Java

# Index	0
CreatedAt	@September 28, 2022
Person	A Ally Hyeseong Kim
Status	In Progress
Tags	JAVA
UpdatedAt	@September 28, 2022

References

Java의 정석 - YES24

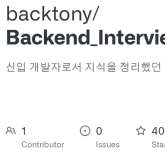
인하대학교에서 물리학을 전공하였으며 삼성SDS에서 첫 직장생활을 시작했다. 삼성멀티캠퍼스에서 자바를 강의하였으며, 여러 직장을 거쳐 현재는 프리랜서로 집필과 강의를 하고 있다. http://www.javachobo.com과 http://cafe.naver.com/javachobostudy.cafe를 운영하고 있

http://www.yes24.com/Product/Goods/24259565

Backend_Interview_for_Beginner/Java.md at master · backtony/Backend_Interview_for_Beginner

예비 답안 보기 (👈 Click) 예비 답안 보기 (👈 Click) 객체 지향 프로그래밍이란 프로그램 구현에 필요한 객체를 파악하고 객체들 간의 상호작용을 통해 프로그램을 만드는 것 을 말한다.

https://github.com/backtony/Backend_Interview_for_Beginner/blob/master/Java.md



면접 시리즈1 - Java

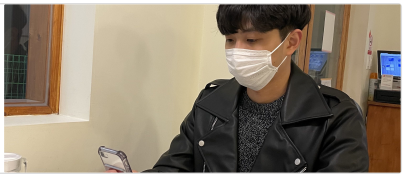
면접 시리즈 전체 링크 (👈 Click) 컴파일 타임에 null값에 대한 잘못된 접근을 감지 -> null 안전성 간결한 코드 -> data class와 같이 보일러플레이트 코드를 줄일 수 있다. 자바 호환성 장점 운영체제에 독립적 JVM에서 동작하기 때문에 플랫폼에 종속적이지 않다.

https://velog.io/@backtony/%EB%A9%B4%EC%A0%91-%EC%8B%9C%EB%A6%AC%EC%A6%881-Java

'전공공부/자바 (JAVA)' 카테고리의 글 목록

Junha's Study Note

https://jooona.tistory.com/category/%EC%A0%84%EA%B3%B5%EA%B3%B5%EB%B6%80/%EC%9E%90%EB%B0%94%20%28JAVA%29



'👩‍💻TIL/기술면접' 카테고리의 글 목록

'👩‍💻TIL/기술면접' 카테고리의 글 목록

https://devuna.tistory.com/category/%F0%9F%91%A9%E2%80%8D%F0%9F%92%BBTIL/%EA%B8%B0%EC%88%A0%EB%A9%B4%EC%A0%91

[Java] 객체지향 설계 5원칙 - SOLID란 무엇일까?

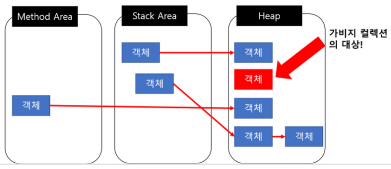
객체 지향 설계의 정수라고 할 수 있는 5원칙이 집대성됐는데, 바로 SOLID 입니다. SRP(Single Responsibility Principle): 단일 책임 원칙 OCP(Open Closed Priciple): 개방 폐쇄 원칙 LSP(Listov Substitution Priciple): 리스코프 치환 원칙 ISP(Interface Segregation

https://devlog-wjdrbs96.tistory.com/380

[Java] 가비지 컬렉션(GC, Garbage Collection) 총정리

가비지 컬렉션은 영어로 Garbeage Collection으로 줄여서 GC라고도 부릅니다. 가비지 컬렉션은 자바의 메모리 관리 방법 중의 하나로 JVM의 Heap 영역에서 동적으로 할당했던 메모리 영역 중 필요 없게 된 메모리 영역을 주기적으로 삭제하는 프로세스 를 말합니다.

https://coding-factory.tistory.com/829



References
1. Java vs Python
   1.1. 인터프리터 언어와 컴파일 언어의 차이
   1.2. 자바는 컴파일 언어가 아니다.
   1.3. 정적타입 언어와 동적타입 언어의 차이
2. Java vs C
   2.1. 컴파일(Compile), 링크(Link), 빌드(Build)
   2.2. 정적라이브러리, 동적라이브러리
3. Java의 장단점
4. OOP(객체 지향 프로그래밍) 특징
   4.1. 캡슐화 방법
5. OOP 5대 원칙: SOLID
   5.1. SRP(단일 책임 원칙, Single Responsibility Principle)
   5.2. OCP(개방 폐쇄 원칙, Open Closed Principle)
   5.3. LSP(리스코프 치환 원칙, Liskov Substitution Principle)
   5.4. ISP(인터페이스 분리 원칙, Liscov Segregation Principle)
   5.5. DIP(의존성 역전 원칙, Dependency Inversion Principle)
6. 객체 지향 프로그래밍 vs 절차 지향 프로그래밍
7. 클래스, 객체, 인스턴스
8. JVM(Java Virtual Machine)
   8.1. Class Loader
   8.2. Execution Engine
   8.3. Garbage Collector
   8.4. Runtime Data Area
9. 가비지 컬렉션(Garbage Collection)
   9.1. 가비지 컬렉션의 대상이 되는 객체들
   9.2. Mark and sweep 알고리즘
   9.3. 가비지 컬렉션 동작 과정
10. 접근제한자
11. String vs Char
   11.1. == vs equals
12. 데이터 타입
   12.1. 기본형 변수와 참조형 변수를 힙과 스택에 나누는 이유
13. Call by value vs Call by reference
14. hashcode
   14.1. hashcode 계산 시 31을 사용하는 이유
15. Wrapper class
   15.1. 박싱, 언박싱
16. non-static vs static
   16.1. 제어자: static
   16.2. main이 static인 이유
   16.3. 제어자: final vs finally vs finalize
   16.4. try with resources
17. 제네릭
   17.1. static 제네릭 메소드
   17.2. 와일드 카드(Wild card)
18. 직렬화, 역직렬화
   18.1. 참조 형식의 데이터를 직렬화해야하는 이유
   18.2. SerialVersionUID 관리
   18.3. 자바 직렬화 사용을 지양해야 한다.
19. 오버로딩, 오버라이딩
20. 추상클래스 vs 인터페이스
21. error, exception
   21.1.Error
   21.2. Exception
   21.3. Checked Exception, unchecked Exception
22. Java collections
   22.1. Map
   22.2. Set
   22.3. List
23. String, StringBuilder, StringBuffer
25. Blocking vs non-blocking
   25.1. Blocking

25.2. Non-Blocking
26. sync vs async
26.1. Sync(동기)
26.2. Async(비동기)
27. 리플렉션
28. stream
29. Fork, Join, Pool
30. lambda
31. optional
32. Java 8 → Java 11
32.1. Java 8
32.2. Java 11
33. 함수형 프로그래밍
34. 멀티스레드 프로그래밍
34.1. 스레드 상태
34.2. 스레드 스케줄링
34.3. 우선순위
34.4. 스레드 그룹
35. 동기화
36. Synchronized, lock&condition 동기화
37. 디자인 패턴
38. 강한 결합 vs 느슨한 결합
39. DDD

1. Java vs Python

Java

- 컴파일 언어로 한 번에 컴파일 된다. → 자바는 컴파일언어이기도 하고 인터프리터 언어이기도 하다.
- 정적 타입 언어(Dynamically typed language)로 데이터 타입 명시가 필요하다.
- 운영체제 독립성: JVM으로 실행되어 OS에 관계없이 동작한다.

