요약

프로세스

프로세스는 보통 컴퓨터에서 연속적으로 실행되고 있는 컴퓨터 프로그램을 의미합니다. 조금 더 자세히 이야기하면

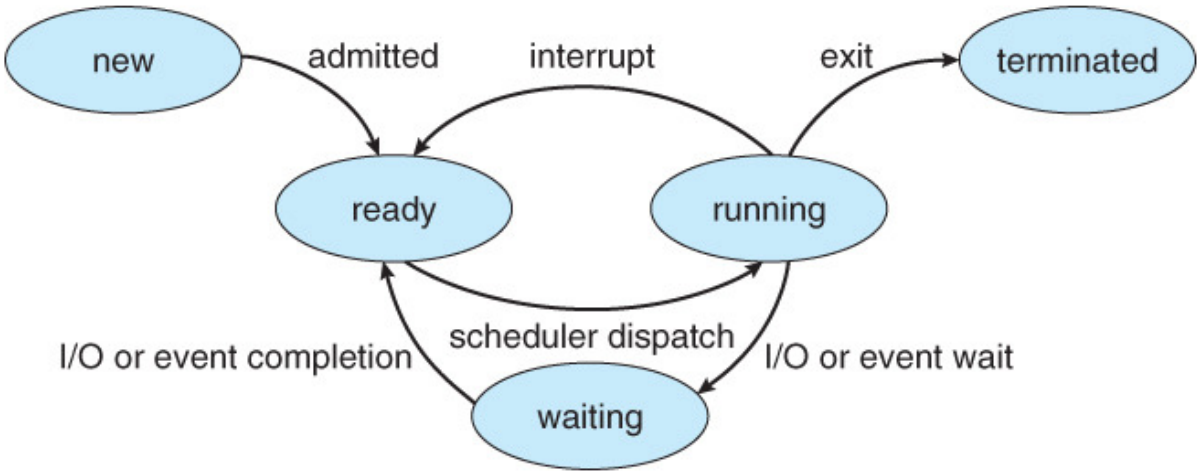
2차 저장장치인 보조 기억 장치에 저장되어 있는 실행 코드(프로그램)을 메모리에 적제(로딩)시켜 CPU의 시간을 할당 받을 수 있는 상태를 의미합니다.

이 때 프로세스는 코드, 데이터, 힙, 1개 이상의 스레드로 구성됩니다. 코드는 실행 코드이고 데이터는 전역 변수, 힙은 동적 데이터가 대표적으로 위치합니다.

스레드는 프로세스 안에 있는 다른 스레드와 코드, 데이터, 힙을 공유하면서 스레드마다 스택이라는 공간을 가지고 있습니다. 이 스택 공간에는

함수의 매개변수, 복귀주소, 로컬 변수등이 저장됩니다.

프로세스 상태



프로세스가 생성이 되면 **new** 상태가 됩니다. 그리고 메모리가 프로세스가 올라갈 수 있는 상황이 되면 **ready** 상태가 되고 프로세스가 CPU를 점유하게 되면 **running** 상태가 됩니다. 그리고 프로세스가 끝나게 되면 **terminated** 상태가 되고 점유시간을 다 써서 인터럽트가 걸리면 다시 **ready** 상태가 됩니다. 만약 점유시간 안에 I/O 요청이 들어오면 **waiting** 상태가 되고 I/O 요청이 완료되면 다시 **ready** 상태가 됩니다.

멀티 프로세스

하나의 응용프로그램을 여러개의 프로세스로 구성하여 각 프로세스가 하나의 작업을 처리하도록 하는 것입니다. 하나의 프로세스가 죽었을 때, 다른 프로세스에

영향이 가지 않는다는 장점이 있지만 운영체제에서 context switching을 할 때, 캐쉬 메모리를 초기화등의 무거운 작업을 해야하기 때문에 오버헤드가

발생하고 프로세스간의 데이터를 IPC(메시지 큐, 파이프, 세마포어)라는 복잡한 통신 기법을 사용해야한다는 단점이 있습니다.

