جامعة حلب
كلية الهندسة المعلوماتية

مقدمة إلى لغة ADA

إعداد المهندسة

تولاي شاهين

الجمعة، 06 أيار، 2005

sweet_tolay@hotmail.com

الفهرس

مقدمة وتاريخ مختصر عن ADA :

لغة ADA هي لغة عالية المستوى صممت من أجل برمجة الأنظمة ذات الزمن الحقيقي , وعلى مستوى عال وضخم .

الأسم ADA مشتق من أوغستا ADA بيرن ابنة الأديب لورد بيرن , وعرفت بأول مبرمجة في العالم .
اخترعت ADA نتيجة تصور وكالة الدفاع الأمريكية أنه لا توجد لغة مناسبة جداً لتطبيقات أنظمة الزمن الحقيقي لنظم التحكم والأنظمة Embedded Systems .
Embedded Systems: التي هي عبارة عن نظام كمبيوتر موجود داخل نظام ما مثل الفرن الذكي, أو الصواريخ الموجهة.

شكل برنامج مكتوب بلغة ADA :

```
With Ada.text_Io;
Procedure Hello is
Begin
Ada.Text_IO. Put_Line (" Hello!!");
End Hello;
```

ناتج تنفيذ البرنامج هو طباعة
Hello!!
على الشاشة

Ada.Text_IO عبارة عن Package تحتوي على إجرائيات مختلفة منها Put_Line التي تأخذ متحولاً واحداً من النوع String , وتظهره على الشاشة .
يعتبر Hello البرنامج الرئيسي وهو يقوم با ستدعاء الإجراء Put_Line.

التصريحات المحلية :

```
With Ada.text_Io;
Procedure Hello is
I: integer;
X,y:Float;
Begin
Ada.text_Io.Put_Line (" Hello");
End Hello;
```

I, X, Y : متحولات داخلية لا يمكن الإشارة إليها من خارج الإجرائية

الثوابت :

الثوابت التي لم يحدد نمطها تدعى الأرقام المسماة .

```
PI: Constant: =3.1415;
Two_PI: Constant Float: = 2.0*PI;
```

* لغة ADA تسمح لتصريحات الثابت أن تكون ممزوجة مع تصريحات المتحولات .
* لغة ADA ليست Case Sensitive : أي مسألة الحرف الكبير أو الصغير غير مهمة , ولكن المتعارف عليه أن الكلمات المحجوزة تكتب بالحروف الصغيرة , بينما التصريحات تبدأ بحرف كبير .
* تبدأ الملاحظات في ADA بالإشارتين _ _ وتكفي لسطر واحد فقط .
* التصريحات أو المتحولات يمكن أن تكون بأسماء طويلة حتى طول السطر , وبذلك يمكن أن تكون ذات معنى, لذلك تسمى البرامج بلغة ADA أنها ذاتية النص .

قواعد تسمية المتحولات والثوابت :

يمكن أن تمتلك أحرف و أرقام و إشارة _ (Under Line) , لكن يجب أن تبدأ بحرف , وألا تتضمن فراغات , ولا يمكن استخدام إشارتين متتابعتين من _ (Under Line) , والفراغات غير مسموحة وبالنسبة للأرقام إشارة _ لا تعتبر مهمة , أي أن الرقم 12345 والرقم 12_345 هما متطابقين , ويجب أن تحاط إشارة _ بخانات رقمية .

* يمكن للبرنامج أن يستدعي التوابع والإجرائيات التي يتضمنها ال Package مباشرة من دون ذكر اسمه , إذا أضفنا العبارة
Use Ada.Text_IO; بعد عبارة With Ada.Text_IO;

كما يمكن استعمال كلمتي With & Use لعدة Packages .

With Ada.Text_IO, Ada.Calender, My_Pkg; ← هذا الطرد من تشكيلنا :
Use Ada.Text_IO, Ada.Calender, My_Pkg;

* الإجرائية Put_Line (Param) تظهر السلسلة المذكورة ضمن القوسين مع سطر جديد , بينما الإجرائية Put تظهر السلسلة بدون سطر جديد , الإجرائية New_Line تولد سطر جديد .
Param: متحول من النوع String .

