Compiler By Ahmed Sabry

رحلة الكود من الكتابة إلى التنفيذ في .NET

عندما تكتب كود C# (أو أي لغة .NET أخرى)، لا يتم تحويله مباشرة إلى لغة الآلة (Machine Code) التي يفهمها المعالج. بدلاً من ذلك، يمر بعملية من مرحلتين تتيح لـ .NET ميزاتها الرئيسية مثل العمل على أنظمة تشغيل مختلفة (Cross-platform) وإدارة الذاكرة.

1. مرحلة الترجمة (Compilation)

عندما تقوم بالـ **Build** لمشروعك، يقوم الـ **Compiler** الخاص باللغة (مثل C# Compiler) بترجمة الكود المصدري (Source Code) إلى تنسيق وسيط:

- **الكود المكتوب (C# Code) إلي لغة وسيطة (IL)**
  - **الفائدة:** هذه الخطوة هي سر **مرونة .NET**. بدلاً من الاعتماد على معالج محدد أو نظام تشغيل، يتم تحويل الكود إلى صيغة موحدة يمكن تشغيلها في أي بيئة تدعم .NET.

2. مرحلة التنفيذ (Execution)

بعد مرحلة الترجمة، يأتي دور **CLR** لتنفيذ هذا الكود الوسيط.

🧠 المفاهيم الأساسية وعلاقاتها

إليك شرح للمفاهيم التي طلبتها وكيف تتشابك معًا:

1. ⚙️ CLR (Common Language Runtime)

هو **قلب** إطار عمل .NET، ويمكنك اعتباره **الآلة الافتراضية** (Virtual Machine) التي تقوم بتنفيذ تطبيقات .NET.

- **الفائدة:**
  - **تنفيذ الكود:** هو المسؤول عن تشغيل الكود المترجم (IL).
  - **خدمات رئيسية:** يوفر خدمات حيوية مثل **إدارة الذاكرة** (Garbage Collection)، **التعامل مع الاستثناءات** (Exception Handling)، و**الأمان** (Security).
- **العلاقة:** الـ CLR هو **البيئة** التي تعيش فيها وتعمل كل المكونات الأخرى (IL، CTS، CLS).

2. 📝 IL (Intermediate Language)

يُعرف أيضًا باسم **MSIL** (Microsoft Intermediate Language) أو **CIL** (Common Intermediate Language). هو لغة التجميع (Assembly Language) ذات المستوى الأدنى التي يتم تحويل الكود المصدري إليها.

- **الفائدة: الاستقلال عن اللغة والمنصة.** الكود المترجم إلى IL لا يحتاج إلى إعادة ترجمة لكل لغة أو نظام تشغيل. الـ CLR هو من يقوم بتحويله إلى لغة الآلة **في وقت التنفيذ**.
- **العلاقة:** الـ **Compiler** يترجم الكود المصدري إلى **IL**، ثم الـ **CLR** يأخذ هذا الـ **IL** ويستخدم آلية تُسمى Compiler (Just-In-Time) **JIT** لتحويله إلى لغة الآلة وتنفيذه.

3. 📦 EXE / DLL (Assembly)

ملف **exe.** (تنفيذي) أو **dll.** (مكتبة ديناميكية) هما ما يُعرفان باسم **Assembly**. هذا الملف هو الحاوية التي تحتوي على الكود الوسيط (**IL**) وبعض البيانات الوصفية (Metadata) التي تصف هذا الكود.

- **الفائدة:** هو المنتج النهائي لمرحلة الـ Build، وهو الوحدة التي يتم تحميلها وتشغيلها بواسطة الـ **CLR**.
- **العلاقة:** الـ Assembly هو الملف الذي يحتوي على **IL**، والـ **CLR** يقوم بتحميل هذا الـ Assembly لتنفيذ الـ **IL** داخله.

