آخر للكتابة في ملف نصي:

```
    public void writeIntoTextFile(String fileName, String lines){

        try {

            // Check file existence
            if (!checkExistence(fileName)) {
                createFile(fileName);
            }

            FileWriter writer = new FileWriter(fileName);
            writer.write(lines);
            writer.close();

        } catch (Exception ex){
            System.out.println("Error while writing into file: " + ex.toString());
        }
    }
```

نلاحظ أننا استخدمنا الإجرائين: *checkExistence* و *createFile* الموروثة من الفئة الأم *FileManager* وكأنها موجودة معنا في نفس الفئة الحالية *TextFileManager*

في الإجراء الرئيسي أنشأنا كائن من الفئة الجديدة، ونلاحظ أنه يمكننا نداء إجراءات الفئة الأساسية *FileManager* بالإضافة لإجراءات الفئة الجديدة:

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

        TextFileManager manager = new TextFileManager();

        if (! manager.checkExistence("/home/motaz/testing/first.txt")) {
            manager.createFile("/home/motaz/testing/first.txt");
        }
        String files[] = manager.listFiles("/home/motaz/testing");

        // List files
        for (String filename: files){
            System.out.println(filename);
        }

        manager.writeIntoTextFile("/home/motaz/testing/second.txt",
             "This is a test file\n written using inheritance\n");
        manager.readTextFile("/home/motaz/testing/second.txt");
    }
```

بهذه الطريقة لم نُعدل على الفئة *FileManager* لكن عملنا إضافات خاصة في فئتنا الجديدة *TextFileManager* المتخصصة في الملفات النصية. يمكن كذلك عمل فئات متخصصة في أي نوعية أخرى من الملفات بعد الوراثة من الفئة الرئيسة *FileManager*.

نكون كذلك قد حققنا إعادة استخدام الكود بدلاً من تكراره كل مرة مع الفئات المتخصصة الجديدة، حيث يكون هناك دائماً إجراءات مشتركة. و كل هذا يُعد من الطرق المثلى لكتابة البرامج.

تكرار حدث بواسطة مؤقت

في هذا المثال نُريد كتابة التاريخ والساعة في الشاشة كل فترة معينة، مثلاً كل ثانية. ولعمل ذلك أنشأنا مشروع جديد سميناه timer ثم أضفنا MainForm من نوع JFrame Form، ثم أضفنا فئة جديدة new class أسميناها MyTimer فكان تعريفها بالشكل التالي:

```
public class MyTimer {
```

لكن غيّرنا هذا التعريف لنستخدم الوراثة التي ذكرناها سابقاً، وذلك بدلاً من كتابة فئة كائن جديد من الصفر نستخدم فئة لديها خصائص مشابهة ثم نزيد فيها. وفئة هذا الكائن اسمها TimerTask. نرثها بهذه الطريقة:

```
public class MyTimer extends TimerTask{
```

ثم نُعرّف كائن myLabel بداخله حتى نعرض التاريخ والوقت فيه، ثم نكتُب إجراء التهيأة كالتالي:

```
    JLabel myLabel;

    public MyTimer(JLabel alabel){
        super();
        myLabel = alabel;
    }
```

وفي هذا الإجراء نستقبل الكائن alabel ثم نحفظ نسخة منه في الكائن myLabel.

بعد ذلك نكتُب الإجراء الذي سوف يُنادى كل فترة وأسمه run بهذه الطريقة:

```
    @Override
    public void run() {
        Date today = new Date();
        myLabel.setText(today.toString());

    }
```

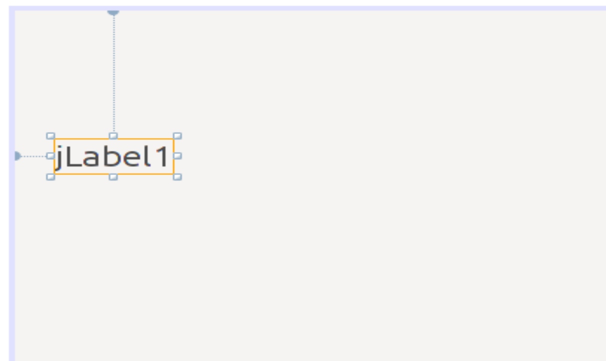
ويمكن إضافة تعريف هذا الإجراء تلقائياً بواسطة implement all abstract methods والتي تظهر في سطر تعريف الكائن MyTimer بالطريقة التالية:

```
12   *
13   * @author motaz
14   timer.MyTimer is not abstract and does not override abstract method run() in java.util.TimerTask
     public class MyTimer extends TimerTask{
16   Implement all abstract methods
17       public MyTimer(JLabel alabel){
18           super();
19           myLabel = alabel;
20       }
21
```

وهذا هو كود الكائن كاملاً:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package timer;

import java.util.Date;
import java.util.TimerTask;
import javax.swing.JLabel;

/*
 * @author motaz
 */
public class MyTimer extends TimerTask{
    JLabel myLabel;
    public MyTimer(JLabel alabel){
        super();
        myLabel = alabel;
    }

    @Override
    public void run() {
        Date today = new Date();
        myLabel.setText(today.toString());

    }

}
```

نضع jLabel في الفورم الرئيسي MainForm ونزيد حجم الخط فيه ليكون بالشكل التالي:



في إجراء تهيئة هذا الفورم نعدل الكود إلى التالي:

```
    public MainForm() {
        initComponents();
        java.util.Timer generalTimer = null;
```

```
        MyTimer timerObj = new MyTimer(jLabel1);
        generalTimer = new java.util.Timer("time loop");
        generalTimer.schedule(timerObj, 2000, 1000);
    }
```

في هذا السطر:

```
        java.util.Timer generalTimer = null;
```

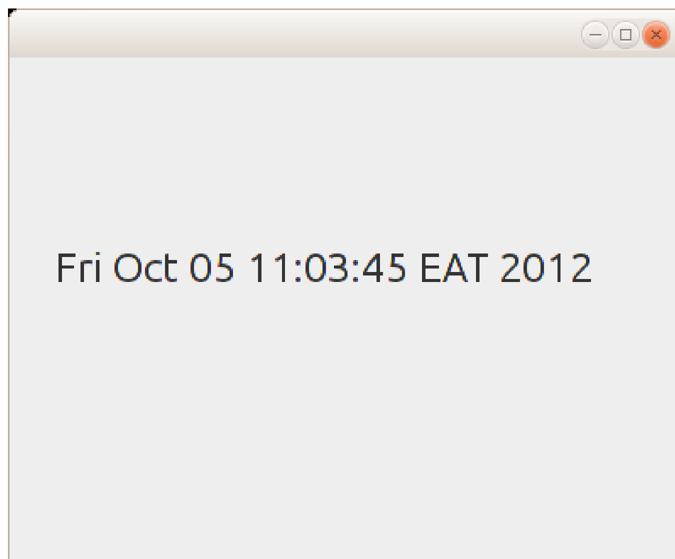
عرّفنا الكائن generalTimer من النوع Timer. وهذا الكائن لديه خاصية تكرار الحدث بفترة زمنية محددة.

ثم عرّفنا الكائن timerObj من الفئة MyTimer والتي كتبناها لإظهار التاريخ والوقت كل ثانية.

ثم هيأنا الكائن generalTimer. وفي السطر الأخير شغّلنا المؤقت schedule وأرسلنا له الكائن timerObj ليُنفّذ الإجراء run كل فترة معينة. والرقم الأول 2000 هو بداية التشغيل أول مرة، وهو بالملي ثانية، أي ينتظر ثانيتين قبل التشغيل أول مرة.

الرقم الثاني 1000 هو التكرار بالملي ثانيةً. حيث يُظهر التاريخ والوقت كل ثانية.

ننفذ البرنامج لنرى أن الثواني تتغير في الفورم الرئيس:



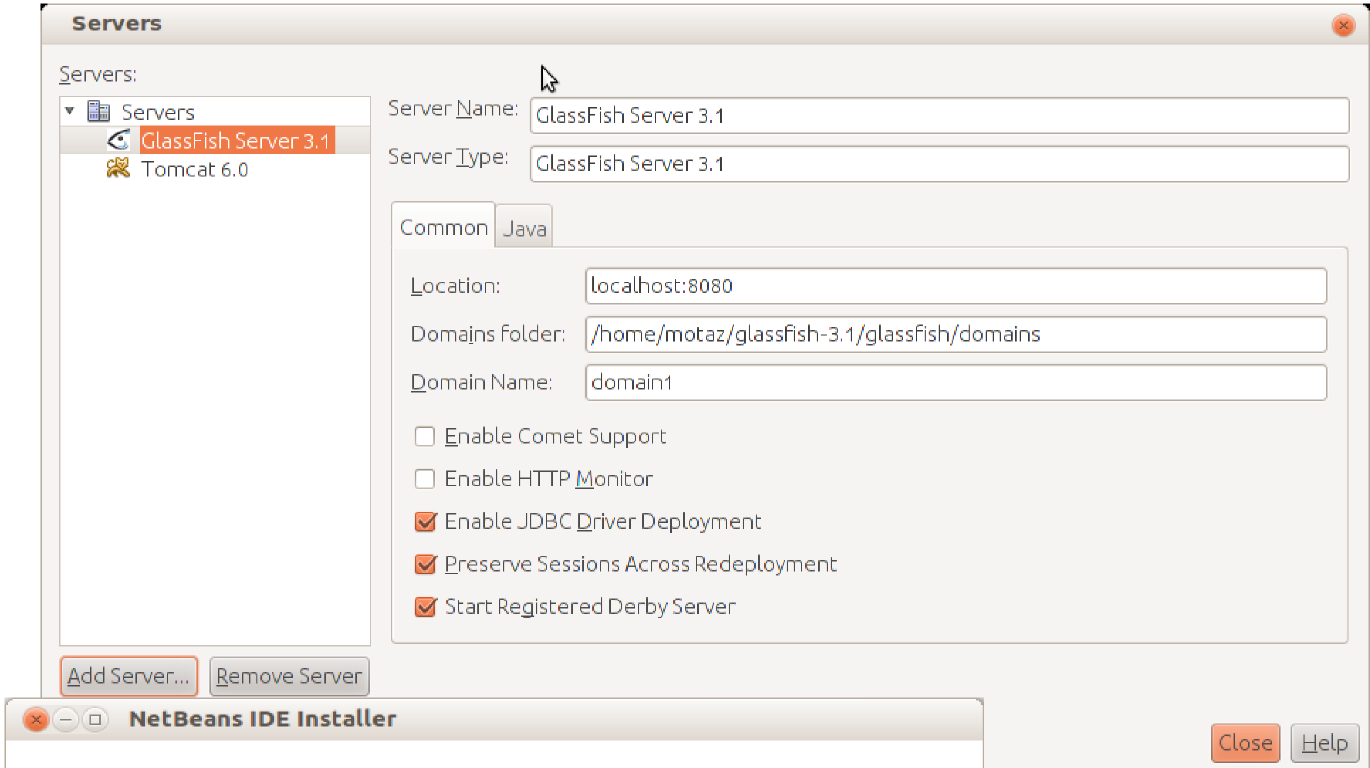
وسوف يُنفّذ هذا الإجراء تلقائياً إلى إغلاق البرنامج.

برمجة الويب باستخدام جافا

تعد لغة جافا من أهم اللغات التي تدعم برمجة الويب web applications وخدمات الويب web services مثل الـ SOAP والـ RESTfull. والتقنية التي استخدمناها هنا في شرح برامج الويب هي تقنية Servlet و JSP. وفي النهاية يُشغّل برامج الويب وخدمات الويب المكتوبة بجافا في مخدم ويب خاص مثل Apache Tomcat أو GlassFish.

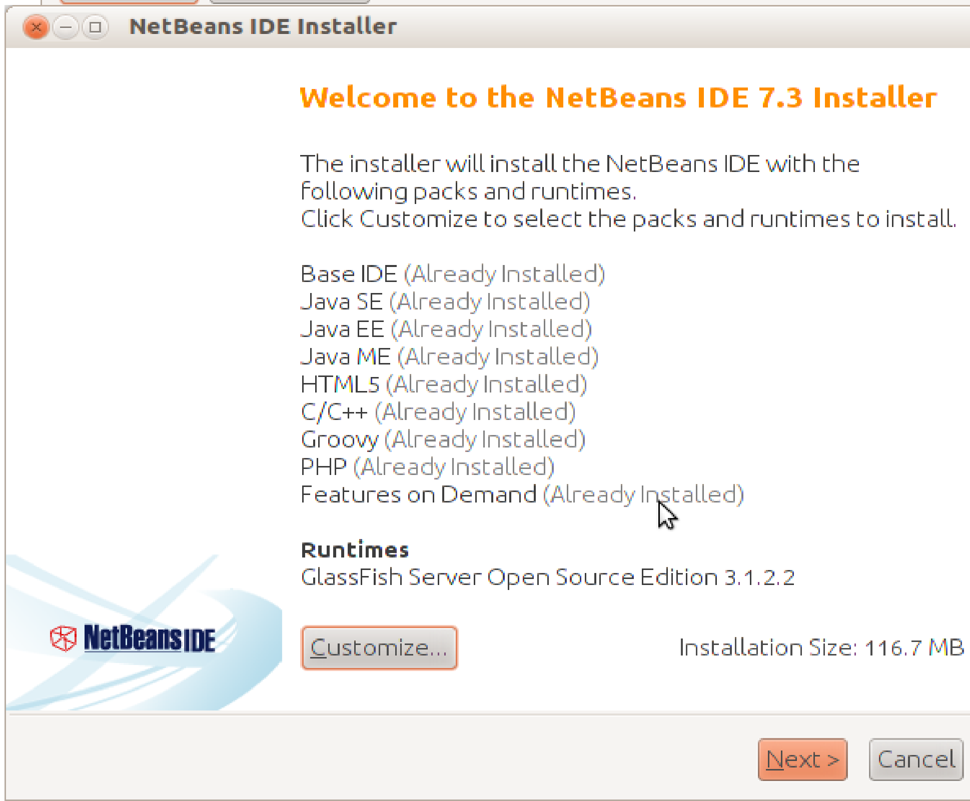
تثبيت مخدم الويب

قبل البداية في كتابة برامج ويب يجب التأكد من أنه يوجد مخدم ويب خاص بجافا وأن له إعدادات في بيئة التطوير NetBeans وذلك عن طريق Tools/Servers فتظهر هذه الشاشة:

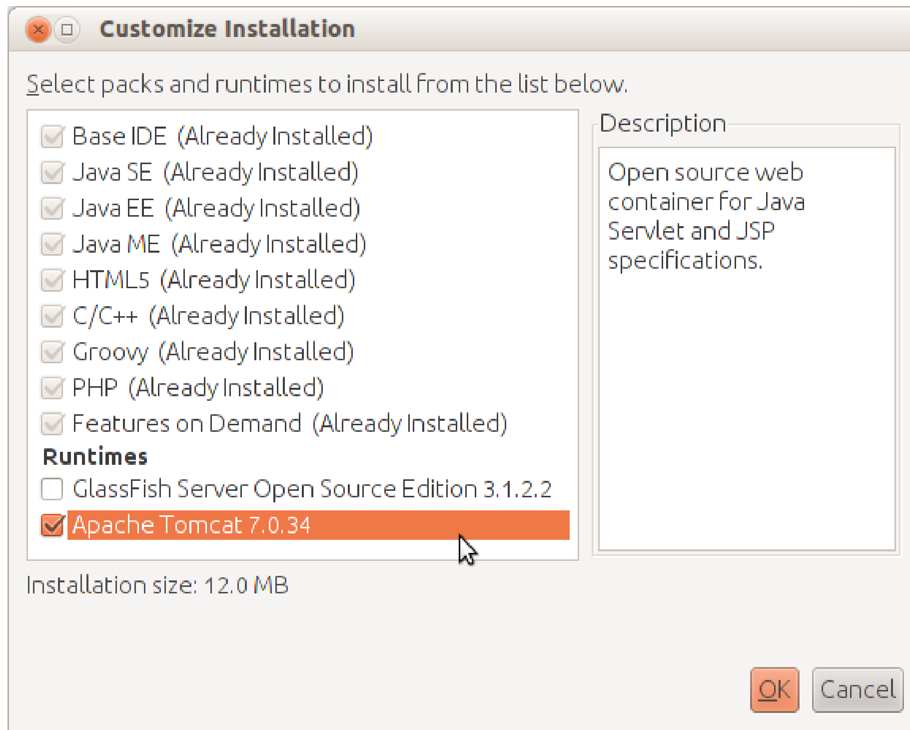


ويظهر فيها وجود GlassFish و Tomcat .