* الاستدعاء Put_Line (2+2) هو استدعاء خاطىء , لأن الإجرائية Put_Line لا تأخذ بارامترمن نوع Integer , والبرنامج الصحيح لحساب 2+2 وإظهار النتيجة :

```
With Ada.Text_IO; Use Ada.Text_IO;
Procedure Add is        اسم اختياري
          Package My_Int_Io is new Integer_IO (Integer);
          Use My_Int_Io;                طرد عام
Begin
     Put (2+2);
New_Line;
End Add;
```

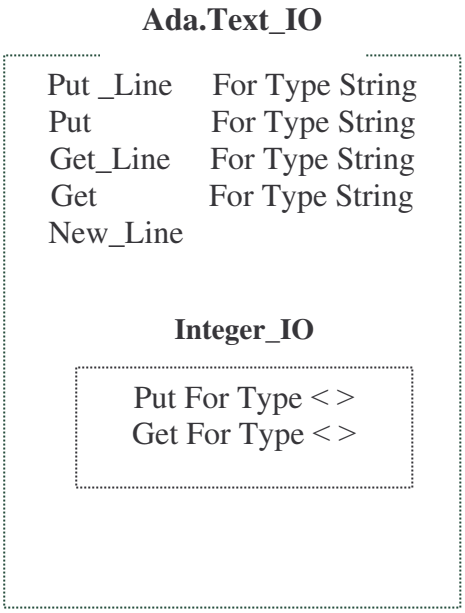
إن الطرد Ada.Text_IO يحتوي على إجرائيات لنوع أو نمط String , وهذا الطرد يعتبر جاهزاً للأستعمال , لكن داخل هذا الطرد يوجد طرد آخر يدعى : Integer_IO والذي يعتبر غير جاهز للأستعمال , يدعى الطرد العام Generic Package , لأنه يمتلك صندوقاً فارغاً < > مكان النمط , يمكن أن نجعله جاهزاً للأستعمال باستخدام جديد New , وإعطاء النمط الذي نريد

```
My_Int_IO IS new Integer_IO (Integer)
```
اسم اختياري

لقد صرحنا عن الطرد My_Int_IO بشكل محلي داخل الإجرائية Add , وقد أصبح هذا الطرد يملك نفس الاجرائيات والتوابع التي يمتلكها الطرد العام Integer_IO , ولكن مع إملاء الصناديق الفارغة بالنمط .(والشكل (1) يوضح ذلك)

الطرد Ada.Text_IO يعتبر جاهزاً للاستعمال من أجل النمط String , بينما Integer_IO هو طرد عام ويجب أن نأخذ مشتقات منه لنمط Integer , والسبب يعود إلى أن البرامج تستخدم عادة نمط واحد من

النوع String , لكن يمكن أن تحتوي على العديد من أنماط ال Integer , حيث يمكن توليد عدد كبير من أنماط Integer .



الشكل (1)

إن الاستدعاء Integer_IO.Put هو خاطىء, لأنه لا يمكننا استعمال طرد عام, ولكن نستخدم مثال عنه (اشتقاق منه) مثل My_Int_IO

الطرود الجاهزة للإستعمال تكون في الأصل مترجمة , لذلك يمكن التعامل معها مباشرة باستخدام With & Use , أو بدون استخدامها , كما في المثال التالي :
حيث يمكن كتابة البرنامج Test.Ada الذي يستدعي وينفذ كلاً من الاجرائيات Hello & Add.

```
With Hello, Add ;
     Procedure Test IS
     Begin
          Hello;
          Add;
     End Test;
```

نتيجة التنفيذ سيظهر
Hello & 4
كل واحدة في سطر

- العدد من نمط Float يجب أن يكون فيه على الأقل رقم قبل الفاصلة , ورقم بعدها .
- تسمح ADA بتحول أحد المتحولات من نمط لآخر مثلاً

K: Integer ; f:Float ; ⟹ k:=Integer(f)+30 ; عبارة صحيحة

وعند الانتقال من النمط Float إلى Integer فأن الرقم يدور rounded ولا يقصر truncated

الأنماط اللوائحية : (Enumeration Types)

```
Type Rainbow_Color IS (Red,Orange,Yellow,Green,Blue,Violent);
RC: Rainbow_Color;
```

صرحنا عن RC متحول من النمط Rainbow_Color
يمكن أن تكون قيم اللائحة على شكل محرف واحد , فيوضع ضمن إشارة ' '

```
Type Event_Digit IS ('0','2','4','6','8');
```

من الممكن مزج المحارف والمعرفات في نفس التصريح :

```
Type Mixed Is (Big,Small,'x','9');
```

من غير المسموح كتابة أنماط المعالجات :

```
Type Processor IS (8048,Z80,….);
```

الأنماط الجزئية (Sub Types) :

```
SubType Day_SubType IS Integer range 1..31;
D: Day_SubType;
```

إذا كانت I من نوع Integer فيمكن أن نكتب D:=D+I; , لأن D&I تملكان نفس النمط .
يوجد نمطين جزئيين في لغة ADA هما :

```
SubType Positive IS Integer range 1..Integer,Last;
SubType Natural IS Integer range 0..Integer,Last;
```

يمكن استخدام Positive & Natural مباشرةً في برامجنا .