4. 🧬 CTS (Common Type System)

هي **المواصفات** التي تحدد كيف يجب أن تُعرف **الأنواع** (Types) - مثل الـ `class` ، `string` ، `int` - وكيف يجب أن تتفاعل فيما بينها في بيئة .NET.

- **الفائدة: التبادلية بين اللغات (Language Interoperability).** يضمن أن المتغير من نوع `int` المكتوب في C# يمكن أن يفهمه كود مكتوب في F# أو VB.NET. كل لغات .NET يجب أن تلتزم بهذا النظام الموحد.
- **العلاقة:** الـ **CTS** هو **القانون** الذي يجب على كل لغة .NET اتباعه، مما يضمن أن الـ **CLR** يمكنه إدارة كل هذه الأنواع بشكل موحد.

5. 📜 CLS (Common Language Specification)

هي **مجموعة فرعية** من قواعد **CTS** التي تحدد الميزات التي يجب أن تدعمها أي لغة .NET لكي تكون متوافقة بشكل كامل مع اللغات الأخرى.

- **الفائدة: التكامل بين اللغات.** تضمن أن مكتبة (DLL) مكتوبة بلغة ما (مثل C#) يمكن استخدامها بسهولة وفعالية من قبل لغة أخرى (مثل VB.NET). على سبيل المثال، الـ CLS تمنع استخدام الأحرف الكبيرة والصغيرة فقط (Case-sensitive only) لأسماء الدوال في الواجهات العامة.

- **العلاقة:** الـ **CLS** هي **المعيار** الذي يجب على مصممي اللغات أن يتبعوه لضمان **التوافق** (Interoperability) الكامل بين اللغات المختلفة التي تُترجم إلى **IL** وتُشغّل بواسطة **CLR**.

6. ⚡ JIT Compiler (Just-In-Time Compiler)

الـ **JIT Compiler** هو أحد المكونات **الرئيسية** داخل الـ **CLR**، وهو الذي يقوم بالتحويل السحري من **IL** إلى **كود الآلة** في **وقت التنفيذ**.

ما هو الـ JIT؟

ومعناه الحرفي "المترجم في اللحظة المناسبة". **Just-In-Time Compiler**، هو اختصار لـ JIT

كيف يعمل؟

1. **التحميل:** يقوم الـ **CLR** بتحميل ملف الـ Assembly (EXE/DLL) الذي يحتوي على **IL**.
2. **التنفيذ الجزئي:** عندما يحتاج برنامجك لاستدعاء دالة (Method) لأول مرة، يقوم الـ **JIT** بالآتي:
   - يأخذ كود **IL** الخاص بهذه الدالة فقط.
   - يترجمه فوراً إلى **كود الآلة (Native Machine Code)** الخاص بالجهاز الحالي.
   - يخزن كود الآلة هذا في الذاكرة المؤقتة (Cache).

1. **التشغيل والاستخدام المتكرر:**
   - يتم تشغيل كود الآلة المترجم حديثاً.
   - في المرات التالية التي يتم فيها استدعاء نفس الدالة، لا يحتاج الـ JIT لإعادة الترجمة؛ بل يستخدم كود الآلة المخزن مباشرة، مما يجعل التنفيذ أسرع.

💡 الفائدة: التوازن بين السرعة والمرونة

- **مرونة:** لا يحتاج البرنامج لإعادة الترجمة الكاملة لكل نظام تشغيل.
- **سرعة:** الكود يتم ترجمته إلى كود الآلة الأصلي (Native) لحظة استخدامه، مما يجعله أسرع من التنفيذ عبر مترجم خطوة بخطوة (Interpreter).

🖼️ ملخص بالعلاقات

يمكنك تخيل الأمر بهذه الطريقة:

- **المحرك CLR** هو.
- **الوقود IL** هو.
- **العبوة** هي (IL) التي تحتوي على الوقود **EXE/DLL**.
- **دفتر الشروط** هي (Types) الذي يحدد شكل ونوع الأجزاء **CTS**.
- **قواعد المرور** هي التي يجب أن يلتزم بها كل من يسير على الطريق (لضمان **CLS** التفاهم بين كل اللغات).