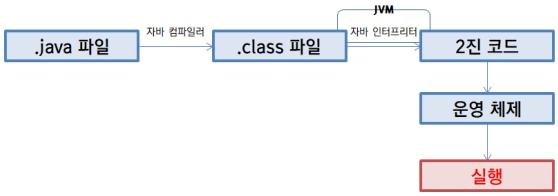
Python

- 인터프리터 언어로 한 줄 씩 번역과 동시에 실행된다.
- 동적 타입 언어(Statically typed language)로 데이터 타입이 동적으로 입력된다. → 타입을 명시하지 않아도 된다.
- 문법이 직관적이고 매우 간단하다. ex. print()

1.1. 인터프리터 언어와 컴파일 언어의 차이

- 인터프리터 언어(Python)는 한 줄 씩 번역(인터프리터)과 동시에 실행이 이루어지지만, 컴파일 언어(C, C++)는 번역(컴파일러) 후 실행 파일이 생성되고 그 실행 파일을 한번에 실행하게 된다.
  - 속도: 큰 규모의 소스일 경우, 컴파일 언어는 소스를 기계어로 컴파일하는 과정에서 시간이 매우 오래걸리고 메모리도 많이 차지할 것이다. 인터프리터 언어는 별도의 컴파일 과정이 없다. 그래서 별도의 컴파일 시간 없이 바로 실행할 수 있다. 하지만 컴파일된 실행프로그램을 실행하는 것은 컴파일 언어가 번역을 같이 해야하는 인터프리터보다 빠르다.
  - OS 이식성: 다른 OS에서 실행하고자 할 때, OS마다 실행할 수 있는 기계어가 다를 수 있다. 그래서 컴파일 언어가 컴파일된 실행파일이 다른 OS에서 실행되지 않고, 해당 OS에 맞는 컴파일러로 다시 컴파일해야하므로 ***이식성이 좋지 않다***. 하지만 인터프리터 언어는 OS에 맞는 인터프리터로 바로 실행할 수 있으므로 ***이식성이 좋다***.

1.2. 자바는 컴파일 언어가 아니다.



- `컴파일 언어`는 소스를 기계어로 번역하여 해당 운영체제에서 실행할 수 있는 실행 파일을 생성하지만 자바는 `자바가상머신(Java Virtual Machine, JVM)`이 실행할 수 있는 `자바 바이트코드(.class)`로 번역(컴파일러)하여 실행 파일을 생성한다. 그리고 `자바가상머신(JVM)`의 `인터프리터`가 `인터프리터 언어`처럼 실행 파일의 `자바 바이트코드`를 한 줄 씩 기계어로 번역하며 실행한다.
- 따라서 자바는 `컴파일 언어`와 `인터프리터 언어`를 혼합한 `하이브리드 언어`이다.

1.3. 정적타입 언어와 동적타입 언어의 차이

- `정적 타입 언어`(C, C++, Java)는 컴파일 시 타입이 결정되므로 변수의 타입을 직접 명시해야하지만, `동적 타입 언어`(Python)는 실행 시 타입이 결정되므로 타입 없이 변수만 선언할 수 있다.
  - `안정성`: 변수의 타입을 명시하면 타입에러의 문제점을 초기에 발견할 수 있으므로 `정적 타입 언어`는 타입의 ***안정성이 좋다***. 반면 `동적 타입 언어`는 실행 중에 타입에러가 발생할 수 있으므로 타입의 ***안정성이 좋지 않다***.

---

2. Java vs C

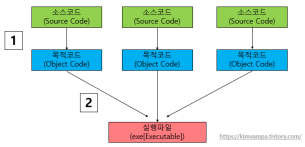
Java

- `운영체제 독립성`: JVM으로 실행되어 OS에 관계없이 동작한다.
- `컴파일 과정`: 별도의 링크 과정 없이 런타임에 필요한 클래스들이 JVM에 동적으로 링크되어 클래스 로더가 동적으로 필요한 클래스 파일을 로딩한다.
- 메모리 관리를 `가비지컬렉터(Garbage Collector)`가 대신한다.
  - OS 레벨의 메모리 영역에 직접 접근하지 않고 JVM을 이용해서 간접적으로 접근한다.

C

- `운영체제 종속성`: OS가 다르면 Object Code와 실행파일이 달라야하므로 다시 컴파일해야한다.
- `컴파일 과정`: `컴파일러`가 Object Code를 생성한 후 `링커`가 필요한 라이브러리를 합쳐 실행 가능한 파일을 생성한다.
  - `정적라이브러리`는 실행 파일에 포함되고 `동적라이브러리`는 런타임에 링크가 일어난다.
- 개발자가 직접 메모리 관리를 해야한다.
  - OS 레벨의 메모리에 직접 접근하기 때문에 `free()`로 할당한 메모리를 명시적으로 해제해야한다. 그렇지 않으면 `memory leak`이 발생할 수 있다.

2.1. 컴파일(Compile), 링크(Link), 빌드(Build)



- `컴파일(Compile)` : Source Code → Object Code
- `링크(Link)` : Object Code → 실행파일
  - `정적 링크(static link)` :
  - `동적 링크(dynamic link)` :
    - `묵시적/암시적 링크(Implicit link)` : `동적라이브러리`의 `.lib` 파일을 이용하여 링킹하는 것으로, 프로그램을 시작할 때 로딩되어 프로그램을 종료할 때 `메모리`에서 해제된다.
    - `명시적 링크(explicit lint)` :
- `빌드(Build)` : `컴파일` + `링크`

2.2. 정적라이브러리, 동적라이브러리

- `라이브러리` : 소프트웨어를 만들 때 쓰이는 클래스나 서브루틴들의 모임. 다른 프로그램들과 링크되기 위하여 존재하는 하나 이상의 서브루틴이나 함수들의 집합 파일 → 링크될 수 있도록 컴파일된 형태인 Object Code로 존재함
  - `정적라이브러리` : 프로그램을 컴파일하는 과정에서 포함시키는 Object 파일들
    - `.lib` : 라이브러리 전체 코드를 포함하는 바이너리 파일
  - `동적라이브러리` : 완성된 프로그램을 실행할 때 포함시키는 동적라이브러리
    - `.lib` : `동적라이브러리`가 제공하고자 하는 함수 정보를 가지는 파일 → 링크 시 필요
    - `.dll` → 실행 시 필요

---

3. Java의 장단점

👩‍💻 **Java의 장점**

- `운영체제 독립성` : JVM으로 실행되어 OS에 관계없이 동작한다.
- `객체지향 언어` : `캡슐화`, `상속`, `추상화`, `다형성` 등을 지원하여 `객체지향 프로그래밍`을 할 수 있다.
- 자동 `메모리` 관리: `가비지컬렉터(Garbage Collector)`가 자동적으로 메모리 관리를 해주기 때문에 따로 메모리 관리를 하지 않아도 된다.
- `동적 로딩`을 지원: 실행 시에 모든 클래스가 로딩되지 않고 필요한 시점에 클래스를 로딩하여 사용할 수 있다. 일부 클래스가 변경되어도 전체 애플리케이션을 다시 컴파일하지 않아도 된다.