المعاملات العلائقية :

- =/ تعني لا يساوي , والكلمة المحجوزة IN تختبر فيما إذا كان أي شيء هو داخل المجال وتعطي جواباً بولياناً (منطقياً) , فالعبارتين التاليتين متطابقتين :

```
If X IN Lower …Upper Then …….. End If;
```

Lower & Upper متحولات من النوع Float .

```
If X >= Lower  And X <=  Upper Then …….. End If;
```

أشكال الدارة القصيرة (Short Forms) :

```
If D=0.0 Or Else N/d >=10.0 Then
……. ;
 ……. ;
……. ;                          ( Block Of Code )
……. ;
……. ;
End If;
```

إذا تحقق الشرط على يسار Or Else وهو D=0.0 , فأنه لن يتم فحص الشرط على يمين Or Else , وبالتالي لا تحصل مشكلة القسمة على صفر , لأنه لم يتم تفعيل التعبير الثاني

```
N/d >=10.0
```

إن Or Else تشبه || في لغة C & C++

```
If D/=0.0 And Then  N/d >=10.0 Then
……. ;
 ……. ;
……. ;                          ( Block Of Code )
……. ;
……. ;
End If;
```

إذا كان التعبير الأول غير محقق (False) فأن العبير الثاني لن يكون مفعلاً , أي لن يفحصه , أما إذا كان التعبير الأول محقق (True) فأن التعبير الثاني يجب تفعيله لتحديد نتيجة التعبير كاملاً , وهكذا نضمن أنه لن تحصل عملية قسمة على صفر .

البنى التحكمية :

IF

```
* If A>= B and c=A+D Then
……. ;
……. ;
End If;
```

While Loop

```
* While I < 10 Loop
……. ;
……. ;
End Loop;

* * While True  Loop
……. ;          حلقة لا نهائية يمكن الخروج منها بتعليمة
……. ;                  Exit
End Loop;
```

For Loop

```
* For IX IN 1..10 Loop
……. ;
……. ;
End Loop;

** For IX IN Reverse  1..10 Loop
……. ;
……. ;
End Loop;
```

Ix يصرح عن نفسه داخل حلقة For, ويختفي عند الخروج من الحلقة .

Case Expression

```
Case C IS
   When '*'=>
      …….. …… ;
      ……;
      When '#'| '$'=>
      …….. …… ;
      ……;
      When '0'..'9'=>
      …….. …… ;
      ……;
      When 'A'.. 'z'| 'a'..'z'=>
      …….. …… ;
      ……;
      When Others=>
      …….. …… ;
      ……;
End Case;
```

* | تعني Or
* C من النمط Character والتعبير المختبر يجب أن يكون من النمط discrete أي (eumeration Integer)
* C لا يجوز أن تكون من النمط Float أو أي نمط جزئي منه .
* إذا أردنا ألا ننفذ شيء في الحالة العامة (أي إذا لم تأخذ) أياً من القيم المشار إليها يمكن أن نكتب حيث Null لا تقوم بفعل شيء

.

التوابع والإجرائيات:

```
Procedure Proc_Demo Is
     X:Float:=1.2;
     Y:Float;
Function Twice (Dammy:IN Float) Return Float IS

Answer :Float;
Begin
Answer:=Dummy*2.0;
Return answer;
End Twice;
Begin _ _ main program
Y:= Twice(X);
End Proc_Demo;
```

في حالة التوابع فأن نظام جميع البارامترات يجب أن تكون في حالة IN , أما الإجرائيات فيمكن أن تكون IN Out , Inout .

إذا لم نكتب نظام البارامترات (mod) فهذا يعني أنه دخل (IN) .

السجلات (Records):

```
Type   Month_Type IS (Jan,Feb,Mar,…..,Nov);
Subtype Day_Subtype IS Integer range 1..31;
Type Date IS
Record
Day: Day_Subtype;
Month: Month_Type;
Year: Integer;
End record;
Usa: Date;
```

تعطى القيم لحقول السجل كما يلي :

```
Usa.Day: =4;
Usa.Month: =Jul;
Usa.Year: =1776;
```

أو بالشكل Usa:=(4,jul,1776) وهذه الطريقة الوضعية aggregate ويكون الترتيب مهماً .

أو بالشكل

```
Usa:=( Month=>jul, Day=> 4, Year=> 1776)
```

وهنا الترتيب غير مهم .

المصفوفات (Arrays) :

```
B: array (Integer range -10..10) of Float
```

B متحول من نوع مصفوفة مكونة من 21 عنصراً من نوع Float مع مفهرس من نوع Integer.
كما يمكن أن نعرف نوع مصفوفة , ثم نصرح عن عدة متحولات من نوع مصفوفة

```
Type Vector A IS array (1..100) Of Float;
D, E, F: Vector;
```

ويمكن إعطاء قيم لعناصر مصفوفة كما يلي :

```
V: Vector A: =(10|15|30|50=>5.5,Others =>0.0);
```

السلاسل (Strings) :

يوجد تصريح مهم لمصفوفة ذات نمط مهم يأتي مع لغة ADA ويجب أن لا يكرر في برامجنا هو :

```
Type String IS array (Positive range <> ) of Character;
```

فيمكن أن نصرح S:string (1..5) , ولا نستطيع أن نكتب S:String .
والمعامل & يقوم بربط سلسلتين معاً , أي يسلسل مصفوفتين من نفس النوع (النمط) , أو عنصر مفرد مع مصفوفة لها نفس نمط العنصر .