🗄️ Metadata و Class Loader

سؤال ممتاز، دعنا نفصل بين دور كل من الـ **IL** والـ **Metadata** داخل ملف الـ Assembly.

1. Metadata (البيانات الوصفية)

الـ **Metadata** هي البيانات التي تصف الكود الموجود في ملف الـ Assembly (EXE/DLL).

- **ماذا تحتوي؟** هي مثل "فهرس" أو "خريطة" للملف. تحدد:
  - الأنواع الموجودة (Classes, Interfaces).
  - الأعضاء الموجودة في كل نوع (Methods, Properties, Fields).
  - الاعتمادات (Dependencies) على مكتبات خارجية أخرى.

2. Class Loader (محمل الفئات)

الـ **Class Loader** هو مكون **داخل الـ CLR** وظيفته الأساسية هي قراءة واستخدام هذه الـ **Metadata**.

- **وظيفته:** عندما يحتاج تطبيقك إلى استخدام فئة (Class) معينة لأول مرة، يتولى **Class Loader** مهمة:
  1. البحث عن Assembly المناسب (DLL).
  2. قراءة الـ **Metadata** الموجودة داخل الـ Assembly للتحقق من وجود الفئة المطلوبة وتفاصيلها.
  3. تحميل هذا Assembly إلى الذاكرة وإعداده للعملية التنفيذية.

الـ **Class Loader** بيراجع التركيبات الأساسية فقط مثل:

- هل الـ Class موجود فعلاً؟
- هل الـ Method اللي هتستدعيه موجود؟
- هل signatures لـ parameters متطابقة؟
- هل الـ Types اللي بيعتمد عليها الكلاس متاحة؟
- هل الـ Inheritance صح؟
- هل الـ Assembly المحمّل متوافق مع الإصدار اللي اتبنى عليه؟

🧩 علاقة Class Loader بـ CLS و CTS

هناك علاقة مباشرة جداً بين هذه المكونات:

المكون	دوره	العلاقة بالمكونات الأخرى
CTS (Common Type System)	هو **القواعد** التي تحدد كيف يجب (Classes, int, string) أن تبدو الأنواع في ملفات الـ Assembly.	الـ **Metadata** (التي يقرأها Class Loader) يتم كتابتها وتكوينها بناءً على قواعد **CTS**.
CLS (Common Language Specification)	هي **المعايير الدنيا** لضمان التبادلية بين اللغات.	معين، يتأكد Assembly عند تحميل **Class Loader** (باستخدام Metadata) من أن الأنواع التي يتعامل معها تلتزم بقواعد **CLS** لضمان التوافق.

الخلاصة: هل هم داخل الـ CLR؟

نعم، كل هذه المكونات تعمل وتدار من داخل الـ CLR:

- الـ **CLR** هو **البيئة الجامعة**.
- الـ **JIT Compiler** هو جزء **داخل** الـ **CLR**.
- الـ **Class Loader** هو جزء **داخل** الـ **CLR**.
- الـ **CTS** و **CLS** هما **المواصفات والقواعد النظرية** التي يلتزم بها كل من الـ **Compiler** (أثناء الترجمة) والـ **CLR** ومكوناته (مثل JIT و Class Loader) أثناء التنفيذ.

