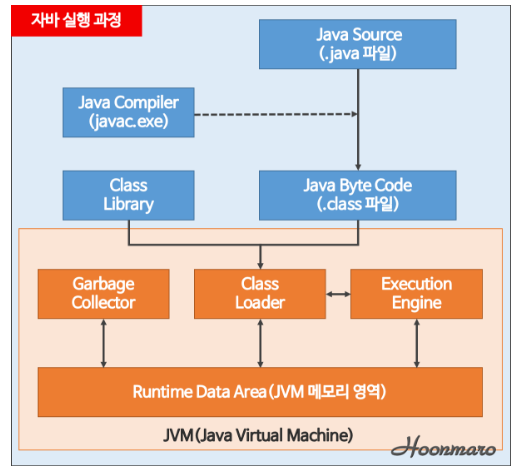
자바 메모리 구조

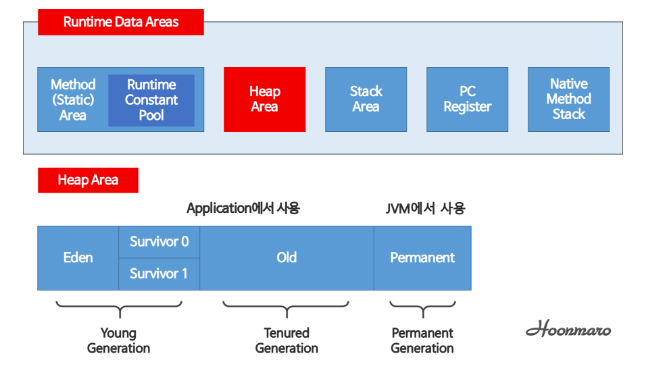
JVM 구조

* 실행될 클래스 파일을 메모리에 로드 후 초기화 작업 수행
* 메소드와 클래스 변수들을 해당 메모리에 배치
* 클래스로드가 끝난 후 JVM은 main 메소드를 찾아 지역변수, 객체변수, 참조변수를 스택에 쌓음
* 다음 라인을 진행하면서 상황에 맞는 작업 수행(함수호출, 객체할당 등)



* Class Loader : JVM내로 클래스를 로드하고 링크를 통해 배치하는 작업을 수행하는 모듈로써 런타임시 동적으로 클래스를 로드한다.
* Execution Engine : Claa Loader를 통해 JVM 내의 런타임 데이터 영역에 배치된 바이트 코드를 실행한다. 이 때 Execution Engine은 자바 바이트 코드를 명령어 단위로 읽어서 실행한다.
* Garbage Collector : JVM은 Garbage Collector를 통해 메모리 관리 기능을 자동으로 수행한다. 애플리케이션이 생성한 객체의 생존 여부를 판단하여 더 이상 사용되지 않은 객체를 해제하는 방식으로 메모리를 자동 관리한다.
* Runtime Data Areas : JVM이 운영체제 위에서 실행되면서 할당받는 메모리 영역이다. Class Loader에서 준비한 데이터들을 보관하는 저장소이다.

Runtime Data Areas 구조



* Method(Static) Area : JVM이 읽어들인 클래스와 인터페이스에 대한 런타임 상수 풀, 멤버변수(필드), 클래스 변수(Static 변수), 생성자와 메소드를 저장하는 공간이다.
* Runtime Constatnt Pool : 메소드 영역에 포함되지만 독자적 중요성이 있다. 클래스 파일 constant\_pool 테이블에 해당하는 영역이다. 클래스와 인터페이스 상수, 메소드와 필드에 대한 모든 레퍼런스 저장한다. JVM은 런타임 상수 풀을 통해 해당 메소드나 필드의 실제 메모리 상 주소를 찾아 참조한다.
* Heap Area : JVM이 관리하는 프로그램 상에서 데이터를 저장하기 위해 런타임 시 동적으로 할당하여 사용하는 영역이다. New 연산자로 생성된 객체 또는 객체와 배열을 저장한다. 힙 영역에 생성된 객체와 배열은 스택 영역의 변수나 다른 객체의 필드에서 참조한다. 참조하는 변수나 필드가 없다면 의미없는 객체가 되어 GC의 대상이 된다.
* Stack Area : 각 스레드마다 하나씩 존재하며, 스레드가 시작될 때 할당된다. 메소드를 호출할 때마다 프레임을 추가하고 메소드가 종료되면 해당 프레임을 제거하는 동작을 수행한다. 메소드 정보, 지역변수, 매개변수, 연산 중 발생하는 임시 데이터 저장. 기본(원시) 타입 변수는 스택 영역에 직접 값을 가진다. 참조 타입 변수는 힙 영역이나 메소드 영역의 객체 주소를 가진다.
* PC Register : 현재 수행중인 JVM 명령 주소를 갖는다. 연산 결과값을 메모리에 전달하기 전 저장하는 CPU내의 기억장치
* Native Method Stack Area : 자바 외 언어로 작성된 네이티브 코드를 위한 Stack. 즉 JNI(Java Native Interface)를 통해 호출되는 C/C++등의 코드를 수행하기 위한 스택이다. 네이티브 메소드의 매개변수, 지역변수 등을 바이트 코드로 저장한다.

객체지향의 SOLID 원칙

* 유지보수와 확장에 유용하고, 유연하며, 재사용성이 좋은 시스템을 만들 수 있게 해준다.

SOLID는 이러한 객체지향의 5가지ㅏ 핵심원칙 , SOLID원칙을 지킨다면 효율적으로 객체지향 프로그래밍을 구현할 수 있으며, 기존의 코드는 리팩토링하여 개선할 수 있다.

* 단일 책임 원칙(Single Responsibility Principle) -> 한 클래스는 하나의 책임만 가져야 한다. 클래스는 자신의 책임을 완전히 캡슐화 해야한다. 강아지 클래스는 강아지로서의 기능을 제공해야하며 그 이상의 책임을 가져서는 안된다.
* 개방-폐쇄 원칙(Open-Closed principle) -> 개체는 확장에 대해서는 열려있어야하고, 수정에 대해서는 닫혀있어야한다. 수정을 하게 된다면 모든 클래스에 영향이 갈 수도 있다. 하지만 확장은 기존의 코드 변경 없이 새로운 기능을 추가하거나 기존의 기능을 변경 할 수 있다. JAVA와 같은 객체 지향 언어는 추상화와 관련된 기능을 제공한다. 이를 통해 유연성,재사용성, 유지보수성 등을 보다 효율적으로 얻을 수 있다. 추상화 과정에서 is-a 관계인지 has-a 관계인지 파악하여 구현해야 한다. Ex)브루들링 움직임
* 리스코프 치환원칙(Lskov Substitution Principle) -> 다형성을 말할 수 있다. 하위클래스인 S클래스는 상위 클래스인 T클래스의 규칙을 지켜야 하며 , 그 성질과 책임이 유지되어야 한다. 즉 S클래스와 T클래스는 is-a 관계이어야한다. Ex)마우스
* 인터페이스 분리 법칙(Interface Segregation Principle) -> 큰 덩어리의 인터페이스를 구체적으로 나누어 클라이언트에게 꼭 필요한 메서드들만 이용할 수 있게 하는 것. 이는 인터페이스에 단일 책임 원칙으로 볼 수 있다. 인터페이스의 세분화 & 기능을 분리해야 한다.
* 의존관계 역전 원칙(Dependencty Inversion Principle) -> 상위 클래스는 하위 클래스에 의존해서는 안된다. 상위와 하위 모두 추상화에 의존해야한다. 상위클래스와 하위클래스의 관계를 느슨하게하는 것이다.