معـايير في تقييـم لغـات البرمجة

بماذا يجب أن نفكر عند اختيار لغة برمجة .

عروة عيسى . ٢٠٠٤/١٢/٥ نسخة مسودة ، للحصول على النسخة النهائية راجع : www.orwah.net


New
معايير فى تقييم
لغات البرمجة
Y - OS Integrity
X - language
بقلم: عروة عيسى
C++ - PERL - DELPHI - .NET - JAVA
www.orwah.net

الفهرس ..

بسم اللّه الرحمن الرحيم

لغات البرمجة هي أدوات خاصة صممت لتحقيق أهداف مختلفة، وكل أداة منها تملك ميزات تناسب مجموعه حالات أكثر من حالات أخرى، وهذه الميزات نفسها يمكن اعتبارها نقاط قوة أحيانا ونقاط ضعف أحيانا أخرى ، حسب المشروع والغاية التي نريد تحقيقها . وأمور متعددة أخرى تتعلق بمستخدم الأداة وظروف الاستخدام.

هذه اللغات موجَّهة لحاجات مختلفة ، ووجدت لتحل مشكلات مختلفة بطرق مختلفة ، وتستخدم في بيئات برمجيّة متباينة .

المشكلة في لغات البرمجة أن فلسلفة كل لغة متعلقة بالأهداف التي بنيت اللغة على أساسها، ومزايا لغة برمجة ما هي ناتج دمج هذه الأهداف من ناحية والضريبة المترتبة على أولوية هذه الأهداف على غيرها.

خذ مثلاً بعض اللغات الشائعة:
- لغة C++: من أهدافها القوه والتحكم، تتميز بالكثير من التعقيد، ولها سرعة مميزة في التنفيذ.
- الدلفي: والهدف هو التركيز على برمجة النوافذ ، وزيادة الإنتاجية، مميزة في برامج E_Business
- الجافا: تسعى لقابليه النقل بدون كبير اهتمام بالسرعة، مطرزه بالعديد من مزايا القوّة، مميزة في التطبيقات الموزعة والويب .
- VB : تهدف إلى السهولة وسرعة التطوير . وتبسيط برمجة النوافذ ، مميزة في تطبيقات أوفيس

يمكننا ببساطه أن نقرر أن نجاح كل من هذه اللغات لا يعود إلى المميزات التي ذكرناها فقط، فما هي إلا غيض من فيض، وتوجد عشرات الأسباب الأخرى المهمة لتقرير أهميّـه لغة برمجة ما أو نجاحها في الانتشار والتسويق.

ما سأقدمه في هذه العجالة هو عرض مبسط للمزايا الأساسية التي تحكم أهميه ونجاح لغات البرمجة من ناحية ، وما هي المعايير التي ينبغي وضعها في الحسبان عند اختيار لغة برمجة ما من أجل مشروع معين .
وتبقى العديد من الجوانب بحاجه لمزيد من التوسع والتدقيق ويبقى المشروع قابلا للتعديل وإضافة معايير وبنود أخرى إلى هذه القائمه المبسطة والمختصرة.

وليعذرني القاريء إذا وجد بعض النقص هنا، وبعض الركاكة وعدم الوضوح هناك. فالمبتغى هو رضوان اللّه أولا ولفت النظر لبعض الجوانب المهمة ثانيا، ويترك للقاريء حرية الرأي والقرار بعد الإطلاع عليها.

. واللّه وليّ التوفيق.

لمن هذا الكتاب

هذا الكتاب موجه أساسا للأخوة التقنيين المهتمين بالإطلاع على الميزات والفروقات بين لغات البرمجة الشائعة
ويبقى مفيدا لكل للأفراد الذين يريدون البدء باختيار لغة برمجة تناسبهم وتناسب نوعيه البرامج التي يرغبون بإنتاجها
أو للشركات المعنية بإنتاج تطبيقات برمجيّة وتهتم بمعرفه لغات البرمجة الأنسب لكل مشروع .

ملاحظات لابد منها

نحن لا نقدم في هذا الكتيب نتائج جاهزة وخلاصات ثابتة عن المشاريع المناسبة لكل لغة أو مقارنات ثابتة بينها، ولانهتم بتقديم ذلك أصلا، وذلك من منطلق الاقتناع أن هذا قد يكون رأي شخصي وقد أختلف به مختصون بالتقنيات وعلوم الحاسب حول العالم، ولامجال لنحشر أنفسنا في هذه المعمعات المبنية على أساس التعصب بالرأي لصالح لغة برمجة دون أخرى.
إنما هدفنا ذكر مزايا تقنية وفنيه مؤثره على لغات البرمجة وشرحها ومناقشتها، ثم قد نورد أمثلة على دعم بعض لغات البرمجة لهذه المزايا للاستئناس ليس أكثر، ويبقى المعيار المطروح هو الهدف والغاية أما المثال بلغات البرمجة فهو قابل للعلاج والمناقشة وإبداء الرأي الشخصي، وما ورد من تحديد لأسماء لغات برمجة هو رأيي الشخصي بها وطبعا يخصني وحدي ويقبل الخطأ والتصحيح..

ترتيب المعايير في هذا البحث ليس بحسب الأهمية، إنما وضعت بحسب التسلسل الزمني في كتابتها
والغاية هي تبيان أن أهميه كل معيار قد تختلف حسب نوع المشروع الذي نرغب بتطويره.

هناك فرق كبير بين لغة البرمجة كنحو Syntax وكمعايير أساسيّة وبين بيئة تطوير اللغة DE (منفّذ اللغة أو المحرر). حيث قد يوجد للغة واحده أكثر من محرر وقد تختلف هذه المحررات فيما بينها في دعم بعض مزايا، وسنتناول المحررات الأكثر شهره في هذا الكتاب أو المزايا المشتركة بينها .

يوجد تشابه كبير في بعض الحالات بين بعض اللغات المشتركة، مثل لغات NET. مثلا ، أو نسخ معدله من اللغة تشترك معها بالمزايا الأم ، وفي هذه الحالة لن نشير إلى الأسماء المنفصلة لكل لغة مالم يوجد اختلاف يستحق الذكر ، لذلك فليعذرني القاريء إذا وجد إجحاف في ذكر بعض اللغات المهمة بالاسم ، وذلك إما لتشاركها مع غيرها ، أو للتقصير في الحديث عنها .

معايير في تقييم لغات البرمجة

المقروئية Readability وتنظيم اللغة :

وهي عامل مهم لوضوح لغة البرمجة وسهولة استخدامها. تزداد أهمية هذا العامل في النظم الكبيرة، وفي النظم التي تأخذ وقت تطوير طويل ويشارك فيها كميه كبيرة من المبرمجين.
وتعتبر الكثير من الجامعات أن وضوح النص النحوي يعتبر دليلا هاما على تطور وتنظيم اللغة وأدرجته بعضها في مقاييس بناء لغات برمجة عصرية ، فكلما قاربت لغة البرمجة الكلام البشري العادي والنص المكتوب كلما زادت في دعم عامل المقروئية ، مما يساعد القادمين الجدد على اللغة والمبتدئين على الانطلاق بسرعة أكبر .
لاحظ مثلا أن التطور من لغة الآلة إلى اللغات العليا ماهو إلا تطور في نحو اللغة ومقروئيتها ، وتبسيط لمفاهيم كتابه العبارات البرمجيّة ، حتى في هذه الأيام بدأنا نشاهد طرق تقنية جديدة لكتابه الشفرة مثل مناهج UML و مرمزات سكربت تعتمد على الصورة وتستخدم الأشكال بشكل تفاعلي لتبسيط عمليه التكويد ، بحيث تبني برنامج متكامل بواسطة الفأرة –(ماعدا البيانات) . الهدف من ذلك هو درجة أعلى من التفاعلية مع فكر المستخدم. ...
من اللغات التي تبدو فيها خاصية المقروئية هي لغة Pascal، حيث تملك نموذج نحوي جميل يحوي كميه أكبر من الحشو لنحصل على شفرة مقروءة جيدا ، ونظرا لتنظيم الباسكال ونحوها الأنيق فإنها تعتمد في الكثير من الجامعات كبداية لابد منها لتدريس لغات البرمجة .
عندما صممت الباسكال كان الحديث عن لغة ذات مقروئية عالية بهدف تسهيل عملية التكويد .

Eiffel كذلك تملك نحو واضح وبسيط ومقروء بشكل ممتاز ،

Smalltalk قريبه من الاثنين أيضا ، البعض يراها غريبة بشكل جميل والبعض يعتبرها ذات مقروئية عالية ، الذي أعجبني في Smalltalk أن شفرتها واضحة وقريبه إلى الكلام العادي ، فإذا أردنا أن نقول بالإنكليزية :"*Jack pass the ball to Jill*" فإنها ستكتب في Smalltalk : "*Jack passtheballto: Jill*" أو "*Jack pass : theball to: Jill*" .لقد صممت لتكون سهلة التعلم والاستخدام .

Python كذلك تعتبر لغة برمجة أكاديمية سهلة القراءة والفهم والتعلم –حتى على مستوى التعلم كأول لغة برمجة– وتتجنب الغموض وتتميز برتابتها الشبيهة برتابة لغة الباسكل .
خذ مثلا لغة البرمجة lisp: إن lisp تحاول أن توفر كل ما تقدمه بقيه لغات البرمجة من ميزات لتكون موجهه إلى جميع الناس (وهذا برأيي لا يجعل لها طابعها خاص بها)
وتعتبر من أسوأ اللغات في المقروئية ربما بسبب إسرافها الغير طبيعي في استخدام الرموز والأقواس وبطريقة غير مفهومه أبداً بالنسبة للوافدين الجدد على هذه اللغة

خذ على سبيل المثال :

Python	y = m*x + b
LISP	(SETQ y (+ (* m x) b))

وكما ترى فالفرق واضح بين هاتين اللغتين، وتعتبر اللغة lisp لغة صعبه جداً للقراءة والتعلم .

C++ تملك بنية نحويه معقده ، بلا شك ليست مثل Lisp ، ولكنها تبدو كأنها لغة من عصور قديمة لا يمكنك دائما فهما مهما بدت السطور مألوفة لك ، تركز C++ بشكل أساسي على الشفرة المختصرة وتحاول أن تكون عمليه قدر الإمكان لذلك من المألوف أن تجد رموز وعبارات مختصرة في شفرتها أكثر من وجود كلمات مفهومه ، لاحظ استخدام الأقواس {} بدل Begin و End في باسكال مثلا ، أو لاحظ استخدام المعاملات (&& أو | |) بدل (And و Or).
أو حتى دوال الدخل/خرج مثل <<cin و >>cout الخ .. كما أنك ستجد عده طرق لكتابه نفس البلوك من البرنامج ، مما يربك القادمين الجدد على اللغة ويصعب عليهم فهم شفرة غيرهم . ربما يقول البعض أن ذلك يعتبر مرونة عالية..... أقول لكل هؤلاء :
أن تقوم بأكثر من شيء بنفس الطريقة أفضل بكثير من أن تقوم بالشيء نفسه بأكثر من طريقة.
إن هذا الاستخدام المفرط للرموز والعبارات البديلة والمختصرة يبعد اللغة عن الاقتراب من اللغة البشرية المحكية ويجعلها لغة **ترميزية** أكثر من كونها لغة بشرية .
مثلا النسخة الحديثة من C++ والمسماة C++/CLI والتي صدرت حديثا كتمثيل كلي لدعم.NET بعد MC++ (Managed C++) ((والتي كانت أول .net. Visual C++ تصدر)) . تحوي تغييرات كثيرة تماشيا مع المواصفات العصرية ولاسيما المقروئية والتنظيم .
لاحظ مثلا هذا الكود البسيط لبرنامج HelloWorld مكتوب بـ C++/CLI:

```
using namespace System;

int main()
{
Console::WriteLine("Hello World");

return 0;
}
```

إنه قريب جدا من كود الC++ العادية، ولكن أول مالفت نظري فيه استخدام الداله Write Line الموجودة داخل الفئة Console . لاحظ وضوح هذا الأمر وطوله مقارنه بتعليمه cout مثلا ، حتى أنه أوضح وأطول من Pascal التي تكتب Writeln .
يمكن استخدام الأمر القديم Cout ولكن لاينصح بذلك لأنه لن يستطيع التعامل مع بعض الأنواع الجديدة في الدوت نيت ، لذلك يستخدم للتوافقية الإرجاعية مع الشفرات القديمة بالدرجة الأساسيّة .
ما قصدته من هذا أن الشعور بأهمية المقروئية العالية غدا أمرا بديهيا في وقتنا الحاضر ويتنامى الاهتمام مع التطور في تقنية المعلومات ، وهذا ما يفسر إتجاه مطورو C++ نحو كود أكثر مقروئية .

Java تملك نحو مشابه للغة C++ ، ولكن أفضل منه قليلا .

C# لغة متطورة وعصرية ، اتجهت أكثر لمراعاة تنظيم الشفرة و قواعد النحو أكثر من الاختصار في الشفرة الذي كان سائدا في C++ . كما أنها أصبحت تلتزم أكثر بالقواعد وتكتب الشفرة بطريقة واحده، لاحظ مثلا القيود التي فرضتها على قلب الأنواع والتي كانت C++ السابقة تتساهل معها: حيث لا يوجد تحويل مباشر من Int إلى Boolean في C# وبالتالي العبارة التالية خطـــأ "if (1)".
كما أنه لا يوجد في السي شارب ملفات رأسيه و لا استخدام للماكرو في ظل وجود preprocessor support for conditional code أي يتم التحقق من الشروط قبل الترجمة.
و هذا يجعل اللغة أسهل، كما أنه يزيد سرعة المترجم، ويجعل من السهولة أكثر فهم شفرة الـ C#. وكأننا نلاحظ أن C# كلغة مطوره عن ثلاث لغات (بنية java وهدف وهيكل Delphi ونحو C++) C فهمت أهميه زيادة المقروئية والتنظيم في نحو اللغة.
لن تجد في C# مثلا استخدام لـ "::" أو "->" .

VB.net تملك مقروئية ممتازة هي الأخرى . وانا أرى انها أكبر من مقروئية C# وأقرب إلى مقروئية Delphi .. كما ان مايكروسوفت توجّة Visual Basic .net نحو مفاهيم مشابهة للمقروئية و تبسيط عملية التكويد أكثر من C# .
لاأعرف إذا كانت أفضل من Delphi بالمقروئية ولكني أرجح الأخيرة أكثر(سيتهمني الآلاف الآن بالتحيز إلى دلفي ، ويقولون مبرمجو الدلفي غير منصفون !! يا جماعة هذا رأيي الشخصي ولن يغير في الحقيقة شيء ، انا أرى دلفي هي الأكثر مقروئية من وجهة نظري) .

Perl أيضا تعتبر لغة صعبه القراءة بشكل ملحوظ ، رغم أنها يمكن أن تكتب بطريقة أنيقة ولكن الانطباع العام أنها لغة ضعيفة المقروئية .

- ســـرعة التكويـد :

ولكن علينا الملاحظه أن المقروئية والتنظيم ربما يؤثران على سرعة التكويد ، وبالتالي لغة مثل C++ ذات مقروئية ضعيفة لأنها تركز على الكود المختصر وتستخدم عبارات مرمزة ، ولكن ذلك يزيد من سرعة التكويد ويسمح للمبرمج بالوصول إلى نتيجة الشفرة نفسها بسرعة أكبر ولاسيما عندما يكون العمل كله كتابه شفرة . (*طبعا لا أعني أن إنتاجيه C++ عالية "راجع بند الإنتاجية"*) .
قابلت ذات مره خبير C++ وتناقشت معه في عدة مواضيع ، ولاحظت أنه لم يعر موضوع المقروئية اهتماما واعتبر أن اختصار الكود مفيد بالنسبة له ويجعله ينجز الأعمال بسرعة أكبر .

إن التكويد السريع يزيد سرعة تطوير بعض أنواع البرامج. كتلك المعتمدة على كتابه الشفرة أكثر من اعتمادها على أدوات ومكتبات اللغة، مثلا عند كتابه الأصناف، مما يعطي عاملا مهم لنأخذه بالحسبان أن اللغات التي تملك شفرة مختصرة، ولو كانت أقل قابليه للفهم والقراءة فهي ربما تعطي سرعة إضافية في برمجة وكتابه شفرة المشروع لأول مره.

وهذا شيء رائع ويعطي للمبرمج القابلية على تجريب أفكار برمجيّة بسرعة أكبر وبلا شك له عده فوائد ، ولكني لازلت منحازا إلى جانب المقروئية ولاسيما إذا كنا نحقق سرعة التكويد هذه على حساب المقروئية .

تنظيم اللغة يعني التقيد الصارم بالقوانين مما يحمي المطور من أخطاء كثيرة مستقبلا كأخطاء قلب الأنماط ، ويزيد من سهولة تطوير منتج واحد من قبل فريق ضخم بسبب تحسين قدرتهم على فهم شفرة بعضهم بعضا .

الكثير من اللغات مثل Python تحاول الاختصار والتسريع في التكويد كذلك، حيث لاداعي لتعريف المتحولات والثوابت بنفس الطريقة التقليده المتبعة في اللغات الأخرى .

Smalltalk تعتبر لغة مختصرة كذلك وتركز على مبدأ تصغير الشفرة . فهي مثلا لاتحتاج مساحات من الشفرة لتعريف المتغيرات كما في اللغات الأخرى ، إنها ليست بحاجه لـ Type من أجل تعريف كافه أنواع المعطيات أو الصفوف والكائنات.