برامج بسيطة بلغة ADA

- **برنامج لحساب مجموع الأعداد الفردية المحصورة بين 1..... 39.**

```
With Ada.Text_IO, Ada.Integer.Text_IO;
Use Ada.Text_IO, Ada.Integer.Text_IO;
     Procedure Add Is
          Oddnumber:Integer;
     Begin
          Oddnumber: =1;
          While Oddnumber<=39 Loop
          Put (oddnumber);
          Oddnumber: =oddNumber+2;
     End Loop;
  End  Add ;
```

- **برنامج لأدخال سلسلة أرقام , يتوقف الإدخال عندما ندخل العدد (0) ويكون ناتج الطباعة ضرب هذه الأرقام .**

```
With Ada.Text_IO, Ada.Integer.Text_IO;
Use Ada.Text_IO, Ada.Integer.Text_IO;
     Procedure Multiplay Is
          Stop: Constant Integer: =0;
          Number: Integer;
          Mul:Integer;
     Begin
     Put ("Enter 0 When Finished");
```

```
     Put ("Enter the First Number");
     Mul: =1;
     Get (Number);
     While (Number/=Stop) Loop
       Mul: = Mul*Number;
     Put ("Enter the Next Number);
     Get (Number );
End Loop;
Put ("The Result Is ");
Put (Mul);
End Multiplay;
```

- **برنامج لطباعة مجموعة أعداد ومربعاتها :**

```
With Ada.Text_IO, Ada.Integer.Text_IO;
Use Ada.Text_IO, Ada.Integer.Text_IO;
     Procedure Squre Is
          Function Sqr(Num:Integer) return Integer Is
          Result: Integer;
          Begin
             Result: =Num**2;
Return Result;
End Sqr;
Maxnumber: Constant Integer: =10;
Sqrnumber: Natural;
Begin _ _ Main Program
New_Line;
Put ("_ _ _ _ _ _ _ _ _ _ _ _ _ ");
New_Line;
For I IN 1..Maxnumber Loop
  Put (I);
Sqrnumber: =sqr (I);
Put (Sqrnumber);
New_Line;
End Loop;
  End Squre;
```

نتيجة التنفيذ :

```
 1          1
 4          2
 9          3
16          4
25          5
36          6
49          7
64          8
81          9
100         10
```

المهام في ADA (TASKS):

في مجال مسألة العالم الحقيقي , تحدث عادة عدة نشاطات بنفس الوقت , فعلى سبيل المثال , يمكن لطيار آلي أن يراقب , وبشكل مستمر حساسات الرياح , وزاوية الهجوم , وينتظر طلبات المستخدم , ويسيطر على عدة أجهزة مستقلة عن بعضها , مثل الأجنحة وغير ذلك .

فإذا أردنا تطوير حل برمجي لمسائل من هذا النوع , سنجد بأن معظم لغات البرمجة عالية المستوى , تقدم قليلاً أو لا تقدم المساعدة للتعبير عن هكذا نشاطات موازية , أما بلغة ADA فأن المهمة ببساطة تمثل كياناً يعمل على التوازي مع عدة وحدات برمجية أخرى , حيث تكون المهام مستقلة منطقياً , ويمكن تنفيذها على عدة نظم حاسوبية , أو نظم متعددة المعالجات .

على سبيل المثال : لنعتبر نظاماً ذا مهمتين , المهمة الأولى تأخذ عينات من لوحة المفاتيح , وتجمعها على شكل أسطر نصية , ولتكن هذه المهمة (Producer) والمهمة الثانية تأخذ الأحرف المحفوظة بأسطر وترسلها عبر مودم , ولتكن هذه المهمة (Consumer) توجد طريقتان رئيسيتان تعبران عن الاتصال بين هاتين المهمتين .

فالطريقة الأولى : مشابهة لتوجيه رسالة من خلال صندوق البريد ,والمهمة (Producer) تبني مجموعة من مدخلات لوحة المفاتيح , ومن ثم , عند الانتهاء من تجميع سطر طبيعي , يوضع ما تم تجميعه في منطقة ذاكرة مشتركة ومعروفة بالنسبة للمهمتين (صندوق بريد) , وبعدها يتم تحديد بعض أنواع المؤشرات , للتنويه بأن صندوق البريد يحتوي رسالة , وبالتالي فالمهمة (Consumer) تجمع الرسائل المتواجدة في صندوق البريد , وحسب قواعد هذا النوع من الاتصال , فقط يمكن لمهمة واحدة أن تدخل صندوق البريد , في وقت واحد (المنح المتبادل) , وإذا حاولت كلتا المهمتين الدخول بنفس الوقت , فيجب على إحداهما الانتظار , كيلا تتداخل مع الأخرى .