👩‍💻 **Java의 단점:** `속도`

- 개선: 바이트코드를 기계어로 바로 변환해주는 JIT 컴파일러와 Hotspot의 도입으로 JVM의 기능이 향상되었다.
- 옵션: Xml, Xms를 통해 힙크기를 조정하여 `가비지컬렉터(Garbage Collector)`의 횟수를 조정할 수 있다.

---

4. OOP(객체 지향 프로그래밍) 특징

- `객체지향 프로그래밍` : 프로그램 구현에 필요한 객체를 파악하고 객체들 간의 상호작용을 통해 프로그램을 만드는 것

👩‍💻 `OOP(객체지향 프로그래밍)` 특징

- `캡슐화(Encapsulation)`
  - `정보 은닉`: 데이터와 코드의 형태를 외부에서 알 수 없게 한다.
  - 높은 응집도, 낮은 결합도 → 유연함, 유지보수성 증가
- `추상화(Abstraction)`
  - 사물들의 공통적인 특징을 파악해서 하나의 개념(집합)으로 다루는 것 → 구현하는 객체들이 가진 공통적인 데이터와 기능을 도출해 낸다.
- `상속(Inheritance)`
  - 상위 개념의 특징을 하위 개념이 물려받음 → 하나의 클래스가 가지고 있는 특징을 그대로 다른 클래스에 물려주고자 할 때 사용한다.
  - 코드의 재사용성을 증대시킬 수 있다.(같은 기능을 다시 구현할 필요 없음)
- `다형성(Polymorphism)`
  - 형태가 같은데 다른 기능을 함
  - `오버라이딩(Overriding)`: 부모 클래스의 메소드를 자식 클래스에 재정의
  - `오버로딩(Overloading)`: 메소드의 이름이 같지만 파라미터가 다름

4.1. 캡슐화 방법

- 멤버 변수 앞에 `접근 제어자` `private`을 붙인다.

접근제어자	설명
`public`	모두 접근 가능
`protected`	상속 혹은 같은 패키지 내 클래스에서 접근 가능
`private`	해당 클래스만 접근 가능

- 멤버 변수에 값을 넣고 꺼내올 수 있는 메서드를 만든다.(`getter` / `setter`)

---

5. OOP 5대 원칙: SOLID

`OOP 5대 원칙: SOLID`

- `SRP: 단일 책임 원칙(Single Reponsible Principle)`
  - 객체는 단 하나의 책임만을 가져야 한다.
  - 어떤 변화에 의해 클래스를 변경해야 하는 이유는 오직 하나뿐이어야 한다.
- `OCP: 개방 폐쇄 원칙(Open Closed Principle)`
  - 확장에는 열려있고 변화에는 닫혀있어야 한다.
- `LSP: 리스코프 치환 원칙(Liskov Substitution Principle)`
  - 자식 클래스는 최소한 자신의 부모 클래스에서 가능한 행위는 수행할 수 있어야 한다.
- `ISP: 인터페이스 분리 원칙(Interface Segregation Principle)`
  - 특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다.
  - 인터페이스가 명확해지고, 대체 가능성이 높아진다.
- `DIP: 의존성 역전 원칙(Dependency Inversion Principle)`
  - 의존 관계를 맺을 때 변화하기 쉬운 것 또는 자주 변화하는 것보다는 어려운 것, 거의 변화가 없는 것에 의존해야 한다.
  - 구현 클래스에 의존하지 않고 인터페이스에 의존해야 한다.

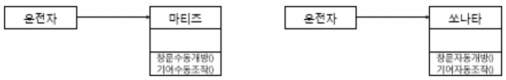
5.1. SRP(단일 책임 원칙, Single Responsibility Principle)

SRP 위배: 책임이 너무 많음

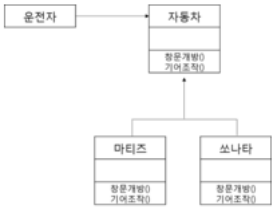
책임 분리

- 작성된 클래스는 하나의 기능만 가지고 클래스가 제공하는 서비스는 그 하나의 책임만 가져야 한다. → 책임 영역이 확실해진다.
- 따라서, 어떤 변화에 의해 클래스를 변경해야 하는 이유는 오직 하나여야한다. → 책임 분배로 인해 코드의 가독성이 향상되고, 유지보수가 용이해진다.

5.2. OCP(개방 폐쇄 원칙, Open Closed Principle)



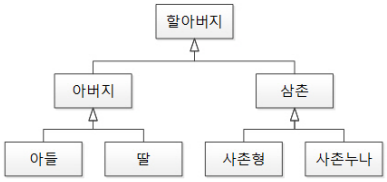
OCP 위반: 기어가 수동/자동이냐에 따라 행동이 달라져야 함 → 변화가 있을 때 운전자에 영향



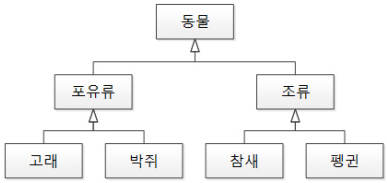
상위클래스로 묶음

- 확장에는 열려 있고, 변화에는 닫혀 있어야 한다. → 기존 코드를 변경하지 않으면서 기능을 추가할 수 있도록 설계되어야 한다.

5.3. LSP(리스코프 치환 원칙, Liskov Substitution Principle)



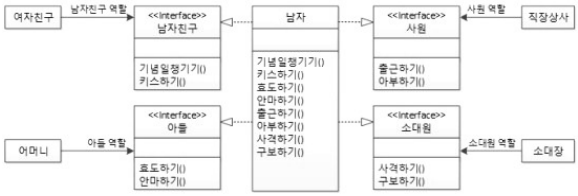
LSP 위배: 하위 타입이 상위 타입으로 치환될 수 없음



LSP 만족

- 서브 타입은 자신의 기반 타입으로 교체할 수 있어야 한다.
- 하위 클래스의 인스턴스는 상위 클래스의 인스턴스 역할을 하는데 문제가 없어야 한다.