ويُفضل تثبيت Tomcat أثناء تثبيت NetBeans باختيار Customize كالتالي:



ثم اختيار Tomcat بدلاً عن Glassfish:



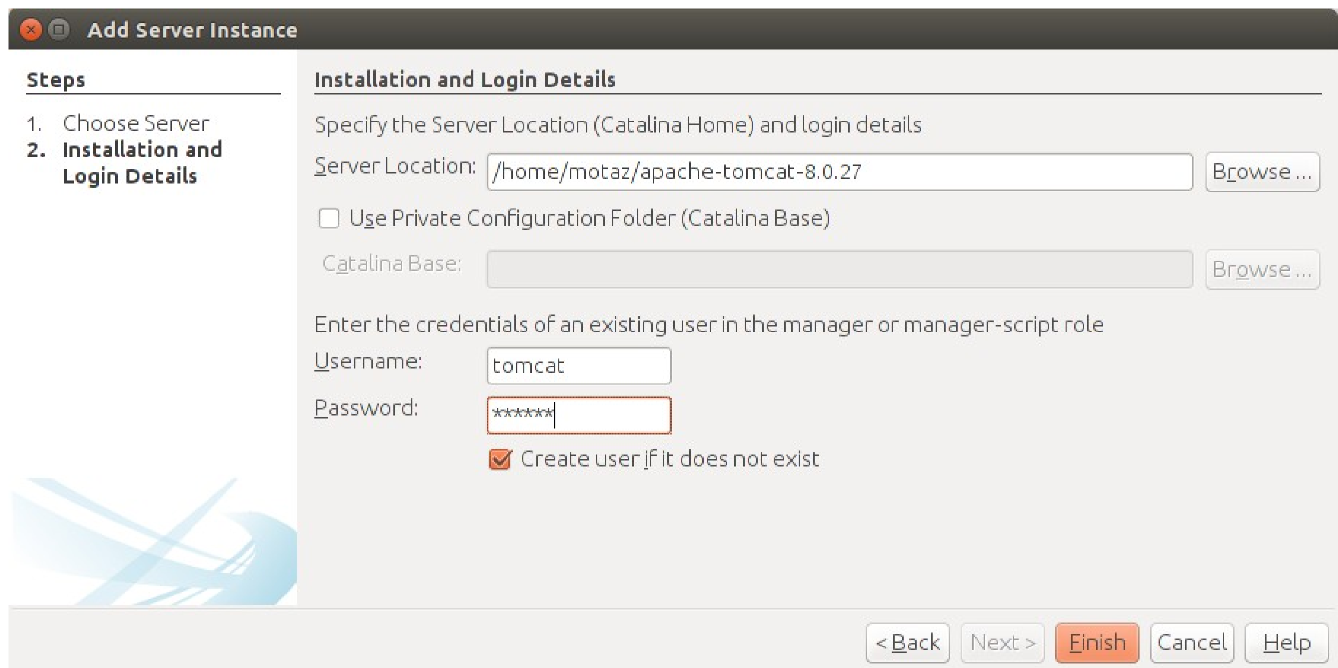
وفي حالة أننا لم نختار الخيار السابق أثناء التثبيت فيكون مخدم Tomcat غير مُثّبت، حينها يمكننا أن نتحصل عليه من موقع tomcat.apache.org . و نختار منه الملف الذي ينتهي بالإمتداد zip. كما يظهر في هذه الشاشة:

Please see the README file for packaging information. It explains wha

Binary Distributions

- Core:
  - zip (pgp, md5, sha1)
  - tar.gz (pgp, md5, sha1)
  - 32-bit Windows zip (pgp, md5, sha1)
  - 64-bit Windows zip (pgp, md5, sha1)
  - 32-bit/64-bit Windows Service Installer (pgp, md5, sha1)
- Full documentation:
  - tar.gz (pgp, md5, sha1)

بعد ذلك نفك الملف في مكان معروف، ثم نُضيف مخدم جديد عن طريق الزر Add Server ثم نختار رقم نسخة Tomcat التي ثبتناها ثم نُدخل الدليل الذي توجد فيه كما في هذا المثال:



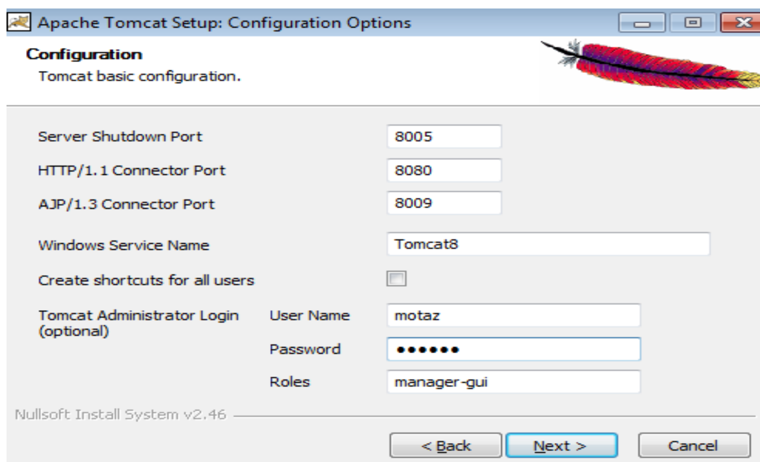
عند الحاجة لتثبيت برامج الويب في أجهزة أخرى (مخدم مثلاً) نُثبّت نسخة Apache Tomcat بطريقة مختلفة حسب نظام التشغيل المستهدف، من الموقع:

http://tomcat.apache.org/

في بيئة وندوز نختار الملف:

32-bit/64-bit Windows Service Installer (pgp, md5, sha1)

ثم نُدخل إسم المستخدم والذي سوف نستخدمه لاحقاً لرفع البرامج وإدارة مخدم الويب.



أما في نظام لينكس فيمكن تثبيته عن طريق مدير الحزم، في نظام اوبونتو أو انظمة ديبيان عموماً نكتب:

```
sudo apt-get install tomcat7 tomcat7-admin
```

ويمكن استبدالها بـ tomcat8 أو tomcat9 حسب إصدارة لينكس المستخدمة.

كذلك يمكن تثبيت نُسخة تعمل مع جميع أنظمة لينكس بتحميلها من موقع tomcat واختيار الملف الذي ينتهي بالإمتداد .zip

بعد ذلك نبحث عن الملف tomcat-users.xml في دليل الإعدادات حسب نظام التشغيل، وحسب نسخة الـ tomcat مثلاً في بيئة لينكس يكون في هذا المسار:

```
/etc/tomcat7/tomcat-users.xml
```

في نظام وندوز يكون في هذا المسار، وذلك إذا احتجنا إلى تغيير اسم الدخول أو كلمة المرور:

```
C:\Program Files\Apache Software Foundation\Tomcat 8.0
```

نُضيف هذا السطر لمستخدم جديد أو نُعدّل مستخدم موجود لإعطائه الصلاحيات الكافية:

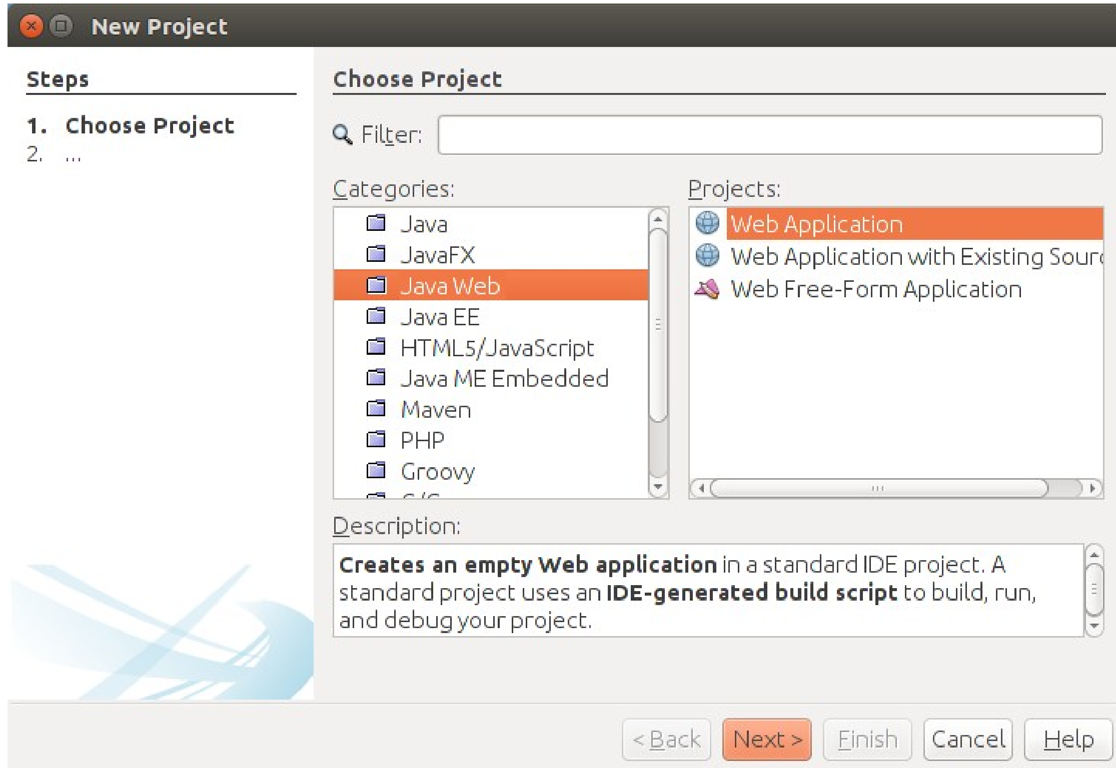
```
<user username="motaz" password="tomcat" roles="manager-gui,manager-script"/>
```

عندها تكون البيئة جاهزة لتثبيت وتشغيل برامج أو خدمات ويب.

أول برنامج ويب

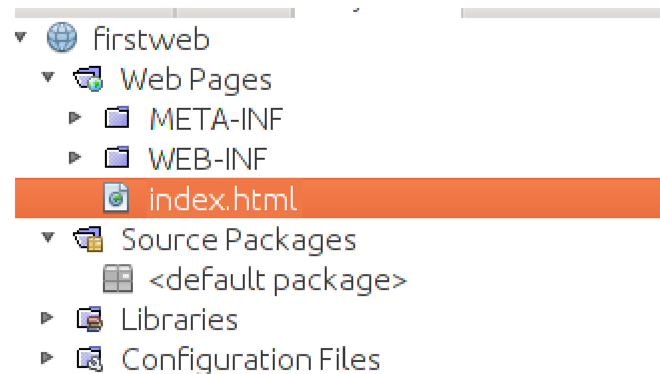
لعمل برنامج الويب الأول في جافا نُنشيء مشروع جديد عن طريق

```
New project\Java Web\Web Application
```



ثم نختار إسم للمشروع مثلاً *firstweb* ثم نختار المخدم، وهو في هذه الحال Tomcat ثم نضغط على زر Finish.

فيظهر البرنامج في محرر NetBeans كالتالي:



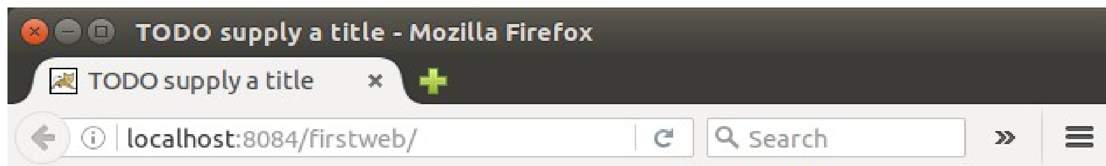
وعند فتح الملف index.html يظهر كود الـ HTML التالي:

```
<!DOCTYPE html>
<!--
To change this license header, choose License Headers in Project Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>TODO write content</div>
    </body>
</html>
</html>
```

يُمكنن تعديل عبارة TODO write content إلى

```
        <h1>Hello World!</h1>
```

ثم حفظ التغييرات و تشغيل البرنامج مباشرة من بيئة NetBeans بواسطة المفتاح F6 لعرض هذه الصفحة بواسطة تشغيل مخدم Tomcat لتظهر على متصفح خاص بالشكل التالي:



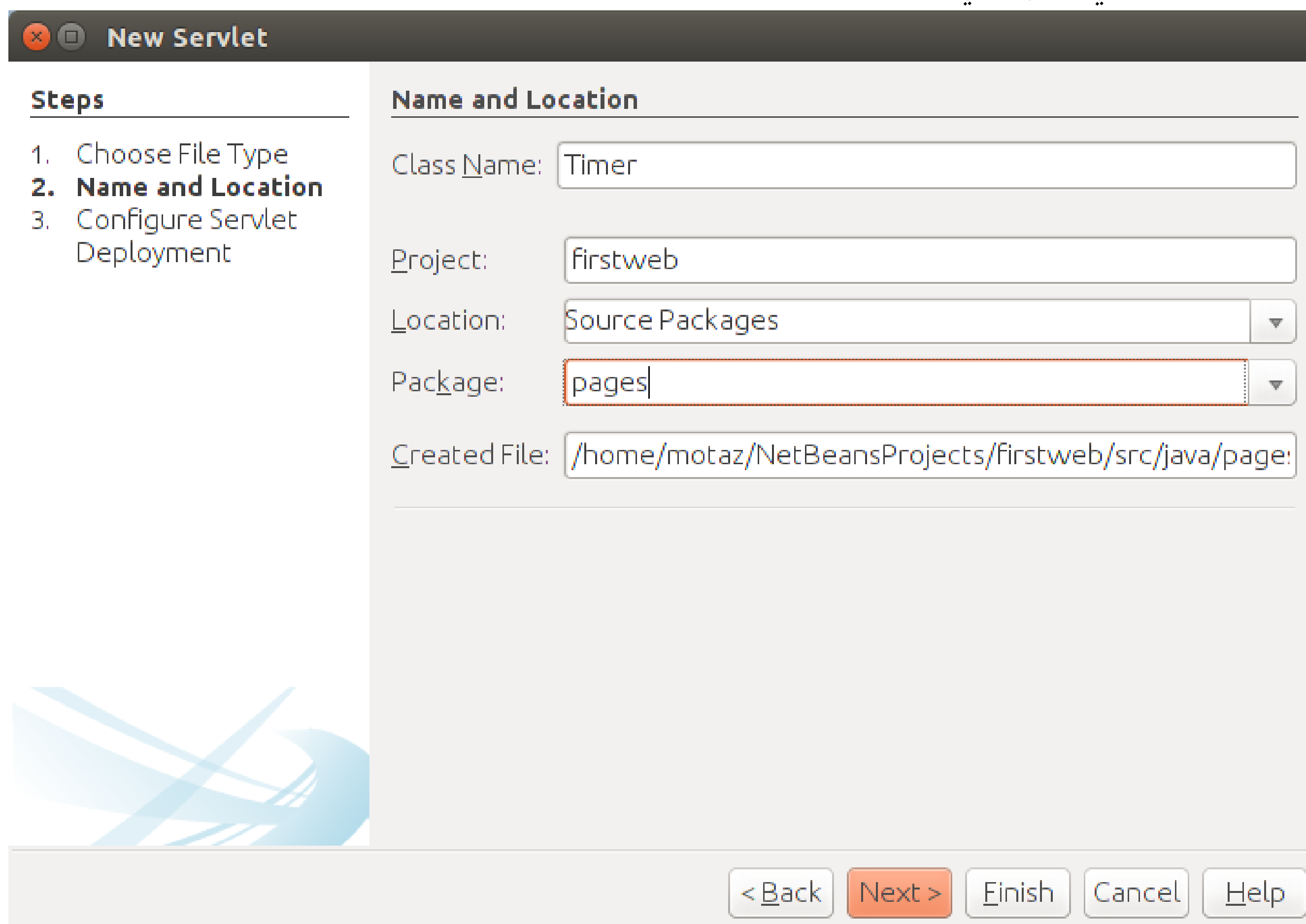
نلاحظ أن العنوان هو:

http://localhost:8084/firstweb/

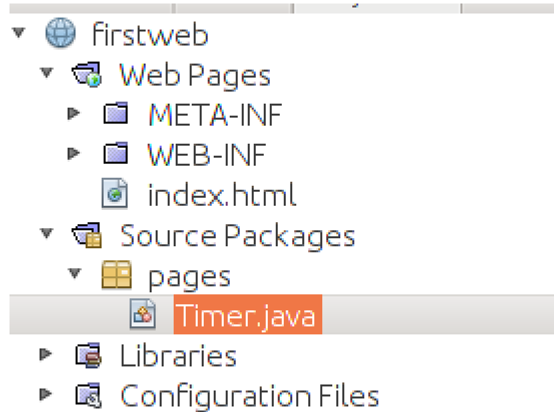
ويظهر فيه رقم المنفذ 8084 والمنفذ الإفتراضي لمخدم الـ tomcat هو 8080 لكن الأخير يُستخدم في حالة التثبيت النهائي للبرنامج لاستخدامه من قبل مستخدمي النظام، أما المنفذ الأول فيُستخدم مع NetBeans لتطوير برامج الويب، ويمكن أن يحتوي الكمبيوتر على أكثر من نسخة tomcat يعملان في نفس الوقت، لكن كل واحد لديه منفذ مختلف.

نغلق المتصفح لنرجع للبرنامج لنضيف فيه محتوي تفاعلي، حيث أن الصفحة السابقة كانت صفحة ثابتة static أو أنها تستخدم تقنية مختلفة وهي HTML فقط.

في شجرة المشروع وفي الفرع Source Packages نُضيف Servlet عن طريق الزر اليمين للماوس ثم New Servlet ثم نُسميه Timer كما في الشكل التالي:



نُسمَي الحزمة package بإسم pages، فيصبح شكل المشروع كالتالي:



وعند فتح Timer.java يظهر الكود التلقائي الذي يحتوي على الإجراء *ProcessRequest* كما في المثال التالي:

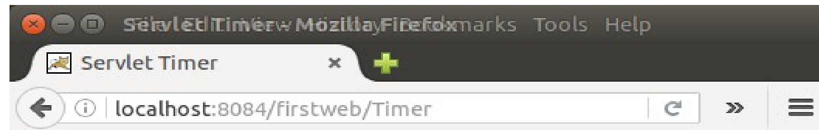
```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Timer</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet Timer at " +
                  request.getContextPath() + "</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }
```

نُضيف كود إظهار التاريخ والوقت ليصبح الكود كالتالي:

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Timer</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet Timer at " +
                  request.getContextPath() + "</h1>");
```

```
            Date today = new Date();
            out.println("Time in server is: <b>" + today.toString() + "</b>");

            out.println("</body>");
            out.println("</html>");
        }
    }
```

ثم نُشغّله مرة أخرى لكن نضيف في عنوان التصفح إسم ال Servlet وهو Timer/ كالتالي:



Servlet Timer at /firstweb

Time in server is: **Fri May 27 12:01:47 EAT 2016**

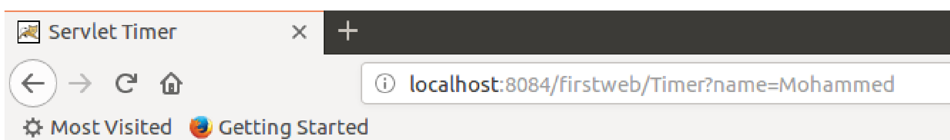
لاستقبال مُدخلات لبرنامج الويب، نقرأها بواسطة request.GetParameter. هذه المُدخلات تُرسل مع العنوان URL بالشكل التالي:

```
http://localhost:8084/firstweb/Timer?name=Mohammed
```

نُضيف السطر التالي للـ Servlet Timer :

```
 out.println("<br>Your name is: <font color=green>" + request.getParameter("name")
             + "</font>");
```

فتكون النتيجة كالتالي في المتصفح:



Servlet Timer at /firstweb

Time in server is: **Tue Sep 25 07:35:52 EET 2018**
Your name is: Mohammed

تثبيت برامج الويب

بعد الإنتهاء من تطوير برامج الويب نُنتج نسخة تنفيذية بواسطة Clean and Build لنتحصل على الملف firstweb.war في الدليل dist من المشروع، وهو ملف مضغوط يحتوي على ملفات الـ Byte code و المكتبات التي يتحاج إليها البرنامج بالإضافة إلى ملفات HTML , JSP بالإضافة للصور وغيرها من محتويات برامج الويب. يُمكن وضعه في مخدم Tomcat الذي نُريد تثبيت البرنامج فيه. ونضعه في الدليل webapps ضمن دليل برنامج Tomcat. مثلاً في نظام لينكس يكون إسم الدليل هو:

```
/var/lib/tomcat7/webapps/
```

وفي نظام ويندوز نجده في الدليل :

```
Program Files\Apache Software foundation\Tomcat 7\webapps
```

هذه المرة نستخدم نُسخة tomcat المعدة للتثبيت النهائي للنظام والتي تكلمنا عنها سابقاً والتي تستخدم المنفذ 8080 لانحتاج لنسخ الملف يدوياً إلى الدليل webapps بل نستخدم مدير برامج الويب في مخدم Tomcat عن طريق المتصفح، حيث نكتب العنوان التالي:

```
http://localhost:8080
```

ثم نضغط رابط Manager Webapp ونكتب إسم الدخول الذي اضفناه سابقاً لتظهر لنا الشاشة التالية في المتصفح:

في الجزء Deploy/War file to deploy

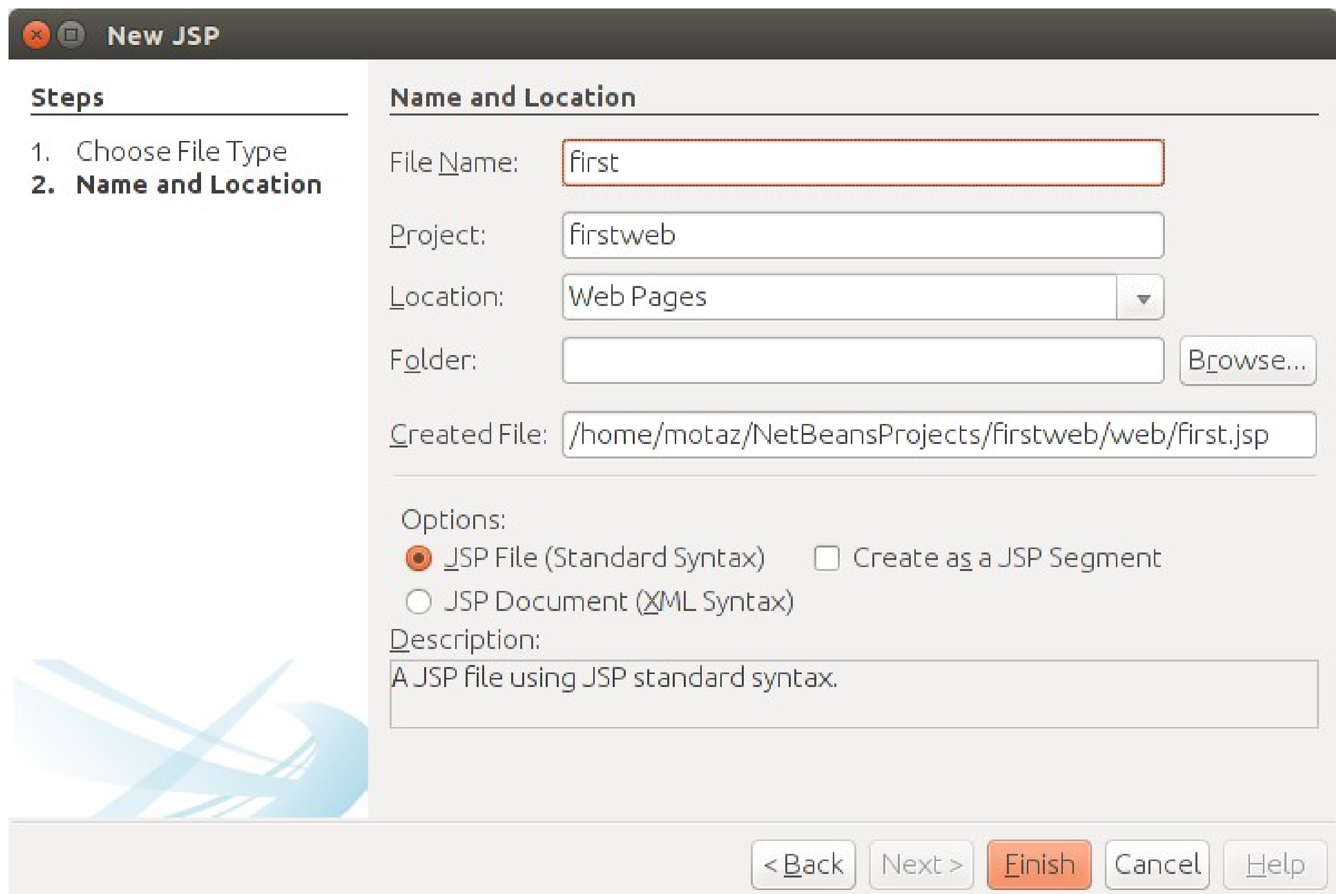
نختار الملف ثم نضغط على زر Deploy ليُرفع الملف في دليل webapps ليصبح جاهزاً للإستخدام.

تقنية JSP

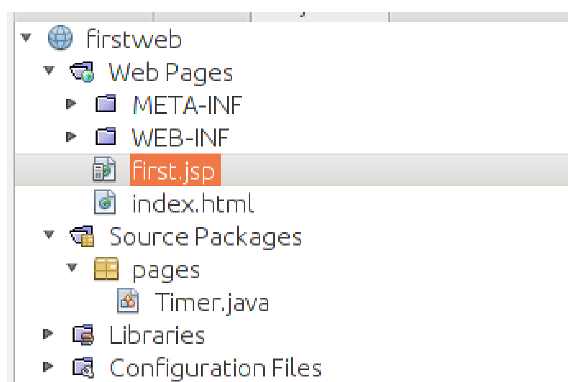
كلمة JSP هي اختصار لـ Java Server Page وهي مشابهة لطريقة برمجة الويب في لغة PHP أو ASP حيث تُضاف صفحة تنتهي بالإمتداد .jsp ويمكنها أن تحتوي على HTML وكود جافا. ويمكن عمل برنامج جافا ويب يحتوي على تقنيتي JSP و Servlet في آن واحد.

نُضيف ملف جديد في نفس المشروع firstweb، لكن هذه المرة في web pages بدلاً عن Source Package والتي نضيف فيها الـ Servlet.

نضغط بالزر اليمين على Web Pages ثم نختار New/JSP ثم نُسميه first كالتالي:



وهذا هو موقع الملف الجديد first.jsp في المشروع:



ومحتويات الملف first.jsp الإبتدائية هي:

```
<%--
    Document   : first
    Created on : May 27, 2016, 12:10:27 PM
    Author     : motaz
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

عند تشغيله في المتصفح، نضيف first.jsp في عنوان المتصفح بعد إسم البرنامج ليظهر كالتالي:

Hello World!

وهي عبارة عن صفحة ثابتة، أي فقط HTML، و لإضافة كود جافا لها نكتب الكود بين علامتي

```
<%   %>
```

كالتالي:

```
<%--
    Document   : first
    Created on : May 27, 2016, 12:10:27 PM
    Author     : motaz
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <%
            int x = 10;
            out.println("X = " + x);
         %>
    </body>
</html>
```

تُستخدم تقنية الـ JSP في حال أن كود الـ HTML اكثر من كود جافا، و تُستخدم تقنية Servlet في حال أن كود جافا أكثر من الـ HTML. او بمعني آخر يُستخدم الـ Servlet في حال أن كود جافا أكثر كود العرض presentation.

تتميز ملفات JSP على أنها توضع في المخدم كما هي source code ويمكن تعديلها بعد تثبيتها في المخدم – مع أن هذا لا يُنصح به ولا يتوافق مع استخدام الـ source control – ، أي يمكن الوصول إليها داخل دليل tomcat/webapps فنجدها مكتوبة كمصدر برنامج يمكن قراءته وتعديله، بخلاف الـ Servlet والتي نجدها في المخدم في شكل ملفات class byte code لا يمكن قراءتها وتعديلها.

