**1. BeanFactory**

 - BeanFactory 인터페이스가 정의됨

 - 기본적인 의존성 주입을 지원하는 가장 간단한 형태의 컨테이너

**2. ApplicationContext**

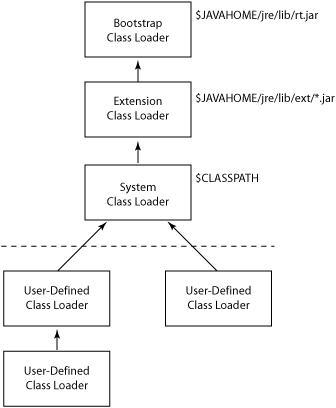
 - ApplicationContext 인터페이스로 정의됨

 - 프로퍼티 파일의 텍스트 메세지를 해석하는 능력이나 어플리케이션 이벤트를 관련된 이벤트 리스너에 발행하는 능력등과 같은 어플리케이션 프레임워크 서비스를 제공하는 BeanFactory의 개념위에 구현된 컨테이너

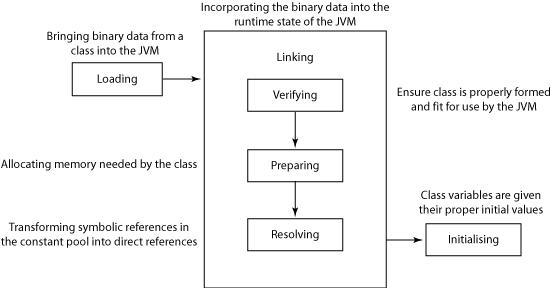
**3. Class Loader**

3.1 개요

**Class Loader** 란 abstrace class로써 Bytecode를 읽어 들여서 class 객체를 생성하는 역활을 담당한다.   
  
Class Loader가 Class를 Loading하는 시점은 ComplieTime이 아닌 Run Time에 Loading이 된다.   
  
프로그래머 가 SampleTest aaa = new SampleTest(); 라는 코드를 처음 실행하면 JVM은 SampleTest라는 Class를 Class Loader를 통해서 SampleTest.clas의 ByteCode를 최초로 메모리에 Load하게 된다.   
  
3.2 ClassLoader 기술적 특징  
   - Hierarchical   
     Class Loader는 Hierarchical 하게 생성이 가능하고, Parent class laoder에서, child class loader를 갖는것과 같이 **Class Loader간의 계층형 구조**를 갖게 된다. 최상위 Class Loader는 "**bootstrap**"   Class Loader가 위치 하게 된다.   
   - Delegate load request  
      Class loading에는 일정한 규칙이 필요하다. 이는 Class Loader가 계측형의 구조를 가지고  
      있기 때문에, 어느 시점에 Class 로딩 요청을 받으면 상위 Class Loader가 Loding한 Class가   
      우선권을 가진다.   
   - Have visibility constraint   
     Child Class Loader는 Parent Class Loader의 Clss를 찾을수 있지만, 그 반대의 경우는 불가능하다.  
   - Cannot unload classes  
      Class Loader에 의해서 Loading된 Class들은 unload 될수가 없다.   
3.3 Class Loader 위임  
   -



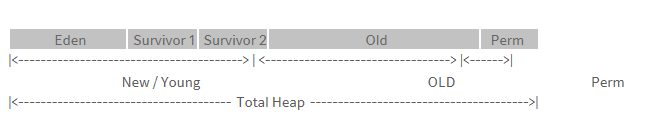
옆의 그림은 class loader delegation model 이다.   
     이는 Loading요청을 서로 에게 전달 하는 ClassLoader의 그래프이다.   
  
     옆의 그림과 같이 Class Loader들은 하나의 delegation parent와 함께 생성되고 다음 위치 에서 Class를 찾는다.   
  