أمن اللغة Language Safe :

لا أقصد أي من مفاهيم الأمن المتعلقة بالاختراق أو حماية البرامج.
ما أقصده هنا هو تساهل اللغة مع أخطاء المستخدم والسماح له بإنتاج تطبيقات غير مستقرة . اللغة الآمنة تبعد المستخدم عن الخطر قدر المستطاع، وتوجههُ إلى إتباع الطرق الآمنة أكثر في الحل، وتعطيه أدوية ذات أعراض جانبيه قليلة قدر المستطاع.
لاحظ مثلا الجافا .
جافا جيده جدا من هذه الناحية، تسمح لك بتنفيذ الأشياء من وجهه النظر الآمنة فهي تجبرك مثلا على التعامل مع الأغراض في بنيتها غرضية التوجه الصرفة حتى في برامجك الصغيرة ، مما يخدمك في المستقبل في حال توسيع برنامجك . حيث تكون قد بنيت قاعدة سليمة وفق قوانين البرمجة الغرضية التي تسهل إعادة استخدام الشفرة وتوسيعها مما يجعلك تزيد حجوم برامجك بثبات وإستقرارية بعيدا عن الأخطاء التي كان من الممكن أن تقع بها لولا أن أجبرتك بالعمل على الأصول من البداية والتي ربما كلفتك إنتاج برامج غير مستقرة .. ،
كما أنها مثلا تحميك من الوقوع في أخطاء المؤشرات الحسابية ولاتسمح لك باستخدامها أو تعريفها مع أنها تعتمد عليها داخليا وتستفيد من ميزاتها .
نعم تعتبر المؤشرات أدوات فائقة القوه ورائعة للتحكم والسرعة، ولكن حديثا تم اعتبار المؤشرات أنماط غير آمنه، وهذا ما يفسر التخوف الحالي من دعم المؤشرات .
مثلا في بيئة .NET لا يوجد دعم كامل للمؤشرات كما كان في Win32 ، حتى الآن الدعم يتم بطريقة غير مباشره وبحالات خاصة في معظم الأحيان . (ربما C++/CLI هي اللغة الوحيدة التي ستبقى مع المؤشرات ، مع أني لم أستطع جمع معلومات كافية عنها بعد ، لكن يقول أصحابها بتأكيد اخبار الدعم الجيد للمؤشرات فيها) .
كل من VB.Net و C# و .NET Delphi لا يدعم المؤشرات ، ولكن توجد التفافات برمجيّة تسمح لك بالقيام بأعمال المؤشرات بطرق آمنه أكثر ، بشكل عام باستخدام الأغراض ، أو مثلا بالعمل تحت نمط Unsafe الذي يتيح استخدام المؤشرات . ولكن نادرا ما يستخدم إلا إذا أردنا معالجه شفرة قديمة وتستخدم بدل ذلك المراجع reference وهي مشابهه لما هي موجود في الـ C++ لكن مع تخطي بعض القصور .

مع أن المؤشرات قويه جدا وسريعة جدا ومهمة جدا ، قامت لغة عريقة مثل Java بالاستغناء عنها ، لصالح مبرمج الجافا .

كما أن جافا تسمح بالتمرير_بواسطة_القيمة فقط (pass-by-value) . وبالتالي تجعل جافا شفرتك أقل عمومية وأكثر أمانا فهي لا تسمح بالتمرير بواسطة المرجع (pass-by-reference). اللغة C القديمة تشاركها هذه الخاصة ، لكن كل من C ++ و باسكال يختلفان معها.

– حتى الإدارة الآلية للذاكرة ومجمع النفايات المتقدم في جافا تعتبر مزايا مهمة من ناحية أمن اللغة ، فهي تحمي المستخدم من الوقوع في أخطاء إدارة الذاكرة . بلا شك لها سلبياتها هي أيضا، فهي تفقد اللغة بعض مزايا القوه والتحكم وقد تسبب لها بطىء إضافي في التنفيذ، ولكن النتيجة أن تطبيقات مبرمجي جافا خالية من أخطاء المستخدم في إدارة الذاكرة إلى مدى بعيد مقارنه بمبرمجي C أو C++.

دلفي تملك نظره مشابهه ولكنها مختلفة عمليا عن جافا. فهي تعتمد مبدأ تغليف احتياجاتك من نظام التشغيل بأدوات خاصة – (تذكر الكميه الكبيرة من الأدوات الموجودة في شريط أدوات دلفي) – وهذا يجعلك تحل المسألة من وجهه نظر دلفي نفسها وبطريقتها الخاصة،مما يزيد من أمن برنامجك ضد الأخطاء التي قد ترتكبها ، أو ضد نقص في معالجتك لكامل الحالات البرمجيّة . كما أن دلفي استفادت من النحو الدقيق للغة ObjectPascal . لاحظ مثلا القيود الصارمة التي تفرضها على نسب وقلب المعاملات. الحالة الإفتراضية أنك ستبرمج على دلفي بالطريقة التي تجعلك هي تبرمج بها ، آخذه بيدك بعيدا عن العديد من المخاطر ، مالم تقم أنت بطلب دخول أكثر تعمقا ،
إنها ليست كالجافا تماما ، فهي تسمح لك بالدخول في نهاية المطاف ولكنها تجعل الحلول الآمنة هي الحلول الإفتراضية . وعندما تقرر أن لا تعمل وفق هذه الحلول عليك أن تتحمل مسؤولية شفرتك.

C# قفزت قفزه مهمة في هذا المجال إذا ما قارناها بC++ القديمة فهي تولد شفرة آمنه أكثر منها ، لن تستطيع في C# إنجاز تحويل غير آمن مثل تحويل قيمه رقميه (Double) إلى قيمه منطقيه (Boolean) ، كل الأنماط ذات القيم (Value Types) يتم تهيئتها إلى صفر وكل الأنماط ذات المرجع يتم تهيئتها إلى Null أوتوماتيكيا من المترجم .
ملاحظه: على كل حال لاتزال NET. مشوبة بكثير من الأخطاء والثغرات الأمنية ، ونحن عند وعد مايكروسوفت بحل كل ذلك .

VB.Net شبيهة بإختها C# إيضا ولكنها حسب إعتقادي أكثر أمنا منها ،
السبب هو طبيعة VB.Net ذات الوضوح العالي ، وهذا يضاعف ميزة الأمن بشكل أسي ، فلا تتوقع خروقات أمنية كبيرة في ظل عملك في بيئة مثل Visual Basic .

C++ بلا شك هي خاسر مهم لهذه الناحية ، ودون أدنى تردد أكرر ما قيل سابقا "للقوه ضريبتها" وعلى C أن تدفع ثمن التفصيل الكبير واليدوية اللتان تجبر مستخدمي السي على إتباعهما . ما أراه في C++ أنها عكس جافا تماما ، فهي تجبرك على فهم المؤشرات والتعامل معها عندما تريد

كتابه برامج مصفوفات مثلا ؟؟؟؟ لاحظ الفرق ، لاتريد مؤشرات ولكنها تقحمها أمام عينيك دائما .

أنا لا أقول أن كل استخدام للمؤشرات سيولد أخطاء برمجيّة من المستخدم ، ولكن على الأقل نسبه من المستخدمين (غير الخبراء) سيقعون بالكثير من الأخطاء ويغرقون ببعض تفاصيل إدارة الذاكرة إذا قررو التعامل مع مشاريع من الحجم الكبير .

كما أن C++ تزودك بكافه أدوات القوه دون أدنى قيد و شرط .. تخيل أن يستطيع الأطفال في المنزل الوصول إلى السكاكين والأسلحة والعبث بها دون رقيب أو حسيب.. النتيجة ستكون كارثية مالم يكن المبرمج على قدر المسؤولية .

انطباعي عن لغة C++ أنها لغة الخبراء ، ولا أستطيع تخيل تطبيقات مستخدمي C++ لازالوا في بداية الطريق من دون أخطاء . فإذا أخذنا أن متوسط مستخدمي C++ لن يكون من الخبراء عندها سأصف C++ أنها لغة غير آمنة .

لغة Perl لغة جميلة وجيدة، ولكنها ببساطه تسمح لك أن تكتب شفرة بغاية السوء بنفس البساطة التي تكتب بها شفرة أنيقة، وهي مثل C++ يمكن أن تتساهل مع أخطاء المستخدم ويمكن أن تسمح له بإنتاج شفرة غير آمنه.

يقول (Larry Wall) منتج Perl: " The very fact that it's possible to write messy programs in Perl is also what makes it possible to write programs that are cleaner in Perl than they could ever be in a language that attempts to enforce cleanliness ".

Python : لغة متساهلة كثيراً مع أخطاء المبرمج . لدرجه أنها لن تظهر لك أي تنبيه في حال لم يتم تعريف التابع أو الحقل أو المتحول ، أو عند تمرير بارامتر غير صحيح للتابع أو أي شيء آخر وبذلك فإنه لا سبيل لاكتشاف الخطأ إلا بالوقوع به عند الأمر بتنفيذ البرنامج .
مع العلم أن بايثون يؤمن معالجة جيدة جداً للاستثناءات وذلك في زمن التنفيذ.

Lisp لا تتسامح أبداً مع أخطاء المبرمج وسوف يظهر لنا تنبيه لجميع الأخطاء النحوية على خلاف بايثون وذلك في زمني التصميم والتنفيذ .

وصـول منخفض المستوى low-level access :

يقصد به قدرة لغة البرمجة على إنجاز وظائف غير تقليديه مع العتاد أو قدرتها على الوصول بمستوى منخفض إلى تفاصيل نظام التشغيل وتفاصيل المشغلات والمنافذ والمكونات .

أمور كثيرة تلعب دورا في تحديد قدرة لغة البرمجة على الوصول المنخفض . ومنها دعم أساليب الولوج المنخفض مثل دعم كتابه شفرات بلغة الأسمبلي ضمن اللغة (الأسمبلي المضمن Inline Assembler)، كذلك الدعم الجيد للمؤشرات لا يقل أهميه عن سابقه، بالإضافة إلى تكوين اللغة بشكل عام وآليتها بتنفيذ المهام، هل تعتمد التفصيل والشمولية، أم أنها تدعم التقنيات من مستوى المستخدم نفسه، أو ما يسمى القشرة.

أيضا مره أخرى تفوز ال C/C++ ، ..
طبعا تملك السي وصولا منخفض المستوى إلى مناهج ودقائق العتاد ! ، أصلا نظام التشغيل مبني عليها بشكل شبه كامل (*بعض الترقيعات بلغة Assimply لاتضر أحيانا*) .

الوصول منخفض المستوى في كثير من الأحيان ينظر آلية أنه أهم عوامل قوه لغة البرمجة ، لست هنا بصدد مناقشه ذلك ، فأنا ميال لعوامل أخرى مثل Reusability و Extensibility أكثر منه ، ولكن رأيت أن ذلك يستحق الذكر والأخذ بعين الحسبان .
و++C تملك كل العوامل التي تجعلها لغة مناسبة لتنفيذ وصول منخفض والمنافسة معها في هذا المجال تنحسر كثيرا .

جافا لا تملك دخولا منخفض المستوى، فلا تستطيع أن تتعامل مع مواقع ذاكرة، أو تقرأ من منافذ الخ .. ربما منع ذلك من أجل السرية، ولكنه بالنهاية غير مسموح ولايمكن تحقيقه، لذلك لايمكن أن تفكر بأن تكتب مشغلات أجهزه في جافا (device driver)، ربما يمكن الالتفاف على كل ذلك باستخدام مناهج محليه ، ولكن تقنيا لا يمكني اعتبار أن ذلك يندرج تحت إطار الوصول المنخفض

دلفي أفضل بكثير من جافا ، فهي تدعم التقنيات السابقة الذكر بشكل ممتاز، يمكنك كتابه أجزاء بلغة الأسمبلي أو استخدام المؤشرات بالحدود الكاملة ، يمكنك الوصول إلى مختلف المنافذ وتحقيق كثير من الأمور التي لم تعتد عليها ، ولكن للأمانة العلمية نقول أن لغة الدلفي مختلفة في تكوينها عن لغة ++C وأنها ليست جديرة أن تصمم برامج مشغلات تقارن ببرامج ++C . وإنما ما تمنحه من الوصول المنخفض ينصب أساسا لخدمه المبرمج في بعض الحالات .

وهدفه شحن الدلفي بجرعة إضافية من القوه التي قد يضطر المستخدم إلى كشف الغطاء عنها أحيانا لتنفيذ وصولات محدده . وعدنا إلى نفس المقولة السابقة ، دلفي ليست اللغة الأقوى ولكنها لغة قويه بشكل كافي .
وتختلف Delphi.NET كثيرا عن Delphi win32 التقليدية في هذا المجال ، حيث خسرت جولات مقارنه معها وربحت اخرى .... وربما كان حري بي أن أذكر Delphi .Net بشكل منفصل عن Delphi32 .
حيث لم يعد يوجد دعم للأسمبلي المضمن في النسخ الجديدة ، كما خسرت الكثير من دعم المؤشرات السابق .
ولكنها ربحت العديد من المزايا المتوفرة من منصة net. والتي أكسبتها وصول رسمي ومنظم مثلها مثل غيرها ك #C و VB.NET إلى أصناف الNet. المتوفرة .. وللصراحة لم يعتد مبرمجوا الدلفي على هذة المساواة مع منتجات مايكروسوفت وعلى أنظمة مايكروسوفت نفسها !!.

لغات أخرى مثل الـ Python تملك قصورا واضحا هنا أيضا . من الأفضل استخدام البايثون لإنشاء التطبيقات الخدمية للمستثمر بمختلف أنواعها مع الأخذ بعين الاعتبار الابتعاد عن البرامج التي تتعامل مع بنية الحاسب بشكل مباشر فلغة البايثون لغة عالية المستوى وغير مناسبة للتعامل مع ركائز النظام والحاسب .(برامج تعاريف الأجهزة مثلاً) .

VB.net ليست سيئة في هذا المجال ، لكني أقرأ في فنجان مايكروسوفت أنها تريد توجية مثل هذة الميزات نحو #C أكثر من VB.NET . وهذا سر تخوفي من التباعد المستقبلي بين اللغتين اللتان ولدتا متشابهتين .

#C الآن ليست أفضل بكثير من VB.Net ولكن يتوقع ان تكون اللغة الأكثر إهتماما بهذة الجوانب بين اللغتين السابقتين مع مرور الزمن .

السرعة Speed :

يعتبر هذا المعيار من أهم معايير تقيم لغات البرمجة، ويعتبر من المعايير البديهية التي تخطر على بال أي منا عند الحديث عن لغات البرمجة . ودائما عند الوصول في الحديث إلى مقارنات بين اللغات الجميع يهتف قائلا ماذا عن سرعة لغة البرمجة هذه أو تلك .

السرعة مفهوم أكبر قليلا مما يبدو عليه للوهلة الأولى، سأحاول أن أشرح نظرتي لمفهوم السرعة.
تقسم السرعة إلى قسمين :

- سرعة تطوير التطبيق ((الإنتاجية))
- سرعة تنفيذ التطبيق

أولاً : سرعة تطوير التطبيق "الإنتاجية" Productivity :

إذا أردنا تعريف الإنتاجية كمعادله رياضيه فالإنتاجية هي ناتج قسمه النتيجة\الزمن . وكلما استطعنا تحقيق نتائج أفضل بزمن أقل كلما زادت إنتاجيتنا الصرفة . إنتاجية لغة برمجة ما تزداد بازدياد قدرتها على بناء تطبيق جيد ومستقر في زمن أقل، ويدخل في تقييم الإنتاجية كل من

- **جوده التطبيق** (سرعة وثوقية إستقرارية حجم محمولية الخ ...)
- **زمن التنفيذ**

إذا أردنا سرد العوامل ألمؤثره على إنتاجيه لغة ما بناء على ما سبق ، رما كانت النتيجة كالتالي :

١- السهولة والبساطة Simple & Easy :

لماذا علينا أن نختار لغة برمجة قويه جدا غير قادرة على بناء التطبيق الذي نريده بسرعة كافيه ؟
قوه لغة البرمجة كثيرا ما تتولد نتيجة التفصيل الكبير في الدوال والمكتبات ، حيث تتيح لنا هذه اللغات العمل بمستوى أدنى لنحصل على جرعه من التفصيل الشديد وبالتالي التحكم الكامل .
وعندها سنجد حلول لأي حاله تعترضنا ، ولكن للأسف نكون قد خسرنا السهولة .
وعندها علينا أن نلاحظ أن هذه الصعوبة المرافقة للغة البرمجة القوية التي تزيد بشكل ملحوظ من خطوات العمل و تعقيده تؤثر من جانبين :

- زمن تطوير أكبر ، مما يخفض الإنتاجية .
- مزيد من الوقوع في الأخطاء مما يخفض مستوى جوده النتيجة، حيث جرت العادة في اللغات التي تتطلب شفرة طويلة ومعقده أن المستخدمين يقعون في كميه كبيرة من الأخطاء البرمجيّة ، والتي تطلب سنوات من الخبرة للتغلب عليها .

في حين أن اللغة الأسهل و الأكثر إنتاجيه تسمح للمبتدئين ببناء تطبيقات أكبر وأكثر استقرارا بسرعة كأنهم يعملون منذ سنوات ولديهم خبره جيده .

إذا كانت لغتان قادرتان على بناء تطبيق ما وإحداهما تحتاج زمن أكثر لتحقيق ذلك ، فاللغة الثانية هي الأكثر إنتاجيه وهي الأفضل لهذا المشروع لأنها توفر الوقت والجهد والمال .
الإنتاجية تعتمد مبدأ أساسي في العمل : ((إتّبع أبسط وأسهل طريق يحقق كامل الهدف))
. لاحظ كلمه "كامل الهدف" ، لا يجب أن نتبع طرق سهلة ولكنها غير قادرة على تحقيق متطلباتنا على أكمل وجه .
وهذا المبدأ مستوحى من الأفكار التي تقول . لماذا علينا أن نضيع الوقت بكتابه الشفرة ، يجب أن يضيع الوقت على التفكير بالبرنامج وتكون كتابه الشفرة أسهل ما يمكن .