멀티 스레드

하나의 응용프로그램을 여러개의 스레드로 구성하여 각 스레드로 하여금 하나의 작업을 처리하도록 하는 것입니다. 프로세스를 새로 만들 필요 없고 스레드 간의 데이터 통신이 간편하고 스레드의 작업량이 작기 때문에 context switching이 빠릅니다. 그리고 스레드 안의 스택 영역을 제외하곤 다른 스레드와 공유하기 때문에 통신 부담이 적습니다. 단, 하나의 스레드에 문제가 발생할 때 다른 스레드에 영향을 주고 자원의 동기화 문제가 발생하며 서로 다른 프로세스에서 스레드를 제어할 수 없다는 단점이 있습니다.

프로세스 스케줄링

운영체제는 CPU에 여러 개의 프로세스를 교체하면서 실행을 시키는데 이를 context switching이라고 부르고 어떤 방법으로 CPU에 할당하는가를

스케줄링이라고 부른다.

스케줄링의 종류

- FCFS : 먼저 온 서비스를 먼저 처리한다. 서비스가 다 처리될 때까지 CPU를 점유하기 때문에 호위 효과가 단점이다. (너무 긴 프로세스가 CPU를 차지하여 다른 프로세스가 CPU 점유를 기다리면서 효율성이 낮아지는 현상)
- SJF : CPU 점유시간이 작은 프로세스부터 먼저 CPU를 할당한다. 그러다보니 긴 프로세스는 계속 기다려서 기아가 발생할 수 있다.
- SRT : 남은 CPU 점유시간 짧은 프로세스부터 처리한다. 그렇기 때문에 나중에 온 프로세스의 점유시간이 짧으면 선점한다. 이 스케줄링도 SJF와 마찬가지로 기아가 발생할 수 있다.
- 우선순위 스케줄링 : 우선 순위가 높은 스케줄링을 먼저 실행한다. 기아나 무기한 봉쇄가 일어날 수 있다. 이는 aging을 통해서 해결할 수 있다.
- Round Robin : 각 프로세스에 동일한 크기의 CPU 할당 시간을 줘서 계속 context switch를 행하는 것이다. 이 할당 크기를 적절히 줘야되는데 너무 크면 호위 효과가 발생하고 너무 작으면 운영체제가 계속 context switch를 행해야하기 때문에 오버헤드가 발생한다.

스케줄러

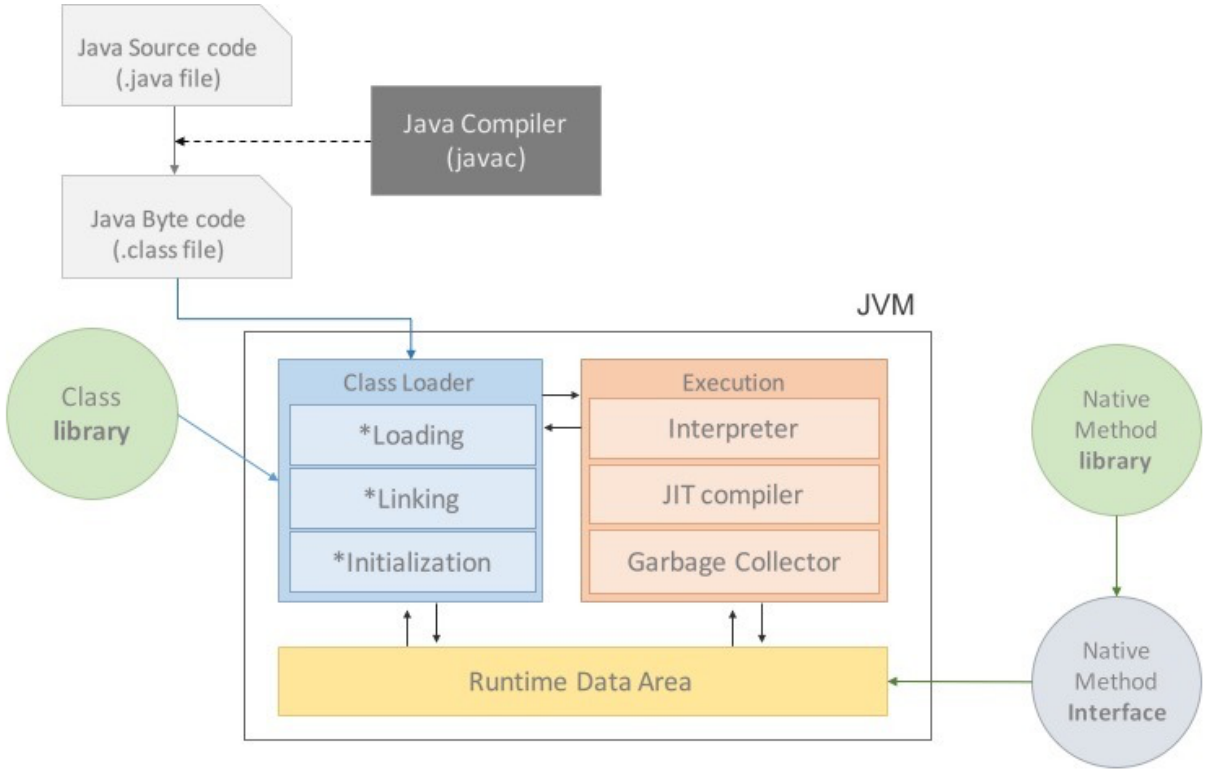
프로세스는 생성되는 시점에 보통 2차 저장 장치인 디스크에 있는 **job queue**에 머무르게 된다. 그리고 메모리의 공간이 허락하면 메모리에 있는 **ready queue**로 이동하게 되는데 이를 해주는 스케줄러가 **장기 스케줄러**다. 그리고 프로세스는 **CPU**와 **ready queue**를 이동하면서 **ready**->**waiting**->**running** 상태를 유지하는데 이를 결정하는 스케줄러가 **단기 스케줄러**다. 그리고 메모리의 공간이 부족한 경우 프로세스를 메모리에서 다시 디스크로 내쫓기도 하는데 이를 **suspended**라고 하며 중기 스케줄러가 담당한다.