لذلك إذا لم يوجد أي دخل , فأن المهمة Consumer تنتظر في صندوق البريد , ومن جهة أخرى إذا كانت المهمة Consumer مشغولة في معالجة رسالة , فأن المهمة Producer ببساطة تتابع توجيه الرسائل وإيداعها في صندوق البريد .

وبمفهوم لغات البرمجة يمكننا زرع ما يدعى سيمافور (Semaphores) أو مونيتور (Monitors) لتحقيق المتطلبات السابقة.وعند الحاجة لأكثر من مهمة من النوع (Producer) , ولأكثر من مهمة من النوع (Consumer) , بدلاً من مهمتين فقط , فمن الضروري الحصول على نظام أفضليات من أجل حجز المنابع , وعلى الغالب فأن هذا النوع من اتصال المهام غير متزامن. يوجد طريقة أفضل من استخدام السيمافورات وهي :

الطريقة الأكثر طبيعية: لمعالجة تفاعل المهام , هي معالجة كل مهمة كإجراء تسلسلي متصل كما في الشكل :

فبدلاً من أن تكون المهام غير متزامنة , فأن بعض المهام تكون متزامنة بالزمان والمكان ,عندما تكون متصلة بطريقة مشابهة لشخصين يتحادثان فيما بينهما,

وباستخدام المثال السابق: فعند تجميع سطر دخل من قبل المهمة (Producer), فإنها تستدعي مدخلاً (Entry) من المهمة (Consumer), وهذا يشير أن المهمة جاهزة للاتصال , وفي اللحظة التي تقبل (Accept) فيها المهمة المدخل , تمر الرسالة , وبعدها

تعمل المهمتان بشكل منفصل عن بعضهما البعض , حتى تصبح المهمة (Producer) جاهزة لتسليم رسالة أخرى , وأي تزامن صريح يعرف بموعد Rendezvous .

إذا كانت إحدى المهام جاهزة للدخول أو القبول قبل أن تكون المهمة الثانية في نقطة الموعد , فأن لتلك المهمة ثلاث خيارات :
1_ الانتظار بشكل غير محدد .
2_ أو الانتظار لفترة زمنية معينة .
3_ يمكنها الدخول / القبول مهمة أخرى جاهزة للاتصال .

الفائدة من هذه الطريقة هو أن الاتصال يكون أكثر وثوقية , فإذا تعطلت أحدى المهام , أو تأخرت يمكن في هذه الحالة لمهمة أخرى أن تكتشف ذلك وتأخذ قياسات الأفضلية .
أما إذا كان اتصال المهام متزامناً (المهمتان تجهزان في نقطة الموعد) , فيمكن بسهولة التعبير عن العلاقات المتزانة بين مهمتين.

شكل المهام بلغة ADA :

تعتبر المهمة إحدى ثلاث وحدات برمجية رئيسية بلغة ADA (الوحدتان الباقيتان هما البرامج الجزئية , والحزم البرمجية) .

تتكون المهمة من قسمين: قسم توصيف المهمة , وقسم جسم المهمة , ويمثل قسم توصيف المهمة Specification , الواجهة بين المهمة وبقية الوحدات البرمجية , بينما يتكون جسم المهمة من التعليمات القابلة للتنفيذ .

توصيف المهمة (Task Specification) :

```
Task Consumer Is
     Entry Recive_Message (A_message: in String);
End Consumer;
```

Consumer : اسم المهمة (Optional) أي اختياري.
Recive_Message : اسم المدخل وهو اختياري.
A_message : بارامترات (معاملات صورية) تصف نوع الرسائل التي يمكن تمريرها , وقد تأخذ أحد النماذج In, Out , In Out فتحدد جهة التدفق لكل رسالة .
يمكن استدعاء مدخل مهمة من أي نقطة مسموح منها استدعاء برنامج جزئي (أي من برنامج جزئي أو برنامج رئيسي أو من مهمة أخرى , أو من جسم حزمة برمجية Package , لكن لا يوجد معنى لأن تستدعي مهمة , مداخلها الخاصة , فإذا حاولت مهمة أن تتصل مع نفسها , يمكن خلق شروط موت Dead Lock , لأن الموعد مستحيل , ويحدث الموت عندما تنتظر المهام منبعاً , لا يمكن مطلقاً أن يصبح حراً .
ليكن لدينا توصيف لمهمة كما يلي :

```
Task Protected_Stack Is
     Entry Pop (Element: Out Integer);
     Entry Push (Element :In Integer);
End Protected_Stack;
```