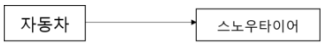
5.4. ISP(인터페이스 분리 원칙, Liscov Segregation Principle)



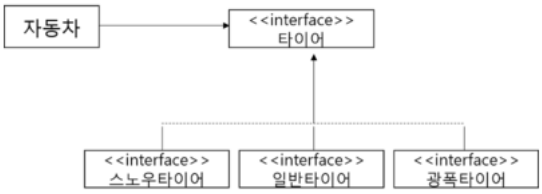
각 역할에 맞는 인터페이스(어머니-아들, 여자친구-남자친구) 분리

- 각 역할에 맞게 인터페이스를 분리해야 한다.
- 클라이언트는 자신이 사용하지 않는 메소드에 의존 관계를 맺으면 안된다.
- SRP와 ISP는 같은 문제에 대한 두 가지 해결책이다.(SRP > ISP)
- 상위 클래스는 풍성할수록 좋다. → 풍성할수록 하위 클래스에 많은 기능을 확장시켜준다.(형변환, 코드 중복 줄여줌)
- 인터페이스 내의 메소드는 최소한일수록 좋다.
  - 인터페이스: 하위 클래스에게 구현을 강제함

5.5. DIP(의존성 역전 원칙, Dependency Inversion Principle)



DIP 위반: 자주 바뀌는 것에 의존하면 안됨



추상화된 인터페이스에 의존 → 타이어가 변경되어도 자동차에 영향이 없음

- 자신보다 변하기 쉬운 것에 의존하던 것을 추상화된 인터페이스나 상위 클래스를 두어 변하기 쉬운 것의 변화에 영향을 받지 않게 한다.
- 상위 클래스, 인터페이스, 추상 클래스일수록 변하지 않을 가능성이 높다.
  - 하위 클래스, 구체 클래스에 의존하면 안된다.

---

6. 객체 지향 프로그래밍 vs 절차 지향 프로그래밍

👩‍💻 객체지향 프로그래밍

- 구현해야 할 객체들 사이의 상호작용을 프로그래밍하는 방식으로 상속, 다형성, 추상화, 캡슐화를 통해 코드의 재사용성을 높인다. 하지만 느리다는 단점이 있다.
- 사람의 사고와 가장 비슷하게 프로그래밍하기 위한 방식이다.

👩‍💻 절차지향 프로그래밍

- 실행하고자 하는 절차를 정하고 순차적으로 프로그래밍하는 방식이다. → 빠르다.
- 순서가 정해져 있어 비효율적이고 유지보수가 어렵다.

---

7. 클래스, 객체, 인스턴스

👩‍💻 클래스

- *객체를 위한 설계도, 틀*
- 객체를 정의해 놓은 것
- 용도: 객체를 생성하는데 상용된다.

👩‍💻 객체

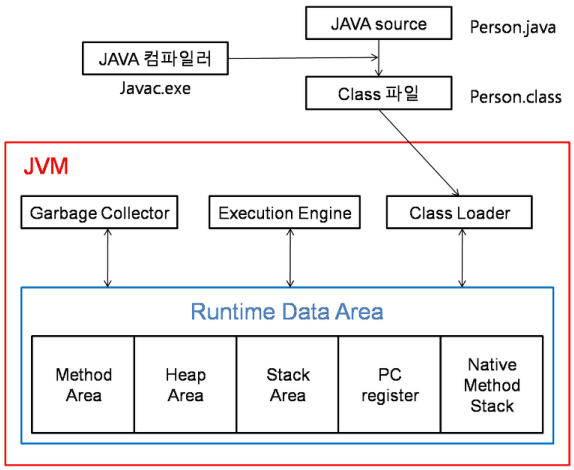
- 실제로 존재하는 것. 사물 또는 개념
- *모든 인스턴스를 대표하는 포괄적 의미*
- 용도: 객체가 가지고 있는 기능과 속성에 따라 다르다.
- 클래스의 인스턴스화 : 클래스로부터 객체를 만드는 과정

👩‍💻 인스턴스

- 클래스로부터 만들어진 객체
- *메모리에 할당되어 실제 사용될 때 인스턴스라고 한다.*

---

8. JVM(Java Virtual Machine)



`JVM(Java Virtual Machine)`

- 자바 프로그램을 실행하는 역할: 컴파일러로 변환된 바이트코드 파일을 JVM에 로딩하여 실행한다.
- 스택 기반으로 동작한다.
- JVM의 구조: `Class Loader`, `Execution Engine`, `Garbage Collector`, `Runtime Data Area`

`JVM(Java Virtual Machine)를 이용해 실행하는 과정`

1. JVM은 OS로부터 메모리(`Runtime Data Area`)를 할당 받는다.
2. `컴파일러(javac)`가 `소스코드(.java)`를 읽어 `바이트 코드(.class)`로 변환한다.
1. `Class Loader`로 class 파일을 `JVM` 내 `Runtime Data Area`로 로딩한다.
1. 로딩된 class 파일을 `Execution Engine` 통해 해석 및 실행

8.1. Class Loader

- JVM 내로 클래스 파일을 로드하고 링크를 통해 배치하는 작업을 수행한다.
- 런타임 시 `동적 로딩`을 한다.

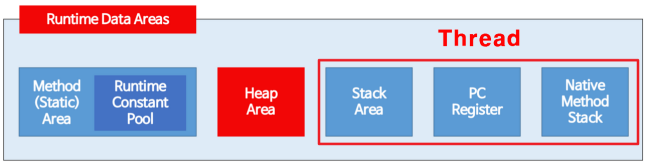
8.2. Execution Engine

- 클래스 로더를 통해 JVM 내의 `Runtime Data Area`에 배치된 바이트 코드들을 명령어 단위로 읽어서 실행한다.
- 최초 `JVM`이 나왔을 당시에는 인터프리터 방식이었기 때문에 속도가 느렸다. → JIT컴파일러 방식으로 보완: 인터프리터 방식을 사용하다가 일정 기준이 넘어가면 JIT컴파일러 방식으로 실행한다.

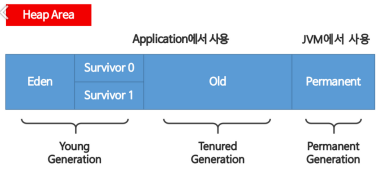
8.3. Garbage Collector

- `힙 메모리 영역`에 생성된 객체들 중 참조되지 않은 객체들을 탐색 후 제거한다.
- 정확히 언제 역할을 하는지는 알 수 없다.

8.4. Runtime Data Area



- `JVM`의 `메모리` 영역으로 자바 애플리케이션을 실행할 때 사용되는 데이터를 적재하는 영역이다.
- 모든 스레드가 공유해서 사용하는 영역 → `Garbage Colection` 대상
  - `Method Area`: 프로그램 실행 중 어떤 클래스가 사용되면, `JVM`은 해당 클래스의 클래스 파일(.class)을 읽어서 클래스에 대한 정보를 이곳에 저장한다.
    - 클래스 멤버 변수의 이름, 데이터 타입, 접근 제어자 정보와 같은 각종 필드 정보들
    - 메서드 정보
    - 데이터 type 정보
    - constant pool
    - static 변수
    - final class 등이 생성되는 영역
  - `Heap Area`: 인스턴스가 생성되는 공간. 프로그램 실행 중 생성되는 인스턴스는 모두 이곳에 생성된다. → 인스턴스변수들이 생성되는 공간
    - new 키워드로 생성된 객체와 배열이 생성되는 영역
    - 주기적으로 `GC`가 제거하는 영역 → 효율적인 제거를 위해 3가지 영역으로 나눈다.