في الجانب العملي تُستخدم التقنيتين في معظم برامج الويب المكتوبة بلغة جافا، حيث يُستقبل المستخدم بواسطة صفحة JSP الذي يمثل واجهة المستخدم الموجود فيها الـ HTML و الـ Java script والـ CCS وكود الجافا ، وعندما نملأ الفورم مثلاً تُستقبل هذه المدخلات و تعالج بواسطة Servlet.

لتوضيح ذلك نُضيف هذا الكود في الملف first.jsp

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <%
            int x = 10;
            out.println("X = " + x);
         %>
        <form action="Timer">
            Please enter your name
            <input type="text" name="username" />

            <br/>
            <input type="submit" />

        </form>
    </body>
</html>
```

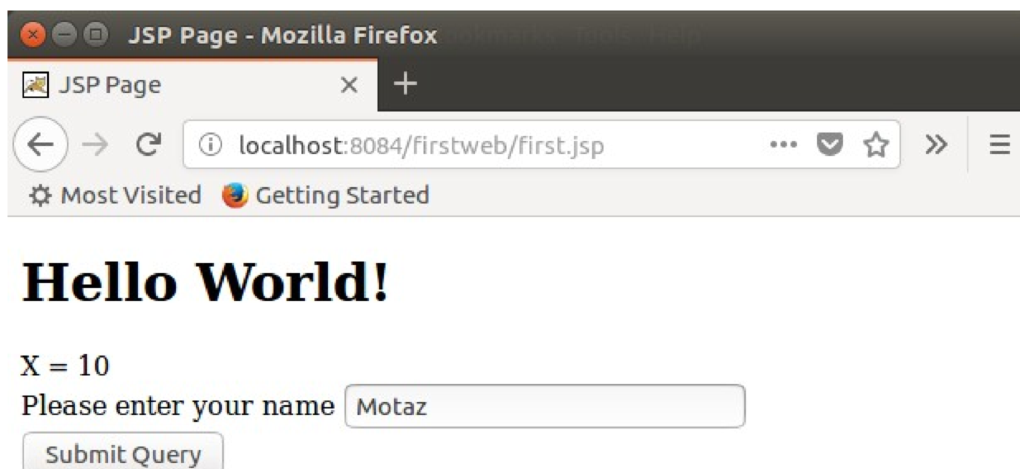
ثم نُضيف العبارة التالية للـ Servlet Timer كالتالي:

```
    out.println("<br>Hello <font color=blue>" +
      request.getParameter("username") + "</font>");
```

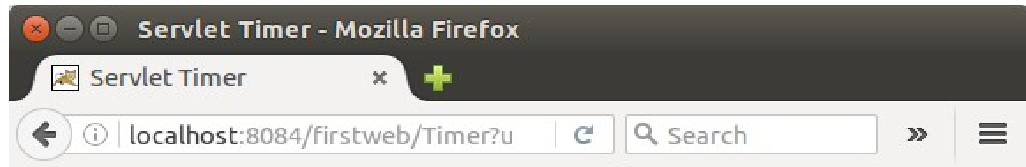
ليصبح كود الدالة processRequest في الـ Servlet هو:

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse
                                                                          response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Timer</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet Timer at " + request.getContextPath() +
                 "</h1>");
            Date today = new Date();
            out.println("Time in server is: <b>" + today.toString() + "</b>");
            out.println("<br>Your name is: <font color=green>" +
                   request.getParameter("name") + "</font>");
            out.println("</body>");
            out.println("</html>");
        }
    }
```

ثم عند عرض الـ JSP تظهر شاشة الإدخال، وعند كتابة الإسم وضغط زر Submit Query يُنادى الـ Servlet كالتالي:



فنكتب الإسم ثم نضغط على الزر Submit Query لترسل محتويات الفورم إلى الـ Servlet المسمى Timer ليظهر بدوره الصفحة التالية:



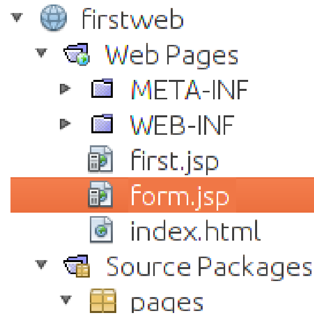
Servlet Timer at /firstweb

Time in server is: **Fri May 27 12:29:13 EAT 2016**
Hello Motaz

تضمين ملف jsp داخل Servlet

يمكن تضمين ملف jsp يحتوي على HTML أو حتى كود جافا داخل Servlet، وعندما يُنادي المستخدم رابط الـ Servlet يُشغّل الأخير كود الـ jsp، وذلك للإستفادة من سهولة كتابة الـ HTML في ملفات jsp.

كمثال نُضيف ملف جديد نسميه form.jsp في نفس البرنامج السابق firstweb ضمن الـ Web pages كالتالي:



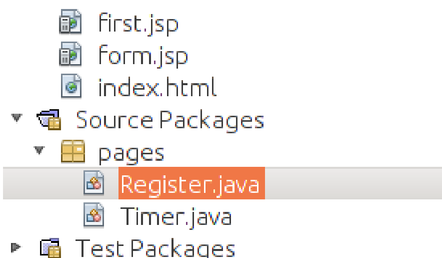
ثم نحذف كود الـ HTML تحت هذا السطر:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

ثم نكتب كود HTML التالي لإظهار فورم لتسجيل معلومات مستخدم:

```
<%--
    Document   : form
    Created on : Sep 27, 2018, 11:33:22 AM
    Author     : motaz
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<h2>Registration Page</h2>
<form>
    Your Name <input type="text" name="yourname" /><br/>

    Your Phone <input type="text" name="phone" /><br/>

    Your Address <input type="text" name="address" /><br/>

    <input type="submit" name="register" value="Register" />
</form>
```

ثم بعد ذلك نُضيف Servlet جديد ضمن الحزمة Pages نسميه Register كما في الشكل التالي:



ثم نحذف كود الـ HTML التلقائي ضمن الدالة processRequest ونستبدلها بالكود التالي:

```
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            request.getRequestDispatcher("form.jsp").include(request, response);

            if (request.getParameter("register") != null){
                out.println("Hello <b>" + request.getParameter("yourname") +
                   "</b>");
            }
        }
    }
```

نلاحظ أن السطر:

```
request.getRequestDispatcher("form.jsp").include(request, response);
```

يُضمّن محتويات الملف form.jsp في هذا الموضع من الـ Servlet فإذا كانت المحتويات HTML فقط نُضمّنها كاملة، وإذا كان بها كود جافا يُنفذه ويرسل له معلومات الـ request ثم يستقبل النتيجة لتضمينها داخل الـ Servlet

وعند عرض صفحة التسجيل والضغط على زر Register تُرسل محتويات الحقول إلى الـ Servlet، بعد ذلك نختبر أنه ضُغط على الزر register بواسطة الكود التالي:

```
            if (request.getParameter("register") != null){
                out.println("Hello <b>" + request.getParameter("yourname") +
                   "</b>");
            }
```

وبعد التأكد نكتب عبارة Hello متبوعة بإسم المستخدم الذي أدخلناه في المربع yourname

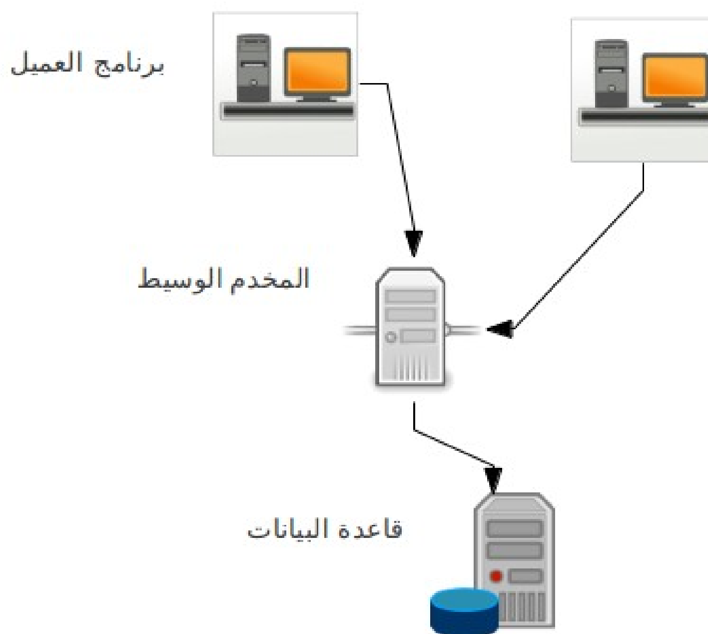
بهذه الطريقة نكون قد دمجنا كود Servlet والتي هي مناسبة لكتابة كود جافا، مع صفحات jsp والتي هي مناسبة مع صفحات ومحتويات الـ HTML.

خدمات الويب Web services

خدمات الويب هي عبارة عن برامج مشابهة لبرامج الويب إلا أن الفرق الأساسي هو واجهة الإستخدام: حيث أن **برامج الويب** واجهة إستخدامها هو المتصفح والذي يتعامل معه المستخدم النهائي، وتستخدم نسق HTML لعرض المعلومات، كذلك تستخدم تقنية التنسيق المعروفة بـ CSS الخاصة بشكل الخطوط وحجمها والألوان، وغيرها، كذلك تُستخدم لغة جافا سكريبت لتشغيل كود معين في المتصفح.

أما **خدمة الويب** فتُستخدم بواسطة برنامج آخر هو عميل لخدمة الويب web service client. مثلاً يكتُب المبرمجون خدمات ويب للتعامل مع حسابات في البنك المركزي للدولة، ويتيحون تلك الخدمات لبقية البنوك للتعامع مع البنك المركزي، ليس كبرنامج ويب وإنما كخدمات ويب أو ما يُعرف بالـ API Application Programming Interface مثلها مثل المكتبات، كأننا نريد تشغيل إجراء أو دالة في مكتبة أو وحدة برمجية أخرى، لكن مكتبات بعيدة وليست في نفس الجهاز كباقي المكتبات، إنما نصل لها عن طريق النت أو شبكة خاصة. يستفيد من تلك الـ API مبرمجو البنكوك الأخرى التي تتعامل مع البنك المركزي، أو أي مؤسسة أخرى لها علاقة بذلك البنك، حيث يُضمّنون نداء تلك الإجراءات في برامجهم الخاصة ببنكهم أو مؤسستهم، فيتسنى لموظفي تلك البنوك والمؤسسات استخدام نفس برامجهم الخاصة ببنكهم مثلاً للوصول لعمليات في البنك المركزي من نفس البرنامج. وهذا يُسمى معمارية الخدمات SOA Service Oriented Architecture وهي تعني توفير خدمات ويب بدلاً من توفير برامج. بهذه الطريقة يستطيع المبرمج الذي يطور برامج للمستخدم النهائي أن يختار اللغة المناسبة له لنداء تلك الخدمات أو تضمينها في أي من برامجه.

وهذا هو مثال لطريقة تصميم نظام به خدمة ويب ويُسمى نظام متعدد الطبقات:



نجد أنه في التصميم

توجد ثلاث طبقات:

- **الطبقة الدنيا** هي طبقة قاعدة البيانات database layer والتي تحتوي على بيانات المؤسسة. وحسب المثال السابق فهي تُمثل قاعدة بيانات البنك المركزي من حسابات وسعر صرف وغيرها من المعلومات.

- **الطبقة الوسطى** middle-ware أو طبقة الـ API هي برامج خدمات الويب و تحتوي على إجراءات تحتاجها البرامج العميلة لتشغيل إجراء معين (مثلاً تحويل بين الحسابات) و هذا الإجراء يُؤثر على قاعدة البيانات التابعة لمؤسسة معينة.

- **الطبقة العليا** وهي الأقرب للمستخدم وهي طبقة برنامج العميل client application أو واجهة المستخدم النهائي وبه نداء لإجراءات الطبقة الوسطى. وفي المثال السابق تُمثل برامج البنوك والمؤسسات الأخرى المستفيدة من الربط مع البنك المركزي

طريقة معمارية تعدد الطبقات لها عدة فوائد منها:

1. عزل قاعدة البيانات ومخدمها عن الأجهزة العميلة، وهذا يُقلل نقاط الإتصال على قاعدة البيانات. فإذا كانت مؤسسة بها مائة عميل مثلاً، فبدلاً من أن تُوصل مائة نقطة إتصال مباشر مع قاعدة البيانات من أجهزة العملاء، تُجمّع تلك الإتصالات في مخدم وسيط واحد أو إثنين وبدورها تتعامل تلك الأجهزة الوسيطة مع قاعدة البيانات.

2. لانحتاج لتثبيت مكتبات للوصول لقاعدة البيانات في أجهزة العملاء، فقط يكفي تثبيت مكتبة التعامل مع قاعدة البيانات في الأجهزة الوسيطة. فإذا تغيرت تلك المكتبة أو حتى إذا غيرنا محرك قاعدة البيانات مثلاً فنعدّل الكود في البرامج الوسيطة فقط.

3. زيادة تأمين وسرية قاعدة البيانات. حيث عزلنا العميل عن قاعدة البيانات. فيُمكن أن تُحصر سماحية الوصول إلى قاعدة البيانات للأجهزة الوسيطة فقط.

4. وسيلة إتصال ومخاطبة برمجية بين المؤسسات المختلفة: فلا يمكن لبنك مثلاً أن يسمح لبنك آخر أو أي مؤسسة أخرى للدخول على قاعدة بياناته لتنفيذ عمليات معينة، إنما تُكتب خدمات ويب محصورة في هذه الخدمات التي يطلبها البنك الآخر و يُعطى صلاحية لندائها، مثلاً إجراء لتحويل بين حسابين.