الخلاصة :

لا أريد أن أقول أن القوه والإنتاجية متعاكستين ، وهذا فهم خاطيء لكلامي . ولكن ما أريد قوله أن الصعوبة والإنتاجية متعاكستين. إذا كانت لغة البرمجة التي تعتمدها تحقق القوه عن طريق الصعوبة والتعقيد فهي أقل إنتاجية. ومن الطبيعي على كل حال أن تكون اللغات التي تهتم للقوه بشكل أساسي أقل إنتاجيه من غيرها، لأنها بحاجه لتفصيل كافي. ولكن ذلك ليس حالة عامة.
تعتبر كثير من اللغات مثل جافا و C++ ضعيفة الإنتاجية. وهذه اللغات عندما بنيت تم تصميمها لتكون أكثر قوة أو أكثر محمولية مثلاً ، و أهمل إلى مدى بعيد موضوع إنتاجية اللغة ..

لغة Visual Basic تعتبر لغة سهلة ، وهي لذلك لغة عالية الإنتاجية نظرا لسهولة كتابة تطبيقات جيدة فيها . مشكلتها أنها لا تقدر على التعامل مع بعض المشاريع الكبيرة ، ولكن في حال كان المشروع مناسبا لها فإنها تعتبر ذات إنتاجيه مميزة، وهذا هو سر انتشارها الواسع. ولا أرى في ذلك عيبا أبدا .
كل لغة مناسبة لمشاريع دون أخرى ، ولكنها تخدمك بشكل أنيق في المشاريع التي تناسبها . وهذا حال VB الموسومة بإنتاجية تستحق التقدير .

٢- قابليه إعادة الاستخدام Reusability.

وهي تمثل قدرة اللغة على الاستفادة من موارد سابقه عند الحاجة لها.
مثلا : الوراثة من الأصناف تسمح لنا بإعادة استخدام شفرة صنف سابق كنا قد أعددناها في الصنف الجديد دون الحاجة لإعادة كتابتها من جديد مما يضاعف سرعة العمل ويختصر تكرار الشفرة ويسمح لنا ببناء نموذج متطور عن سابقه بزمن قياسي .

لغات OOP بشكل عام داعم أساسي لهذه التقنية ، تختلف بدعمها لها بحسب نوع وطريقة تمثيلها لمفهوم الـ OOP . *لمزيد من المعلومات راجع بند الـ OOP* .

٣- التوسعية Extendibility.

وهي القدرة على التخاطب مع نظم وتقنيات مختلفة مما يسمح بالإستفادة من الميزات المتوفرة بكل نظام بدلا من إعادة بنائها، كما يسمح دعم التقنيات المختلفة بتحقيق أعمال معقده ومتطورة بمستوى أعلى من السرعة والأمن.

سأشرح ذلك :

- **القدرة على التخاطب مع نظم مختلفة:**

تسمح بالتكامل بين هذه النظم ودمج الميزات المهمة في كل نظام . لا يجب أن تكون لغة البرمجة منطوية على نفسها وغير قادرة على التواصل مع لغات ونظم أخرى.وكثيرا ما تحتاج بناء برامج متكاملة تغطي عده جوانب تقنية ، ولكل جانب من هذه الجوانب نظم عمليه وقياسية قادرة على التعامل معه مثل نظم إدارة قواعد البيانات وخدمات الويب و مشغلات الملتيميديا و نظام التشغيل الخ ....

مثلا تملك دلفي دعم خاص في شريط أدواتها لقواعد بيانات Dbase ، مما يعطي إمكانية هائلة لإدارة هذا النوع من قواعد البيانات ، حيث يستثمر تطبيقنا ميزات Dbase بشكل أمثل ويحقق لنا أفضل أداء تقدر Dbase عليه ، كما أنه يسمح لنا بالتحكم الكامل بنظام إدارة قواعد البيانات العلائقية هذا ، ويضيف إمكانيات الإدارة التي لا تستطيع أدوات أخرى إضافتها دون أطنان من الشفرة .

كل هذه الميزات يمكن تحقيقها يدويا دون الحاجة للتكامل مع نظام Dbase ولكنها ستأخذ شهور من البرمجة وستكلفنا كميه كبيرة من الأخطاء أي وببساطه ستسلب الإنتاجية العالية منا وستصبح خيارا سيئا عندها .

الإنتاجية في هذه الحالة تعني أن المبتدئ الجديد على اللغة يستطيع بناء برنامج قواعد بيانات يعتمد على Dbase ويحقق أداء أمثليا وأقل نسبه ممكنه من الأخطاء خلال زمن تطوير صغير نسبيا.

والخبير يستطيع تحقيق نتائج تامة من الجودة والموثوقية بزمن قياسي . وبأقل تكلفه ممكنه للجهد والمال .

كذلك يمكن لدلفي التخاطب مع لغات برمجة أخرى . فهي تتشارك مع لغة C_Builder بمكتبه المكونات ,(VCL) كما أنها يمكنها استخدام وبناء مكتبات ActiveX (ملفات OCX) التي تدعمها لغات Visual Studio مثلا.

لاحظ مثلا تكامل لغات الـ .NET. هذا ما أسميه توسعيه حقيقية ، حيث أن القدرة على التخاطب بين اللغات المختلفة يجعل من السهل تقسيم العمل إلى أجزاء وبناء كل جزء باللغة الأكثر مناسبة له .ومن السهولة بمكان استخدام كومبونينت من لغة .NET. في لغة أخرى منها .

فكلّ لغات VS.Net تستخدم واجهه واحده، مليئة بالأدواتِ التي تُسهّل بطريقة مدهشه عمليّه تصميم البرنامج.. إن هذه الميزة تسمح لك بإنشاء تطبيقات تدخل فيها أكثر من لغة برمجة، دون أن تحتاج لفتح أكثر من واجهه.. إنّها واجهة واحده فقط لكلّ المبرمجين.

قلدت بورلاند مايكروسوفت هنا . ووضعت C#Builder و Derlphi2005 في بيئة واحدة لNet. سمتها Delphi Studio .

لغة Python مثلا تملك حوار خاص مع C++ التي كتبت بها بعض أجزاء الPython ، وتسمح Python باستخدام شفرات C++ في بعض الأماكن التي تعاني فيها Python من البطء من أجل تسريع البرنامج ، مما يعطي سرعة كبيرة في تطوير التطبيق (Python) وسرعة في تنفيذ التطبيق (C++) .وكذلك إمكانية تطوير وتعديل وحدات كتبت بلغات مثل C و C++ و Java و Fortran.
حتى لغة Perl يمكنها أن تترجم إلى ملفات C سريعة التنفيذ أكثر . كما أنها تملك تكامل جيد مع نظام التشغيل . حيث تسهل من عمليات معالجه النصوص والولوج لتوابع نظام التشغيل . كذلك الأمر كل من لغتي Smalltalk, lisp تسمح لنا بتضمين يفره بلغة الـ C .

دعم تقنيات مختلفة Many tech Support:

دعم لغة برمجة ما لتقنية يعني توفير واجهه جاهزة لاستخدام ميزات هذه التقنية بشكل مريح. مثلا دعم اللغة لتقنيات الويب ، تدعم C# تقنيات ASP.Net مما يجعلك تصمم تطبيقات ويب تفاعليه بمنتهى السهولة والمتعة ، الدعم الجاهز والمضمن مع اللغة لهذه التقنيات يسمح لك بالعمل مع التقنية من خلال واجهه مريحة جدا تغلف إمكانيات التقنية بشكل سهل ، حيث يكفي السحب والإفلات بالفأرة مثلا لتصميم واجهات ويب WYSIWYG متكاملة وضبط خصائصها دون الحاجة لبرمجة ذلك من البدء وكتابه آلاف السطور.

تقنيات مختلفة وكثيرة مثل COM, DCOM, CORBA, XML, SOAP, Web, Databases, الخ... يمكن أن تكون مدعومة بشكل جيد من لغة البرمجة مما يزيد بشكل كبير القدرة على بناء تطبيقات متطورة، ويخفض الزمن والتكلفة اللازمة لذلك.

٤- توفر مكتبه أدوات واسعة Wide Component Library:

بغض النظر عن دعم التقنيات المختلفة والتكامل مع النظم المختلفة. تحوي لغات البرمجة مكتبه أدوات تسمح لك بإنجاز المهام المتكررة وتصميم برامج مفيدة بسهولة أكبر ومجموعه أكبر من الخيارات.

أنا أؤمن بالمقولة التي تقول:

" When all you have is a hammer, everything looks like a nail. And when you have a complete tool chest, you can choose the right tool for the job instead of just pounding away"

مكتبه الأدوات هي الترجمة الفعلية للبندين السابقين، حيث يتم دعم نظم وتقنيات مختلفة بتوفير مجموعه أدوات مناسبة في مكتبه أدوات اللغة.

توفر مكتبه أدوات واسعة تغطي مجموعه كبيرة من الاحتياجات يفيد في ثلاث جوانب:

- يوفر الوقت والجهد.
- يقلل من الأخطاء ويزيد من الفاعلية، لأننا نستخدم شفرة قياسية وضعها أخصائيون وجربت حول العالم لوقت كافي.
- يسمح لنا ببناء أنواع جديدة من التطبيقات لم نكن قادرين على بنائها قبل توفر الأدوات بسبب عدم قدرتنا على إنتاج هذه الشفرة المعقدة المكونة لها.

- خلاصـه :

إنتاجيه لغة برمجة ما برأيي هي أحد عوامل الفصل التي تقرر إمكانية استخدامنا للغة البرمجة هذه أو تلك . ولاينكر أحد أهميه وجود لغة برمجة قادرة على تحقيق ما تريد في زمن أقل من غيرها.

من اللغات الأكثر إنتاجيه لغة البرمجة دلفي (Borland Delphi) التي تدعم الأفكار السابقة بشكل فريد ، ربما تكون دلفي اللغة الأشهر التي تتميز بالإنتاجية كصفه أساسيّة . فهي منتج بورلاند للتطوير السريع للتطبيقات -RAD– (Rapid Application Development) وقد صممت دلفي منذ البداية تحت راية الإنتاجية العالية. فهي رغم أنها لغة قويه جدا، ولغة تملك وصول منخفض المستوى وقدرة ممتازة على العمل تحت الحطام، استطاعت انتزاع السهولة والبساطة عن طريق فلسفه التغليف بالأدوات. حيث قام مطورو دلفي بتزويدها بكميه ضخمه من الأدوات والمكتبات القادرة على تغليف كميه غير مسبوقة من الاحتياجات البرمجيّة،

مما أخفى التعقيد والصعوبة تحت قناع الأدوات والمكتبات التي تستطيع القيام بالأمور الصعبة بجهد أقل .

والأجمل من ذلك أن مكتبه الأدوات هذه مفتوحة المصدر ومبنية بشكل كامل على دلفي نفسها ، مما أعطى مستخدمي دلفي القدرة على الإطلاع على آلية تنفيذ مكون ما وبالتالي الاستفادة منها والتعديل فيها بما يناسبهم . وبنفس الوقت ظلت دلفي قادرة على العمل بالمستوى الأدنى متى لزم الأمر .

لغة VB كذلك تعتبر ذات إنتاجيه عالية ، ولو أنها تختلف مع دلفي بفلسفة الإنتاجية ، حيث حفظت دلفي خط الرجعة وبقيت محتفظة بمخزون كافي من القوه ، أما VB اعتمدت على البساطة النحوية والسلاسة في الاستخدام ، والتكامل مع ميزات Office ونظام التشغيل .

VB.Net ضاعفت إنتاجية VB لإنها أحتفظت بالصفات الجيدة فيها ، وحفظت خط الرجعة هذه المرة . VB.Net هي برأيي الشخصي ثاني لغة بعد دلفي تملك ميزة الإنتاجية . حيث تغلبت عليها دلفي بحجم مكتبة الأدوات ، في حين تساويتا بالسلاسة
وأذكر هنا الصنف My المميز للغة VisualBasic.net . والذي أراه سيف ذهبي يؤيد الإنتاجية المهمة التي تملكها VB .

لغة C++ ذات إنتاجيه ضعيفة مقارنة بغيرها ، ورغم الكميه الكبيرة من البرامج التي بنيت على C++ إلا أن مواقع الويب والكتب ونشرات المجلات المعنية تؤكد مرارا أنها لغة ذات إنتاجيه منخفضة ، السبب أن C++ بنيت أساسا من أجل أهداف أخرى فهي تركز على القوه والسرعة وصغر حجم التطبيق ولا تعير بالغ اهتمام إلى موضوع الإنتاجية التي قد تتعارض أحيانا مع المزايا السابقة .

جافا كذلك إنتاجيتها منخفضة ، إنها مثل الـ C++ في كثير من الجوانب ولطاما تشابهت معها . وهاهي الجافا تفضل البقاء مع صديقتها C++ في الخندق المعادي للإنتاجية ، .
ويعتبر الكثيرون أن إنتاجيه جافا أفضل من إنتاجيه C++ في الكثير من الجوانب وهذا هو رأيي الشخصي كذلك .

Smalltalk تعتبر لغة ذات إنتاجيه عالية، لعده أسباب منها سهولة اللغة نحويا فهي من أقرب اللغات إلى اللغة البشرية المحكية . و سرعة تنقيح الشفرة . كذلك الاستغناء عن Type وبالتالي عدم إهدار الوقت في عمليات التعريف للمتحولات والأغراض .
وأخيرا الواجهة التصميمية المساعدة على تصميم واجهات سهلة وسريعة .

لغة Python ذات إنتاجيه جيده ، وتأخذ البرامج المكتوبة بواسطة Python وقت أقصر بحوالي ٣ – ٥ مرات من البرامج المكتوبة بواسطة جافا مثلا .

ثانيـاً : السرعة في التنفيذ Executing Speed :

وهي سرعة التطبيق الناتج من لغة البرمجة في تنفيذ التعليمات المرمزة داخله ، في كثير من الحالات تعتبر سرعة التطبيق الناتج العامل الأهم والأول للأخذ بعين الاعتبار عند البحث عن مزايا لغة برمجة ما .حيث يلزمنا في كثير من الأوقات تطبيقات أو أجزاء من تطبيقات يجب أن تنفذ بأقصى سرعة ممكنه لنحصل على أداء أمثل ، مثلا في الألعاب ثلاثية الأبعاد حيث يضطر مطورو هذه الألعاب في كثير من الأحيان الاستغناء عن بعض المؤثرات البصرية (كالاستغناء عن الظلال والإنارة وتخفيف دقه العرض) من أجل الحصول على نسبه جيده لما يسمى FPS (Frame peer second) والذي يعبر عن عدد الإطارات التي تتم معالجتها وإظهارها في الثانية الواحدة . وهذه بلا شك تتعلق بشكل مباشر بسرعة تنفيذ التعليمات ..

وعلى كل حال سرعة تنفيذ تطبيقات لغة برمجة ما يحتل مواقع متقدمة في مقارنة وتقييم أداء وأهمية هذه اللغة .

تتفاوت لغات البرمجة بشكل كبير في سرعة تنفيذ تطبيقاتها بتأثير عوامل مختلفة تتعلق بفلسفة اللغة والهدف الذي صممت من أجله ، والكثير من اللغات تبدي خصائص ومميزات مهمة أخرى على حساب السرعة .
خذ مثلا لغة الجافا ،
كذلك لغات منصة NET. التي أنخفض فيها منسوب السرعة قليلا .

لغة C/C++ من اللغات الرائدة في سرعة التنفيذ ، دعنا نقل أنها الأفضل في هذا المجال ،تحديدا لغة C التي ولدت لكي تكون الأسرع .
لغة برمجة سريعة جدا في التنفيذ مناسبة للعمل بسرعة وفعالية في حالات ومنصات مختلفة . صممت أساسا وموضوع السرعة كان أحد أهم العوامل التي روعيت وأخذت بعين الاعتبار كأولوية أساسيّة إلى جانب القوه الخارقة .
كما لاحظنا في بند الـ OOP وفي كثير من الحالات تستخدم C الطريقة الساكنة في التنميط ، وهذا الشيء يزيد بشكل فاعل من السرعة على حساب بعض الأمور الأخرى ، ما أردت توضيحه هنا أن C مستعدة للتضحية بالكثير من أجل الحصول على سرعة عالية ، وباختصار هذه اللغة سريعة بما فيه الكفاية وتطبيقاتها تشهد بذلك .

لغات NET. الحديثة مثل C# مثلا أو VB.NET للأسف تعتبر أبطأ قليلا من غيرها ، البعض يقول حوالي ٣ مرات ، والبعض من ٥-٧ مرات ذهب آخرون إلى أنها قد تكون أبطأ بمعدل ٢٠ مره في بعض الحالات .

أي أن تطبيق C# أبطأ بالتنفيذ من تطبيق C أو C++ بعده مرات على المؤكد . وهذه إشارة كبيرة على أن السرعة رغم أهميتها الفائقة في كثير من الحالات قد لاتكون العامل رقم ١ في تحديد اللغة المناسبة أكثر لحالتنا .

أنا مثلا إذا أردت شراء سيارة (وسيطة نقل) لن أقول أني أريد أقوى وسيطة نقل (شاحنه بضائع مثلا) ولن أقول أنني أريد أسرع وسيطة نقل (سيارة سباق باهظة الثمن ، غير اقتصاديه ، لايمكن التحكم بها بسهولة) ، بل سأختار السيارة المريحة والعصرية ذات الشكل الجميل والتقنيات الحديثة ، والسيارة الاقتصادية بالاستهلاك والسهلة القيادة ... ؟؟
ربما يختار مزارع شاحنه ، ويختار أبن مسئول سيارة السباق ... ولكل منا مذهب وكل منا محق فيما يرى .

وهذا ما يبرر استغناء مايكروسوفت عن بعض السرعة في منتجها من أجل تقنيات أخرى مهمة .