JVM

Java Virtual Machine의 줄임말로 Java를 실행시키기 위해 물리적인 머신과 유사한 머신을 소프트웨어로 구현한 것입니다. JVM은 OS와 Java 프로그램 간의 중개자 역할을 함으로써 Java가 OS에 구애받지 않도록 합니다.

자바 프로그램이 실행되는 과정

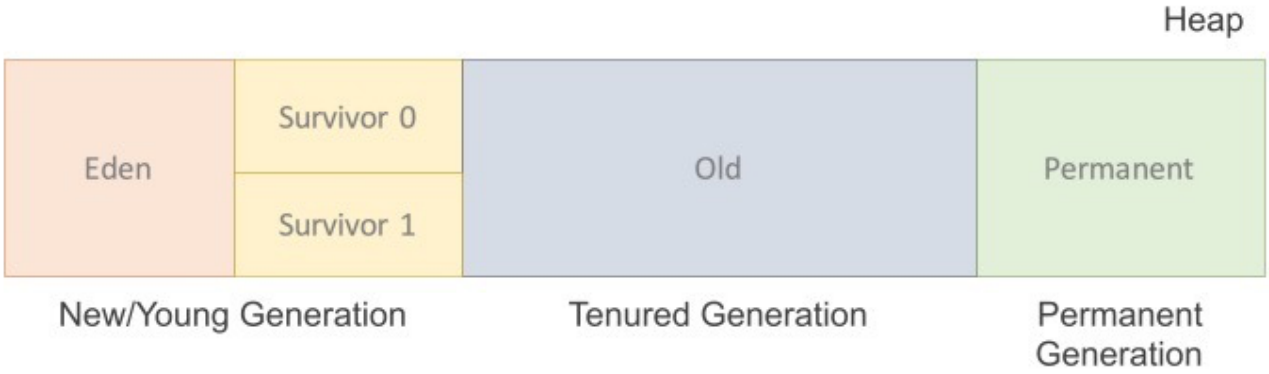
1. 프로그램이 실행되면 JVM이 OS로부터 프로그램을 실행시키는데 필요한 메모리를 할당 받습니다.
2. JDK에 존재하는 javac 컴파일러가 자바 소 파일을 바이트 코드로 변환합니다.
3. JRE에 존재하는 Class Loader를 통해서 class 파일들을 JVM에 로딩합니다.
4. 로딩된 파일들은 Execution Engine을 통해서 실행됩니다.
5. 해석된 바이트 코드는 Runtime Data area에 배치되어 실질적인 수행이 이루어집니다.