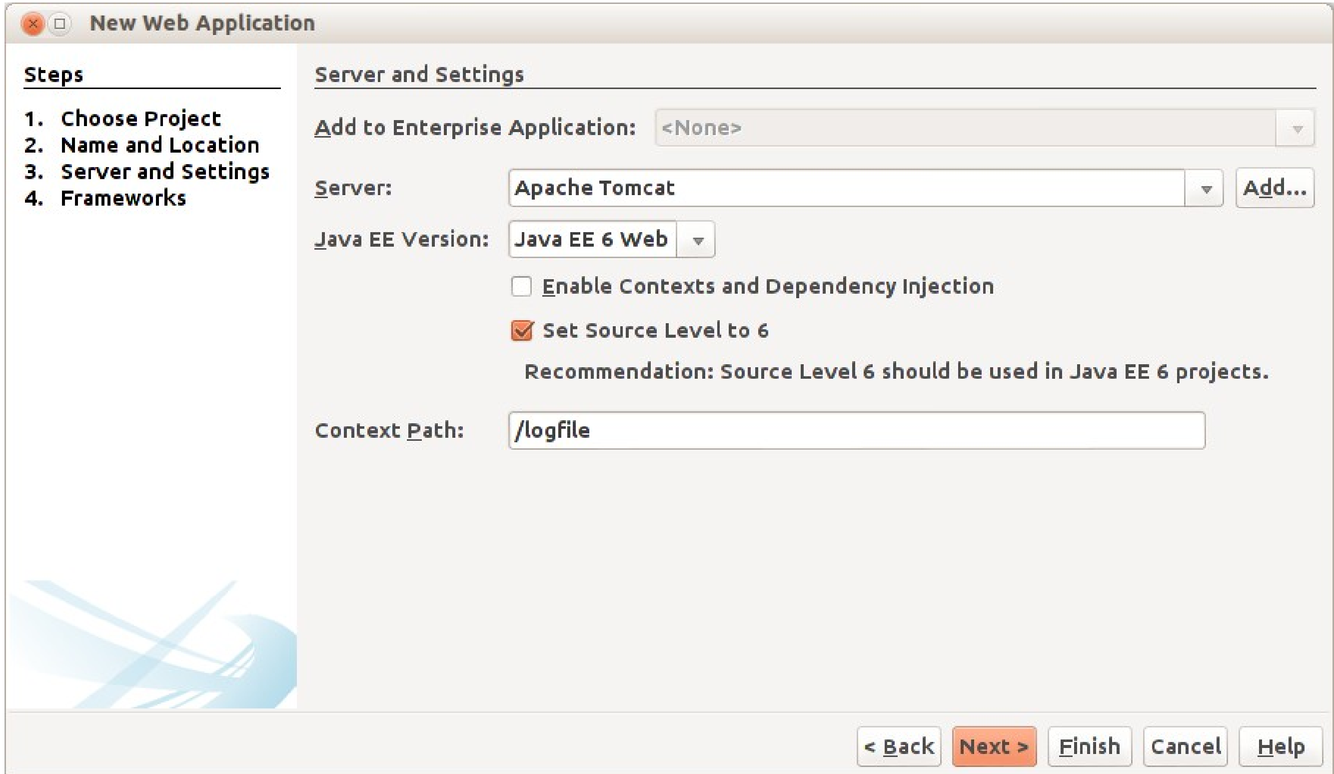
5. وسيلة إتصال بين الأنظمة المختلفة في المؤسسة الواحدة. حيث يُمكن لمؤسسة أن يكون لديها أكثر من نظام من جهات مختلفة، ولتكامل تلك الأنظمة مع بعضها يُمكن أن يوفر كل نظام خدمات ويب تسمح للأنظمة الأخرى الإستفادة منه. فمثلاً إذا كان هُناك نظام لإرسال رسائل نصية فبدلاً من إعطاء البرامج الأخرى صلاحية على قاعدة البيانات لإرسال تلك الرسائل يُفضل أن يكون لديه خدمات ويب لإرسال وإستقبال الرسائل الموجهة إلى البرامج الأخرى.

6. التقليل من التحديثات المستمرة في برامج العملاء. ففي أغلب الأحيان يكون التحديث والإضافات في النظام تحدث في قاعدة البيانات والطبقة الوسيطة ولاتتأثر البرامج الطرفية في أجهزة العملاء بهذا التغيير، فنقلل بذلك تكلفة صيانة ومتابعة النُسخ عند المستخدمين.

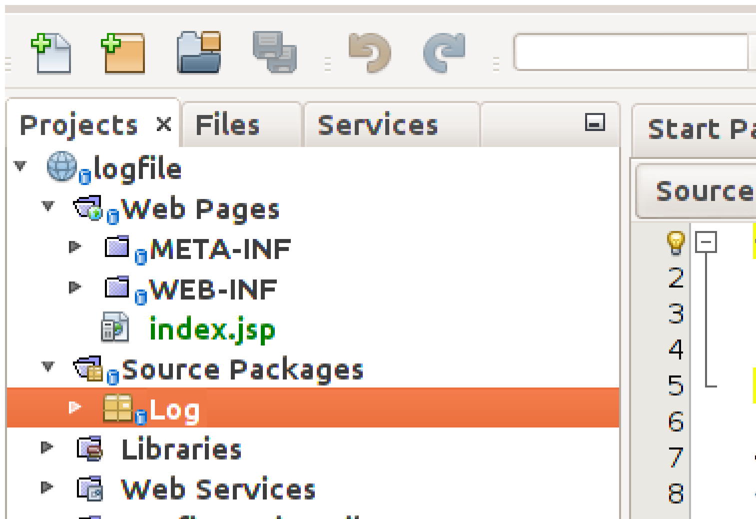
برنامج خدمة ويب للكتابة في ملف

في هذا المثال نريد كتابة خدمة ويب من نوع تقينة الـ SOAP بها إجراء لإستقبال نص وكتابته في ملف نصي، ثم كتابة إجراء آخر لقراءة محتويات الملف النصي الذي كتبنا فيه.

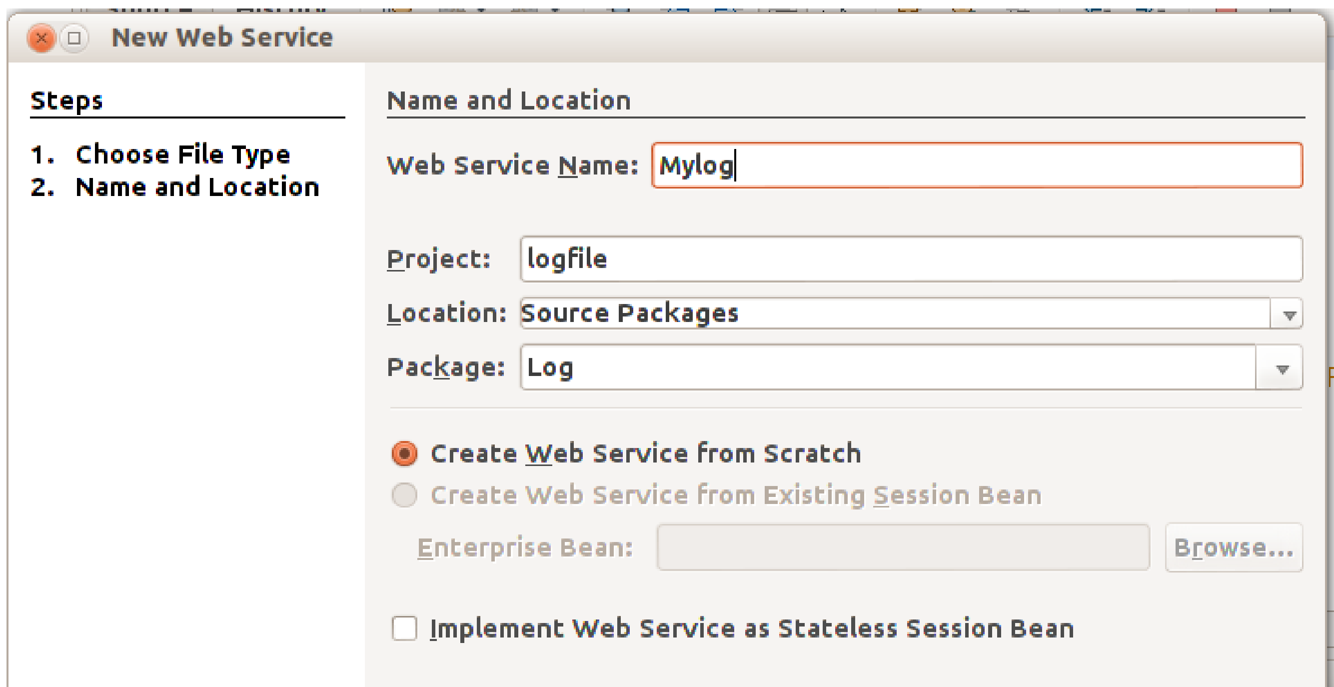
نُنشيء برنامج جديد من نوع Java Web/Web Application ونسميه *logfile* ثم نختار tomcat كمخدم ويب له:



بعد ذلك نُضيف حزمة جديدة نُسميها Log في Source Packages:



وفي الحزمة الجديدة Log نُضيف Web Service نسميها Mylog كما في الصورة التالية:



بعدها نُنبه على أنه سوف تُضاف مكتبة METRO، فنختار الموافقة.

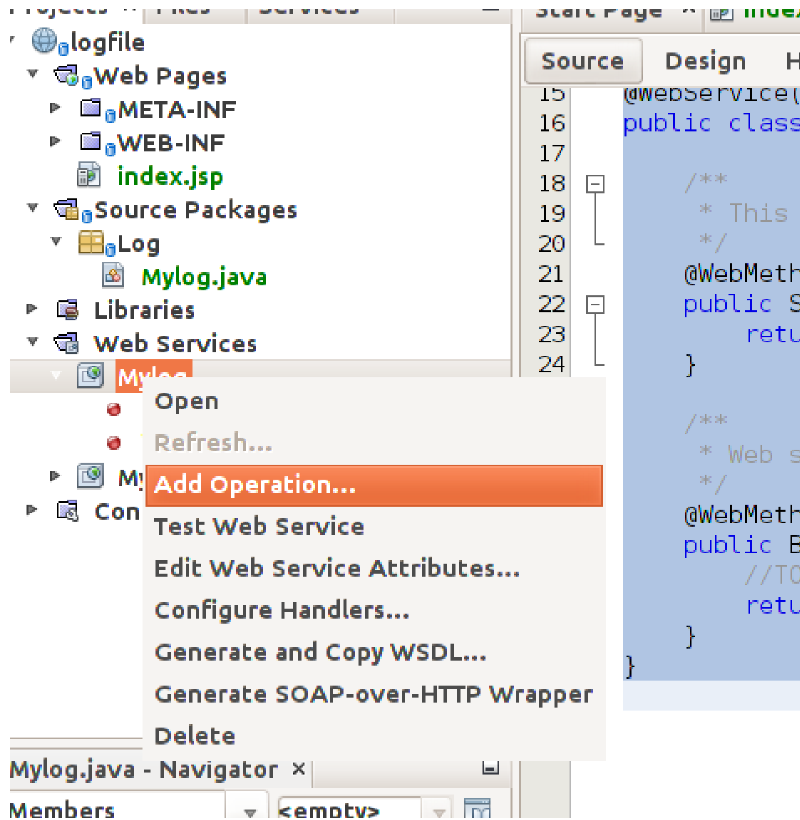
يُضاف الكود التلقائي التالي في الملف Mylog.java:

```
*/
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package Log;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
/**
 *
 * @author motaz
 */
@WebService(serviceName = "Mylog")
public class Mylog {

    /**
     * This is a sample web service operation
     */
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hello " + txt + " !";
    }
}
```

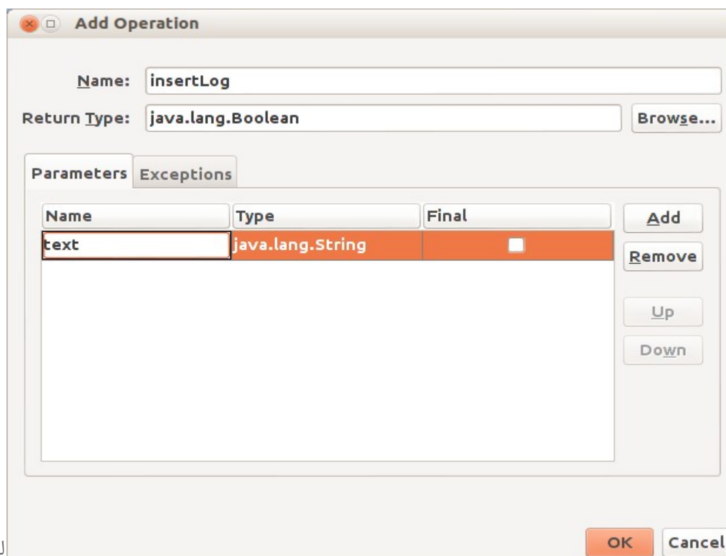
أولاً نُضيف متغير مقطعي اسمه lastError في بداية تعريف خدمة الويب لنضع فيه الأخطاء التي تحدث:

```
@WebService(serviceName = "Mylog")
public class Mylog {

    public String lastError = "";
```

ونجد أنه أضيف فرع جديد في المشروع إسمه Web Services عند فتحها نجد Mylog، فنضيف إجراء جديد فيه بواسطة Add Operation



نُسمّي ذلك الإجراء insertLog. وهو يرجع النوع boolean ويستقبل متغير اسمه text من النوع المقطعي String كما في الصورة التالية:



لكمبيوتر

وإذا رجعنا مرة إخرى للملف Mylog.java نجد أنه أضيف الإجراء insertLog :

```
@WebService(serviceName = "Mylog")
public class Mylog {

    /**
     * This is a sample web service operation
     */
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hello " + txt + " !";
    }

    /**
     * Web service operation
     */
    @WebMethod(operationName = "insertLog")
    public Boolean insertLog(@WebParam(name = "text") String text) {
        //TODO write your implementation code here:
        return null;
    }
}
```

بعدها استلفنا إجراء الكتابة في ملف نصي من مثال سابق وعملنا عليه بعض التعديلات:

```
    private boolean writeToTextFile(String aFileName, String text)
    {
     try{
       FileOutputStream fstream = new FileOutputStream(aFileName, true);

       DataOutputStream textWriter = new DataOutputStream(fstream);

       textWriter.writeBytes(text);
       textWriter.close();
       fstream.close();
       return (true); // success


      }
        catch (Exception e)
      {
        lastError = e.getMessage();
        return (false); // fail
      }

   }
```

وأضفناه في نهاية الملف Mylog.java ليُستدعى من الإجراء insertLog بالطريقة التالية:

```
    @WebMethod(operationName = "insertLog")
```

```
    public Boolean insertLog(@WebParam(name = "text") String text) {

        boolean result = writeToTextFile("/tmp/mylog.txt", text);

        return result;
    }
```
ثم اضفنا إجراء آخر للقراءة أسميناه readLog بواسطة Add Operation كما في المثال السابق لكن بدون أن تكون له مدخلات، فقط مخرجات في شكل مقطع. فتضاف بالشكل التالي:

```
    @WebMethod(operationName = "readLog")
    public String readLog() {
        //TODO write your implementation code here:
        return null;
    }
```
ثم كتبنا إجراء القراءة من ملف نصي لإرجاع الملف كاملاً في متغير مقطعي بدلاً من عرضه على الشاشة:

```
 private String readTextFile(String aFileName)
    {
     try{
       BufferedReader reader = new BufferedReader(new FileReader(aFileName));
       String contents = "";
       String line = reader.readLine();

       while (line != null) {
           contents = contents.concat(line + "\n");
           line = reader.readLine();

       }

       reader.close();

       return (contents);

      }
        catch (Exception e)
      {
        lastError = e.getMessage();
        return (null); // fail
      }

   }
```
نادينا إجراء القراءة من الملف النصي في الإجراء readLog بالشكل التالي:

```
    @WebMethod(operationName = "readLog")
    public String readLog() {

        String filetext = readTextFile("/tmp/mylog.txt");
        return filetext;
    }
```
وفي النهاية كتبنا إجراء لإرجاع آخر خطأ حدث وأسميناه getLastError:

```
    @WebMethod(operationName = "getLastError")
```

```java
    public String getLastError() {
        //TODO write your implementation code here:
        return lastError;
    }
```

حيث يستخدمه العميل لمعرفة الخطأ الذي حدث في خدمة الويب أثناء ندائها.