لغة Pascal / Delphi تعتبر لغة سريعة إلى حد ما ، إنها ليست أسرع من C++ . ولا تحاول أن تكون كذلك . لكن Delphi-32 أسرع من نسخ لغات .NET. (أسرع كذلك من Delphi8.net مثلا) .
دلفي ليست الخيار الأفضل إذا كنت تبحث عن السرعة فقط ، ولكنها خيار سريع على كل حال ، . يوجد ألعاب ثلاثية الأبعاد مبنية على دلفي بالكامل وتعمل بسرعة جيده وتقنيات مقبولة تماما . كما أنها توفر سرعة خاصة في بعض الحالات مثل تطبيقات قواعد البيانات وتطبيقات الشبكات . دلفي ليست مع المجموعة **فائقة السرعة** ولكنها حتما مع المجموعة **السريعة** .

Smalltalk سريع في التنفيذ ، يمكن أن تنفذ كتل برمجيّة أسرع عده مرات من تنفيذها في لغات شائعة .
Perl تعتبر بطيئة ، حيث تستغرق برامجها فتره حتى تبدأ حيث يجب تحميل مفسر ضخم قبل البدء بتنفيذ أول سطر كتبه المستخدم ، وعلاوة على ذلك إذا كان البرنامج يستخدم مكتبات محمله بشكل ديناميكي فهنا يوجد تأخير إضافي في زمن التشغيل لقراءة المكتبات من القرص .

جافا لغة بطيئة لأنها تعمل على virtual machine وهي فكره قديمة استخدمت في UCSD Pascal ، Prolog وبعض اللغات الأخرى . تتولى الآلة الافتراضية عمليه التفسير لكل بنية تشغل جافا ، مما يعطي الاستقلالية في التعامل ، وهو سر المحمولية العالية التي تتمتع بها جافا .

ولكن بالمقابل فإن أي مرحلة وسيطية أو طبقه تفسير إضافية ستضيف عمل إضافيا وتسبب بطء إضافي .

يقول Jan Newmarch (Distributed Information Laboratory) في جامعه University of Canberra : إن برنامج مبني على جافا قد يعمل أبطأ ٢٠ مره من برنامج C++ . ولكنه يعقب أن ذلك ليس دائما يشكل مشكله لأنه في برامج ويندوز سيتم تنفيذ كميه كبيرة من شفرة بناء النوافذ المعمولة على C وبالتالي تختلف النسبة .

قرأت في كتاب C++ Programming How To مايلي :

"تعمل C++ بسرعة فائقة وهي في الحقيقة أسرع من جافا بحوالي ١٠ – ٢٠ مره . تعمل جافا ببطء شديد لأنها لغة byte–code–interpreted language وتعمل فوق منصة آله افتراضيه . ستعمل جافا بسرعة أكبر مع مترجم JIT (Just–In–Time) compiler)) ولكنها رغم ذلك ستبقى أبطأ من C++ . وسيبقى تطبيق C++ محسّن (optimized) أسرع ٣ إلى ٤ مرات من جافا مع مترجم JIT ."

ليست Java الأسوأ دائما في موضوع السرعة . لغة Python أبطأ من الجافا بشكل عام .

وبكل تأكيد مفهوم البطء والسرعة يختلف باختلاف حالات كثيرة ويتأثر بعدد من العوامل ، وقد تكون لغة أسرع من أخرى في حاله وأبطأ منها في حاله ، وهذا يتطلب دراسة تفصيلية لسنا في صدد دراستها في هذا القسم .

حجم التطبيق Application Size :

قدرة لغة البرمجة على إنتاج تطبيقات صغيره الحجم سريعة التنفيذ كان ولازال من أهم معايير تقييم اللغة ، حجم التطبيقات التي تنتجها اللغة يلعب دورا حاسما في كثير من الحالات . ما أروعها لغة البرمجة التي تشفر الكود الذي تكتبه على شكل برنامج صغير ، يسهل عليك نسخه ونقله ، إرساله بالبريد ، نشره على إنترنت ، يوفر بمساحه التخزين ويزيد بالمحمولية .

حجم تطبيق مكتوب بـ C/C++ يعتبر صغير جدا .
اللغات التي تعتمد على منصة ، قد تكون حجوم تطبيقاتها صغيرها ، ولكن لاشيء يضمن لنا توفر نسخه من المنصة عند المستخدم مما يجبرنا في كثير من الحالات لتضمين نسخه من المنصة مع مشروعنا ، على الأقل أول مشروع يحتاج لتنصيب المنصة وستعمل بقيه المشاريع دون مشكله عليها نفسها . وبالتالي قد يعتبر البعض أن حجم المنصة سيضاف إلى حجم التطبيق وهذا يجعل لغات البرمجة التي تعتمد على المنصة ذات حجم ملف كبير نسبيا .

وهذا موضوع خلاف ، فإذا ضمنت المنصة مع نظم التشغيل سيعتبر حجم التطبيق صغيرا ، وإذا لم تضمن المنصة وجب عليك نقل المنصة مرة على الأقل . بالنسبة لمدير تسويق ناجح لن يغامر بتوزيع منتج دون نشر المنصة مع القرص المضغوط ، وبالتالي حجم كبير ،
وبالنسبة لملفات يتم تحميلها على الإنترنت ، يكفي تحميل الملف دون المنصة في معظم الحالات وبالتالي حجم صغير.

اللغات التي تنشر مكتبات Runtime مع الملف التنفيذي يمكن مناقشتها بنفس المبدأ تماما مثل لغات المنصة ، تحتاج لملفات الـ Runtime على الأقل أول مره ، ولاشيء يضمن توفر نسخ منها عند المستخدم أو توافق هذه النسخ مع الإصدارات اللاحقة .

Smalltalk حجم تطبيقها يتراوح من ٣/١ إلى ٢/١ حجم التطبيق نفسه في اللغات الشائعة

دلفي ذات حجم ملفات أكبر من C++ بالحالة العادية ، يمكن خفض حجم الملفات بحيله معروفه في عالم البرمجة وذلك بالاعتماد على توابع API بدلا من بناء كل شيء من الصفر ، ملف دلفي لا يحوي سوى زر واحد سيكون أكبر بكثير من نفس الملف مكتوب بـC++ ، عند زيادة حجم المشروع فإن ملف الدلفي لن يزيد بنفس النسبة وسيبقى تطبيق دلفي كبير بحجم مقبول ، ولكن لايمكننا القول أن تطبيق دلفي صغير يملك حجم صغير .

دلفي تنتج ملفات لاتراعي فيها الحجم كأولوية ، وفي حال لم يكن تطبيقك كبيرا ، يتوقع عندها ان يكون حجم الملف كبير بالنسبة لحجم المشروع .
وبالتالي من أجل إنتاج ملفات Console صغيره جدا لا تعتبر دلفي مناسبة .
في حين أن حجم الملف لايلفت النظر في المشاريع المتوسطة والكبيرة .

دعم المجاري المتعددة Multi Threading :

نقصد بذلك إمكانية إنشاء مجرى معالجه جديد منفصل عن مجرى جسم البرنامج الأساسي ، مما ينتج سرعة إضافية نتيجة العمل المتزامن لأكثر من بلوك برمجي . والأهم من ذلك يتم تنفيذ المجرى الجديد بغض النظر عن توقفات المجرى القديم ، فإذا كان تطبيقنا يعمل على مجرى واحد وحدثت حاله عدم استجابة بسبب انتظار رد من طرفيه ما فإن برنامجنا كله سيتوقف عن الاستجابة ، حتى الواجهة الرسومية ستتأثر بذلك وسنحصل على حاله تجمد (Freeze) في البرنامج .
تقنية المجاري المتعددة تعالج ذلك ، فإذا قمنا بفصل إجراء قراءه الطرفية بمجرى مستقل فإن توقف استجابة هذا المجرى لن يؤثر على مجرى البرنامج الرئيسي بنفس الأثر ، وسيصبح الأمر مشابها لحاله برنامج آخر على سطح المكتب قد توقف عن الإستجابه ولكني لازلت أستطيع الوصول إلى بقيه البرامج ولو حدث بعض البطء الإضافي الناجم عن إستنزاف طاقه المعالج .

تعتبر هذه الميزة من مزايا اللغات القوية ، وتدعمها الكثير من اللغات مثل C و Delphi و C# و VB.net وJava
(يمكنها تحديد أولوية المجاري برمجيا).
Python تدعم threads ولكنها ليست بقوه وإتقان دعم Java لها مثلا .

Smalltalk تدعم هذه التقنية في بعض إصداراتها وليس جميعها .

أما اللغة Perl فإنها لا تدعم المجاري المتعددة .

معالجه الاستثناءات exception handling :

لنفرض أن بلوك برمجي (قطعه من الشفرة) يجب أن ينفذ دفعه واحده ، أي أما أن تنفذ كل التعليمات الموجودة ضمنه معا أو لا ينفذ أي منها ، مثلا إما أن نسحب مبلغ من الرصيد الأول ونودعه في الرصيد الثاني مباشره أو إذا حدث أي خطأ نتراجع عن عمليه السحب بالكامل ، فلا يجوز نتيجة الخطأ أن نحصل على مبلغ مسحوب من البنك وغير مودع في البنك الثاني .

ولنفرض مثلا أن جزء من الشفرة يجب أن ينفذ بعد كتله من التعليمات سواء نفذت التعليمات بالكامل أو لا ، مثلا بعد حجز الذاكرة لمكون ما يجب تفريغها في نهاية المطاف من الذاكرة مهما حدث من استثناءات وأخطاء في استخدام هذا المكون ، وسواء نفذت بشكل صحيح أو لم تنفذ

– لا يجب أن يتوقف التنفيذ في نقطه قبل تحرير الذاكرة ، وفي حال حدث خطأ في هذا البلوك فإن التنفيذ لن يخرج من البرنامج قبل أن ينفذ جزء تحرير الذاكرة.

معالجه الاستثناءات تحل كل هذه الأنواع من المشاكل وأكثر منها بكثير ، وتعتبر من الأدوات المهمة لكتابه تطبيقات قوية ، كما أنها تسمح لك بمعالجة أخطاء متوقعة على طريقتك الشخصية بدلا من تركها تعالج عل يد نظام التشغيل .

لم أجد الكثير من الفروقات لتذكر في هذا البند ، مع أن معالجه الاستثناءات من العوامل الغاية في الأهمية ، إنما خلال بحثي عن المعلومات لم أجد نتائج مميزة تختلف فيها اللغات الحديثة عن بعضها البعض في هذه النقطة ...
تدعم معظم اللغات السابقة معالجة جيدة للاستثناءات ، ويبدو أنها تستعير من بعضها الكثير وتقلد بعضها بعضا .

دعم الرسوميات والواجهات GUI و graphics :

ربما يعتبر البعض أن هاتين نقطتين مختلفتين .. نعم ربما
ولكن بالنهاية الدعم الجيد لتقنيات الرسوم يسهل بشكل كبير الدعم الجيد للواجهة المرئية ويبسط تعقيداتها ، ويسهل من إنتاج أدوات مرئية مناسبة .

جافا سيئة بدعم هاتين النقطتين . فهي لا تملك سوى بعض إجرائيات الرسم مثل drawLine(), drawRect() وأخرى مشابهه ، وهذه الإجرائيات لها تقييدات كثيرة ، (الخط لايمكن أن يرسم سوى بعرض 1pixel وإذا أردت عرض أكبر عليك القيام برسم العديد من الخطوط المتجانبه (مهما وجدت حلول لذلك فإن هذه الحالة تمثل تصرف اللغة الإفتراضي)

كما أنها لا تملك دعم 3_D متقدم ولا تملك العدة الكافية من الأدوات المرئية GUI Objects التي ربما يحتاجها برنامجنا .

مزايا C++ السابقة الذكر كالوصول المنخفض والقوه والتفصيل ، تجعل إمكانيات الرسم جيده جدا وتؤمن دوال كافيه إلى حد ما لتصميم برامج متكاملة معنية بالرسم والتصميم الهندسي . مجموعه الأدوات المرئية في VC++ جيده إلى حد ما ، ولكنها ليست الأفضل . كما لاحظنا من فقره (توفر مكتبه أدوات واسعة) فهي تعتمد توفير المقدرة عن طريق الشفرة وليس عن طريق مكتبة الأدوات .

دلفي جيده في مجال Graphics وتؤمن دوال كافيه وملائمة تختلف بين السرعة والسهولة والقوه يمكنك الاختيار بينها ، وهي لاتقل شأنا عن C++ في مجال معالجه الصور ولو أن الأخيرة أفضل منها برأيي الشخصي نظرا للكمية الكبيرة من البرامج المتعلقة والموارد والكتب المتوفرة والتي تكون عاده عن C++ في حين أنك ستجد كميه أقل من أجل Delphi (حاول أن تبحث عن كتب OpenGL مثلا ، ستجد فارق كبير بين نسبه الكتب والأمثلة المتوفرة لصالح C++ عنها المتوفرة لصالح Delphi)
ولكن دلفي تتفوق في مكتبه الأدوات ليس فقط على C++ بل على معظم اللغات الأخرى ، إنها تملك مكتبه أدوات ضخمه وفاعله ، مفتوحة المصدر . هي سر إنتاجيه دلفي العالية ، هذا المخزون من الأدوات يجعل من السهل عليك بناء أنواع مختلفة من التطبيقات منها التطبيقات المرئية التي تعتمد على الواجهات (GUIs) وتطبيقات الرسوميات عموما .

لغات MSVisual Studio .NET دعمت مكتبه أدوات قياسية وملائمة لكميه أكبر من التطبيقات ، مكتبه الأدوات الجديدة قياسية لكل لغات MSVS.Net ، وهذا يحقق نقطه مهمة

في تكامل هذه اللغات العضوي والمنطقي مع بعضها البعض . الجديد هو في مكتبه +GDI هناك إمكانيّات هائلة في مجال الرسم والتلوين تمنحها لك مكتبه GDI+.. يكفي أن تعرف أنّ بإمكانك الآن رسم منحنيات معقّده، وتكوين أشكال مركّبه من مجموعه خطوط ومضلّعات ومنحنيات، وتلوين السطوح بألوان متدرجة، وتحديد شكل مساحه الرسم، وتحديد درجه الشفافية، وتدوير الرسوم وتغيير مقاييسها تكبيرا أو تصغيرا.... إلخ.

كما أن مكتبه أدواتها المرئية جيده باعتراف الجميع . وتملك ميزات عديدة لاتقل أهميه كثيرا عن مكتبه الأدوات المرئية في دلفي

يقدم البايثون أدوات رسومية مميزة تسهل بناء البرامج الرسومية بشكل كبير
وكذلك يتميز الـ Smalltalk بدعم مثالي للبرامج الرسوميه بمختلف أنواعها وحتى الثلاثية الأبعاد فهو يمتلك مجموعه مميزة من الأدوات للقيام بذلك كما أنه يمتلك واجهه تصميميه سهلة جداً .

الانتشار، التسويق و الدعم الفني Marketing and Support :

لاتنغش بورود هذا البند في آخر البنود السابقة ، فهو من أهم البنود التي تؤثر على انتشار ورواج لغة برمجة ما .
انتشار لغة برمجة ما يؤثر على توفر الدعم الكافي لهذه اللغة على كافه الأصعدة ، وبشكل خاص الإنترنت . اللغة المنتشرة يمكن أن تجد لها مئات الكتب وآلاف المواقع وكميه كبيرة من فرص العمل . كل ذلك يسهل كثيرا تعلم اللغة لوجود كميه كافيه من الموارد والأمثلة ، كما أنه يشعرك بالأمان عندما تتعامل معها ، حيث ستجد وظيفة وعمل محترم بواستطها وستشعر أنها لغة تستقطب كميه كبيرة من الناس وفي حال حدث أي مكروه فإنه سيصيب كل هؤلاء جميعا ولست لوحدك وبالتالي نضمن على الغالب توفر حلول بديله لأعاده تقديم هذه الشريحة الواسعة من الناس .

برأيي أن لغات Microsoft بشكل عام تملك هذه الشعبية الواسعة ، ليس من الضرورة أن تجتمع الأسباب التقنية لنقول أن لغات MS أفضل من غيرها .. لا أبدا ، إنها لغات الشركة التي تملك نظام التشغيل ، أكبر شركه ، الشركة الأكثر تشعبا . تقف ورائها دول ومنظمات وشركات وهيئات الخ ... مايكروسوفت تملك الكثير ، على الأقل أنها تملك نظام التشغيل الأكثر انتشارا وقادرة أن تستعبد الناس تحت لوائه .
أي شخص يعمل مع أحد منتجات مايكروسوفت يشعر أنه أمتلك مايكروسوفت كلها ، وغالبا ما يشعر مستخدمو مايكروسوفت بالأمان .
حيث أن خطر توقف مايكروسوفت عن دعم عملائها منخفض كثيرا ، ولن يحدث مالم تتوقف قبلها شركات كثيرة . كما أن مايكروسوفت بارعة بالتسويق بشكل لا يصدق ، وتملك أقسام دعم فني متفانية بالعمل ، لاحظ شهادات مايكروسوفت المصدقة التي أصبحت بديلا مهما للتقنيين والمختصين ، لاحظ أنك تستطيع بسهولة من معظم بلدان العالم الحصول على هذه الشهادات (في حين أني تعبت كثيرا لأجد بلدان قريبه تقدم شهادة DCP من بورلاند ، حيث أقتصر الموضوع على البلدان الأوربية الشهيرة وأميركا وبعض بلدان الشرق مثل الهند) . هذا التسويق الذكي يزيد من أهميه منتجات مايكروسوفت ، والأدهى من ذلك أن مايكروسوفت تؤمن دعم التكامل بين أدواتها المختلفة ، .
نعم ربما يكون أوراكل أفضل من MS_Sql server البعض يرون ذلك . البعض يرون دلفي أفضل من C# البعض يرون Java أفضل من C++ البعض يرون لينكس أفضل من ويندوز ... ربما أنا لا أقول لا ... ولكن من المؤكد أنك إذا اخترت أن تكون كل الحلول من مايكروسوفت فإنها جميعا ستكسب قوه إضافية ، لاحظ إمكانية إنشاء منظومة متكاملة من نظام التشغيل وبرامج إدارة قواعد البيانات ولغات البرمجة وتقنيات الويب الخ ... مما يجعل حلول مايكروسوفت المتكاملة

أحد أهم الاختيارات أمامك ، مع أن كل منها لوحده قد لا يكون الحل الأمثل ، ولكنها تملك قوه الانتشار وقوه الدعم الفني ، التي نتحدث عنها الآن .