Garbage Collection

우리 말로 하면 "쓰레기 수집"입니다. 좀 더 Java에 맞게 해석을 하면 "사용자 되지 않는 인스턴스로부터 메모리를 회수하는 행위"를 의미합니다.

Java에서 힙영역은 아래와 같이 구분됩니다.



1. Java 프로그램이 실행하면 새로운 인스턴스는 **Eden** 영역에 생성된다. **s0**과 **s1**은 비워진 상태로 시작한다.
2. **Eden**영역이 가득차면 **Minor GC**가 실행된다. 이를 통해 현재 참조되고 있는 인스턴스는 **s0**로 이동되고 **Eden** 영역은 클리어된다.
3. **Eden** 영역에 있던 인스턴스가 **s0**으로 이동하는 것을 반복하면 **s0**의 영역이 가득차게 되는데 이 때, **s0**에 있던 인스턴스들 중에 사용되지 않는 인스턴스들은 메모리를 반납되고 사용되는 인스턴스는 **s1**로 이동한다. 그리고 해당 인스턴

스의 **age**라는 값이 증가된다.
4. **Minor GC**가 발생하면서 **s1** 또한 가득차게 되면 다시 **s0**으로 인스턴스가 이동한다. **survivar**영역을 옮겨 다닐 때마다 **age** 값이 증가하고 일정 값이 이상이 되면 **old** 영역으로 인스턴스가 옮겨진다.
5. 위와 같은 과정이 반복되면서 **old**영역이 가득차게되면 **Major GC**가 발생하는데 이 때, **old** 영역에서 사용되지 않는 인스턴스의 메모리가 반환된다.

GC가 실행될 때마다 **GC**가 실행되는 스레드를 제외하고 멈추게 되는데 이를 **stop the world**라고 부릅니다. 그래서 코드를 작성할 때,

JVM 스스로 **GC 알고리즘**에 게 자원을 회수하도록 해야하고 직접 **GC**를 호출해서는 안됩니다.

Java의 특징

- 운영체제에 독립적이다.
- 객체지향언어로 비교적 이해하고 배우기 쉽다.
- GC에 의해 자동적으로 메모리를 관리해준다.
- 오픈 소드다.
- 동적 로딩을 지원한다.
- 비교적 속도가 느리다
- 예외처리가 불편하다.

객체 지향 프로그램의 특징

- 캡슐화 : 객체가 손상되지 않도록 객체의 내부 구현 내용을 숨기는 것을 의미한다.
- 상속 : 이미 구현되어 있는 기능을 다른 객체가 그대로 사용하거나 변형해서 사용할 수 있도록 해주는 기능이다.
- 다형성 : 선언한 메소드나 클래스를 다양한 방법으로 동작시키는 것을 의미한다.
- 추상화 : 특정 사물에 대해서 공통적인 속성 또는 기능을 정의하는 행

오버로딩과 오버라이딩

- 오버로딩 : 메소드를 재정의해서 사용하는 행위다.
- 오버라이딩 : 메소드의 이름을 중복해서 사용하는 행위다. 단, 메소드의 이름은 같아야 하고 반환형은 달라도 되며 파라미터의 갯수나 자료형이 달라야한다.

추상 클래스와 인터페이스

추상 클래스와 인터페이스 모두 특정 사물에 대해서 공통적인 속성 또는 기능을 정의하는 추상화 개념을

정의하는데 사용되고 이런 추상화 과정을 통해서 비슷한 기능이지만 결과나 구현 내용이 다르게 하여

다형성을 확보하는데 용이합니다.

공통점

- 추상 메소드, 디폴트 메소드, 정적 메소드를 선언할 수 있습니다.
- 해당 클래스를 직접 사용하기보단 별도의 클래스를 상속 또는 구현해서 사용합니다.

사용법에서의 차이점

- 인터페이스는 객체화할 수 없습니다.
- 인터페이스는 변수를 가질 수 없고 상수만 가질 수 있습니다.
- 추상 클래스는 다중 상속이 불가능합니다.