     \* 다음 위치란  
**Cache, Parent, Self**  
  
    CalssLoader가 최초 하는 일은 전에도 같은 Class를 Loading 하도록 요청을 받았는지를 먼저 파악한다.   
    만약 요청을 받은적이 있으면, Cache에 저장된 Class를 리턴 한다. 그렇지 않을 경우, Parent Class를 Loading 할 기회를 준다.   
    이러한 두 단계들이 반복되고 Depth First 방식으로 탐색이 진행 된다.   
    만약 Parent Class에도 해당 Class가 존재하지 않으면 (ClassNotFoundException) ClassLoader는 Class의 소스에 대한 공유(작업한 코드)의 경로 (Self)를 검색한다.   
    Parent ClassLoader는 Class를 처음으로 loading할 기회가 주어지기 때문에 Root에 가장 가까운 ClassLoader에 의해 로딩 된다.   
   가장 핵심적인 bootstrap class들은 java.lang.Object같은 정확한 클래스 버전이 로딩되었는지를 확인하고, Class Loader가 그 자체 또는 부모(Parent)에 의해서 Loading 되었는지를 Class가 불 수 있도록 한다. 위에서 말했다 싶이 자식에 의해 Loading된 Class 들은 볼 수 없다.   
   - Bootstrap Class Loader  
   다른 ClassLoader들과는 달리 **Bootstrap Class Loader** 는 자바 코드에 의해서 instance화 될 수 없다.   
   Bootstrap ClassLoader 는 Boot ClassPath에서 핵심 시스템 Class들을 로딩하는데 이것은 일반적으로 JavaHome의 jre/lib 디렉토리에 있는 JAR 파일들이다.   
   단 **-Xbootclasspath** 옵션으로 classpath를 추가 수정할 수 있다.   
   - Extension Class Loader  
   주로 jre/lib/ext 디렉토리에 위치한 Extension 디렉토리에서 Class 들을 Loading 한다.   
   사용자의 classpath를 수정하지 않고 다양한 보안 확장 같은 새로운 확자으로 쉽게 갈 수 있는 기능을 제공 한다.   
   - System Class Loader   
   CLASSPATH 환경 변수에 지정된 결로에서 코드를 로딩 한다.   
   기본적으로 이 Class Loader는 사용자가 생성한 모든 ClassLoader의 Parent 이다.      
3.4 Class Loding 단계   
   - Class Loadig은 Loading, Linking, initialzing 단계로 나누어 진다.   
   - 대부분의 Class Loading과 관련된 문제는 이 세단계 중 한 단계에서 발생한다.   
   - 로딩 단계는 필요한 Class File을 배치하고 ByteCode로 Loading하는 단계 이다.  
   - JVM의 로딩 프로세스는 Class 객체에 매우 기본적인 메모리 구조를 제공한다.  
     메소드, 필드, 기타 참조 Class 들이 이 단계에서는 다루어지지 않는다.



   - Linking은 세 단계 중에서 가장 복잡한 단계이다. 다름과 같이 세 개의 주요 단계들로 나뉜다.   
     ㄱ. ByteCode 확인 - ByteCode가 정상적으로 작동 하는지 확인 한다.   
     ㄴ. Class Preparation - Class 에서 정의된 필드, 메소드, 인터페잇 들을 나타내는 데이터 구조를 준비한다.   
     ㄷ. Resolving - Class 에 다양한 방식으로 참조된 모든 Class들을 Loading한다.   
 - super class, interface, fild, method signature, local fild  
   - initialize 단계 동안, Class 내에 포함된 정적 initializer들이 실행된다. 이 단계의 끝에 가서는 기본 값으로 초기화 된다.   
   - 이 세 단계를 지나면 Class는 완전히 로딩되고 사용할 준비가 된다.   
     Class Loading은 Lazy방식으로 수행 될 수 있기 때문에 일부 Class Loading Processor 는 Loading 시점외에 처음 클래스를 사용할 때 수행될 수도 있다.      
 - Explicit Loading vs Implicit Loading  
   - 클래스가 로딩되는 두가지 방식의 차이   
   - Explicit : 다음과 같은 메소드 호출들 중 하나를 사용하여 로딩될 때 발생한다.   
                  - (java.lang.ClassLoader).loadClass()  
                  - Class.forName()  
   - 이러한 메소드들 중 하나가 호출 되면 이름이 인자로 지정된 클래스가 클래스 로더에 의해서 로딩된다.   
     그렇지 않을 경우, 로더는 클래스를 로딩하기 위해 delegation model을 사용한다.    
   - Implicit : Class 가 레퍼런스, 인스턴스화, 명시적 메소드 호출을 제외한 상속의 경우 로딩되는 방식

이 경우 슬그머니 로딩작업은 초기화되고 JVM은 필요한 레퍼런스를 결정하여 만약 로딩이 되어 있으면 레퍼런스만 리턴되고 그렇지 않을 경우 델리게이션 모델을 사용하여 클래스를 로딩하게 된다.