لغات شركه بورلاند Borland تعتبر من أهم لغات العصر ، وربما تحتل بعد مايكروسوفت المرتبة الثانية مباشره ، لأنها تقدم عدد من الحلول ولغات البرمجة (أيضا المتكاملة نسبيا فيما بينها) ماذا عن C Builder و Delphi و J Builder و Kylix و Dbase و FireBird الخ ...

لكنها بالنهاية تحتل المركز الثاني وليس الأول ، وبفارق ملحوظ وهو ليس صغيرا .. بورلاند غبيّة بالتسويق .
إذا حدث يوما وتوقفت بورلاند لن يكون السبب تقنيا أو علميا ، لطالما كانت هذه الشركة المميزة شركه إبداعات وثورات تقنية . برأيي سيكون السبب ضعف الانتشار والتسويق أمام عملاق السوق وعدم تملكها الأدوات اللازمة لإذهال المستخدم عن بقيه الشركات . بورلاند ليس بارعة بالتسويق ، وأغرب ما سمعته حولها أن شركه مايكروسوفت تملك نسبه معقولة من أسهم بورلاند . وهذا بالتالي يجعل مايكروسوفت من مالكي بورلاند ويسمح لها بالتدخل في سياستها وهذا ما يفسر عدم قيام بورلاند بأي خطوات عدائيه أو ضاره ضد مايكروسوفت مؤخرا ، وانضمامها تحت جناح مايكروسوفت . ربما يكون لذلك بعض الفوائد ، فقد لاحظنا سرعة بورلاند بدعم منصة NET. (جائزة أول شركه دعمت NET. بمنتجها دلفي ٧ "دعم تقنية NET. كان موجود منذ النسخة ٦" أي قبل أن تقوم مايكروسوفت بإصدار التقنية بشكل رسمي بخوالي ٦ أشهر . وهذا حسب ما ورد على موقع مايكروسوفت) كيف تفسر ذلك ؟؟ .

ذهب بعض الكتاب على مواقع الويب مثل الموقع الشهير About للقول بأن تقنية NET. تعود أصولها لشركه بورلاند قامت مايكروسوفت بشرائها مقابل مبالغ ضخمه ، وأضافتها إلى منصة NET. وخلقت ما يسمى غسيل الدماغ الكلي والانقلاب المسمى NET. ، إذا كان ذلك الكلام صحيحا فإنه أكبر مثال على كون الانتشار والتسويق والدعم الفني مهم أكثر من المزايا التقنية للغة البرمجة ، ولو كانت NET. من إنتاج بورلاند لما سمع بها العالم هكذا .

شركه Sun تملك حصة كبيرة من الكعكة أيضا مثلها مثل بورلاند ، وربما أفضل منها ولكني فضلت بورلاند لأن لها مجموعه كبيرة من لغات البرمجة التي تنتجها ومجموعه كبيرة من الحلول ، في حين تركز Sun على لغة JAVA بالدرجة الأساسيّة ، ولكن لها رصيد مهم من المنتجات الأخرى ، و كل ما ينطبق على بورلاند من الأفكار السابقة ينطبق عليها ، سأذكر مثلا المشكلة بين مايكروسوفت وصن ، حيث توقفت مايكروسوفت عن دعم وتضمين منصة JVM في نظم تشغيلها ، وسيضطر المستخدم إلى تنصيبها من الإنترنت مثلا ... لاحظ ذلك مايكروسوفت

قادرة على لوي ذراع صن لأنها شركه كبيرة أكثر منها وتملك سلاح فتاك هو عبارة "نظام تشغيل مايكروسوفت ويندوز".

رغم تفوق لغات مايكروسوفت تعتبر كل من لغات بورلاند ولغات صن ، خيارا حيا ومهما للغات البرمجة . في حين تنحسر المنافسة كثيرا مع منتجات ولغات أخرى حتى لا تكاد تسمع بها أو بمن يستخدمها إلا بالجامعات والمنظمات اللاربحية والمراكز وما إلى ذلك

ناحية سلبية لـ python مثلا هي قله المراجع الرسمية كالكتب المتعلقة بالغة فهناك على الأكثر بضعه كتب فقط (لا أعرف ربما ثلاث كتب فقط) مهما كانت هذه الكتب ممتازة فهي عامه ، بإمكانك أن تجد في لغات أخرى كتاب كامل يتحدث عن موضوع واحد بكافه تفاصيله (عن الإنترنت أو API أو الشبكات أو الألعاب أو أو ..) ومهما كانت هذه الكتب الثلاث شامله فهي لن تقدم معلومات تفصيلية بنفس مستوى كتاب مخصص .
لكن ومن ناحية أخرى يوجد دعم كبير لهذه اللغة على الإنترنت من حيث إمكانية إيجاد حلول للمشاكل والدروس وغيرها عبر مقالات ونشاطات لمبرمجي البايثون أنفسهم.

يمكن أن تجد أرشيفات مجانية على الإنترنت عن Perl مثلا والتي أصبحت مناسبة من أجل Unix ، كما أن لها توثيق جيد وموسع للتعلم على شكل HTML أو PDF ..

وجدير بالذكر أنه توجد عوامل أخرى عديدة تؤثر على انتشار لغة البرمجة ، خذ مثلا ملاءمتها للغات وهيئات مختلفة حول العالم . كدعم اللغة الصينية والعربية وغيرها ، وما إلى ذلك من القدرة على التراصف لليمين أو من الأعلى للأسفل في كتابه النصوص .
الكل يعلم أن تأخر دعم لينكس للغة العربية كان من أهم أسباب ضعف انتشاره في أوساط منطقتنا العربية .(التأخر بسبب العرب أنفسهم الذين تكاسلوا في العمل على هذه النظم المفتوحة وتطويرها ، ومالم ندعم اللغة العربية على هذا النظام لن يدعمها من أجلنا الغرباء)

خلال إطلاعي على لغات مثل Smalltalk أو python أو Power Builder أو أو ..
وجدت أنه في العالم الكثير من اللغات التي لا نعرفها ولم نسمع بها ، وغير منتشرة في منطقتنا ومن الجدير بنا الاضطلاع على هذه اللغات القوية والمجهولة لا بل تعلمها أيضاً . وربما تبقى المشكلة خلف سوء انتشارها هي عدم دعمها الكامل للغتنا العربية أو عدم توفر مراكز عربيه تنشر هذه اللغات وتعلمها بشكل جيد .

المحمـوليـــه Portability :

١- على مستوى نظم التشغيل .
٢- على مستوى إصدارات نظام التشغيل .
٣- على مستوى نظام تشغيل واحد_أجهزه مختلفة .

أولا : المحمولية على صعيد نظم التشغيل . تعدد المنصات Cross_PlatForm :

هناك بعض اللغات القادرة على العمل على عدّه نظم تشغيل ، ومن أبرز هذه اللغات اللغة Java ، والتي تستطيع العمل حتى على أجهزه ذكيه أخرى غير الحواسيب كالموبايلات والساعات مثلا ..
تختلف فلسفه لغات البرمجة بين بعضها لتحقيق هذه الغاية :

أولاً – اللغات التي تعمل على منصات خاصة :
وتكون لهذه اللغات منصة خاصة بها ، تتعامل اللغة معها بشكل كامل بدلا من نظام التشغيل ، وتقوم هذه المنصة بدورها بترجمة احتياجات اللغة إلى نظام التشغيل . يمكن اعتبار المنصة طبقة برمجيّة بين اللغة ونظام التشغيل ، بحيث تلبي احتياجات اللغة حسب نظام التشغيل الذي تعمل عليه ، وفي هذه الحالة يكون للمنصة نفسها عده إصدارات من أجل نظم تشغيل مختلفة ، مثلا إصدارة من المنصة تعمل على لينكس وأخرى تعمل على ويندوز وهكذا ....
وبما أن المنصة طبقه برمجيّة لايراها المستخدم ولايتأثر بتغيراتها ، لأنه يهتم فقط بالواجهة المرئية للغة البرمجة والتي تبقى ثابتة ، عندها لن يتأثر المستخدم بشكل كبير بالإصدارات المختلفة للمنصة من أجل كل نظام تشغيل .

ومن هذه اللغات :

١- جافا (Java) . Multiplatform

تعتبر جافا أكثر لغة برمجة تدعم هذه المحمولية ، وصممت أساسا من أجل مراعاة دعم التطبيقات الموزعة وتطبيقات الإنترنت مثلا - في هذا النوع من الحالات تكون الزبائن تعمل على نظم تشغيل مختلفة ومتباينة وبالتالي من المهم القدرة على التخاطب مع هذه النظم المختلفة من لغة واحده ، وقدرة هذه اللغة على العمل عليها جميعـــاً . وهذا هو حال جافا ملكه المحمولية الأولى بين اللغات .

وليس فقط ذلك بل أن جافا تدعم أنواع أخرى عديدة من الأجهزة ألإلكترونيه وقادرة على العمل عليها ، لوجود نسخ من منصة جافا قادرة على العمل في هذه الأجهزة ،

كلنا نرى تطبيقات الموبايل المبرمجة بواسطة جافا ، أو الساعات الرقمية الحديثة . أو الإجهزة الكهربائية التي تعمل على وحدات معالجه صغيره .. (هل تستغرب إمكانية جافا من العمل على ثلاجة مثلا ؟)

والخلاصة أن جافا لغة تدعم محمولية تطبيقاتها على نظم مختلفة بنطاق واسع .

-لغة Python أيضا متميزة في هذا المجال حتى أن عشاق Python يرون أن شفرتها أكثر محمولية من جافا نفسها ، حتى أنك تستطيع تشغيل كود بايثون من منصة جافا نفسها (Java Virtual Machine)

أو تشغيله من الأنظمة التي تدعم JVM (Java Virtual Machine) باستخدام JPython . وتستطيع Python العمل على Windows, Linux, Unix, Macintosh, OS/2, Amiga, وأكثر من ذلك بدون تعديل على شفرتها .

لغة Smalltalk تعمل كذلك وفق منطق الـVM (Virtual Machine) بشكل مشابه لما ورد . والفرق أن Smalltalk تحفظ كل الكود في ملف (Image) واحد فقط ، وتحفظ فيه حال النظام كله (المجاري النشطة وما إلى هنالك) ..

تستطيع Squeak العمل على Macintosh,Unix,OS/2,Ms-Dos,Win9x/Nt/2000, and windows CE . المنصة الخاصة بها تم عملها بواسطة C من أجل سهولة نقلها إلى نظم مختلفة .

٢- **لغـات الـ NET.** التي قدمتها مايكروسوفت (Microsoft) ، تعمل هذه اللغات على فلسفه شبيهه بفلسفة منصة جافا مع بعض التغييرات البينية ، حتى أني أظن أن منصة NET. تقليد لفكره منصة جافا .

أصدرت مايكروسوفت لغات جديدة تدعم هذه التقنية ، مثلا كل من C# و VB.NET كما قامت بعض اللغات الأخرى بدعم هذه التقنية مثل Delphi .NET و C# Builder.Net من شركه بورلاند ، وغيرها كثير .

Net. لازالت حديثة العهد ، ولازالت الإصدارات منها المخصصة للعمل على نظم أخرى قيد التجريب والتطوير (توجد نسخه NET. تجرب من أجل لينكس تسمى Mono راجع : http://www.go-mono.com/) وبالتالي في حال تمّ إعداد نسخ من المنصة قادرة على العمل تحت نظام تشغيل ما فإن هذه اللغات تستطيع *على الأرجح* العمل على هذه النسخة .

ثانياً – نسخ من اللغة مخصصه للعمل على نظم مختلفة :

وهنا يطرح مفهوم Multiplatform vs CrossPlatform نفسه .
لاحظ Cross تملك دعم خاص لكل نظام مما يمكن اللغة من التعامل مع النظام الحالي بشكل جيد جدا ، حيث توجد نسخه لكل نظام، أما Multi فهي نفسها لكل النظم وبالتالي لايمكنها دعم الأمور الخاصة

وهذا حال اللغة دلفي ، حيث دعمت شركه بورلاند بعد الإصدارة السادسة منها (Delphi 6) مكتبه CLX المخصصة لدعم تعدد-المنصات Cross-Platform ، والذي مكّن بورلاند من إنتاج نسخه من دلفي (تمّت برمجتها على دلفي) قادرة على العمل تحت لينكس ، وسمتها Kylix.
أي أن دلفي لا تملك منصة خاصة بها كما حدث مع الحالتين السابقتين ، ولكن يوجد منها نسخه للعمل على نظام مايكروسوفت ويندوز ، ونسخه للعمل على نظام لينكس .
فكره CLX هي الابتعاد قدر الإمكان عن استخدام توابع API النظام ، لأن هذه التوابع لن توجد في نظام آخر (Api ويندوز غير موجودة في لينكس) ، لذلك على دلفي تضمين كل شيء في الملف التنفيذي نفسه ،(وهذا هو سبب عدم صغر حجم ملفات دلفي) .

نتيجة ذلك أننا إذا بنينا برنامج على CLX_Delphi ضمن ويندوز ، فإننا سنستطيع فتح شفرته مباشره من كايلكس ضمن لينكس وإنتاج نسخه من المشروع تعمل تحت لينكس . أي شفرة واحده ــ ملفين تنفيذيين : الأول لويندوز الثاني للينكس .

وهذا يسمى المجمولية على صعيد الشفرة ، بدلا من المحمولية على صعيد الملف

ثالثاً – محررات مختلفة تدعم نظم مختلفة :

حيث تقوم شركات ومنظمات مختلفة بدعم النحو (Syntax) الخاص بلغة برمجة ما ، مثلا تستطيع أن ترى محررات للغة C/C++ موجودة على لينكس ، ومحررات من جهات أخرى موجودة على ويندوز . هذه المحررات ليست نسخه من المحررات السابقة بل هي منتجات جديدة تدعم شفرة لغة برمجة ما . ولاعلاقة لها بالمحررات السابقة سوى بالتشابه النحوي للغة .

وعادة من الصعب نقل شفرة محرر إلى محرر آخر في المشاريع الكبيرة ، لإنه كثيرا ما تستقل المحررات بدعم مزايا خاصة بها تخرج عن مواصفات القياسية المعروفة وربما تملك مكتبة أدوات مختلفة ١٠٠٪ ..

لم أنجح بنقل شفرة برنامج GUI من Borland C++ Builder إلى MS Visual C++ وأظن أن الكثيرون غيري أصتضدموا بنفس المشكلة .

ثانيًا : المحمولية على صعيد إصدارات نظام التشغيل المختلفة :

تظهر في كثير من الأحيان إشكالات في عمل البرامج على إصدارات مختلفة من نظام التشغيل نفسة .
مثلا Win 95 و Win 98 و Win 2000 و Win XP .

كيف أضمن أن التطبيق الذي يعمل على Win 95 قادر على العمل على Win XP ؟ وهل للغة البرمجة دور في ذلك . ..... في الحقيقة نعم

- لغات البرمجة التي تعتمد المنصات ستكون قادرة على ذلك بالتأكيد لأنها تتعامل مع منصتها ، ولكن ستكمن المشكلة بالمنصة نفسها ، فإذا لم تتمكن المنصة من العمل على الإصدارات الجديدة يجب إنتاج إصداره جديدة من المنصة تدعم ذلك .
-

في هذا المضمار تعتبر جافا حتى الآن أفضل من NET. من ناحية توافقيه المنصة مع الإصدارات المختلفة .

- اللغات التي تستخدم الحد الأدنى من توابع API النظام هي الرابح الأكبر في هذا المجال .
C++ ستحتل مرتبه متقدمه في ذلك ، لأنها تعتمد على توابع اللب (Core) والتي نادرا ما تتغير لأنها أساس نظام التشغيل من المستوى الأدنى ،
ولكن نبقى عند المشكلة التقليدية ل C++ أنه توجد فيها كل ماتريدة ، ولكن عليك إخراجة من فم السبع (السبع=C++) .
VB سوف تتراجع كثيرا ، لأن فلسفتها المبنية على أساس المفسر (Interrupter) تقوم بكميه كبيرة من الاستدعاءات والتي قد تحوي كميه كبيرة من توابع القشرة (Skin) ، وهذه عاده ما تتغير كثيرا وتحدث فيها تطويرات دائمة . وهذا يولد احتمال أكبر أن لا يتوافق تطبيقنا مع إصدارات مستقبليه من نظام التشغيل .
تعتمد برامج الملتيميديا على كم كبير من توابع القشرة ، لذلك كثيرا ما نلاحظ الحاجة لتنصيب نسخه محدثه من البرنامج لتعمل على نظامنا الجديد .

دلفي ستحقق مرتبه متفوقة في هذا السبق ، السبب الأساسي كما ورد سابقا هو مبدأ دلفي في العمل القائم على أساس " نظام التشغيل عدو لذلك لا أحتاج منه شيء وأستطيع بناء كل ذلك بنفسي " . المقاربة أدبيه إلى حد ما فلا بد لدلفي من التعامل مع نظام التشغيل شاءت أم أبت ، ولكن من المؤكد أنها تقتصد قدر الإمكان في ذلك .