لا يمكن تطبيق العبارة Use على مداخل المهام , لذلك من الضروري دائماً إسباق المدخل المستدعى باسم المهمة .

```
Protected_Stack.Pop (My_Value);
Protected_Stack.push (36);
```

ويمكن إعادة تسمية المداخل مثل الإجرائيات :

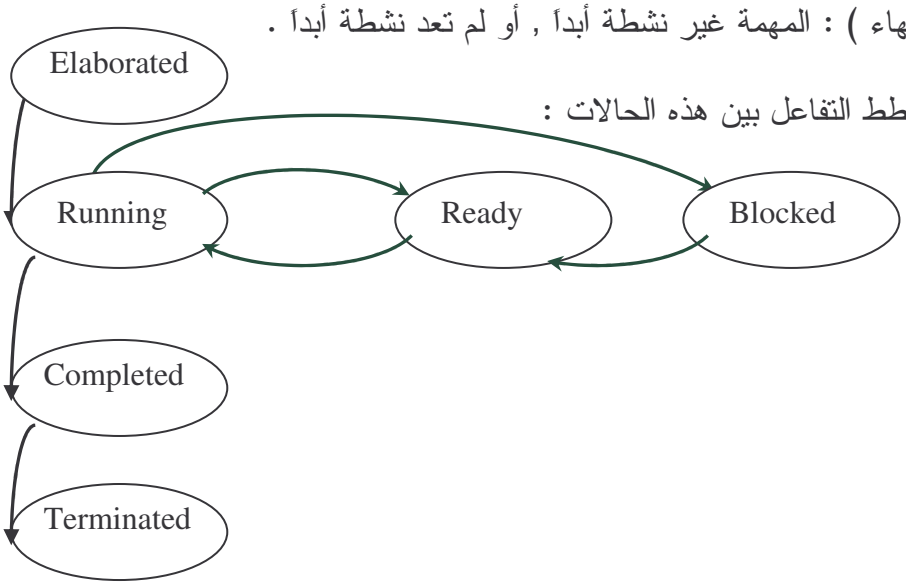
```
Procedure Protected_Pop (Element:Out Integer) Renames
          Protected_Stack.Pop;
```

وإعادة التسمية مفيدة جداً عندما يكون اسم المهمة , وأسماء مداخلها طويلة , ونرغي بتقصيرها , وأعطائها اسماً ذا دلالة لعمل المدخل.
إن استدعاء المدخل شبيه باستدعاء برنامج جزئي , لكن يختلف بما يلي :
إذا استطاعت عدة مهام استدعاء نفس البرنامج الجزئي , عندئذ يمكن لعدة مهام فعلياً تنفيذ نفس البرنامج الجزئي في وقت واحد , وعندها نقول بأن ترميز البرنامج الجزئي مشترك أو (متعدد الدخول (Reentrant)) , وعلى أية حال يوجد رتل ضمني (Implicit queue) , فإذا استدعت عدة مهام نفس المدخل , فقط مهمة واحدة (ابتداءً من المهمة التي استدعت المدخل أولاً) يسمح لها بتنفيذ الموعد (Rendezvous) , وسننتظر جميع المهام الأخرى في الرتل الضمني وفق ترتيب الوصول لكل مهمة

(First_ In_ First_ Out) , وليس حسب درجة الأفضلية للمهمة , وفي حال استدعاء مهمتين لنفس المدخل في لحظة واحدة , فسيتم اختيار المهمة اختيارياً , ويمكن لمهمة ترك الرتل قبل إكمال الموعد في حال وجود الشرط (Time Out) : بعد مضي القسم الأعظم من زمن الانتظار , ويمكن لمهمة نشطة أن تتواجد في إحدى الحالات الخمس التالية:

1_ Running (تنفيذ) : المعالج يخدم المهمة في الوقت الحالي .
2_ Ready (جاهزية) : المهمة منفكة (ليست في حالة تأخير , وليست منتظرة موعداً) وجاهزة من أجل المعالجة .
3_ Blocked (انتظار) : المهمة في حالة تأخير , أو منتظرة موعداً .
4_ Completed (اكتمال) : المهمة أنهت تنفيذ سلسلة تعليماتها .
5_ Terminated (انتهاء) : المهمة غير نشطة أبداً , أو لم تعد نشطة أبداً .

والشكل التالي يوضح مخطط التفاعل بين هذه الحالات :



خلال الحياة النشطة للمهمة

في البدء يجب إنجاز المهمة Elaborated أي كتابة تصريحها مع جسمها , وخلال حياتها النشطة , فأن المهمة تأخذ أحدى الحالات Blocked , running , Ready وإن الحالة Running تشير بأن المهمة تمتلك منابع المعالجة (معالج , ذاكرة) والحالة Ready تشير بأن المهمة تنتظر منابع المعالجة , لكنها جاهزة للتنفيذ , والحالة Blocked تشير بأن المهمة تنتظر حدثاً مثل موعد .