نلاحظ أنه لابد أن نستخدم دليل به صلاحية للمستخدم tomcat6 أو tomcat7 والذي يُستخدم مع نظام الشغيل عند التعامل مع خدمات الويب. وفي هذا المثال السابق استخدمنا الدليل tmp/ بإعتبار أن به صلاحية لكافة المستخدمين في بيئة لينكس.

في الواقع العملي تكون إجراءات خدمة الويب مرتبطة بتنفيذ إجراءات في قواعد بيانات مثلاً إدخال قيد محاسبي، إدراج معاملة بنكية، دفع فاتورة هاتف. كذلك يُمكن أن تُنادي خدمات الويب خدمات ويب أخرى، فيصبح المعمارية ذات أربع طبقات: عميل —> خدمة ويب —> خدمة ويب أخرى —> قاعدة بيانات.

بعد ذلك يُمكن تشغيل البرنامج و يُفتح متصفح الويب تلقائياً لتظهر الشاشة التالية:

بعد نهاية عنوان الويب نُضيف إسم خدمة الويب Mylog ليُصبح العنوان هو:

http://localhost:8080/logfile/Mylog

فيظهر لنا معلومات خدمة الويب Mylog:

Web Services - Mozilla Firefox

Web Services

localhost:8080/logfile/Mylog

Web Services

Endpoint		Information	
Service Name:	{http://Log/}Mylog	Address:	http://localhost:8080/logfile/Mylog
Port Name:	{http://Log/}MylogPort	WSDL:	http://localhost:8080/logfile/Mylog?wsdl
		Implementation class:	Log.Mylog

وعند الضغط على عنوان الـ WSDL يظهر لنا ملف XML وهو وصف لخدمة الويب ويُستخدم عند عمل البرامج العميلة لخدمات الويب:

Mozilla Firefox

http://localhos...ile/Mylog?wsdl

localhost:8080/logfile/Mylog?wsdl

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
-<!--
   Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2-hudson-740-.
 -->
-<!--
   Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2-hudson-740-.
 -->
-<definitions targetNamespace="http://Log/" name="Mylog">
  -<types>
    -<xsd:schema>
       <xsd:import namespace="http://Log/" schemaLocation="http://localhost:8080/logfile
       /Mylog?xsd=1"/>
     </xsd:schema>
   </types>
  -<message name="insertLog">
     <part name="parameters" element="tns:insertLog"/>
   </message>
  -<message name="insertLogResponse">
     <part name="parameters" element="tns:insertLogResponse"/>
```

بذا نكون قد إنتهينا من كتابة وتشغيل خدمة الويب في مخدم Tomcat.

وهذا هو رابط الـ WSDL:

http://localhost:8080/logfile/Mylog?wsdl

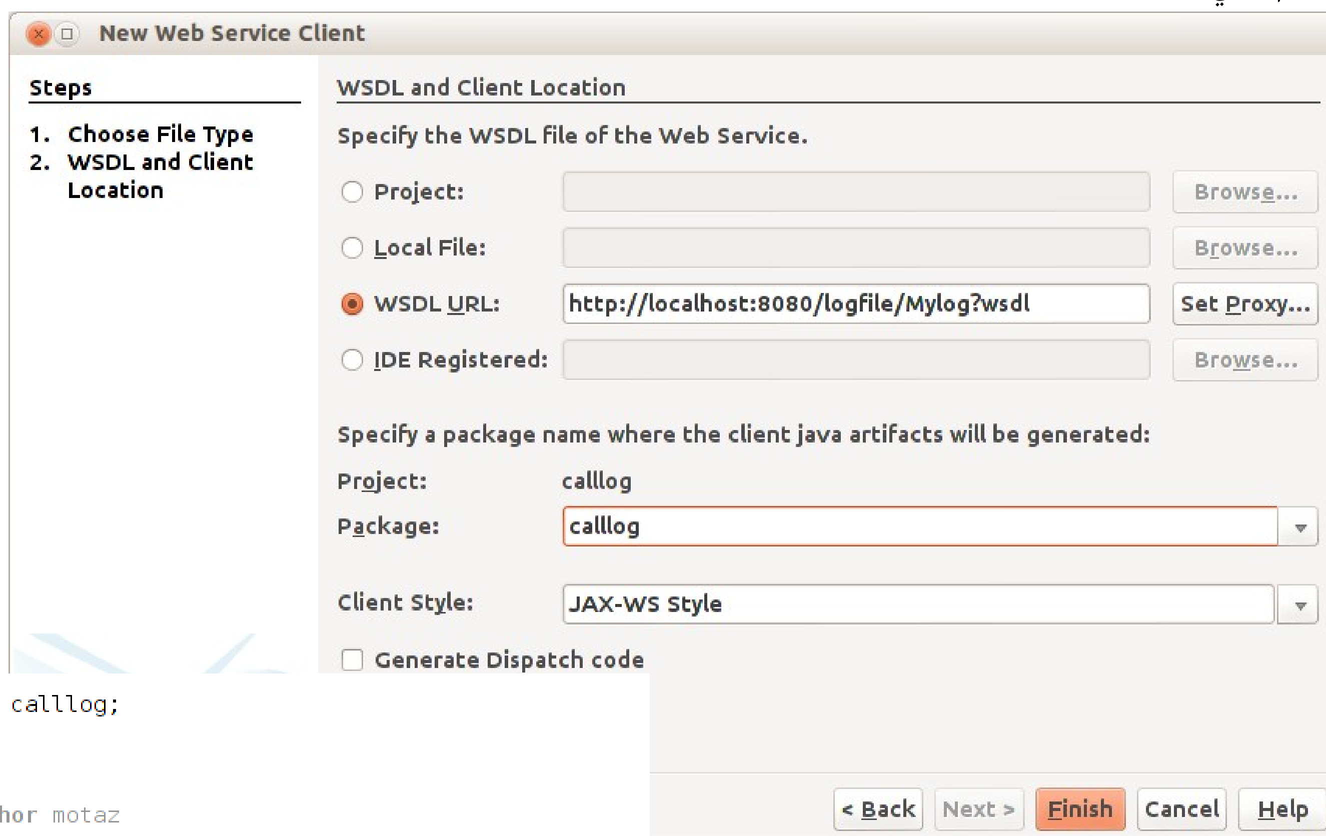
وهذا هو الشيء الوحيد الذي يحتاجه المبرمج لكتابة برنامج عميل لإستخدام خدمة الويب. ويمكن أن نستخدم أي لغة برمجة تدعم تقنية الـ SOAP لنداء الدالتين insertLog و readLog.

برنامج عميل خدمة ويب Web service client

يُمكن أن يكون إجراء نداء خدمة الويب في أن نوع من البرامج، مثلاً يُمكن أن يكون في برنامج سطح مكتب Desktop application أو برنامج ويب أو حتى برنامج سطر الأوامر كما في مثالنا التالي.

نُنشيء برنامج جديد من نوع Java/Java application نُسميه calllog.

بعدها نجد أن هُناك حزمة اسمها calllog في البرنامج. نُضيف عميل خدمة ويب بواسطة new Web service client فيظهر لنا الفورم التالي:



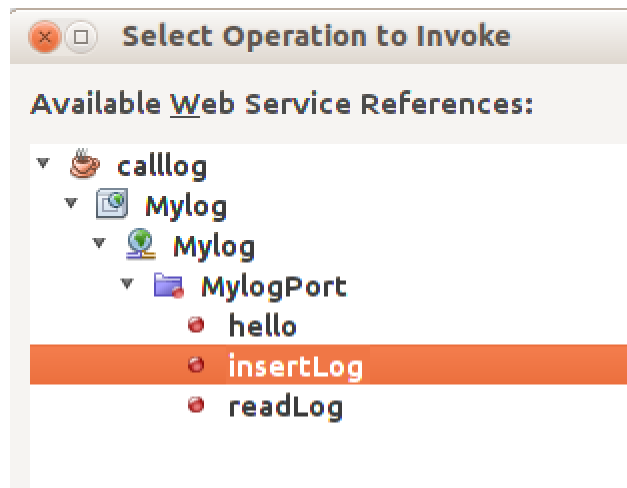
حيث نختار WSDL URL نضع فيه عنوان الـ WSDL لخدمة الويب. ثم نختار الحزمة calllog في Package ثم نضغط الزر Finish

```
package calllog;

/**
 *
 * @author motaz
 */
public class Calllog {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

    }
}
```

Generate
- Constructor...
- Logger...
- toString()...
- Override Method...
- Add Property...
- Call Web Service Operation...
- Generate REST Client...

نرجع لملف الكود الرئيسي calllog.java فنضغط الزر اليمين للماوس داخل الإجراء main ونختار Insert Code ثم Call Web Service Operation كما في الصورة التالية:

ثم إختيار insertLog:



فيُضاف إجراء جديد لنداء خدمة الويب بالشكل التالي:

```
    private static Boolean insertLog(java.lang.String text) {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        return port.insertLog(text);
    }
```

ونكرر نفس العملية السابقة لإضافة نداء الإجراء readLog:

```
    private static String readLog() {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        return port.readLog();
    }
```

ثم عدلناها بإضافة إظهار الخطأ الذي يحدث في خدمة الويب:

```
    private static Boolean insertLog(java.lang.String text) {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        boolean res = port.insertLog(text);
```

```
        if (!res) {
                System.out.println("Error: " + port.getLastError());
        {
        return(res);
    }

    private static String readLog() {
        calllog.Mylog_Service service = new calllog.Mylog_Service();
        calllog.Mylog port = service.getMylogPort();
        String result = port.readLog();
        if (result == null) {
            System.out.println("Error: " + port.getLastError());
        }
        return (result);
    }
```

ثم استدعينا الإجرائين في الدالة الرئيسة للبرنامج:

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

        Date today = new Date();
        insertLog(today.toString() + ": Sample text\n");
        String result = readLog();
        System.out.print(result);

    }
```

عند تشغيل البرنامج نتحصل على المخرجات التالية:

```
Fri Mar 29 12:34:48 EAT 2013: Sample text
Fri Mar 29 12:34:53 EAT 2013: Sample text
```

عند تنفيذ أي من الإجراءات في الجهاز العميل والتي تُنادي خدمة ويب، فإن هذا التنفيذ يحدث في المخدم.
وفي الواقع تكون خدمة الويب في جهاز منفصل والبرنامج العميل يكون متصلاً به عبر شبكة محلية أو شبكة الإنترنت، وكل تعقيدات الإتصالات بقواعد البيانات يكون في جهة خدمة الويب، ويكون برنامج العميل مبسطاً بقدر الإمكان لتحقيق فوائد معمارية تعدد الطبقات.

القراءة من مخدم ويب بواسطة HTTP

من اسهل الطرق للإتصال أو الحصول على معلومة من برنامج أو مخدم في النت أو مخدم داخلي هو استخدام بروتوكول الـ HTTP.
وكما فعلنا سابقاً باستخدام بروتوكل الـ SOAP فإن وسيلة الإتصال بين العميل والمخدم هو بروتوكول الـ HTTP والذي يعمل فوق بروتوكول الـ TCP/IP. يمكن استخدام الـ HTTP في لغة جافا للقراءة من رابط معين عن طريق GET أو ارسال بيانات إلى المُخدم باستخدام POST.

استخدمنا الفئة URL والتي تُجهّز العنوان للإتصال ولكنها لا تُنفّذ الإتصال الفعلي:

```
        URL url = new URL(myURL);
```

ثم تمرير الكائن url كمُدخل لكائن من فئة URLConnection والتي تتولى الإتصال الفعلي وتُرسل و تستقبل البيانات:

```
        URLConnection myURLConnection = url.openConnection();
        myURLConnection.connect();
```

ثم تعريف كائن للقراءة من نوع *InputStreamReader* وذلك بتهيئته من الكائن السابق للإتصال:

```
        InputStreamReader reader;
        reader = new InputStreamReader(myURLConnection.getInputStream());
```

ثم قراءة المحتويات من الإتصال، وتخزينه في المتغير المقطعي outputResult. وهذا هو كود الإجراء كاملاً:

```
    public static String callURL(String myURL) {
        try {

            URL url = new URL(myURL);
            URLConnection myURLConnection = url.openConnection();
            myURLConnection.connect();

            InputStreamReader reader;
            reader = new InputStreamReader(myURLConnection.getInputStream());

            String outputResult= "";
            char buf[] = new char[1024];
            int len;
            while ((len = reader.read(buf)) != -1){
                String data = new String(buf, 0, len);
                    outputResult = outputResult + data;
            }

            reader.close();

            return(outputResult);

    } catch (Exception ex) {
        return("error: " + ex.toString());
    }
    }
```

ويمكن نداء هذا الإجراء لقراءة محتويات صفحة من الدالة الرئيسة main كالتالي:

```
   public static void main(String[] args) {
        String content = callURL("http://localhost");
        System.out.println(content);
   }
```

يمكن استخدام هذه الطريقة للربط بين برنامجين بطريقة أكثر بساطة من بروتوكول الـ SOAP حيث يمكن أن يكون المخدم هو برنامج ويب يحتوي على Servlet يستقبل معلومات عن طريق GET ويمكن ندائه كالتالي:

```
        String content = 
callURL("http://localhost:8080/JavaWebApp/GetCustomerInfo?customerid=1");
        System.out.println(content);
```

وفي جانب الـ Servlet نقرأ المدخلات كالتالي:

```
            String id = request.getParameter("customerid");
```

ثم يُنفّذ كود معين، مثلاً قراءة معلومات من قاعدة بيانات ثم إرجاعها في شكل JSON أو حتى XML.
وهذه الطريقة يمكن استخدامها لتبادل البيانات بين برنامج أندرويد في الهاتف مع مخدم في الانترنت. ويمكن تحويل تلك الطريقة لتصبح بروتوكول *parameters* بدلاً من إرسالها في شكل مدخلات JSON في شكل POST قليلاً (ارسال المعلومات باستخدام RESTFull أو ما يُسمى بالـ REST.