      - `Young Generation (Minor GC)`: 자바 객체가 생성되자마자 저장되고, 생긴지 얼마 안된 객체가 저장되는 공간. `heap 영역`에 객체가 생성되면 최초로 `Eden 영역`에 할당되고 `Eden 영역`에 데이터가 어느정도 쌓이면 참조 정도에 따라 `Survivor`의 빈 공간으로 이동되거나 회수된다.
      - `Old Generation (Major GC/Full GC)`: `Young Generation` 영역이 차게 되면 참조 정도에 따라 `Old 영역`으로 이동되거나 회수된다. `Old Generation` 영역이 차게 되면 `Old 영역`의 모든 객체들을 검사하여 참조되지 않는 객체들을 한꺼번에 삭제하는 `GC`가 실행된다.(`Stop the World`: `GC`를 실행하는 스레드를 제외한 모든 스레드는 작업을 멈추게 한다.)
      - `Permanent Generation`
- 스레드마다 하나씩 생성되는 영역
  - `Stack Area`: 지역변수, 파라미터, 리턴 값, 연산에 사용되는 임시 값 등이 생성되는 영역
    - 메소드 호출 시마다 개별 스택을 생성한다.
  - `PC Register`: 스레드가 생성될 때마다 생성된다. 프로그램 카운터(현재 스레드가 실행되는 부분의 주소와 명령)을 저장한다.
  - `Native Method Stack`
    - 자바 외의 언어(C, C++ 등)으로 작성된 네이티브 코드를 실행할 때 사용되는 메모리 영역 → 일반적인 C 스택을 사용함

---

9. 가비지 컬렉션(Garbage Collection)

`가비지 컬렉션(Garbage Collection)`

- 자바의 메모리 관리 방법
- `JVM`의 `heap 영역`에서 동적으로 할당했던 메모리 영역 중 필요 없게 된 메모리 영역을 주기적으로 삭제하는 프로세스
- `메모리 누수(memory leak)` 문제를 완벽히 관리하지 않아도 된다.

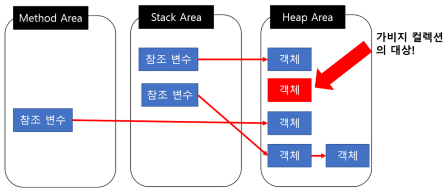
`가비지 컬렉션(Garbage Collection)의 장점`

- `메모리 누수(memory leak)` 문제를 완벽히 관리하지 않아도 된다.

`가비지 컬렉션(Garbage Collection)의 단점`

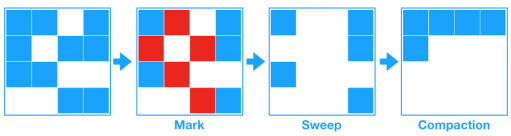
- `메모리`가 언제 해제되는지 정확하게 알 수 없다.
- `GC`가 동작하는 동안 다른 동작을 멈추기 때문에 오버헤드가 발생한다. → `GC`가 너무 자주 실행되면 소프트웨어 성능 하락의 문제가 되기도 한다.(실시간이 중요한 경우 `GC`의 사용이 적합하지 않을 수 있음)

9.1. 가비지 컬렉션의 대상이 되는 객체들



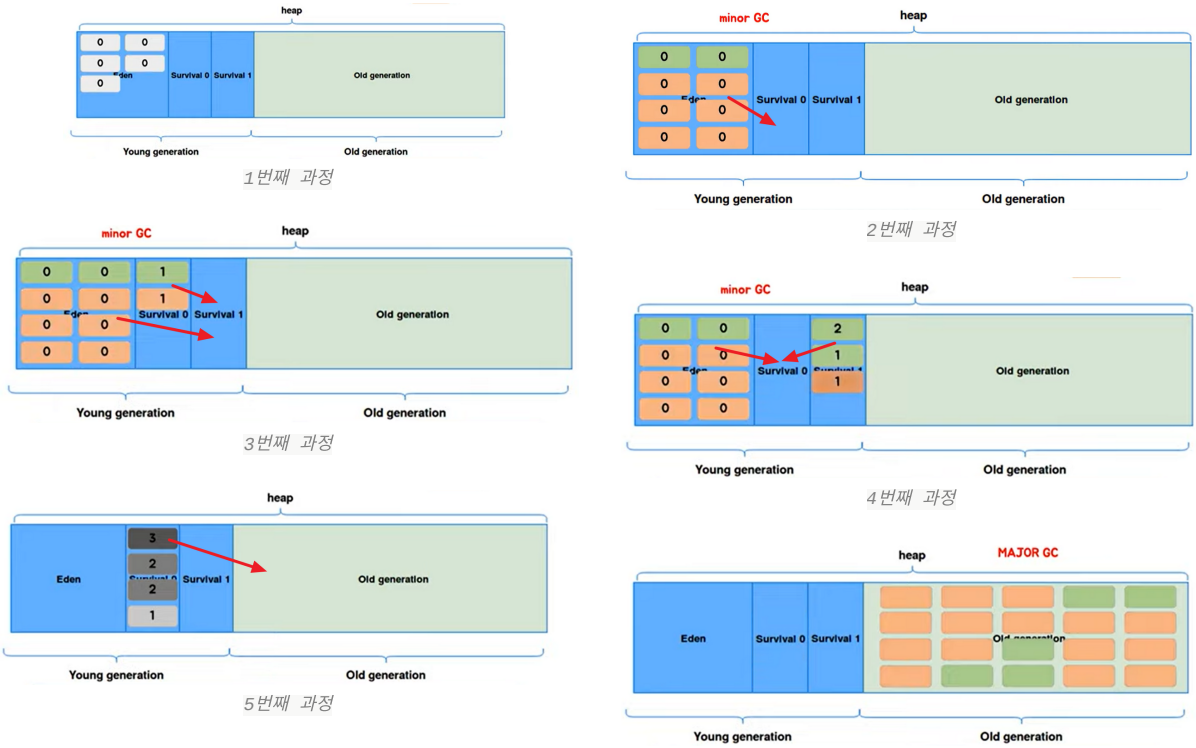
- 객체들은 실질적으로 `heap 영역`에 생성되고 `method 영역`이나 `stack 영역`에서 `heap 영역`에 생성된 객체의 주소를 참조한다.(`reachable`)
- `heap 영역`에 있는 객체의 메모리 주소를 가지고 있는 참조 변수가 삭제되면 그림의 빨간생 객체와 같이 `heap 영역`에서 참조되고 있지 않는 객체들이 발생한다.(`unreachable`) → `GC`의 대상

