Architecture Principles and Design Patterns

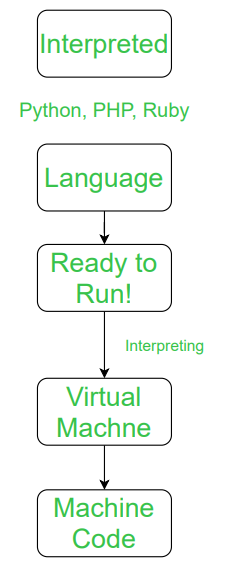
# Compiler and Interpreter

## Compiler

* Converts high level code (Human Understandable) to machine level code which is understandable by the platform.
* It converts the source code present in files directly to machine code for a particular platform (means an OS and Processor).
* Links various source code files into a single executable file(.exe).
* It requires storage for storing machine code to disk.
* **Platform = Operating System + Processor**
* Compiler complies with syntax rule of the programming language.
* A compiler understands the source and the target language.

## Interpreter

* Translates one statement at a time during the program run.
* Overall execution time is slower than the compiler.
* Compiler shows all errors at time of compilation.
* It converts to intermediate code and then takes and executes one by one.
* It doesn’t generate an output program.
* Interpreter exists in memory during program runtime.
* Suitable for all web based environments where load time is critical.
* It is one where the instructions are not directly executed by the target machine, but instead read and executed by some other program. Interpreted language ranges – JavaScript, Perl, Python, BASIC, etc.



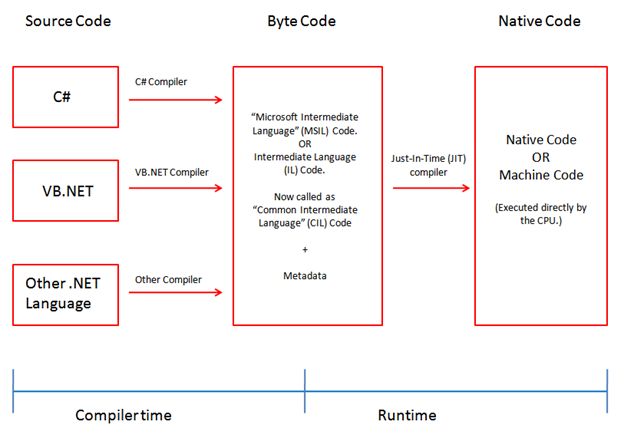
**Platform Independent:** The programs developed can run on any platform.

# C# Compilation

* C# is a purely compiled type safe object oriented language.
* C# is compiled into MSIL, by the c# compiler.
* This IL is then compiled just-in-time (JIT) as it's needed, into the native assembly language of the host machine. This is done to optimize the byte code for the underlying platform.

## .NET Code Execution Process

* The .NET code execution process involves 2 processes
  + Compile time Process
  + Run time Process

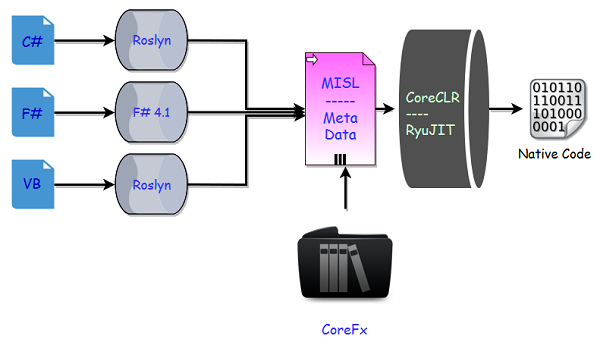
`

* **Compile Time Process**
  + .NET framework supports more than 1 programming languages like C#, F#, VB.NET etc.
  + Any of the compiler translates source code into MSIL aka CIL (Common Intermediate Language) and required metadata. The compiler compiles it into an .exe /dll.
  + The code converted to CIL is compliant with CLI (Common Language infrastructure).
  + The IL code and resources, such as bitmaps and strings, are stored on disk in an executable file called an assembly, typically with an extension of .exe or .dll. An assembly contains a manifest that provides information about the assembly's types, version, culture, and security requirements.
* **Run Time Process**
  + The CLR(Common Runtime Language) has a JIT Compiler for converting MSIL to Native/Machine Code.
  + When C# code is executed, the assembly is loaded into the CL, which might take various actions based on the manifest.
  + CLR provides other services related to automatic garbage collection, exception handling, and resource management. Code that's executed by the CLR is sometimes referred to as "managed code", in contrast to "unmanaged code", which is compiled into native machine language that targets a specific system.

****

* **CLR – Common Language Runtime –** It is a virtual execution system. CLR is implementation of Common Language Infrastructure (CLI).
* .**NET** supports concept of **Language interoperability**. Because the IL code produced by the C# compiler conforms to the Common Type Specification (CTS), IL code generated from C# can interact with code that was generated from the .NET versions of Visual Basic, Visual C++, or any of more than 20 other CTS-compliant languages. A single assembly may contain multiple modules written in different .NET languages, and the types can reference each other as if they were written in the same language.

# .NET Core Code Execution Process



* In .net core we have new series of compilers, Roslyn for C# and VB.NET.
* Roslyn can be also used for .net framework and c# above v6
* .NET core doesn’t have FCL(**Framework Class Libraries),**  it uses CoreFX( It is reimplementation of class libraries in .net core).
* We have a new runtime which is CORE CLR, which leverages JIT.