SOLID

- SRP(Single Resposibility Principle) : 책임의 기본 단위인 객체가 하나의 책임만 가져야 한다는 원리입니다. 하나의 객체에 너무 많은 책임을 가지고 있다면 교체, 변경하는데 너무 많은 코드가 수정되야할 수 있기 때문입니다. 그래서, SRP를 객체가 변경되어야 하는 이유는 단하나여야 한다라고도 설명할 수 있습니다.
- OCP(Open Close Principle) : 확장과 변경에는 개방되어 있어야 하지만, 수정에는 닫혀 있어한다는 원칙입니다. 예를 들어, 의존 객체만 변경한다면 코드는 수정하지 않았지만 기능을 확장하거나 변경할 수 있습니다.
- LSP(Liskov Substitution Principle) : "서브클래스는 슈퍼클래스의 한 종류다." 라고 말을 합니다. 이는 의존 객체를 상속을 통해 만든 서브클래스로 사용 했을 때 문제가 없어야 한다는 의미다. 그래서 LSP를 잘 지키는 가장 쉬운 방법은 오버라이딩을 하지 않는 것이다.
- ISP(Interface Segregation Principle) : 인터페이스를 필요에 따라 분리하여 정의해야한다는 원칙이다. 예를 들어, 복합기라는 인터페이스가 있다면 이를 프린터, 스캐너, 팩스로 분리하여 인터페이스의 책임을 분리해야한다는 의미다.
- DIP(Dependency Inversion Principle) : 주입되는 의존 객체를 변화하기 어려운 추상 클래스나 인터페이스를 주입하라는 원칙이다. 만약 쉽게 변화하는 객체를 주입하면 요구 사항이 변경됬을 때, 해당 코드를 많이 수정해야할 수도 있기 때문이다.

Call By Value VS Call By Reference

함수를 호출할 때, 매개 변수 값을 어떤 방식으로 넘기냐에 따라 크게 **Call By Value**와 **Call By Reference**로 분류한다. **Call By Value**의 경우 값을 복사해서 매개변수의 값을 넘기고 **Call By Reference**의 경우 실제 메모리의 주소 값을 매개변수로 넘긴다. Java의 경우 **Call By Value**로 값을 직접 복사해서 넘기지는 않고 값을 참조할 수 있는 참조 변수를 넘긴다. 그렇기 때문에 이 참조값을 변하지 않지만 함수 내에서 참조 값에 의해 찾을 수 있는 실제 변수 값은 변경할 수가 있다.

Java Collection

- Map : Key Value 형태로 데이터를 저장할 수 있는 구조다.
- List : 순서가 있는 Collection으로 데이터를 중복해서 저장할 수 있는 구조다.
- Set : 집합적인 개념의 Collection으로 순서에 의미가 없고 데이터의 중복이 없다.

리플렉션

구체적인 클래스의 타입은 알지 모르는 상황에서도 그 클래스의 메소드, 타입, 변수에 접근할 수 있게 해주는 자바 API입니다.

이게 가능한 이유는 자바 클래스 파일은 컴파일이 되면 Static 영역에 위치하게 되는데 이 정보를 클래스의 이름만 알면 참고할 수 있기 때문이다.

```
val account = em.find(Account.class, 1L)
```

위는 JPA에서 1L을 아이디로 엔티티를 찾는 코드인데 실제로 실행되는 시점에서는 쿼리에 의한 반환 값이 구체적으로 어떤 클래스를 원하는지 모릅니다.

그런데 리플렉션을 통해서 클래스에 대한 정보를 넘기면 반환 값을 클래스에 대한 정보를 통해서 생성하여 반환합니다.

어노테이션

컴파일러에게 코드 작성 문법 에러를 체크하도록 정보를 제공하거나 실행시 특정 기능을 실행하도록 정보를 제공한다.

빌트인 어노테이션과 커스텀 어노테이션으로 구별되는데 빌트인 어노테이션의 경우 자바 SDK에서 지원해주는 어노테이션을 의미하고

커스텀 어노테이션은 유저가 정의 어노테이션으로 **@interface**를 통해서 정의한다.

제네릭

클래스를 선언할 때 타입을 결정하지 않고 객체를 생성할 때 유동적인 타입으로 재사용하기 위한 것입니다.

== 과 equals

==는 참조 주소를 비교하고 equals는 객체의 동등성을 검사한다.

cf) 동일성 => == && equals, 동등성 equals

자바 메모리 구조

- 메서드 영역 : static 변수, 전역 변수, 코드에서 사용되는 class 정보 등이 올라간다. 코드에서 사용되는 class들을 로더로 읽어 클래스별로 런타임 필드 데이터, 메서드 데이터 등을 분류해 저장한다.
- 스택 : 지역 변수, 함수등이 할당된다.
- 힙 : new 연산자를 통해 생성된 객체들이 저장된다. GC에 의해서 메모리가 관리된다.