خدمات ويب الـ RESTFull

كما ذكرنا في المثال السابق لقراءة معلومة من برنامج web باستخدام الـ URL، وقلنا أنها طريقة بسيطة لعمل خدمة ويب. وخدمات الويب مشابهة تماماً في البروتوكول والتقنيات مع برامج الويب، حيث أن التقنية المستخدمة في برامج الويب يُمكن استخدامها في برامج الويب. أما الإختلافات فتكمن في الآتي:

1. المستخدم المباشر لبرامج الويب هو شخص يستخدم المتصفح لاستعراض والتفاعل مع البرنامج، أما من يستخدم خدمة الويب فهو برنامج آخر وليس إنسان.
2. الرد الذي يرجع من برنامج الويب يكون في شكل HTML حتى يستطيع المتصفح فهمه وعرضه بالشكل المطلوب للمستخدم النهائي، أما خدمات الويب فيكون الرد الذي ينتج عن نداء خدمة ويب أو إجراء فيها هو هيكل بيانات متفق عليه بين خدمة الويب والبرنامج المستفيد منها، مثل أن تكون في شكل XML أو JSON أو بيانات في شكل خصائص، حيث أن طريقة إظهار المعلومة وشكل الفورم النهائي ليس له علاقة بخدمة الويب، وإنما هو مسؤولية البرنامج العميل، و يمكن أن يكون عبارة عن برنامج ذو واجهة رسومية، مثل برنامج الـ Swing.

أما وجه الشبه بينها فيتمثل في:

1. كلاهما يستخدم بروتوكول الـ HTTP لتبادل البيانات بين المخدم والعميل
2. يمكن ان يشتركا في نفس تقنية الويب، مثلاً في لغة جافا يُمكننا استخدام JSP أو Servlet، لكن بالنسبة لخدمات الويب فبما أنها لا تستخدم HTML فالأفضل إذاً استخدام تقنية الـ Servlet.
3. كلاهما يُستضاف في نفس مخدم الويب مثل Tomcat وهذا مرتبط بالتقنية المستخدمة حيث أنها تتطلب هذا النوع من مخدمات الويب والذي يُسمى أحياناً حاوية البرامج container.

لعمل خدمة ويب من نوع الـ RESTFull يمكننا إضافة Servlet جديد في برنامج ويب جديد أو في برامج قديم، حيث يمكن لبرنامج ويب واحد أن يحتوي على خدمات ويب من نوع SOAP و RESTFull وحتى صفحات ويب، لكن الأفضل أن يكون هُناك برنامج ويب منفصل لكل تقنية لأن كل واحدة يمكن أن تتطلب مكتبات مختلفة، كذلك فإن الأفضل دائماً عزل أنواع البرامج لتكون متخصصة في خدمة ما حتى تسهل صيانتها، وتشغيلها، وتطويرها.
أنشأنا برنامج ويب اسميناه RESTServer، ثم اضفنا خدمة ويب بتقنية Servlet اسميناها GetServerInfo.

ومهمتها هي إرجاع بعض المعلومات عن المخدم، مثل نوع نظام التشغيل و تاريخ المخدم ونُسخة جافا، وذلك في شكل بيانات من نوع الخصائص *Properties* بالشكل التالي:

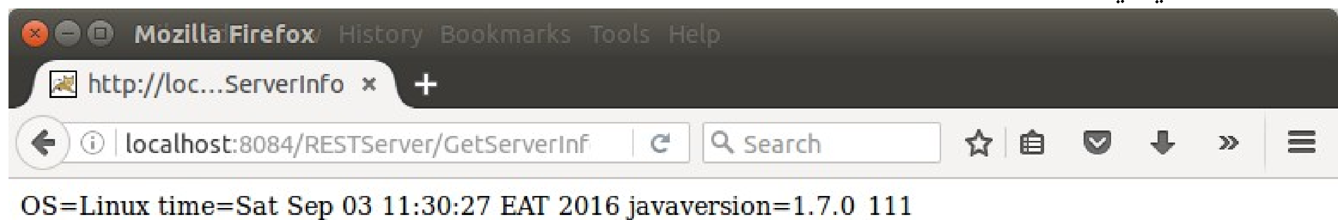
```
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
```

```
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            String os = System.getProperty("os.name");
            Date now = new Date();
            String java = System.getProperty("java.version");
            out.println("OS=" + os + "\n");
            out.println("time=" + now.toString() + "\n");
            out.println("javaversion="  + java);
        }
    }
```

بعد ذلك نُنفذ البرنامج لعرضه في المتصفح وإضافة إسم خدمة الويب *GetServerInfo* إلى العنوان ليصبح كالتالي:

```
http://localhost:8084/RESTServer/GetServerInfo
```

فيكون الرد كالتالي في المتصفح:



نلاحظ أن البيانات ظهرت في سطر واحد، حيث أننا لم نستخدم نسق الـ HTML حتى تظهر بصورة جيدة، وكما ذكرنا فإن المقصود ليس المتصفح وإنما برنامج آخر يُكتب بواسطة المبرمج، لكن يمكن استخدام المتصفح أثناء تطوير البرنامج لأجل عرض البرامج.

يمكن عرضه بطريقة أفضل بعرض مصدر الصفحة من المتصفح

وذلك بالضغط بالزر اليمين على الصفحة ثم اختيار View page source ليظهر لنا النص كما جاء من المخدم بدون تغيير:

http://localhost:8084/RESTServer/GetServerInfo - Mozilla Firefox

```
OS=Linux
time=Sat Sep 03 11:36:09 EAT 2016
javaversion=1.7.0_111
```

كذلك يمكننا استخدام بعض الأدوات البسيطة التي تعرض صفحة ويب في الطرفية مثل برنامج *curl* في نظام لينكس:

```
motaz@L40-laptop:~$ curl http://localhost:8084/RESTServer/GetServerInfo
OS=Linux
time=Sat Sep 03 11:37:06 EAT 2016
javaversion=1.7.0_111
motaz@L40-laptop:~$
```

كذلك يمكننا عمل برنامج يقرأ الـ URL كما كتبنا سابقاً، يمكننا نداء الإجراء التي كتبناه سابقاً *callURL*.

أنشأنا برنامج Java Application جديد اسميناه *CallRest* وأضفنا إليه الإجراء *callURL* ثم استدعيناه في الإجراء main كالتالي:

```
        String content = callURL("http://localhost:8084/RESTServer/GetServerInfo");
        System.out.println(content);
```

فكانت النتيجة:

```
OS=Linux
time=Sat Sep 03 11:42:41 EAT 2016
javaversion=1.7.0_111
```

لكن فإن البرنامج العميل لا يريد مجرد إظهار المعلومات وإنما معالجتها مثلاً وقراءة كل قيمة بمفردها، لذلك نستخدم كائن الخصائص لاستخلاص تلك القيم التي رجعت من خدمة الويب:

```
    public static void main(String[] args) throws IOException {
        // TODO code application logic here
```

```
        String content;
        content =  callURL("http://localhost:8084/RESTServer/GetServerInfo");

        // Load output into stream
        ByteArrayInputStream in = new ByteArrayInputStream(content.getBytes());

        // Parse output
        Properties properties = new Properties();
        properties.load(in);
        String serverOS = properties.getProperty("OS");
        String serverTime = properties.getProperty("time");
        String javaVersion = properties.getProperty("javaversion");

        // Display output
        System.out.println("Server OS is : " + serverOS);
        System.out.println("Server Time is : " + serverTime);
        System.out.println("Server Java version is : " + javaVersion);
    }
```

استخدمنا الفئة *ByteArrayInputStream* لقراءة محتويات المقطع الذي يحتوي على الخصائص لتحويلها إلى سلسلة بيانات stream وذلك لأن كائن الخصائص يستطيع القراءة من سلسلة بيانات ولا يستطيع القراءة من مقطع مباشرة، فكان هذا بمثابة تحويل لآلية نقل البيانات:

```
        ByteArrayInputStream in = new ByteArrayInputStream(content.getBytes());
        Properties properties = new Properties();
        properties.load(in);
```

في هذا المثال ركزنا على إرجاع بيانات من المخدم إلى العميل، لكن نريد إرسال بيانات من العميل إلى المخدم. عدلنا خدمة الويب لقراءة مدخلات من نوع الخصائص، وكمثال طلبنا من البرنامج العميل إدخال اسم الزبون وعنوانه:

استخدمنا كائن من فئة الخصائص في المخدم لقراءة المعلومات المُرسلة بواسطة البرنامج العميل، وذلك بإضافة هذا الكود:

```
            // read input
            Properties properties = new Properties();
            properties.load(request.getInputStream());
            String customerName = properties.getProperty("name");
            String customerAddress = properties.getProperty("address");
```

هذه المرة لم نحتاج للفئة *ByteArrayInputStream* لأن البيانات المُرسلة توجد في شكل سلسلة بيانات ويمكن استخلاص كائن سلسلة البيانات بهذه الطريقة :

```
request.getInputStream()
```

كذلك أرجعنا نفس البيانات المُرسلة، ليصبح كود خدمة الويب كالتالي:

```
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
```

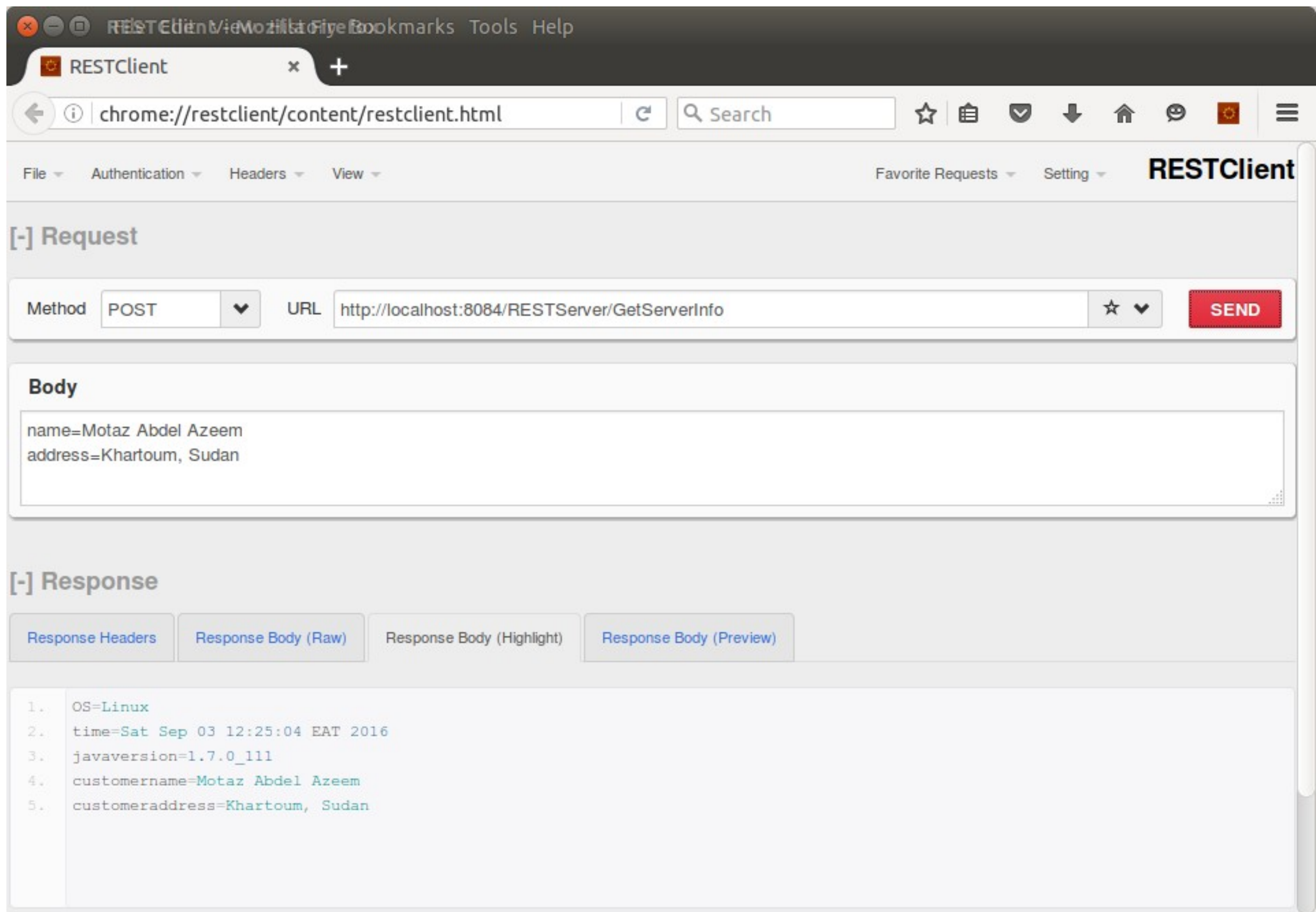
```
        // read input
        Properties properties = new Properties();
        properties.load(request.getInputStream());

        String customerName = properties.getProperty("name");
        String customerAddress = properties.getProperty("address");


        String os = System.getProperty("os.name");
        Date now = new Date();
        String java = System.getProperty("java.version");
        out.println("OS=" + os);
        out.println("time=" + now.toString());
        out.println("javaversion="  + java);
        out.println("customername=" + customerName);
        out.println("customeraddress=" + customerAddress);
    }
}
```

هذه المرّة لا يمكننا نداء خدمة الويب هذه عن طريق المتصفح العادي، لأن إرسال البيانات الذي استخدمناه يستخدم طريقة POST والتي تُرسل البيانات فيها كمحتويات مضمنة بمعزل عن العنوان URL.

لذلك لتجربة خدمة الويب هذه نحتاج لتثبيت أداة إضافية في المتصفح، مثلاً في متصفح Firefox نُضيف الأداة *RESTClient* وذلك عن طريق:

Add-ons/Extensions

ثم نبحث عن *RESTClient* ثم نضيفها إلى المتصفح، ثم نعيد تشغيله، فتظهر في اعلى المتصفح، بعد ذلك نكتُب عنوان خدمة الويب ثم نكتب المُدخلات في شكل خصائص ثم نُحوّل طريقة إرسال البيانات (Method) إلى POST بالشكال التالي:



وكما ذكرنا سابقاً فإن خدمة الويب لا تُستخدم مع المتصفح مباشرةً، لكن هذه الأداة تُستخدم أثناء تطوير البرنامج بواسطة المبرمج أو من يختبر خدمة الويب، أما المستخدم النهائي والمستفيد من النظام فلا يستخدم هذه الأدوات، إنما يستخدم البرنامج العميل.