**4. Java 의 메모리 구조**  
   - Java의 Heap Memory 구조를 살펴 본다.   
   - Java의 Memory는   크케   **New/Young, Old, Perm**영역 이렇게 3가지로 구분된다.      
   - **New Object space :** 새로 생성된 객체들을위해 존재 한다.   
                                 이는 다시 Eden, Survivor1, Survivor2 로 구분되어 진다.   
   - **Old Object space   :** 객체가 만들어진 후 오래된 객체들을 위해 존재 한다.   
   - **Permanent space   :** java class, method Code등이 저장 되는 영역 이다.   
이를 그림으로 표현 하면﻿

  
   - Java 에는 GC - Garbege Collection - 이란 메커니즘이 동작 하는대 이는 Java Application에서 사용하지 않는 메모리를 자동으로 수거하는 기능을 말한다. 이 GC가 Java Heap 에서 어떻게 동작 하는지 알아 보자  
   - 먼저 자바의 객체가 생성되면 재일 먼저 Eden 영역에 저장된다.   
     이렇게 생성된 객체는 minor GC가 발생하면서 살아 있는 객체는 Survivor 2 영역으로 복사하고 Eden 영역을 Clear 한다.

이후 다음번 minor GC가 발생하면 Survivor2 와 Eden 영역에 살아 있는 책체를 Survivor1 으로 복사 되고 이 영역을 Clear 시킨다.

이런 식으로 minor GC를 수행하다가, Survivor 영역에서 오래된 객체는 Old 영역으로 이동하게 된다.   
   - 이렇게 OLD영역에 쌓인 객체는 Full GC가 일어 나면서 Clear 된다.   
   - Minor GC 의 알고리즘을 Copy & Scavenge 이라 한다.   
   - Full GC 의  알고리즘을 Mark & Compact 라 한다.   
   - 주의 할점은 이 Full GC 가 발생할때 Java Application은 완전히 멈춰 버리며 속도 또한 상당히 더디다.

**5. AOP(Aspect-oriented programming)**

##### 5. 1 AOP, Aspect-oriented programming?

- 객체 지향 프로그래밍에서 여전히 중복 코드(횡단 관심사)가 발생한다는 단점을 해결하기 위해 고안된 방식으로, 개발코드에서는 비즈니스 로직에 집중하고 실행시에는 비즈니스 로직 앞, 뒤 등 원하는 지점에 해당 공통 관심사를 수행하게 하여 중복 코드를 줄일 수 있는 프로그래밍 방식.   
  
- 모듈의 핵심 기능 이외의 기능이 하나의 모듈에 응집되지 않도록 하기 위한 기술

##### 5.2 AOP 관련 용어

1. aspect : 공통 기능.  
2. advice : aspect 자체를 의미.  
3. joinpoint : advice를 적용해야 하는 필드나 메소드.  (스프링은 메소드에 적용하는 기능만 지원)  
4. pointcut : advice가 실제로 적용되는 부분.

##### 5.3 AOP의 종류

1. before : 핵심 기능이 실행되기 전에 advice를 실행한다.  
2. after-returning : 핵심 기능이 정상적으로 종료된 후 advice를 실행한다.  
3. after-throwing : 핵심 기능을 실행하는 도중에 exception이 발생되면 advice를 실행한다.   
4. after : 핵심 기능을 실행하는 도중에 exception이 발생 여부에 관계없이 advice를 실행한다.   
5. around : 핵심 기능이 실행되기 전과 실행되고 난 후 exception 발생 여부에 관계없이 advice를 실행한다

**6. Maven or Gradle**

6.1 Maven

 - XML, remote repository를 가져올 수 있음

 - Strong convention but hard to implement custom logic

 - 라이브러리들이 서로 종속성을 가지는 경우 XML이 복잡해진다.

 - XML은 계층적 데이터를 표현하기에는 좋지만 플로우나 조건부 상황을 표현하기는 힘들다.

 - 상속형 (하위 XML이 필요없는 속성도 모두 표기)

6.2. gradle

 - Groovy, remote repository를 가져올 수 있음

 - convention 정책과 함께 custom 빌드 스크립트(groovy) 작성가능

 - 멀티플랫폼 프로젝트에도 적합(상위 스크립트, 하위 스크립트 독립적으로 작성)

 - 상속이 아닌 주입