في ADA لا يحتاج ترتيب المهام إلى خوارزمية لتقسيم الزمن , فعندما تكون المهمة نشطة , يمكنها متابعة التنفيذ حتى تجهز مهمة ذات أفضلية أعلى , وعند انتهاء مهمة من تنفيذ تعليماتها تنتقل لحالة الاكتمال (Completed) , وتنتظر حتى تكتمل جميع المهام المرتبطة بها , وبعد اكتمال المهمة , لا تهتم المهمة بأي موعد , وفي حال انتهاء جميع المهام الأبناء من النشاط , ستنتهي المهمة .

ويرتبط بكل مهمة أفضليات ثابتة تشير إلى درجة الأهمية , حيث يتمثل تأثير الأفضلية بالمساعدة في تخصيص منابع المعالجة (المعالجات , أو أماكن الذاكرة) , إلى مهام متوازية .

* وفي حال وجود أكثر من مهمة جاهزة للتنفيذ , بينما الموادر الجاهزة لا يمكنها استيعاب هذه المهام الجاهزة , عندها سيتم اختيار المهمة ذات الأفضلية الأعلى , ليتم تنفيذها (وضعها في حالة Running) . فإذا كانت المهام الجاهزة ذات أفضليات متساوية , أو أفضليات غير محددة , فأن الترتيب الزمني غير معرف في ADA , ويترك القرار للتنفيذ .

* وليس من الضروري أن نعرف مدخلاً أو أكثر في مهمة , مثال ذلك ما يلي :

```
Task Producer
```

في هذه الحالة تم تعريف مهمة لا تمتلك أي مسار اتصال مرئي (أي مدخل Entry) , وهذا النوع من المهام يسمى Actor Tasks , فهو لا يقدم أي خدمة لأي وحدة برمجية أخرى .
ولكن تبقى دائماً في حالة نشطة , لكن المهمة من النوع Actor Task تستطيع أن تستدعي المداخل المرئية لمهام أخرى .

* كما توجد مهام غير فعالة من النوع Server Tasks , إذا أنهاتمتلك مداخل لكنها لا تستدعي مداخل مهام أخرى .

* ويمكن أن توجد مهمة تحتوي مداخل تستخدم من قبل مهام أخرى , وهي بدورها تستخدم مداخل مهام أخرى .

- المهمة المستدعية يجب أن تعرف اسم المهمة المدعوة , بينما المهمة المدعوة لا تعرف اسم المهمة المستدعية .
-

أجسام المهمة Tasks Bodies

```
Task Water_Monitor;
Task Body Water_Monitor IS
     Begin
        Loop
          If Water_Level> Maximum_Level The
               Sound_Alarm;
          End If
          Delay 1.0;
```

وتستمر هذه المهمة بشكل دائم , وتصدر تنبيهاً عندما يزداد مستوى الماء فوق حد معين (Maximum Level) والتعليمة Delay تجعل المهمة تنتظر على الأقل ثانية واحدة , (وهو شرط يحدده مطور البرنامج ليصف حالة نوم) , وبعد ذلك تتكرر الحلقة بسرعة .

```
    End Loop;
  End Water_Monitor;
```

فأذا عرفنا مداخل لمهمة , يجب على جسم المهمة أن يحتوي على الأقل تعليمة Accept واحدة , موافقة لكل مدخل .

مثال :
توصيف المهمة التي سترسل رسالة إلى المودم

```
Task Consumer IS
     Entry Transmit_Message (A_Message: in String);
End Consumer;
Task Body Consumer IS
     Begin                                                    جسم المهمة
          Loop
            Accept Transmit_Message (A_Message: in String) Do
            Text_IO.Put (Modem, A_message);
            End Transmit_Message;
          End Loop;
     End Consumer;
```

من أجل مهمة من النوع Nonactor ومن أجل تحقيق موعد , يجب أن يتحقق الشرطان التاليان :
_ استدعاء مدخل من خارج المهمة .
_ وتعليمة Accept الموافقة من داخل جسم المهمة .
ومن أجل كل مدخل , يستطيع جسم مهمة أن يحتوي تعليمة أو أكثر من Accept , وتعليمة Accept مؤلفة من كلمة محجوزة وهي Accept , متبوعة باسم مدخل مع أدلة اختيارية (بارامترات) .

```
Accept Do …….. End
```

وفي حال عدم وجود أي عمل نستطيع إهمال عبارة Do …….. End .