# Executing Angular Apps

* JavaScript is interpreted language whereas Angular is compiled language.
* Converting Typescript to JavaScript is the compilation part which happens.
* Angular supports JIT and AOT
* In JIT the angular components are compiled to JS after being loaded to browser on demand.
* In AOT, the compilation happens before the app is loaded in the browser where the JS is executed.
* main.ts  **>>**   app.Module.ts  **>>**  app.component.ts  **>>**  index.html  **>>**  app.component.html

# REST Principles (Representational State Transfer) – Roy Fielding, 2000

* It is an Architectural Style
* Used to design loosely coupled application over HTTP.
* This is for designing the web services

## Architectural Constraints in REST

* **Everything is a resource**: anything that needs to be identified on the web, needs to be treated as a resource and should be uniquely identifiable.
* **Each resource is identifiable by unique address (URI)**
* **Using HTTP standard methods** – GET, PUT , POST , DELETE
* Communications should be **stateless**: Every request should be a stateless request. On server client state is never stored or is passed on.
* **Communications are done via representations** (when we send any request or get any response it is a representation)
* **Cacheable**: every response should include if it is cacheable or not and for how much duration the response should be cached.
* **Client- Server Architecture**: REST app should have a client-server architecture.
* The components of a request are: HTTP Verb, URI, HTTP Version, HTTP Request Header and Http Request body.
* The components of a response are: Response code, http version, response header and body

Design Patterns

Singleton Pattern

* A class which allows only a single instance of it to be created.
* Most commonly, it doesn’t allow parameter to be passed when creating instance
* Instance is created lazily – i.e. not created until first requested.
* A singleton class has following properties
  + Single private paramterless constructor
  + The class is sealed, i.e. cannot act as a base class
  + A static variable which holds a reference to the single created instance.

## Not thread safe singleton class

public **sealed class Singleton**

{

**private static** Singleton **instance=null**;

**private Singleton**() { }

**public static** Singleton **Instance** { get {

if (instance==null) {

instance = new Singleton();}

return instance; } } }

* Two different thread cloud have evaluated if statement and found to be null and created a new instance which is violation. The memory model doesn’t stop us from creating two instances.

## Simply Thread Safe Singleton Class

**public sealed class** Singleton

{

**private static** Singleton instance = null;

**private static readonly object** padlock = new object();

**Singleton()**{ }

**public static Singleton** Instance { get {

**lock (padlock){**

if (instance == null) {

instance = new Singleton(); }

**return instance;** **}** } } }

* The thread takes locks of a shared object and then check if the object has been created or not. This overcomes memory barrier issue.
* This effects performance as every time lock is created once instance is requested.

## Not quite as lazy, but thread-safe without using locks

**public sealed class** Singleton

{

**private static readonly** Singleton instance = new Singleton();

// Explicit static constructor to tell C# compiler

// not to mark type as beforefieldinit

**static Singleton()**{}

**private Singleton**(){ }

**public static Singleton** Instance

{

Get {

return instance;

} } }

* static constructors in C# are specified to execute only when an instance of the class is created or a static member is referenced, and to execute only once per AppDomain.

## When to use singleton pattern?

* . When a class has no attribute, so it has no state and we are passing value as method parameters, in that case there is no need to create multiple objects. By creating single object we can call that methods multiple times.
  + - class Circle{  
                 
                public area (int radius)  
                   {  
                      //display area  
                       area=3.1412\*r\*r;  
                   }  
              }
* When the case is read-only state then at that case also there is no need to create multiple objects ,we can declare singleton
* Singleton pattern is used for logging, drivers objects, caching

## Singleton pattern in .net core

## Yet to Read

# Façade Pattern

## Using Service Façade to reduce controller dependencies

1. Façade is a way of grouping class togther. It provides unified interface to a set of interfaces in a subsystem.

# Mediator Pattern

# Single Responsibility Principles

* **DEF:** A class should have only one reason to change
* Adding too much responsibility in classes or interfaces may make them bloated
* Responsibility can be defined as reason for change.
* While designing classes the name of class should denote the responsibilities

## Benefits of SRP

1. Reduction of code complexity
2. Increased readability, extensibility and maintainability
3. Reusability and Reduced code
4. Better testability
5. Reduced Coupling

# Open Closed Principle – Bertrand Meyer

* Software entities (classes, modules, functions, etc) should be open for extension and closed for modification
* This says to write code to add new functionality to code without modifying existing code
* Bertrand meyer proposed to use inheritance for this
* *“*A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.”
* But inheritance introduces tight coupling if sub classes depend on parent class
* Robert C Martin proposed to use interfaces instead of super classes.
* Interfaces are closed for modification but new features can be added
* Inheritance introduces additional level of decoupling & introduces loose coupling
* **We should strive to write code that doesn’t have to change everytime the requirement changes**

# Liskov Substitution Principle

* It is guideline for creating inheritance hierarchies in which a client can reliably run any class or subclass without compromising the expected behavior.
* If we don’t follow LSP then extension to a class hierarchy (adding a new class) might necessitate change to any client of base class or interface.
* If we follow LSP clients can remain unaware of the changes to the class hierarchy. As long as there is no change to interface there will be no new reason for change.
* This allows implementation of open closed principle and SRP.
* Barbara Liskov: If S is a subtype of T, then objects of type T may be replaced with objects of type S, without breaking the program.
* There are 2 types of rules in LSP **Contact and Variance Rules**

[Read Contract and variance rules]

# Interface Segregation Principle

* Clients should not be forced to depend on interfaces which they don’t use.
* the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.