직렬화

자바에서 입출력을 하기 위해서는 스트림이라는 통로를 이용해야한다.

근데 이 스트림은 바이트형으로만 통과할 수 있다. 그래서 객체를 스트림을 통해 입출력하려면 바이트 배열로 변환하는 것이 필요한데 이를 직렬화라고 한다.

serialVersionUID

직렬화와 역직렬화가 이루어질 때, 클래스의 버전 정보가 달라지는 것을 방지합니다.

만약, 클래스의 버정 정보가 다르면 직렬화된 클래스를 역직렬화 하지 못합니다.

DI

클래스 A, B가 있을 때, A에서 클래스 B를 사용하고 있고 클래스 B의 변화가 A에 영향을 줄 때, 클래스 A는 클래스 B에 의존한다고 한다라고 하고

클래스 B를 의존 객체라고 한다. 그리고 의존 객체를 직접 인스턴스화 하지 않고 외부로부터 주입 받는 것을

의존 주입이라고 한다. 대표적인 의존 주입 방식은 생성자 주입 방식과 수정자 주입 방식이 있다.

IoC

객체의 생성주기를 프로그래머가 아니라 프레임워크에 위임하는 상태를 말한다. 그래서 Inversion of Control,

제어의 역전이라 부른다. 스프링의 경우 스프링 컨테이너가 이 역할을 한다. 스프링을 IoC를 통해서

의존 자동 주입과 AOP를 가능하게 해준다.

AOP

AOP는 Aspect Oriented Programming의 약자로 한글로 말하면 관점 지향 프로그래밍이라고 부릅니다.

AOP의 핵심은 공통기능과 핵심기능을 분리하여 핵심기능을 수정할 때 공통기능을 수정하지 않게 하고

반대로 공통기능을 수정할 때 핵심기능을 수정하지 않게 하는 것입니다.

스프링의 경우 프록시 객체을 생성하여 프록시 객체가 공통 기능과 핵심기능을 모두 호출할 수 있어

적절한 절차에 따라 공통기능과 핵심기능을 호출합니다.

빈 라이프 사이클

스프링 빈은 스프링 컨테이너가 생성되고 초기화 될 때, 빈이 생성되고 초기화 된다. 그리고 스프링 컨테이너가 종료되기 전에 소멸되는데 그 주기는 빈 생성, 의존 설정, 초기화, 소멸의 과정을 거친다.

MVC

Model View Controller의 약자로 애플리케이션을 세가지의 역할로 구분한 개발 방법론이다.

사용자가 Contoller를 조작하면 Controller는 Model을 통해서 데이터를 가져오고 그 정보를 바탕으로 시각적인 표현을 담당하는 View를 제어해서 사용자에게 전달하게 된다.

BigO

알고리즘이 어떤 문제를 해결하는데 걸리는 시간을 시간 복잡도라고 하고 주로 빅오표기법을 이용해 나타낸다.

입력 값이 n개 일 때 걸리는 시간을 비교해 알고리즘의 효율성을 표현하기 위한 방식이다. 이 때, n의 값은 충분히 크다고 가정한다.

O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n!) < O(n^n)

DFS (Depth-First Search)

루트 노드(혹은 다른 임의의 노드)에서 시작해서 다음 분기(branch)로 넘어가기 전에 해당 분기를

완벽하게 탐색하는 방법이다. 넓게 탐색하기 전에 깊게 탐색하는 것으로 모든 노드를 방문 하고자 하는 경우 이 방법을 선택한다.

BFS보다 좀 더 간단하고 단순 검색 속도 자체는 BFS에 비해서 느리다. 스택을 이용해서 구현한다.

BFS

루트 노드에서 시작해서 시작해서 인접한 노드를 먼저 탐색하는 방법이다.

깊게 탐색하기 전에 넓게 탐색하는 방법으로 두 노드 사이의 최단 경로 혹은 임의의 경로를 찾고 싶을 때 사용한다.

큐를 이용해서 구현을 한다.