أمثلة على المهام في لغة ADA :

برنامج لمهمة واحدة :

```
With Ada.Text_IO;
Procedure One_Task IS
Task Type SimpleTask (Message: Character);
Task Body SimpleTask Is
Begin _ _ Simple Task
     For Count IN 1..10 Loop
       Ada.text_IO.Put ("Hello from Task "&Message);
       Ada.Text_IO.New_Line;
     End Loop;
End SimpleTask;
Task_A: SimpleTask (Message=>" A" );
Begin _ _ One_Task
Null;
End One_Task;
```

على عكس الاجرائيات , المهام لا تستدعى بل تنشط (تفعل) آلياً , فالمهمة A ستبدأ بالتنفيذ حالما يصل المترجم إلى هذه النقطة (مباشرة بعد Begin , وقبل أن تنفذ أي تعليمة من تعليمات البرنامج الرئيسي)
نتيجة التنفيذ : تكرر كتابة التعليمة Hello from Task A عشر مرات .

```
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
Hello from Task A
```

برنامج لمهمتين :

```
With Ada.Text_IO;
Procedure Two_Tasks IS
Task Type SimpleTask (Message: Character; HowMany: Positive);
Task Body SimpleTAsk IS
Begin
     For Count IN 1..HowMany Loop
       Ada.Text_IO.Put (Items=> "Hello from Task "&Message);
       Ada.Text_IO.New_Line;
     End Loop;
End SimpleTask;
Task_A: SimpleTask (Message=> "A", HowMany=> 5);
Task_B: SimpleTask (Message=> "B", HowMany=> 7);
Begin
Null;
End Two_Task;
```

صرحنا عن متحولين من نوع مهمة SimpleTask
المهمة A والمهمة B ستبدآن بالتنفيذ حالما يصل التحكم إلى هذه النقطة , وذلك قبل أن تنفذ أي تعليمة من تعليمات البرنامج الرئيسي .
وهنا لا يتم تحديد أي مهمة ستبدأ أولاً .
فعندما يبدأ التنفيذ بالمهمة B فينفذها بالكامل أي يتم طباعة العبارة Hello from Task B سبع مرات ,
وبعدها يتم طباعة العبارة Hello from Task A خمس مرات .
نتيجة التنفيذ :

```
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task B
Hello from Task A
Hello from Task A
Hello from Task A
```

```
Hello from Task A
Hello from Task A
```

استخدام Delay لتحقيق التشارك :

```
With Ada.Text_IO;
Procedure Two_Cooperation_Tasks Is
Task Type SimpleTask (Message: Character; HowMany: Positive) ;
Task Body SimpleTask IS
Begin
For Count IN 1..HowMany Loop
     Ada.Text_IO.Put ( "Hello From Task"&Message);
     Ada.Text_IO.NewLine;
     Delay 0.1;
End Loop;
End SimpleTask;
Task_A: SimpleTask (Message=> 'A', HowMany=> 5);
Task_B: SimpleTask (Message=> 'B', HowMany=> 7);
Begin
Null;
End Two_Cooperating_Tasks;
```

يبدأ التنفيذ حالما يصل التحكم إلى هذه النقطة , ولا يتم تحديد أي مهمة ستبدأ أولاً .
تكون نتيجة التنفيذ

```
Hello From Task B
Hello From Task A
Hello From Task B
Hello From Task A
Hello From Task B
Hello From Task A
Hello From Task B
Hello From Task A
Hello From Task B
Hello From Task A
Hello From Task B
Hello From Task B
```

استخدام start Buttons للتحكم بترتيب بداية المهام :

```
With Ada.text_IO;
Procedure Start_Button IS
Task Type SimpleTask (Message: Character; HowMany: positive) IS
     Entry StartRunning;
End SimpleTask;
Task Body SimpleTask IS
Begin
Accept StartRunning;
For Count IN 1.. HowMany Loop
     Ada.Text_IO.Put (Item=>"Hello From Task"&Message);
     Ada.text_IO.New_Line;
     Delay0.1;
End Loop;
```

```
End SimpleTask;
Task_A: SimpleTask (Message=>'A', HowMany=>5);
Task_B: SimpleTask (Message=>'B', HowMany=>7);
Task_C: SimpleTask (Message=>'C', HowMany=>4);
Begin - Start Buttons
Task_B.StartRunning;
Task_A.StartRunning;          هنا نتحكم بترتيب بداية المهام (أي مهمة ستنفذ أولا )
Task_C.StartRunning;
End Start_Buttons;
```

نتيجة التنفيذ :

```
Hello From Task B
Hello From Task A
Hello From Task C
Hello From Task B
Hello From Task A
Hello From Task C
Hello From Task B
Hello From Task A
Hello From Task C
Hello From Task B
Hello From Task A
Hello From Task C
Hello From Task B
Hello From Task A
Hello From Task B
Hello From Task B
```

تم بحمده تعالى
الخميس، 05 أيار، 2005