ملخص العلاقة بين Class Loader و Metadata و CLR:

المكان	الوظيفة الأساسية	المكون
داخل ملف EXE/DLL	خريطة ووصف لجميع الأنواع والأعضاء في الـ Assembly.	**Metadata**
مكون داخل CLR	يقرأ الـ Metadata لتحديد ما يجب تحميله وكيفية تهيئته.	**Class Loader**
مواصفات نظرية	القواعد والمعايير التي يجب أن تلتزم بها الـ Metadata.	**CTS/CLS**
بيئة التشغيل	البيئة التي تضم الـ Class Loader وتدير التحميل والتنفيذ.	**CLR**

💡 جوهر الفرق: Static vs. Dynamic World

الفارق الجوهري بينهما هو:

- الـ **Compile-Time (ثابت - Static):** الفحص يتم على **النص المكتوب** (Source Code).
- الكومبايلر يرى **الخريطة** فقط.
- الـ **Runtime (ديناميكي - Dynamic):** الفحص يتم على **الواقع الحي** (Assemblies المحملة والبيئة المحيطة).
- الـ CLR يرى **الحركة في المصنع**.

🛠️ أولاً: Compile-Time Checks (فحص ثابت للمنطق والبناء)

هذا هو الفحص الأساسي الذي يقوم به **مترجم اللغة (C# Compiler)**. وظيفته منع الكود "الذي لا معنى له" أو "المستحيل البناء" من أن يرى النور.

🔷 ما الذي يتأكد منه الكومبايلر؟

الكومبايلر لا يستطيع "التنبؤ بالمستقبل"، ولكنه يضمن أن **البناء الداخلي** لكودك سليم منطقياً وقواعدياً.

1. **سلامة القواعد (Syntax):** هل الأقواس والمعاملات (Operators) والكلمات المفتاحية (Keywords) مكتوبة بشكل صحيح؟
   - **مثال تفصيلي:**

```
// Compile-Time Error: يجب أن تكون الفاصلة المنقوطة موجودة
int a = 10 ``
```

1. **سلامة الأنواع (Type Safety):** هل تستخدم المتغيرات بالطريقة التي صُممت لها؟
   - **مثال تفصيلي:**

```
string name = "Ali";
// Compile-Time Error: لا يمكن تقسيم نص على رقم
int result = name / 2;
```

1. **وجود المراجع (Existence Check):** هل الفئات والدوال التي تناديها موجودة بالفعل في المشروع أو في ملفات DLL المرجعية وقت البناء؟
   - **مثال تفصيلي:** إذا حاولت مناداة `NonExistentClass.SomeMethod()` .

🔷 الخلاصة في هذه المرحلة:

> إذا فشل فحص الـ Compile-Time، فلن يتم إنتاج ملف Assembly (EXE/DLL) من الأساس. الكود لن يتحول إلى IL.

🚀 ثانياً: Runtime Verification (فحص ديناميكي للبيئة المحيطة)

هذا هو دور **CLR (Common Language Runtime)** ومكوناته (Class Loader, Verifier, JIT). هنا يتم التعامل مع المتغيرات **التي لا يمكن معرفتها إلا عند بدء التشغيل**.