وسأورد هنا مثالا من تجربتي الخاصة يوضح تعلق اللغة بالنظام بالنسبة للغات السابقة :

عندما نبني تطبيق على VB تحت Win Me ، ونقوم بتشغيل التطبيق تحت Win XP . فإن التطبيق يتصرف بشكل افتراضي مثل تطبيقات XP الأساسيّة ، ونلاحظ هنا أن أزرار التطبيق تصبح كلها جذابة ومضيئة مثل أزرار XP نفسها ... أي أنها أخذت شكل XP .

عندما نكرر نفس التجربة مع دلفي ، فإن أزرار دلفي تبقى كما هي تماما عند تشغيل التطبيق تحت XP ، ولا تأخذ الشكل الجذاب والمضيء تلقائيا ...
((تسمح دلفي بالحصول على شكل XP بوضع الأداة XP_Man على الفورم ، ولكن بالنهاية لم تتم العملية بشكل تلقائي)) . لماذا :

تعقيب :
VB تعاملت مع الصنف زر Button الموجود في نظام التشغيل ، وعندما تغير نظام التشغيل تغير صنف الأزرار في VB تلقائيا لأنها تستمده من النظام .

Delphi لا تتعامل مع الصنف Button الموجود في النظام ، بل تقوم ببناء أصنافها الخاصة .

يمكن ببساطه الإطلاع على شفرة الصنف TButton الذي تزوده دلفي ، وفي حال قمنا بأي تغيير في هذا الصنف ستتغير أزرار تطبيقاتنا بالتوافق مع هذا التغيير بمجرد ترجمة ملف BPL .

ميزة / سيئة : تعتبر التقنية السابقة التي تحدثنا فيها عن دلفي ميزة مهمة ومفيدة من مزايا دلفي ، ولكن لها آثارها السلبية ، عدم اعتماد دلفي على النظام يعني أنها لن تستطيع دعم آخر تحديثات النظام وتقنياته بنفس سرعة اللغات الأخرى ، وسننتظر دلفي في كثير من الأحيان حتى الإصدارة المقبلة حتى تدعم تقنية جديدة . وهذا ما يفسر الكم الكبير من الإصدارات التي تنتجها دلفي (من ١٩٩٦ حتى اليوم توجد ٩ إصدارات دلفي)

لكل صفة في لغة البرمجة آثارها الإيجابية والسلبية ،

ثالثا : المحمولية على صعيد نظام تشغيل واحد_أجهزه مختلفة :

ماذا عن توزيع تطبيقنا على زبائن مختلفة . حتى لو استخدموا نفس نظام التشغيل . هل تتساوى لغات البرمجة بدعم هذه الحالة من المحمولية ؟

١- مكتبات زمن تشغيل (Run Time) :

وهذا هو حال لغة Visual Basic .....
تحتاج بعض لغات البرمجة إرفاق مكتبات خاصة تعتمد عليها التطبيقات المبنية بواسطتها ، ولا تعمل هذه التطبيقات بدون توفر نسخه من هذه المكتبات على الجهاز .
وبالتالي سنطر لتوفير نسخه من هذه المكتبات مع برنامجنا ، لأننا لا نضمن أن المستخدم يملك نسخه من هذه المكتبات ، وبالتالي زيادة بالحجم من جهة
ومن جهة أخرى ، تنسخ هذه الملفات إلى مجلدات خاصة بالنظام ، ولن نضمن أن يعرف المستخدم تنصيب البرنامج وأن يستطيع نسخ هذه الملفات إلى أماكنها الخاصة ،
وبالتالي لابد من توفير برامج التحزيم (برامج التنصيب Install Managers) لكي نقوم بإعداد نسخه Setup من برنامجنا . مما يعني القليل من الحجم الزائد أيضا .

كذلك الأمر دعنا نفرض أن المستخدم قام بفرمتة(Format) نظام التشغيل مره أخرى ، هذا يعني أن مكتبات زمن التشغيل المخزنة فيه كلها محيت ويجب إعادة تنصيبها في النظام ، ولن تعمل برامجنا من دونها حتى لو كانت منسوخة إلى سواقة أقراص مختلفة عن سواقة نظام التشغيل ؟؟؟؟
ببساطه عليك إعادة تنصيب البرنامج كلما حدثت مشكله بالنظام أو كلما قمت بإعادة التنصيب

٢- الأدوات التي استخدمناها في برنامجنا .

في كثير من الأحيان نستخدم أدوات غير الأدوات القياسية المرفقة مع اللغة ، مثلا ننزلها من الإنترنت أو نبنيها نحن لتسهيل البرمجة أو من أحد الأصدقاء ..
بعض اللغات مثلا VB التي تستخدم مكونات Active X (ملفات ذات امتداد OCX) ، عندما نقوم باستخدام أحد هذه المكونات في تطبيقنا ، فإننا مجبرين على إرفاق ملف المكون (ocx.) مع برنامجنا ونسخه أيضا إلى مكتبات نظام التشغيل ... ؟؟؟؟ . وبالتالي كل العيوب التي تحدثنا عنها بالبند السابق من الحاجة لأعاده التنصيب .

ليت الأمر ينتهي هنا .. ستحتاج إلى تسجيل هذه الأدوات في النظام (الرجستري) ، والرجستري هو قاعدة بيانات تحوي كل توصيفات وإعدادات الجهاز ، وكلما زاد حجم هذا البيانات كلما زاد بطء النظام الذي سيضطر للبحث في كميه اكبر من البيانات .
كما أنها أصبحت متاحة لجميع التطبيقات وفي حال نجح أحد ما بتسجيلها واستخدامها ، مبروك عندها يستطيع استخدام هذه الأداة بحرية (يوجد متخصصين كسر حماية لهذه الملفات) دون إذن منك .
أصلا لوناقشنا الموضوع نظريا لوجدنا أن هذه الأدوات يتم استدعائها خارجيا (ملفات مستقلة عن تطبيقنا يتم استدعائها من النظام) وبالتالي سيشكل ذلك بطء في تحميل المشروع .

ربما يبدو كل ذلك عاديا ، حتى أن VC++ تملك نفس المشاكل .
ولكن إذا قارنا ذلك بالغات التي تؤمن دعم ملفات Stand Alone Exe's ، لوجدنا فارق كبير .
لا ننسى أن المحمولية معيار مهم في تقييس لغات البرمجة .

لغة الدلفي الرائدة في مجال المحمولية تسمح لك ببناء تطبيقات Stand-Alone . دون الحاجة لإرفاق مكتبات زمن تشغيل ، كما أتفقنا تطبيق دلفي يعتمد على نفسه قدر المستطاع .

على سبيل المثال إذا بنينا تطبيق بسيط في دلفي ، لايحتاج لملفات موارد (صور أو ما شابه) عندها يكفي نسخ الملف التنفيذي(EXE) وحده ولصقه في أي مكان على جهاز الزبون . تخيل الدرجة العالية من المحمولية .
كما أن نموذج مكونات دلفي (خذ VCL مثلا) يسمح بدمج الأدوات التي استخدمناها وتضمينها مع النسخة التنفيذية دون الحاجة لنقلها مع برنامجنا وتنصيبها كما في الحالة الأولى .

إذا كنت تعد برنامج كبير يحوي العديد من ملفات الموارد ، تستطيع نسخ ملفاتك كلها وتشغيلها من عند الزبون . ونادرا ما تحتاج لأعاده تنصيب برنامجك عند إعادة تنصيب نظام التشغيل .

أما حاله مكتبات DLL وبعض مشغلات قواعد البيانات فيفضل نسخها إلى النظام بدلا من المجلد الحالي ، لكي تستفيد منها تطبيقات أخرى بدلا من إعادة نسخها كل مره وهذه حاله عامه في كل لغات البرمجة .

التكامل مع نظام التشغيل OS integrity:

جميل جدا أن نملك أدوات تملك محمولية عالية وقادرة على التعامل مع نظم مختلفة ، تحدث الإستقلالية عن نظام التشغيل نتيجة عدم الاعتماد على أي من المميزات الخاصة بنظام تشغيل ما ، والابتعاد عن كل خاصية غير موجودة في النظم الأخرى ... لكي تكتب تطبيقات تتوقع نقلها إلى نظم تشغيل أخرى لا تستخدم أبدا توابع API الخاصة بنظام ما ، لأنها ستتغير من نظام لآخر ولا يوجد ما يضمن لك أن نسخ مشابهه لها تعمل على نظام تشغيل آخر .

أن ذلك كما اتفقنا سيكسبنا عمومية وقدرة على التعامل مع أكثر من نظام تشغيل ، ولكن حذاري فإن الانعزال عن نظام التشغيل سوف يحرمنا من المزايا المميزة بكل نظام .

ألا ترى معي أن نفس الأسباب السابقة التي تعطي المحمولية العالية والاستقلالية عن نظام التشغيل هي نفسها التي ستحرمنا من استخدام الجوانب الإيجابية التي يتميز بها نظام ما عن غيره ، وستجعلنا مجبرين على التعامل بشكل عام مع النظام دون القدرة على استنزاف طاقاته والغوص بمميزاته وتقنياته التي ينفرد بها عن غيره .

نعم .. تملك جافا محمولية مذهله ، ولكنها كما هو متوقع غير قادرة على الغوص في ثنايا نظام التشغيل نتيجة لذلك . كما أنها في كثير من الأحيان غير قادرة على دعم أمور تعتبر سهلة وبسيطة في لغات أخرى ، لا تحوي جافا حتى الآن مشغلات MPEG أو حركات أخرى غير GIF متلاحقة (مع أنها تستطيع القيام بذلك بشكل أو بآخر مثلا محرك Macromedia مكتوب بالجافا) .
خذ على سبيل المثال صعوبة إنشاء تحكمات تقبل السحب والإفلات بواسطة جافا (drag and drop)
وبما أن جافا تحاول الاستقلال عن نظام التشغيل فهي ستدعم التقنيات التي تعمل على كل النظم بشكل رائع ولكنها ستخسر الأدوات المميزة بكل نظام ، ماذا عن دعم OLE في مايكروسوفت مثلا . حتى تطبيقات MDI (Multiple Document Interface) ستصطدم بها .

C++ تملك ميزة سحريه ، أن نظام التشغيل مكتوب بها . وبالتالي كميه كبيرة من الوثائق والشروحات التي توضح خفايا نظام التشغيل وآلية التعامل مع توابعه وشرح لهذه التوابع والأمثلة عليها ، كلها مكتوبة بلغة C++ نفسها التي بني عليها النظام . لاحظ وثائق MSDN الأساسيّة من أجل أي مستخدم ، وكتب توابع API ويندوز كلها تدعم C++ في البداية ثم يتم ترجمتها إلى لغات برمجة أخرى .

دليل Win API الذي أعتمده من أجل دلفي ، يرفق المثال الأصلي بالـ C++ لمزيد من التوضيح والدقة ، ما رأيك بتكامل C++ مع نظام التشغيل الآن .

لغات Microsoft تملك هذا الدمج الكبير ،
والسبب أنها لغات من إنتاج نفس الشركة التي أنتجت النظام (أهليه بمحليه = MS*MS) .
وبالتالي تملك لغات مايكروسوفت ميزة تحسد عليها من بقيه اللغات أنها لغات أخوه بالرضاعة مع نظام التشغيل التي تعمل عليه ، وتتشرب كل تقنياته وميزاته .

لاحظ لغة VB التي يجمع الجميع أنها لغة ضعيفة لغويا ولا تملك مزايا اللغات القوية ، ولكنها رغم ذلك قادرة على فعل الكثير الكثير بالنظام ويندوز ، وتسمح لك بدرجه جيده من التحكم به .. أنا متأكد لو أنها تعمل على نظام ثاني لما ملكت هذه القوه الإضافية أو لو أنها من إصدار شركه غير الشركة الغول Microsoft ، لأنها عندها ستخسر تكامليتها مع نظام التشغيل .

أصلا تحاول مايكروسوفت أن تدعم VB من خلال نظام التشغيل ، لاحظ مثلا إمكانية كتابه VB Script من برامج أوفيس ويندوز ، ربما إذا كتبت برنامج على الـ Access تحتاج لبعض شفرات VB ضمنه ؟

في حين أن لغات أخرى أهم منها تقنيا غير قادرة على القيام بما تقوم به من اندماج مع ثنايا نظام التشغيل التي تعمل عليه .

دلفي خسرت كثيرا في نسخ Win32 أمام لغات ميكروسوفت ، إنها ليست مثل جافا حتما ، ولكنها حتما أيضا ليست مثل لغات MS .

Delphi 2005 أستفادت كثيرا على مايبدو من NET. ، وبدأت تتساوى مع لغات ميكو السابقة الذكر بوصولها إلى أصناف Net. مثلها مثل غيرها .
لاأعرف لماذا أصبحت مع الرأي القائل شكرا مايكروسوفت لقد أعدت دلفي .

دعم الويب Web Supporting :

لاحظ أن دعم الويب بالآونة الأخيرة لم يعد مجرد دعم لتقنية عادية ، حيث أصبح التطور التقني يركز على قدرة الأفراد والمؤسسات على التواصل فيما بينها وتبادل شتى أنواع الوثائق ألإلكترونيه والمعلومات ، ويتجه العصر لاعتبار الاتصال بالويب ضرورة حتمية لأي حاسوب وربما أجهزه أخرى كالتلفزيون التفاعلي .
لذلك أفردت دعم الويب كنقطه منفصلة لكي أضع خطين تحت هذا البند .

دعم الويب هو تسهيل القدرة على التحكم فيه واستثمار كافه تقنياته .

جافا تملك دعم خيالي للويب ، وعلى حد تعبير SUN : "جافا هي لغة الويب والشبكات" ، حيث يمكن تنفيذ Applets ضمن وثائق الويب ، معظم مكونات واجهه المستخدم المرئية GUI يمكن أن تستخدم في مستندات الويب لإنتاج تطبيقات تفاعليه من جانب المستعرض .

لغات NET. كذلك توجهت لدعم التقنية الجديدة (NET.) كخيار استراتيجي يضمن تفوقها على غيرها من لغات العصر .
باتت تطبيقات ASP.NET تتصدر قائمه "المشاريع الجديدة" في لغات NET.

منذ متى كان الاهتمام بتطبيقات الويب بهذا الشكل ؟.

أظن .. لابل أجزم أن هذا الاهتمام في تزايد مستمر ، وربما يكون اختصاص العصر القادم .

اللغات مفتوحة المصدر Open Source:

البرامج المفتوحة الشفرة تملك بطاقة ضمان مفتوحة القيمة ، وتؤمن مقدرة التطوير المستقبلي والتحسين والإضافة على البرنامج . كما أنها تؤمن القدرة على إصلاح الأخطاء التي تظهر متأخرة وسد الثغرات التي لم تأخذ بالحسبان ، أو مثلا القدرة على تهيئه نسخ جديدة من المشروع تعمل وفق شروط وبيئات جديدة .
أهميه اللغات المفتوحة المصدر لا تقل أهميه عن البرامج المفتوحة المصدر ، بالعكس إنها تزيدها ميزة سحريه .. فهي تسمح للمبرمج من فهم الدقائق الداخلية التي تتألف منها اللغة ومكتباتها وتوابعها ، وتساعده في فهم مبدأ وآلية عمل الكثير من الدوال والإجرائيات مما يجعله يستخدمها بالشكل الصحيح تماما ويبني دوال ومكتبات جديدة فائقة بمواصفات مشابهه للمواصفات التي وضعها مبدعو اللغة . وتؤمن له توريث صالح ودقيق لأصناف اللغة الموجودة . كما أنه يستطيع التعلم من شفرة اللغة المفتوحة بلا شك .

Python هي لغة برمجة مفتوحة المصدر (أنشأها Guido van Rossum) " أنا معجب بكل شيء مفتوح المصدر"
وكل الشفرات التي تتكون منها Python مفتوحة ويمكن تفحصها والتعديل عليها .

Php من اللغات المفتوحة المصدر الرائعة والتي أخذت تبتلع سوق برمجة مواقع الويب في الفترة الأخيرة ،، وتستحق أكثر من ذلك .

لغة دلفي ليست مفتوحة المصدر بالكامل ، ولكن مكتبه أدوات دلفي مفتوحة المصدر .
ومكتوبة بدلفي نفسها .
المشكلة هل ستحافظ بورلاند على فتح شفرتها في Delphi .Net . انا لاأعرف أين هي شفرة مكونات دلفي الكاملة في Delphi 2005 ؟

القياسية Standard:

بعض لغات البرمجة تكون مملوكه لشركه أو لجهة معينه، ويحق لهذه الشركة تقرير مصير هذه اللغة وتكون هي المسؤله عن كل ما يتعلق بهذه اللغة . القياسية من الأمور المفيدة جدا لانتشار لغة البرمجة وإستمرارية دعمها . خاصة أنها تتيح إنتاج العديد من المنفذات للغات القياسية مما يضيف عامل التنافس والتعدد الذي يعطي خيارات أوسع لمستخدمي هذه اللغة .

C++: هي لغة قياسية غير مملوكه .
لقد استنفذت ANSI/ISO وقتها لكتابه المقاييس الخاصة بالسي ويبدو أن معظم المترجمات تنحى نحو تطبيق هذه المقاييس، بالرغم من وجود بعض الخصوصية والتي تتجلى بشكل واضح في MS Visual C++ التي خرجت عن القياسية في كثير من الجوانب . ورغم أن C++ تملك الاحترام لأنها ANSI قياسية ولكن إمكانية الانتقال الفعلي من مترجم إلى آخر بعيده عن الكمال قليلا.