بعد ذلك عدلنا البرنامج العميل لإرسال تلك البيانات، وأول تعديل كان للإجراء *callURL* حيث اضفنا له مُدخلات:

```
    public static String callURL(String myURL, String input) {
```

ثم اخبار كائن الإتصال أن هناك output

```
            myURLConnection.setDoOutput(true);
```

ثم عرّفنا كائن للكتابة من نوع OutputStreamWriter لإرسال تلك البيانات إلى كائن الإتصال:

```
            OutputStreamWriter writer;
            writer = new OutputStreamWriter(myURLConnection.getOutputStream());
            writer.write(input);
            writer.flush();
            writer.close();
```

فكان هذا هو الشكل النهائي للإجراء *callURL* والذي يُرسل بيانات في شكل POST ثم يُرجع النتيجة:

```java
    public static String callURL(String myURL, String input) {
        try {

            URL url = new URL(myURL);
            URLConnection myURLConnection = url.openConnection();
            myURLConnection.setDoOutput(true);
            myURLConnection.connect();

            // Write Input
            OutputStreamWriter writer;
            writer = new OutputStreamWriter(myURLConnection.getOutputStream());
            writer.write(input);
            writer.flush();
            writer.close();

            // Read result
            InputStreamReader reader;
            reader = new InputStreamReader(myURLConnection.getInputStream());

            String outputResult= "";
            char buf[] = new char[1024];
            int len;
            while ((len = reader.read(buf)) != -1){
                String data = new String(buf, 0, len);
                    outputResult = outputResult + data;
            }

            reader.close();

            return(outputResult);

    } catch (Exception ex) {
        return("error:  " + ex.toString());
    }
    }
```

ثم استدعيناه في الإجراء main كالتالي:

```java
    String input;
    input = "name=Motaz Abdel Azeem\naddress=Khartoum, Sudan";

    String content;
    content = callURL("http://localhost:8084/RESTServer/GetServerInfo", input);
```

ويمكن استخدام كائن الخصائص بدلاً من كتابة كافة المُدخلات في سطر واحد. وهذا هو كود النداء كاملاً:

```java
    public static void main(String[] args) throws IOException {
        // TODO code application logic here

        String input;
        input = "name=Motaz Abdel Azeem\naddress=Khartoum, Sudan";
```

```
        String content;
        content = callURL("http://localhost:8084/RESTServer/GetServerInfo", input);

        // Load output into stream
        ByteArrayInputStream in = new ByteArrayInputStream(content.getBytes());

        // Prase output
        Properties properties = new Properties();
        properties.load(in);
        String serverOS = properties.getProperty("OS");
        String serverTime = properties.getProperty("time");
        String javaVersion = properties.getProperty("javaversion");
        String customerName = properties.getProperty("customername");
        String customerAddress = properties.getProperty("customeraddress");

        // Display output
        System.out.println("Server OS is : " + serverOS);
        System.out.println("Server Time is : " + serverTime);
        System.out.println("Server Java version is : " + javaVersion);
        System.out.println("Customer name : " + customerName);
        System.out.println("Customer address : " + customerAddress);
    }
```

استخدام نسق JSON

نسق الـ JSON هو بديل لنسق الـ XML، فهو اسهل كتابة وأقل حجماً من الـ XML. وهو مناسب للاستخدام لنقل البيانات في خدمات الويب، وهو يسمح بنقل بيانات أكثر تعقيداً مثل المصفوفات (مقارنة بنوع الخصائص). لكن قبل استخدامه في خدمات الويب، لابد من البحث عن مكتبة تدعم هذا النسق لاستخدامها مع برامج جافا التي تحاج إليه.

المكتبة التي استخدمناها اسمها json-simple وهذا مثال لإسم ملف يمكن الحصول عيه من النت:

```
json-simple-1.1.jar
```

لشرحها أولاً قبل استخدامها في خدمات الويب، أنشأنا برنامج جافا عادي ثم اضفنا هذه المكتبة في جزء library بواسطة Add JAR/Folder ثم كتبنا هذا الكود لإضافة بيانات في شكل JSON:

```
        JSONObject myObject = new JSONObject();
        myObject.put("year", 2016);
        myObject.put("service", "Customer Registration");
        myObject.put("valid", true);
        System.out.println(myObject.toJSONString());
```

في هذا المثال استخدمنا كائن من نوع الفئة *JSONObject* للتعامل مع هذه النوعية من النسق.

وناتج التشغيل هو مقطع في شكل JSON:

```
{"valid":true,"service":"Customer Registration","year":2016}
```

بعد إضافة كافة العناصر باستخدام *put* نتحصل في النهاية على المقطع الذي يحتوي على هذا النسق بواسطة *toJSONSting*.

وهذا الجزء يُستخدم في جانب العميل لإرسال بيانات في شكل JSON، أما المخدم فيُعالج هذه المعلومات و يستخلص القيم بواسطة أسماءها.

أضفنا هذا الكود في نفس البرنامج لمعرفة آلية قراءة مقطع JSON وتحويله إلى متغيرات بسيطة:

```
        String myInput = myObject.toJSONString();
        JSONParser parser = new JSONParser();
        JSONObject received = (JSONObject)parser.parse(myInput);
        System.out.println("year is: " + received.get("year").toString());
        System.out.println("service is: " + received.get("service").toString());
        System.out.println("Is valid: " + received.get("valid").toString());
```

هذه المرة استخدمنا كائن من النوع *JSONParser* لتحويل المقطع إلى كائن JSON حتى نتعامل مع البيانات التي بداخله مباشرة، وفي هذا السطر يُحوّل المقطع إلى JSON:

```
JSONObject received = (JSONObject)parser.parse(myInput);
```

نلاحظ أننا استخدمنا ما يُعرف بالـ casting والتي تُحوّل من نوع بيانات أو كائنات إلى اخرى، في هذا المثال تُحوّل نوع الكائن *object* إلى *JSONObject*.

بعد ذلك حاكينا جزئية المخدم، لقراءة البيانات الموجودة في الكائن *received* و حولناها إلى صيغتها البسيطة المقطعية، أو الرقمية أو المنطقية. التحويل إلى الصيغ البسيطة يمكن بهذه الطريقة بدلاً من قراءتها جميعها كأنها مقاطع:

```
        int year = Integer.parseInt(received.get("year").toString());
        boolean isValid = Boolean.valueOf(received.get("valid").toString());
```

بعد ذلك حولنا البرنامج *RESTServer* ليستقبل بيانات من نوع JSON بدلاً من نوع الخصائص.

أولاً اضفنا إجراء جديد لقراءة المحتويات المُرسلة وتحويلها إلى مقطع واحد، وهو مقطع الـ JSON:

```
    public static String readClient(HttpServletRequest request) throws IOException
{
        BufferedReader reader = request.getReader();
        String line;
        String jsonText = "";
        while ((line = reader.readLine()) != null){
            jsonText = jsonText + line;
        }
        return jsonText;
    }
```

وهذا هو الجزء الذي استقبلنا فيه ثم عالجنا المُدخلات ثم حولناها إلى بيانات بسيطة:

```
                String requestStr = readClient(request);

                JSONParser parser = new JSONParser();
                JSONObject obj = (JSONObject) parser.parse(requestStr);
                String customerName = obj.get("name").toString();
                String customerAddress = obj.get("address").toString();
```

ثم أرجعنا الرد في شكل JSON كالتالي:

```
                JSONObject result = new JSONObject();
                result.put("success", true);
                result.put("OS", os);
                result.put("time", now.toString());
                result.put("javaversion", java);
                result.put("customername", customerName);
                result.put("customeraddress", customerAddress);
                out.println(result.toJSONString());
```

والكود الكامل لخدمة الويب GetServiceInfo هو التالي:

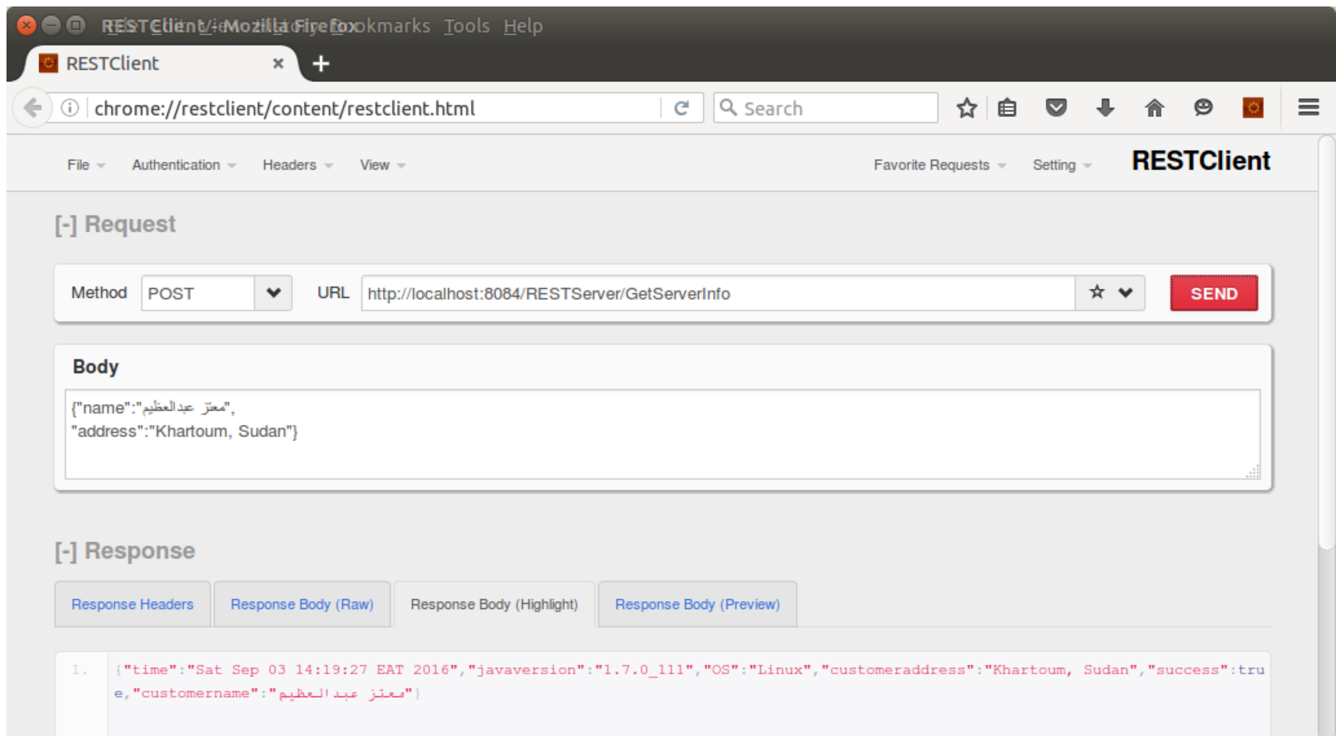
```
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        response.setCharacterEncoding("UTF-8");
        try (PrintWriter out = response.getWriter()) {

            try {
                // read input
                String requestStr = readClient(request);
```

```
                JSONParser parser = new JSONParser();
                JSONObject obj = (JSONObject) parser.parse(requestStr);
                String customerName = obj.get("name").toString();
                String customerAddress = obj.get("address").toString();

                String os = System.getProperty("os.name");
                Date now = new Date();
                String java = System.getProperty("java.version");

                // prepare output
                JSONObject result = new JSONObject();
                result.put("success", true);
                result.put("OS", os);
                result.put("time", now.toString());
                result.put("javaversion", java);
                result.put("customername", customerName);
                result.put("customeraddress", customerAddress);

                // Write output
                out.println(result.toJSONString());
            }
            catch (Exception ex){
                JSONObject obj = new JSONObject();
                obj.put("success", false);
                obj.put("error", ex.toString());
                out.println(obj.toJSONString());
            }
        }
    }
```

يمكن مناداته باستخدام الأداة *RESTClient* في متصفح Firefox بإرسال المُدخلات في شكل نسق JSON كالمثال التالي:

```
{"name":"معتز عبدالعظيم",
"address":"Khartoum, Sudan"}
```

ليظهر لنا الرد في شكل JSON كالتالي:



أما في جزئية العميل (برنامج CallRest) فنضيف المكتبة json-simple ثم نُحوّل المُدخلات لشكل JSON.
ثم نُضيف ترميز لإظهار اللغة العربية وذلك بتحويل البيانات إلى UTF-8 في الإجراء *callURL* كالتالي:

```
            URL url = new URL(myURL);
            URLConnection myURLConnection = url.openConnection();
            myURLConnection.setDoOutput(true);
            myURLConnection.setRequestProperty("content-type",
                  "text/json; charset=utf-8");
            myURLConnection.connect();
```

ثم نداءه بهذه الطريقة:

```
        JSONObject input = new JSONObject();
        input.put("name", "معتز عبدالعظيم");
        input.put("address", "السودان - الخرطوم");

        String content;
        content = callURL("http://localhost:8084/RESTServer/GetServerInfo",
input.toJSONString());
        System.out.println(content);
```

فتكون النتيجة هي مقطع في نسق JSON:

```
{"time":"Sat Sep 03 14:23:59 EAT
2016","javaversion":"1.7.0_111","OS":"Linux","customeraddress":"السودان - الخرطوم","success":true,"customername":"معتز عبدالعظيم"}
```

ويمكننا كذلك معالجة هذه النتيجة للتعامل معها كبيانات بسيطة:

```java
        JSONParser parser = new JSONParser();
        JSONObject outputResult = (JSONObject) parser.parse(content);

        String serverOS = outputResult.get("OS").toString();
        String serverTime = outputResult.get("time").toString();
        String javaVersion = outputResult.get("javaversion").toString();
        String customerName = outputResult.get("customername").toString();
        String customerAddress = outputResult.get("customeraddress").toString();
```

لكن علينا أولاً قراءة القيمة *success* فإذا كانت تحتوي على *true* نقرأ باقي القيم، أما إذا كانت *false* فنظهر الخطأ نفسه *error*:

```java
        JSONParser parser = new JSONParser();
        JSONObject outputResult = (JSONObject) parser.parse(content);
        boolean success = Boolean.valueOf(outputResult.get("success").toString());

        if (success){
            String serverOS = outputResult.get("OS").toString();
            String serverTime = outputResult.get("time").toString();
            String javaVersion = outputResult.get("javaversion").toString();
            String customerName = outputResult.get("customername").toString();
            String customerAddress =
                         outputResult.get("customeraddress").toString();

            // Display output
            System.out.println("Server OS is : " + serverOS);
            System.out.println("Server Time is : " + serverTime);
            System.out.println("Server Java version is : " + javaVersion);
            System.out.println("Customer name : " + customerName);
            System.out.println("Customer address : " + customerAddress);
        }
        else
        {
            System.out.println("Error: " + outputResult.get("error").toString());
        }
```

وفي الختام نتمنى أن تُنال الفائدة من هذا الكتاب.

ملحوظة:

هذا الكتاب مازال يُعدّل ويُنقّح من فترة لأخرى، فنرجو الحرص على الحصول على آخر نُسخة منه من الصفحة

www.code.sd/odjt

عَنْ أَبِي هُرَيْرَةَ رضي الله عنه عَنْ رَسُولِ اللَّهِ صَلَّى اللَّهُ عَلَيْهِ وَسَلَّمَ أَنَّهُ قَالَ :

(مَنْ قَعَدَ مَقْعَدًا لَمْ يَذْكُرْ اللَّهَ فِيهِ كَانَتْ عَلَيْهِ مِنْ اللَّهِ تِرَةٌ ، وَمَنْ اضْطَجَعَ مَضْجَعًا لَا يَذْكُرُ اللَّهَ فِيهِ كَانَتْ عَلَيْهِ مِنْ اللَّهِ تِرَةٌ)

سبحانك اللهم وبحمدك، أشهد أن لا إله إلا أنت، أستغفرك وأتوب إليك

معتز عبدالعظيم الطاهر

كود لبرمجيات الكمبيوتر

code.sd