9.2. Mark and sweep 알고리즘



- `GC`가 동작하는 원리: 루트에서 해당 객체에 접근 가능한지에 대한 여부를 메모리 해제의 기준으로 삼는다.
- 과정
  1. `Mark 과정`: root로부터 그래프 순회를 통해 연결된 객체를 찾아내어 마킹한다.
  2. `Sweep 과정`: `unreachable` 객체들을 `heap 영역`에서 제거한다.
  3. `Compact 과정`: `Sweep` 후 분산된 객체들을 `heap 영역`의 시작 주소로 모아 메모리가 할당된 부분과 그렇지 않은 부분으로 압축한다.(`GC`의 종류에 따라 하지 않는 경우도 있음)

9.3. 가비지 컬렉션 동작 과정



1번째 과정

2번째 과정

3번째 과정

4번째 과정

5번째 과정

6번째 과정

1. 객체가 처음 생성되고 `heap 영역`의 `Eden 영역`에 `age-bit 0`으로 할당된다. → `Minor GC`에서 살아남을 때마다 `age-bit + 1`
2. `heap 영역`의 `Eden 영역`에 객체가 다 쌓이면 `Minor GC`가 한번 일어나고 참조 정도에 따라 `Survivor 0 영역`으로 이동하거나 회수된다.
3. (반복) 계속해서 `Eden 영역`에 신규 객체들이 생성되고 또 다 쌓이면 `Young Generation` 영역에 있는객체들은 `Survivor 1 영역`으로 이동한다. → 살아남은 객체들 `age-bit + 1`
4. (반복) 다시 `Eden 영역`에 신규 객체들이 생성되고 또 다 쌓이면 `Minor GC`가 한번 일어나고 `Young Generation` 영역에 있는 객체들은 `Survivor 0 영역`으로 이동한다. → 살아남은 객체들 `age-bit + 1`
5. `Promotion` : `age-bit`가 특정 숫자 이상이 되는 경우(`JVM`에서 설정해놓은 `age-bit`에 도달할 경우) 오랫동안 쓰일 객체라고 판단하고 `Old Generation` 영역으로 이동한다.
6. `Old Generation` 영역에 할당된 메모리가 허용치를 넘으면 `Old Generation` 영역에 있는 모든 객체를 검사해 참조되지 않는 객체들을 한꺼번에 삭제한다.(`Major GC`, `Stop the World`)

---

10. 접근제한자

접근제어자	설명
`private`	해당 클래스만 접근 가능
`default`	동일한 패키지 내에서 접근 가능
`protected`	상속 혹은 같은 패키지 내 클래스에서 접근 가능
`public`	모두 접근 가능

---

11. String vs Char

`String`

- 제한없이 문자를 담을 수 있다.
- `참조형(reference type)` 변수로 실질적인 문자가 아닌 좌표값을 가지고 있다.
  - new로 생성할 경우: 인스턴스가 `heap`에 생성되고 이를 참조한다.
  - 리터럴로 생성할 경우: 리터럴이 `string constant pool`에 생성되고 이를 참조한다.
- `동등성, 동일성 비교`: 값이 같더라도 생성되는 좌표가 다르므로 `==`로 비교할 수 없다 → `equals()`

`char`

- 1개의 문자를 담을 수 있다.
- `본형(primitive type)` 변수로 직접 문자를 가지고 있다.
- `동등성, 동일성 비교`: 값이 같으면 `==`로 비교할 수 있다.

11.1. == vs equals

- `==` 연산자
  - `기본형(primitive type)` 변수: 값을 비교한다.
    - 엄밀히 말하면, `constant pool`에 있는 특정 상수를 참조하는 주소값을 비교하므로 같은 상수값일 경우 주소값이 같아 동일하다고 판단한다.
  - `참조형(reference type)` 변수: 주소값을 비교한다.
- `Object.equals()`: 주소값을 비교한다.
  - `기본형(primitive type)` 변수에 적용할 수 없다.
  - `overriding`해서 원하는 방식으로 수정할 수 있다.
    - `String.equals()`: 값을 비교한다.(`Object.equals()`를 재정의함)

12. 데이터 타입

`기본형(primitive type)`

- `int`, `char` 등

분류	타입
정수형	`byte`, `short`, `int`, `long`
실수형	`float`, `double`
논리형	`boolean`
문자형	`char`

- 크기가 작고 고정적이므로 `stack 영역`에 저장된다.

`참조형(reference type)`

- `String`, `Integer` 등
- 크기가 가변적이고 동적이므로 `heap 영역`에 저장된다.
  - 실제 데이터는 인스턴스를 생성하여 `heap 영역`에 저장하고 인스턴스의 주소값을 `stack 영역`에 저장한다.
- `stack 영역`에 더이상 참조하는 변수가 없을 때 `GC`에 의해 삭제된다.

12.1. 기본형 변수와 참조형 변수를 힙과 스택에 나누는 이유

- `stack 영역`은 함수의 호출이 완료되면 알아서 소멸되지만, `heap 영역`은 사용자가 직접 관리할 수 있는 메모리 영역이다. (자바는 `GC`가 관리)

13. Call by value vs Call by reference

`Call by Value` → 자바

- `값에 의한 호출`: 함수 호출 시 인자로 전달되는 변수의 값을 복사하여 함수의 인자로 전달한다. → 함수 안에서 인자의 값이 변경되어도 외부 변수의 값은 변경되지 않는다.
- `value`: 값을 담을 수 있는 모든 타입(정수형, 문자열, 주소값 등)
- 자바는 새 지역 변수(다른 주소)를 만들어서 값만 복사하고 할당한다.

`Call by Reference`

- `참조에 의한 호출`: 함수 호출 시 인자로 전달되는 변수의 레퍼런스를 함수의 인자로 전달한다. → 함수 안에서 인자의 값이 변경되면 외부 변수의 값도 변경된다.

14. hashcode

`hashcode`

- 각 객체의 주소값을 변환하여 생성한 객체의 고유한 정수값
- 내용이 같으면 `hashcode`도 같다.

- `Object.hashcode()`: 객체의 `hashcode`를 반환한다.
  - `String.hashcode()`: 문자열의 한 글자씩 가져와서 정수값으로 변경한다.(해당 글자의 ASCII 코드 값)
    - `hashcode` = $\sum_{i=0}^{n-1}(31^{(n-i-1)}$ * s[i])

14.1. hashcode 계산 시 31을 사용하는 이유

```
https://github.com/dolly0920/Effective_Java_Study/issues/25
```

자바 문자열 해시코드 - Java String hash code

String의 hash code에 대해 조사해보았습니다 String.hashCode() 내부 구현 코드 String Class 에서 "String" 값은 `value` 변수에 저장하고 있습니다 해시 코드 값은 `hash`에 저장하고 있습니다 (그래서 hashCode() 함수에서 != 0 이라면 기존에 가지고 hash 멤버 변수 반환이 가능합니다) 우선

https://leedo.me/36

```
al class String
ements java.io.Serializable, Comparable<String>, CharSe
 value is used for character storage.
te final char value[];

he the hash code for the string
te int hash; // Default to 0
```

- 퍼포먼스 이점
- 31이 홀수이면서 소수이기 때문

> *짝수가 안되는 이유?*

- 숫자를 비트로 바꾸면 → ***짝수를 곱하는 것 = Shift 연산***
  - 맨 우측 숫자는 0으로 고정된다. → 가능한 숫자 범위가 많이 좁아진다. → 충돌 가능성 높아짐

> *합성수가 안되는 이유?*

- 상관없긴 하나, 이전부터 소수를 사용하려는 경향이 있다.(실제로 성능 차이는 거의 없지만 홀수에서 최적화를 하려면 소수를 사용해야한다는 신념 같은 것)

> *왜 31인가?*

- 이전 어셈블러에서 최적화가 되기 때문이다.