🔷 ما الذي يتم فحصه وقت التشغيل؟

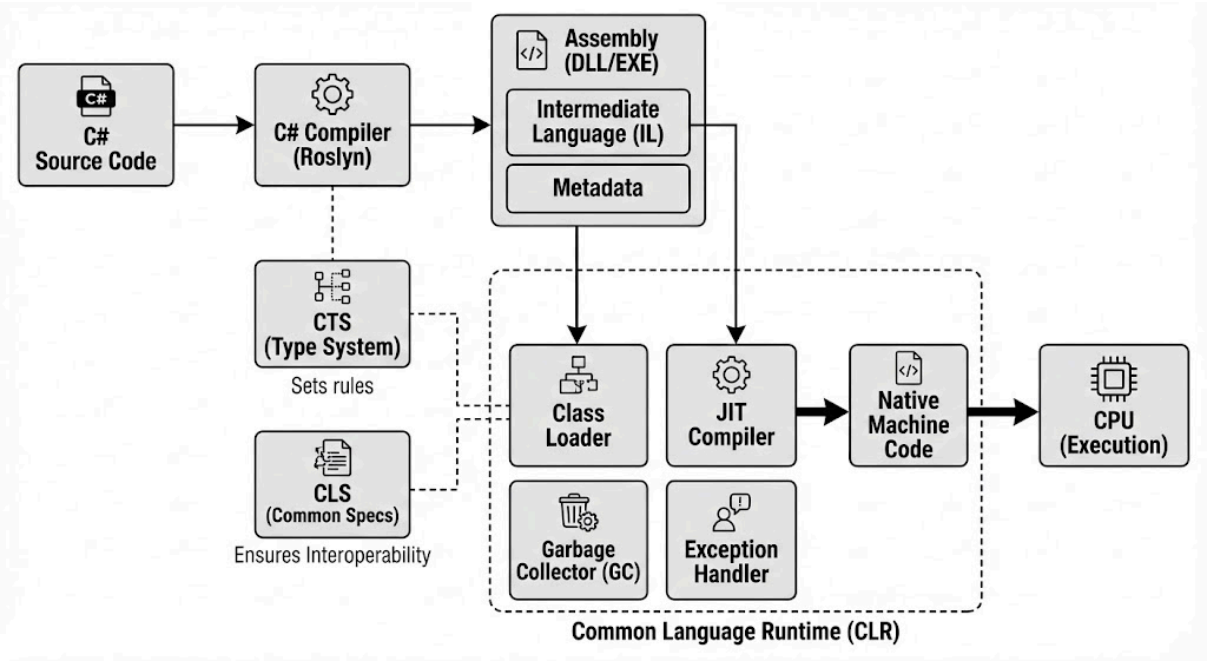
الفحص هنا يركز على سلامة **التفاعل** بين كودك والبيئة المحيطة، خاصةً العناصر التي تأتي من خارج الـ Assembly الخاص بك.

1. **تحميل الاعتمادات (Assembly Loading):** هل الـ DLL الذي يعتمد عليه كودك (باستخدام الـ **Metadata** التي قرأها الـ **Class Loader**) موجود في المسار الصحيح وبنفس رقم الإصدار المتوقع؟
2. **التوافق (Compatibility Check):** المثال الذي ذكرته ممتاز هنا:
   - **سيناريو MissingMethodException:** تم بناء كودك بنجاح لأنه وقت البناء كانت دالة `DoWork()` موجودة. لكن وقت التشغيل، **الـ Class Loader** يقرأ الـ **Metadata** الخاصة بـ `externalObject.dll` ويكتشف أن دالة `DoWork()` **غير موجودة الآن**.
   - **النتيجة:** يرفض الـ CLR استكمال التنفيذ ويرمي **MissingMethodException**.
3. **سلامة الكود (Code Safety / Verifier):**
   - يقوم الـ **Verifier** (مكون داخل الـ CLR) بفحص كود الـ **IL** قبل أن يترجمه الـ **JIT** للتأكد من أنه **آمن (Type-Safe)**. هذا يمنع أي محاولات ضارة للتلاعب بالذاكرة أو تجاوز القيود الأمنية.

🔷 أمثلة إضافية لأخطاء Runtime (لا يمكن للكومبايلر اكتشافها):

نوع الخطأ	متى يحدث؟	المكون المسؤول عن معالجته (داخل CLR)
DivideByZeroException	عندما تكون قيمة المقام صفر في وقت التنفيذ.	**JIT/Execution Engine**
StackOverflowException	عند دخول دالة في استدعاء ذاتي (Recursive Call) لا يتوقف.	**CLR** (إدارة الذاكرة)
InvalidCastException	محاولة تحويل كائن من نوع إلى نوع آخر غير متوافق فعلياً في وقت التنفيذ.	**CLR** الخلاصة

- **Compile-Time Checks:** يضمن **الجودة الداخلية** لوحدتك البرمجية.
- **Runtime Verification:** يضمن **التوافق والاستقرار الأمني** بين وحدتك البرمجية والبيئة المحيطة (الـ Assemblies والعالم الخارجي).



لاتنسوني من دعائكم 💚

a7medsabrii