C# : أيضا هي لغة قياسية ، تم توصيفها من قبل Hewlett-Packard, Intel, Microsoft .
لغة C# ليست مملوكة ، ويوجد لها عدة منفذات . مثلا كل من **MS C#** و **Borland C# Builder**

Eiffel : وهي لازالت مستقرة منذ زمن فهي تملك مكتبه_نواه قياسية (ELKS 95) تتحكم بها إتحاد جمعيات NICE ، وهي الجسم القياسي لـ Eiffel .

الباسكال الشيئية: وهذه لغة مملوكه، لذلك لا يوجد لها مقاييس. تقوم بورلاند بتطوير هذه اللغة مع كل إصدار جديد منها ، ولكني لاأرى ذلك يعني الكثير أمام اللغات القياسية .

الجافا : وهي أيضا لغة مملوكه ولديها أيضا علامة تجاريه للاسم ومع ذلك فإن شركه Sun أكثر من راغبة بتسجيلها لمترجمات أخرى . وقد أعطت جهات داعمة أخرى غيرها الحق بإنتاج منفذات لها . مثل منفذ Borland مثلا .. ورغم ذلك فهي تبقى لغة مملوكة بالنهاية مثل الباسكال .

Perl لغة غير مملوكه ، ويمكن أن تجد شفرتها المصدرية ومترجمها مجانا .

دعم البرمجة غرضيّـة التوجّه (OOP)

البرمجة غرضية التوجه Object Oriented Programming :

البرمجة الغرضية هي تقنية ثورية غيرت مسار البرمجة منذ عده عقود من الزمن .
غرضية التوجه بعيدا عن المفهوم القواعدي لها هي أسلوب كتابة برامج يركز على قابليه إعادة الإستخدام والتوسعيه . *يلغي فكرة موت البرنامج بعد أن تزيد سطورة عن حد معين* ، حيث يصبح من غير الممكن التعامل معه كما هو على شكل وحدة واحدة وإجراء التغييرات عليه دفعة واحدة .

يبنى الأسلوب الغرضي التوجه على أساس تقسيم العمل على شكل وحدات عضوية متكاملة كل وحدة قائمة بذاتها تملك خصائصها وأفعالها ويمكن تطويرها بشكل منفصل عن البرنامج الرئيسي وإستخدامها في برامج أخرى.
إذن :
أولا قابلية إعادة الإستخدام التي تفصل بين باني الصنف ومستخدم الصنف .
الكثير منا يستخدم مكونات برمجيه لايعرف عنها الكثير ، كلنا نستخدم المكون (Button) وقليلون منا من يعرفون برمجته ، تخيل لولا البرمجة الغرضية كم كان الوضع صعبا . ليس فقط كذلك ، بل سيتم التركيز على جعل هذا الصنف وحدة مركزه منفصله وبالتالي عند القيام بأي تغييرات عليه ستطبق التغييرات تلقائيا على كل جزء يستخدم هذا الصنف . (مثلا لوقمت بتطوير نوع أزرار جديد "صنف" وأستخدمته ٢٠ مرة في برنامجك . عندها لتعديل شكل الأزرار في برنامجك يكفي تغيير شفرة الصنف مرة واحدة ، ولن تضطر للمس ال ٢٠ زر البقيه .
وأفترض لو كان للبرنامج عدة أصناف عندها يمكن برمجة كل صنف لوحدة من قبل شخص أو عدة أشخاص وبناء كل صنف لن يرتبط ببناء الأصناف الأخرى إلا من حيث الدخل\خرج .

ثانيا التوسعيه ، وهي مبنيه على كون الصنف المبني وحدة مستقله لاتحتاج شيء من أحد ، وبالتالي بإمكاننا تصديرة وإستخدامه في برامج أخرى غير برنامجنا الحالي أو نشرة على الإنترنت ليستفيد منه آخرون ، أو حتى إستخدامه من لغات برمجة أخرى .

– لا تعتبر البرمجة الغرضية التوجه تقنية برمجة جديدة، بل تعود جذور هذه التقنية إلى Simula-67 المطورة بواسطة العالم النرويجي Kristen Nygaard و Ole-johan Dahl في بدايات ١٩٦٠، وربما يمكننا أن نعتبر أن التمثيل الكامل لها هو Smalltalk-80 (١٩٨٠) والتي تعتبر أول لغة برمجة غرضية (شيئيه) صرفه (Pure OOP) .
أصبحت البرمجة الغرضية التوجه شائعة الاستخدام في النصف الثاني من عقد الثمانينات، وأصبحت مدعومة من معظم اللغات الحديثة، وربما أمكننا القول أن دعم اللغات للـ OOP أصبح من المعايير المهمة التي تحدد أهميه لغة برمجة ما.

مقدمـه عن غرضية التوجـه

وددت أن أستطرد قليلا في شرح مبسط عن غرضية التوجه للتركيز على أهميه هذه التقنية وضرورة فهم ما معنى دعم لغة برمجة ما للـ OOP .

تعتمد هذه التقنية في مضمونها على ثلاث مفاصل :
- التغليف encapsulation.
- الوراثة inheritance .
- تعدديه الأشكال polymorphism.

التغليف Encapsulation :

تعتمد فكره غرضية التوجه على إخفاء البيانات .
وتستخدم الأصناف لتحقيق ذلك، حيث يتم إخفاء البيانات داخل الأصناف الخاصة بها، أو نقول بعبارة أخرى يتم تغليف البيانات داخل الأصناف .

عادة يتم توضيح هذه الفكرة باستخدام ما يسمى الصناديق السوداء (black boxes) ، حيث لا تضطر أن تعرف كيف تتم الأمور بالداخل وما هي المحتويات الداخلية ، وكل ما يهمك هو كيف تتعامل مع واجهه الصندوق الأسود وتعطيه معطياتك وتأخذ النتائج بغض النظر عن ما يتم في الداخل . إن ما يهمك فعليا من الصندوق هو آلية التعامل معه (مع واجهته) ولا تعطي اهتماما كبيرا عن تفاصيل داخل الصندوق.
خذ على سبيل المثال جهاز التلفزيون، إذا اعتبرنا التلفزيون صندوقا أسود، فإن كل ما يهمنا من هذا الصندوق هو كيفيه تشغيله وإطفاءه وتغيير المحطات وبعض الأمور الثانوية الأخرى، دون اهتمام بفهم الدارات الداخلية المكونة له، ودون الدخول في تفاصيل معالجته للإشارة وتحويلها إلى صوره وصوت .
إذن تخزن البيانات داخل الأصناف وعندها يمكننا أن نكتفي بمعرفه كيفيه استخدامها من الخارج .
إن كيفيه الاستخدام تدعى واجهه الصنف (class interface) وهي التي تسمح للأجزاء الأخرى من البرنامج باستخدام الأغراض المعرَّفة من هذا الصنف.

وبالتالي عندما تستخدم غرض ما فإن معظم شفرته تكون مخفيه، ونادرا ما تعرف ما هي البيانات الداخلية له حتى أنه قد لا توجد طريقة لدخول البيانات الخاصة به بشكل مباشر مالم تستخدم المناهج المتاحة على الواجهة والتي تسمح لك بتغيير وقراءه البيانات.

وذلك يعتبر من أهم الفروق بين البرمجة غرضية التوجه و البرمجة الكلاسيكية والتي تكون البيانات فيها عامة لكل الأصناف غير تابعة لصنف محدد كما أنك تستطيع تغيرها مباشره وبالتالي تقع في مطب ظهور أخطاء نتيجة عدم إمكانية اختبار القيمة المدخلة قبل إدخالها وذلك بسبب كون البيانات ظاهرة ويمكن الوصول المباشر إليها....

- **فائدة التغليف للمبرمج**

كما أن للتغليف ميزة سحرية للمبرمج نفسه لأنها تسمح له بتغيير التركيب الداخلي للصنف في التحديثات المستقبلية مع بقاء واجهه الصنف نفسها، وبالتالي ستطبق التغييرات تلقائيا على بقيه الأغراض التي استخدمت هذا الصنف بأقل عناء ممكن، دون الحاجة لتغير شفرتنا في مناطق مختلفة من البرنامج.

وذلك مثل حاله توابع API لها أسماء ثابتة، مهما تغيرت آلية عملها الداخلية طالما بقيت الأسماء كما هي ستبقى برامجنا تستدعيها بنفس الطريقة، وتغيير في بنية هذا التوابع لن يستوجب تغيير في البرامج التي تستدعيها .

الوراثة من أنماط موجودة:

غالباً ما نحتاج بناء نموذج مختلف قليلا من صنف موجود، بدلا من بناء صنف جديد من البداية، ربما نحتاج إضافة مناهج جديدة أو خصائص أو تعديل أخرى موجودة.

والفكرة من ذلك هي أننا نريد المتابعة من نقطه توقف الغير، وليس البدء من الأول لذلك سأقترح أننا نستطيع فعل ذلك بطريقتين:

- نسخ الشفرة من هناك ولصقها هنا . وبذلك ستضاعف شفرتك مرتين ، ناهيك عن الأخطاء ، والغرق في تفصيلات تبعدك عن مشروعك الأساسي ، والخروج عن مبدأ مركزيه الشفرة .

- لماذا لا تقوم بدلا من ذلك باستخدام إحدى أروع ميزات البرمجة الغرضية : ألا وهي الوراثة (inheritance) .

وبالتالي نستطيع ببضعة سطور شفرة فقط أن نبرمج كائن جديد (زر مثلا) يحوي جميع ميزات الزر العادي وزيادة بعض الخصائص والدوال الخاصة بنا .

ملاحظه :

إن البرمجة غرضية التوجه تفضل القدرة على التوسع و إعادة الاستعمال (reusability)، على أي شيء آخر، حيث أنك تستطيع كتابة شفرات تستخدم أصناف من بنية وراثيّة معقدة دون أي معرفة بالأصناف المحددة التي تشكل جزء من هذه البنية ،

وبكلمه أخرى تبقى هذه البنية الوراثيّة وبرامجك التي تستخدم هذه البنية قابله للتوسع والتغيير ، حتى بوجود آلاف السطور من الشفرة التي تستخدمها . ولكن بشرط واحد أساسي : أن يكون الصنف السلف لهذه الشجرة الوراثية مصمم بعناية فائقة .

وبالتالي تضمن أنك لن تتوقف في مرحلة ما عن تطوير المشروع لغرقك في كميه هائلة من الشفرة والتي أصبحت غير مفهومه وغير صالحه لكل الحالات.

وهذه هي تركه غرضية التوجه الأساسيّة، أنها جعلت تطوير البرامج الضخمة أمرا ممكنا وسهلا.

تعددية الأشكال (Polymorphism)

توفر هذه الخاصية القدرة على استدعاء المناهج المحددة في صنف ما اعتمادا على نوع هذا الصنف .

وتتيح الإشارة إلى صنف على أنه صنف آخر . *يكون بالعادة في نفس فرع شجره الوراثة .*

كالإشارة للصنف الابن على أنه أحد الآباء. مما يتيح اعادة الاستخدام للأصناف و جعل البرامج أكثر قابليه للتطور وأكثر تفاعليه في زمن التشغيل (Run Time) .

وبالتالي فالعملية نفسها تتصرف بشكل مختلف في أصناف مختلفة:

The same operation amy behave differently on different classes

تعددية الأشكال هي الجزء الأكثر صعوبة التي يعاني من فهمها الكثيرون .

لتبسيط الموضوع :

تتعلق تعددية الأشكال بدعم تصرفات تبدو كأنها غير قياسية في زمن التشغيل (runtime) حيث تتيح الكثير من الأمور .. خذ مثلا

- نسب متغير من صنف ما إلى متغيير من صنف آخر (ضمن قواعد DownCasting)
- إختبار من أي صنف هو المتغيير الحالي
- تنفيذ المناهج بناء على نوع الصنف الناتج.

أو مثلا المتحول Sender الذي يوجد في العديد من لغات البرمجه

(Delphi C++Builder C# وغيرها ..)

يمكن باستخدام هذا المتحول معرفة من هو الغرض الذي أطلق الحدث والتحكم فيه مع أن Sender حتما ليس من نفس صنف الغرض ، وفي حال قمنا بعملية قلب أنماط للتعامل مع sender على أنه من صنف آخر (زر مثلا Button) فإننا سنستطيع الوصول لكافة مناهج هذا الزر عن طريق المتغير sender .
أي أن sender متعدد الأشكال لإنه يمكنة التلائم حسب نوع الصنف الذي قام بإطلاق الحدث .. وهذه هي تعددية الأشكال بأبسط شكل لها .

منـاقشـــه :

هذه المزايا الثلاث (الأصناف، الوراثة، تعدد الأوجه) لازمة في أي لغة برمجة موصوفة بأنها لغة برمجة شيئية.

فالتغليف هو صلب غرضية التوجه الأساسي ويتم تمثيله برمجيا بالقدرة على بناء الأصناف . وفي حال كانت لغة البرمجة لا تدعم الأصناف فعندها لا أرى سببا من وجهه نظري لتسميتها لغة برمجة غرضية التوجه .

لغة الفيجوال بيزك Visual Basic دعمت الأصناف في الانتقال من الإصدارة ٤ إلى الإصدارة ٥ ، وربما تكون اللغة الأساسيّة بين اللغات المشهورة التي تمتاز بضعف في دعم OOP خاصة في الإصدارات المبكرة .
كما أنها لم تستطع دعم الوراثة بشكل جيد في نسخه VB 6 ، وتم دعمها لاحقا في لغة VB.Net. وأرى كذلك أن JavaScript تفتقد إلى الدعم الجيد الوراثة هي الأخرى.

ربما يمكنني القول أن VB هي الخاسر الأكبر بين اللغات الأكثر شعبيه في معيار دعم الـ OOP . ولكن VB.NET تداركت كل ذلك ، وبشكل عام في هذه الدراسة ما سنذكره عن C# سيكون موجودا في VB.net مالم نذكر خلاف ذلك .
كذلك لغة Perl تعاني ضعفا واضحا في هذه الناحية ، فهي ليست لغة غرضية التوجه تماماً ، وتم دعم الأغراض مؤخرا في دوره حياه اللغة ، كما أنها هي الأخرى لا تدعم الوراثة بشكل جيد ؟ .

لاحظ الشفرة اللازمة لتعريف غرض في Perl:

```
package MyClass;

sub new{

my $class = shift;

my $self;{} =

bless $self, $class

self->initialize();  # do initialization here

return $self;

}
```

ربما لو حاولنا كتابة الشفرة المقابلة لها في لغة Python لنتج معنا :

```
class MyClass:

pass
```

لغات البرمجة الغرضية الصرفة مقابل اللغات الهجينة:

من الفروق الأخرى بين لغات البرمجة التي تدعم البرمجة غرضية التوجه هو مدى دعمها لتقنيات البرمجة التركيبية التقليدية، واللغات الصرفة هي تلك التي تدعم النموذج الشيئي فقط.
فتستطيع التعامل مع الأصناف والمناهج فقط ولا تستطيع تعريف دوال وبرامج فرعيه تقليديه أو تعرف متغيرات عامه (global).
ومن بين اللغات التي نتحدث عنها يمكن اعتبار Eiffel و Smalltalk على أنها لغة شيئية صرفة مئة بالمئة.
كذلك يعتبرون الكثيرون أن Java لغة شيئية صرفة ، وهذا هو رأيي الشخصي .

يبدو للوهلة الأولى أن هذه الميزة سلبية لا سيما وأنك تستخدم العديد من المناهج الثابتة والتي لا تختلف كثيرا عن الدوال والبرامج الفرعية في اللغات الهيكلية، وكل ما تجنيه هو زيادة تعقيد الكود اللازم لتمثيل هذه الإجراءات.
في رأيي الخاص أن اللغات الشيئية الصرفة تقدم الكثير للمبرمجين المستجدين حيث أنها تجبرهم على استخدام تقنيات البرمجة الشيئية وتعلُّم هذا النمط من البرمجة.
وبنفس الوقت قد تنزعج من ذلك في كثير من الأحيان . حيث ستجبرك الكثير من هذه اللغات على فهم فلسفه البرمجة الغرضية كلها ولصقها أمام عينيك قبل البدء بكتابه تطبيقات ولو كانت بسيطة .
مثال لطباعه عبارة Hello world في جافا تحتاج إلى كتابه الشفرة التالية:

```
public class HelloWorld {
        public static void main (String[] args) {
            System.out.println("Hello, world!");
        }
    }
```

يجب أن يعرف المبتدئ العزل ، الأنماط ، المناهج ، الصنف String ، مصفوفات ، مناهج أصناف مقابل مناهج أغراض . يجب أن يعرف كل ذلك من أجل هذه الـ Hello World الصغيرة ؟ . وفوق ذلك يجب أن تحفظ بملف اسمه "HelloWorld.java`` .

في المقابل فإن باسكال الشيئية ولغة C++ و C# وPayton يمثلون نموذج من اللغات الهجينه أو المدمجة (hybrid languages)، فكل منهم يتيح للمبرمج أن يستخدم النموذج التقليدي بالإضافة لغرضية التوجه .

الوراثة المتعددة Multiple inheritance:

بعض اللغات تدعم تقنية في الوراثة تسمى الوراثة المتعددة ، أي أن نقوم بتوريث صنف ما من أكثر من أب (*CALCULATOR_WATCH* inherits from *CALCULATOR* and *WATCH*) وتعتبر لغة Eiffel داعم جيد لهذه التقنية .

الجدير بالذكر أن لغة C++ تدعم الوراثة المتعددة بشكل جيد أيضا (قد تجد في بعض الكتب وملفات بعض مترجمات ال++C تخويفا من استخدامها نظرا للمشاكل التي قد تحدث ، ولكن لازلت عند إعتبار دعم c++ للوراثة المتعددة نقطة متميزة) .

مع أن العديدون يعتبرون أن Eiffel تدعم هذه التقنية أفضل بكثير من C++ .