  `RSB r1, r0, r0, ASL #5 ;` `r1 := - r0 + (r0<<5)` → 일반적인 연산과 비교해서 횟수가 적기 때문에 시간이 적게 소모된다.(지금은 컴파일러 최적화가 잘 되어서 31을 곱하는 것보다 다른 소수를 곱하는게 더 빠른 경우도 있다. ex. Lombok 어노테이션은 다른 소수를 사용함)

---

15. Wrapper class

`래퍼 클래스(Wrapper class)`

- `기본형(primitive type)` 데이터를 객체로 취급해야하는 경우
  - ex. 메서드의 인수로 객체 타입만이 요구되면 데이터를 객체로 변환한 후 수행해야 한다.
- `래퍼 클래스(Wrapper class)` : 각각 타입에 해당하는 데이터를 인수로 전달받아 해당 값을 가지는 객체로 만들어준다.

기본 타입	래퍼 클래스
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Char
boolean	Boolean

- 산술 연산을 위해 정의된 클래스가 아니다. → 인스턴스에 저장된 값을 변경할 수 없다.
  - 값을 참조하는 인스턴스를 생성한다.
  - 값 비교는 `==` 연산자가 아니라 `Integer.equals()`와 같은 메서드를 사용해야 한다.

15.1. 박싱, 언박싱

- `박싱` : `기본형(primitive type)` 데이터 → 인스턴스
- `언박싱` : 인스턴스에 저장된 값 → `기본형(primitive type)` 데이터

```
Integer num = new Integer(17);  // 박싱
int n = num.intValue();         // 언박싱
System.out.println(n);          // 출력: 17
```

- JDK 1.5부터 자바 컴파일러가 알아서 `박싱`/`언박싱` 작업을 해준다.

```
Character ch = "X";     // Character ch = new Character("X"); 오토박싱
char c = ch;            // char c = ch.charValue(); 오토언박싱
System.out.println(c);  // 출력: X
```

16. non-static vs static

`non-static`

- 공간적 특성: 객체마다 별도로 존재함 → `인스턴스 변수`
- 시간적 특성: 객체와 생명주기가 동일함

👩‍💻 `static`

- 공간적 특성: 클래스 당 하나만 생성됨 → `클래스 멤버`
- 시간적 특성: 객체가 생기기 전에 이미 생성되어 객체를 생성하지 않아도 사용 가능
  - 객체가 사라져도 사라지지 않는다.
  - 프로그램 종료 시 사라진다.
- 동일한 클래스의 모든 객체들에 의해 공유된다.

접근 제어자	`public`, `protected`, `default`, `private`
그 외	`static`, `final`, `abstract` 등

16.1. 제어자: static

- `static`은 '클래스의', '공통적인'의 의미를 갖는다. `static` 변수는 하나의 변수를 모든 인스턴스가 공유하기 때문에 클래스변수로 인스턴스에 관계없이 같은 값을 가진다.

대상	의미
멤버변수	모든 인스턴스에 공통적으로 사용되는 클래스변수(인스턴스를 생성하지 않아도 사용 가능하고 클래스가 메모리에 로드될 때 생성)가 된다.
메서드	인스턴스를 생성하지 않고도 호출이 가능한 static 메서드(인스턴스 멤버들을 직접 사용할 수 없음)가 된다.
초기화 블럭	클래스가 메모리에 로드될 때 단 한번만 수행된다.(주로 클래스변수(`static` 변수를 초기화하는데 주로 사용)

16.2. main이 static인 이유

- `static` 멤버는 프로그램 시작 시(클래스 로딩) 메모리에 로드되어 인스턴스를 생성하지 않아도 호출 가능하다.
- class 파일이 로드 될 때 `Runtime Data Area` 중 `method area`에 Class Variable이 저장되는데, `static` 변수도 여기에 포함된다. → `Excution Engine`이 `method area`에 로드된 `main()`을 실행한다.

16.3. 제어자: final vs finally vs finalize

제어자	특징
`final`	변수, 메서드 클래스가 변경 불가능하도록 한다. 1. 기본형 변수: 값 변경 불가능 2. 참조형 변수: 참조 변수가 heap 내 다른 객체를 가리키도록 변경 불가능 3. 메서드: 오버라이드 할 수 없음(오버로딩은 가능) 4. 클래스: 상속 받을 수 없음
`finally`	`try catch` 블록 뒤에서 항상 실행될 코드 블록을 정의
`finalize` 메서드	`GC`가 더 이상의 참조가 존재하지 않는 객체를 메모리에서 삭제하겠다고 결정하는 순간 호출

16.4. try with resources

- Java8 이전 `try-catch-finally`: `try` 구문에서 리소스 생성, `finally` 구문에서 리소스 반납 → 실수 발생의 여지 있음
- Java8 이후 `try with resource`: try 옆 괄호 안에서 리소스를 생성하면 리소스를 따로 반납하지 않아도 리소스를 자동으로 반납함

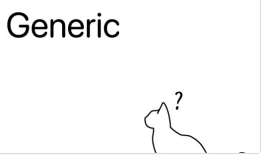
```
public class ResourceClose {
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(new File("input.txt"))) {
      System.out.println(scanner.nextLine());
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } // finally에서 리소스 반납하지 않아도 됨
  }
}
```

17. 제네릭

자바 [JAVA] - 제네릭(Generic)의 이해

정적언어(C, C++, C#, Java)을 다뤄보신 분이라면 제네릭(Generic)에 대해 잘 알지는 못하더라도 한 번쯤은 들어봤을 것이다. 특히 자료구조 같이 구조체를 직접 만들어 사용할 때 많이 쓰이기도 하고 매우 유용하기도 하다. 잠깐 그럼 제네릭(Generic)이란 무엇인지에 대해 알고 가보도록 하자. 제네릭

https://st-lab.tistory.com/153



제네릭(Generic)

- 데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있도록 하는 방법 → 클래스나 메서드에서 사용할 내부 데이터 타입을 외부에서 지정

타입	설명
<T>	type
<E>	element
<K>	key
<V>	value
<N>	number

- type 파라미터에는 참조형(reference type)만 올 수 있다. → 래퍼 클래스(Wrapper class) 사용

제네릭(Generic)의 장점

- 잘못된 타입이 들어올 수 있는 것을 컴파일 단계에서 방지할 수 있다.
- 코드 재사용성이 증가한다.