حتى Python تدعم الوراثة المتعددة بشكل جيد ، في حين أن كل من Java و Delphiو Smalltalk و C# (و VB.net طبعا) لا تدعم هذه التقنية.

ولكن (Java و Delphi و C#) يؤمنون دعم للواجهات Interfaces مما يؤمن القدرة على القيام بما تقوم به الوراثة المتعددة ويسد الثغرة التي خلفتها الوراثة المفردة ، ولكن بهذه الحال سنكون قد خسرنا إمكانية إعادة الاستخدام (Reuse) وسنضطر إلى إعادة اشتقاق الصنف كل مرة نريده بها .

ويقال أن للوراثة المتعددة كميه من المشاكل ، وينظر لها أنها صورة ايجابية من قبل بعض المبرمجين، ومسببه للمشاكل من وجهه نظر الآخرين.

ولكن الشيء الأكيد أن الوراثة المتعددة والمتكررة ستستحضر العديد من الأفكار الأخرى إلى الذهن كالأصناف الوهمية والتي ليس من السهل السيطرة عليها .

و مع ذلك فهي تعتبر قويه جدا .

يقول أحد خبراء Eiffel : "لاتصدق من يقول لك أن الوراثة المتعددة صعبه أو تسبب أخطاء ، هذا صحيح فقط في اللغات التي تدعمها بشكل سيء" .

على كل حال ماهو تفسير استغناء العديد من شركات إنتاج لغات البرمجة عن دعم الوراثة المتعددة ، خذ مثلا في منصة NET. ؟

لماذا لاتدعم C# الوراثة المتعددة ؟ ولماذا لاتدعمها لغة من الوزن الثقيل مثل Java ؟

هل فعلا السر في صعوبة دعم هذه التقنية ؟

لا أعرف ربما أكون مخطئا ولا يكون العيب من هذه اللغات. بل هو مجرد قوه إضافية تتمتع بها اللغات الداعمة للوراثة المتعددة.

طوبى لمبرمجي C++ على هذه الجوهرة الفريدة.

نموذج الأصناف الساكن مقابل نموذج مرجعيه الغرض :

من الفروق المهمة بين لغات البرمجة الغرضية هو كيفية تمثيلها للكائنات في الذاكرة.
فالبعض يستخدم المكدس و يمثل الكائنات بطرق ساكنة (Static) . وفي هذه اللغات سيكون للكائن جزء خاص من الذاكرة كما هي حال لغة C++ أو Eiffel. وبالتالي فإن تعريف متغير من صنف ما سينشيء تلقائيا منتسخا من هذا الصنف ويحجز له الذاكرة مباشره .
توجد لهذا التمثيل بعض الحسنات فهو يحسن مقروئية الشفرة ، ويمكّن المترجمات من التقاط الأخطاء في زمن التصميم قبل تحولها إلى مشاكل زمن-تنفيذ كريهة ، وهو أسهل وأسرع في العمل ، ومن المهم أن نتذكر أن السهولة دائما تقلل من الوقوع في الأخطاء .

مؤخرا ظهر اتجاه لاستخدام نموذج آخر يدعى نموذج مرجعيه الغرض (Object Reference model) في هذا النموذج يتم حجز الذاكرة الخاصة لكل صنف بطريقة ديناميكية في الكومة (heap) وتكون الكائنات فعليا مؤشرات لهذا المكان بالذاكرة.
والفكرة في ذلك أن تعريف متغير من نمط صنف ما لن يخزن الغرض داخله ، ولكنه يخزن مرجع لموقع الغرض في الذاكرة ، أي عبارة عن مؤشر (Pointer) يشير إلى موقع خاص بالذاكرة حيث يتم تخزين جسم الغرض هناك ، وليس ضمن المتغير نفسه ، لذلك يجب حجز ذاكرة للغرض قبل استخدامه وتحريرها بعد الانتهاء منه.
كما أنك تستطيع الإشارة إلى نفس الغرض من أكثر من متحول.
لغات مثل Java وPascal و Smalltalk تعتمد هذا الأسلوب ، حتى لغة C# الجديدة اتجهت إلى هذا الأسلوب .
ومع أن هذا الأسلوب يزيد العمل على المبرمج لكن له مزايا متعددة تزيد من التحكم بالأغراض وكذلك تعطي قدرة كبيرة على إدارة الذاكرة . ومنها :

- **يسمح بتقنيات أفضل في نسب الأغراض :**

بما أن المتغير الذي يحوي الغرض يدل فقط على العنوان الأساسي للغرض في الذاكرة ، فإننا نستطيع تعريف أكثر من متحول تدل جميعها على الغرض نفسه ،
فإذا قمنا بنسب متحول جديد ما من نفس الصنف إلى آخر تمت تهيئته فإننا نستطيع التعامل مع المتحول الجديد للدخول على الغرض نفسه دون أدنى مشكله .

- **يسمح بتمرير الغرض في الإجرائيات :**

أي تستطيع تمرير الغرض بهذه الطريقة على شكل بارامتر خاص بتابع ما ، ونعامل الغرض كمتحول عادي ونمرره إلى مناهج تقوم بمعالجته والتعديل فيه من مكانه .

وهذا يعني التوفير في الذاكرة ويسبب أداء سريع في هذه الحالة.

مثلا نفرض إجرائية لها بارامتر وحيد من الصنف TButton ، نمرر لها زر ما فتقوم بتغيير اسمه مثلا ، *المثال بلغة دلفي* :

```
procedure ChangeCaption (B: TButton);
begin
  B.Caption := B.Caption + ' was Modified';
end;
……
// call...
ChangeCaption (Button1)
```

المتحول الذي قمنا بتمريره كزر أعطى عنوان الذاكرة للإجرائية . وهذه بدورها دخلت آلية وقامت بالتعامل معه مباشرة.

هذا يعني أن الغرض تم تمريره بالمرجع بلا استخدام الكلمة المفتاحين Var التي نستخدمها بالحالة العادية للمتحولات، وبدون أي من التقييدات الأخرى التي تفرضها حاله التمرير بالمرجع (pass-by-reference).

- **تسمح بتحكم أكبر في إدارة الذاكرة :**

بما أننا قبل استخدام الغرض يجب أن نقوم بحجز الذاكرة له (Create)، وبعد استخدامه يجب أن نقوم بتحرير الذاكرة (Destroy) . هذا يعني أننا نملك التحكم الكامل بلحظه حجز الذاكرة ولحظه تحريرها، وبالتالي نستطيع إدارة الذاكرة بشكل ديناميكي ولن نخسر ذاكرة تطبيقنا حتى نحتاج لاستخدام الغرض، وبعد الانتهاء منه يمكننا تحريره مباشره لإعادة استثمار ذاكرته .

والاستخدام البسيط لمؤشرات المناهج وهي الفكرة وراء الأحداث، والفكرة المميزة للخصائص على أنها طريقة الوصول لبيانات الصنف فقد يكون وصولا مباشرا إلى البيانات أو عبر إجراءات محدده وبناءاً عليه فإن أي تغيير في طريقة الوصول لا يتطلب تغييرا مقابلا بالاستدعاء

ملاحظه:

في لغة C++ عاده ما تستخدم المؤشرات والمراجع للتعامل مع الكائنات ، وبهذا تحصل على خاصية تعدد الأوجه والقدرة على العمل بالطرقتين تقريبا (لكن بلا شك يتطلب ذلك ضعف العمل من المبرمج)

في حين أن نموذج تمثيل الكائنات بالمراجع يستخدم المؤشرات بشكل افتراضي ويقوم بعمل جيد لإخفاء التفاصيل عن المبرمج.

في لغة الجافا استثنائياً لا يوجد مؤشرات لكنها موجودة في كل مكان ولا يستطيع المبرمجون التعامل مباشره معها لذلك فإنها لا تشير إلى أماكن عشوائية في الذاكرة لأسباب أمنيه.

الإدارة الآلية للذاكرة ومجمع النفايات garbage collector:

والمقصود آلية استعادة الذاكرة التي تم تخصيصها لكائن ما بعد الانتهاء منه.

مساحات الذاكرة الساكنة المحجوزة بواسطة المكدس في لغة C++ يمكن استرجاعها بسهولة ، في حين أن استرجاع المساحات المحجوزة بشكل ديناميكي عاده ما يكون معقد إلى حد ما .
- انطباعي الأولي أن استخدام النموذج المرجعي للكائنات في C++ سيجعل العملية أكثر صعوبة وتعقيدا .
حيث أن C++ لاتملك مجمع نفايات (garbage collection) في المنفذات التجارية الشائعة .
-GC ليس ميزة قياسية من مجموعه مزايا C++ ، ويعتبر التعامل مع حجز وتحرير الذاكرة عملا صعبا ويتطلب بعض العناء والتعقيد ، وهو بيئة خصبه لتوليد أخطاء إدارة الذاكرة.
مجمع النفايات ليس جزء من الC++ القياسية ولكن يمكنك الحصول على العديد من خوارزميات مجمع النفايات من بعض المكتبات الخارجية.

C# تعمل ضمن بيئة NET. وهذا يعني وجود العديد من العمليات التي لاتحتاج لتحكم وتعامل مباشر من قبل المبرمج ، وبالتالي التخلص من العناصر غير المستخدمة يتم بواسطة مجمع النفايات (GC) التي تدعمه C# (ربما الدعم في مستوى أخفض ، وممكن أن يكون على صعيد منصة Net.) .

الجافا : تملك الجافا Garbage collection ، ليس هناك الكثير ليقال في هذا الموضوع حيث أنها تستخدم خوارزمية خاصة لاسترجاع الذاكرة بعد الانتهاء من استخدامها من خلال ما يسمى (virtual machine) وهذا بدون أدنى تدخل من المبرمج ما يعطي سهولة وراحة مثاليين في العمل . لكن لكل شيء ثمن فقد تحصل على بعض الأخطاء المنطقية بسببها.

الباسكال الشيئية : باسكال لا تملك طريقة لتفريغ الذاكرة كتلك الموجودة في الجافا ، ولاتحوي مجمع نفايات.
الدلفي : تدعم فكرة العنصر الأب الذي يكون مسئولا عن استرجاع الذاكرة التي كانت مستخدمة من قبل كل الكائنات الأبناء. مما يجعل الموضوع بالدلفي بسيط إلى حدما .
كما تحوي بعض التقنيات للتفريغ الآلي للذاكرة فيما يخص بعض الأصناف ك Tstrings وغيرها
- إذا كانت دلفي تملك GC فسيكون ضمن نسخ دلفي المعدة للعمل تحت Net. أما نسخ win32 فلا تحوي مجمع نفايات بالمعنى الحرفي للكلمة.

لغات أخرى مثل Smalltalk و Eiffel تملك Garbage collectionو إدارة جيده جدا للذاكرة

الاستدعاء المتأخر (وتعددية الأشكال) Late Binding and Polymorphism

إذا أعاد صنف ما تعريف منهج (method) تابع لأبيه ، فيجب على لغة البرمجة أن تعرف أي من المنهجين عليها أن تنفذ عند الاستدعاء. (أي أصبح لدينا أب يملك إجراء ما ، وأبن يملك نفس الإجراء . مع العلم ان قواعد down Casting تسمح لنا بنسب متحول الأبن إلى الأب لإنه من نفس الشجرة الوراثية . فإذا حدث ذلك كيف سنعرف أي من المنهجين المختلفين بالإنتماء والمتشابهين بالإسم يجب على لغة البرمجة ان تنفذ ؟)
لتمكين ذلك يجب أن يدعم المترجم عمليه الاستدعاء البعيد (late binding) فجملة الاستدعاء في الشفرة المصدرية لا تقوم بربط عنوان المنهج وقت الترجمة، لكنها تنتظر إلى وقت التنفيذ لتعرف أي منهج سيتم استدعاءه.

C++: في هذه اللغة فإن الاستدعاء المتأخر متاح فقط للمناهج الوهمية (virtual methods) مما يسبب البطيء عند التنفيذ في هذه الحالة .

Object Pascal: يمكن الحصول على هذه الميزة سواء للمناهج الوهمية أو الديناميكية، ويتم ذلك بإعادة تعريفها بالكلمة override.
وتعتبر هذه ميزة فريدة للباسكال الشيئية.

java : وهنا جميع المناهج تستخدم الاستدعاء المتأخر إفتراضيا، إلا إذا قمت يدويا بتمييزها على أنها مناهج نهائيه (final methods)

الفرق بين الجافا و السي في هذه الميزة فالسي تستخدم الاستدعاء المبكر افتراضيا على عكس الجافا ، إنما مردّه لأن السي صممت للحصول على أسرع ما يمكن من البرمجة الشيئية ، والاستدعاء الساكن أسرع في التنفيذ ، ولكنه يعتبر برمجيا محدود أكثر من الاستدعاء المتأخر في حين ان الجافا تضحي بالسرعة من أجل دعم التنوع الذي يغذي محموليتها .

– والباسكال على النقيض من اللغتين الأخريين في هذه الناحية فهي تسمح بتعريف منهج بناء أو هدام وهمي .

معلومات الأصناف في زمن التشغيل: RTTI Runtime Type Identification/Information

في اللغات الغرضية التي تدعم فحص الأنواع ، فإن المترجم يقوم بكل مهام فحص الأنواع لذلك هناك حاجه قليلة لحفظ معلومات النوع أو الصنف على وقت التشغيل ،
من ناحية ثانيه هناك بعض الحالات التي تستدعي حفظ هذه المعلومات مثل (downcast) لهذا السبب فإن لغات البرمجة التي نناقشها تدعم RTTI بشكل متفاوت.

C++: في الأصل إن لغة السي لا تدعم RTTI وقد أضيفت في وقت متأخر بشكل downcast وأتاحت بعض معلومات الصنف . فتستطيع أن تقارن مثلا إذا ما كان صنفين من نفس النوع .

الباسكال: الباسكال الشيئية وبيئة العمل المرئية تدعم وتتطلب RTTI. ليس فقط فحص النوع و downcast (باستخدام is و as) بل إن الأصناف تولد RTTI في حالة الجزء published مثلا.

في الواقع هذه الكلمة(published) تتحكم بجزء من RTTI .
حيث أن كل الفكرة خلف الخواص(properties) في بيئة الدلفي وملفات النماذج كلها تعتمد على RTTL بشكل رئيسي .
الخلاصة : دلفي / باسكال . تدعم وتطلب RTTI بشكل مثالي .

الجافا : وهي مشابهه للباسكال، فلدينا هنا أيضا أب واحد لجميع الأصناف يساعد على تتبع معلومات الصنف ، كما ان التحويل الآمن بين الأصناف هو التحويل الافتراضي ولكل صنف منهج يدعى GetClass الذي يعيد بعض المعلومات لوصف الصنف.

تستطيع في حال دعم RTTI أن تعرف معلومات عن صنف ما في زمن التشغيل .

القوالب والبرمجة الجينية : (Generic Programming)

البرمجة الجينية هي تقنية لكتابه الدوال والأصناف مع ترك بعض الأنواع البيانية بدون تحديد. يتم تأجيل ذلك إلى وقت استخدام هذه الدوال والأصناف في الشفرة المصدرية. أي أننا عندما نقوم بتعريف صف معين يكون له نوع محدد، لكن من أجل استخدام أوسع لنفس الصف يمكن أن يكون للصف نفسه أكثر من نوع و هذا من خلال استخدام الـgeneric و هي خاصية تتيح إمكانيه أن يكون الصف غير معرف النوع و يمكن استخدامه نفسه تحت أي نوع ..
كل شيء يترك يحدد بإشراف المترجم، ولا يؤجل أي شيء لوقت التنفيذ وأفضل مثال على هذا الأسلوب هو الأصناف الحاوية (container classes).

C++: وهي اللغة الوحيدة بين اللغات التي ناقشها التي تدعم هذه الخاصية ويشار إلى هذه الأصناف بالكلمة Template . الC++ القياسية تحتوي على مكتبه ضخمه من قوالب الأصناف تدعى STL التي تدعم أسلوب مميز وقوي بالبرمجة .

C# دعمت هذه الخاصية ، وقد أضيفت في النسخة الأخيرة للNET. وهي Version 2.0 .

الباسكال: لا يوجد لديها أي قوالب والأصناف الحاوية يتم بناءها لتحوي الكائنات من النوع TObject.

الجافا:لا تحتوي قوالب أيضا يمكنك استخدام الحاويات (containers) من الصنف Object أيضا .

التجريد Abstraction :

و هو يعني القدرة على تعميم الأغراض كأنواع (the ability to generalize an object as a data type that has a specific set of characteristics and is able to perform a set of actions.)
و تؤمن لغة البرمجة الدعم لهذه الخاصية من خلال الصفوف classes حيث تعرف الصفوف الخصائص و السلوكيات(properties and methods) ثم يمكن استخدامها كنوع data type ..
و تدعم هذه الخاصية أغلبية لغات البرمجة Smalltalk, Java, Python, Eiffel# Ruby
لكن كل من السي++ و الفيجوال بيسك و البيرل لا تدعمها ..

.. انتهت ..

إعــداد :

عـروه علـي عيسـى :

```
Web Site   :   www.orwah.net
E_Mail     :   WebMaster@orwah.net .
```

جـمـع المعلـومات :

لـغـات Python و Smalltalk و Perl و Lisp :

علاء الخـطـيـب

```
E_Mail     :   Dolphin2@scs-net.org .
```

لـغـات C# و VB.net :

حسـان إسـمـاعـيـل

```
E_Mail     :   HassanMi@scs-net.org .
```

شـكـرا لـمـوقـع AraB Team : www.arabteam2000 :

anwarica
HGB
undo
محـمـد سـمـير
OMLX
AhmedSameh

المناقشة الكاملة في الكتيب على موقعي في منتدى مخصص :
http://www.orwah.net/modules/newbb/viewforum.php?forum=23

```
Syria . 20 / 2 / 2005 .
```