- 제네릭(Generic) 사용 예

```
public class ClassName <T, K> {
  private T type;
  private K key;

  // getter, setter
}

public class Main {
  public static void main(String[] args) {
    ClassName<String, Integer> a = new ClassName<String, Integer>();

    a.method(3);
    a.method("a");
    a.method(new ClassName<String, Integer>());
  }
}
```

17.1. static 제네릭 메소드

- static 메소드를 제네릭(Generic)으로 선언하면, 포함되어 있는 클래스의 인스턴스가 생성되어 타입이 지정되기 전인 프로그램 시작 시에 메모리에 로드되어야 한다. → 메소드에 한정한 제네릭(Generic)이 필요함

```
class ClassName<E> {
  private E element;

  // getter, setter
```

```
public <T> T genericMethod(T type) {  // T: class의 E와 다른 타입
  return type;
}
```

17.2. 와일드 카드(Wild card)

[Java] 제네릭과 와일드카드 타입에 대해 쉽고 완벽하게 이해하기(공변과 불공변, 상한 타입과 하한 타입)

이번에는 공변과 불공변에 대해서 먼저 알아보고, 이후에 제네릭과 와일드카드에 대해 알아보도록 하겠습니다. 많은 분들이 어려워하는 제네릭인 만큼 쉽게 풀어서 설명하려고 노력했는데, 많은 도움이 되기를 바라겠습니다. 제네릭과 와일드카드에 대해 이해하기 위해서 우리는 먼저 공변과 불공변에 대해 알아야 한다. 공변과 불공변은 각각 다음과 같다.

https://mangkyu.tistory.com/241

- 활용 예: `E extends Comparable<? super E>`

```
public class GenericClass <E extends Comparable<E>> { ... }  // Error 가능성
public class GenericClass <E extends Comparable<? super E>> { ... }

public Student extends Person implements Comparable<Person> {
  @Override
  public int compareTo(Person object) { ... };
}

public class Main {
  public static void main(String[] args) {
    GenericClass<Student> a = new GenericClass<Student>();
}
```

18. 직렬화, 역직렬화

자바 직렬화, 그것이 알고싶다. 실무편 | 우아한형제들 기술블로그

자바의 직렬화 기술에 대한 대한 두 번째 이야기입니다. 실제 자바 직렬화를 실무에 적용해보면서 주의해야 할 부분에 대해 이야기해보려고합니다. 자바 직렬화는 자바 개발자 입장에서는 상당히 쉽고 빠르게 사용할 수 있도록 만든 기술입니다. JSON 또는 CSV 등 형태의 포맷을 이용하면 직렬화 또는 역직렬화시에

https://techblog.woowahan.com/2551/

WOOWATECH 우아한Tech 우아한형제들의 기술조직 이야기를 전합니다.

직렬화(Serialization)

- 메모리를 디스크에 저장하거나, 네트워크 통신에 사용하기 위한 형식(텍스트 혹은 바이너리)으로 변환한다.
  - `CSV`, `XML`, `JSON` **직렬화**
  - Binary **직렬화** → **Java 직렬화(Serialization)**
    - 용량이 크다.(`JSON`에 최소 2배 최대 10배 이상의 크기)
- 디스크에 저장하거나 통신할 때는 값 형식의 데이터만 사용할 수 있어 **직렬화**가 필요하다.

Java 직렬화(Serialization)

- 자바 시스템 내부에서 사용되는 객체 또는 데이터를 외부의 자바 시스템에서도 사용할 수 있도록 바이트 형태로 데이터를 변환한다. (JVM의 메모리에 있는 객체 데이터를 바이트 형태로 변환)
- `java.io.Serializable` 인터페이스 구현
- `java.io.ObjectOutputStream` 사용

👩‍💻 `Java 역직렬화(Deserialization)`

- 바이트로 변환된 데이터를 다시 객체로 변환한다. (`직렬화`된 바이트 형태의 데이터를 객체로 변환해서 JVM의 메모리에 할당)
- `java.io.ObjectInputStream` 사용

👩‍💻 `Java 직렬화(Serialization)의 장점`

- `JSON`, `CSV` 등의 포맷은 직렬화, 역직렬화 시 특정 라이브러리를 도입해야 쉽게 개발이 가능하다.
- `Java 직렬화(Serialization)`는 복잡한 객체도 큰 작업 없이 기본 자바 라이브러리로 인터페이스 구현만으로 `직렬화`와 `역직렬화`를 할 수 있다.

18.1. 참조 형식의 데이터를 직렬화해야하는 이유

- 참조한 주소값을 저장하더라도 프로그램을 종료하고 다시 실행하면 기존에 할당되었던 메모리가 해제되고 없어지므로 주소값이 다를 수 있다.
- 각 PC마다 사용하고 있는 메모리 공간의 주소가 달라 네트워크 통신을 했을 때 해당 주소값에 다른 데이터가 있을 수 있다.

18.2. SerialVersionUID 관리

- 기존에 직렬화한 클래스의 구조를 변경하고 변경된 클래스로 직렬화한 데이터를 역직렬화하면 `java.io.InvalidClassException`이 발생한다. (`serialVersionUID`의 정보가 일치하지 않음)
- `Java 직렬화(Serialization)`에서 `serialVersionUID`를 설정하는 원칙: `serialVersionUID`가 선언되지 않으면 클래스의 기본 해쉬값을 사용한다. → 특별한 문제가 없다면 개발 시 `serialVersionUID` 값을 직접 관리해야 한다.
- `serialVersionUID` 값이 같을 경우
  - 멤버 변수 및 메서드 추가는 문제 없음
  - 멤버 변수 제거 및 이름 변경은 오류 발생은 없으나 데이터 누락
  - 타입에 엄격하다. → 멤버 변수명은 같은데 멤버 변수 타입이 바뀌면 `java.io.ClassCastException`이 발생한다.

18.3. 자바 직렬화 사용을 지양해야 한다.

- 외부(DB, 캐시 서버, NoSQL 서버 등)에 장기간 저장될 정보는 자바 직렬화 사용을 지양해야 한다.
  - 역직렬화 대상의 클래스가 언제 변경이 일어날지 모른다. 긴 시간 외부에 존재한 직렬화된 데이터는 쓰레기가 될 가능성이 높다. → 언제 예외가 발생할지 모르는 지뢰 시스템
- 프레임워크 혹은 라이브러리에서 제공하는 클래스처럼 개발자가 직접 컨트롤이 힘든 객체는 직렬화 사용을 지양해야 한다.
  - 객체가 직접 `serialVersionUID` 값을 가지고 있음
  - 버전업 하면서 변경될 수 있음 → 운영에서 문제 발생
  - ex. Spring Security `SecurityContextImpl` 클래스
- `용량` 문제: `Java 직렬화(Serialization)`는 타입에 대한 정보 등 클래스의 메타 정보도 가지고 있어서 다른 포맷에 비해 용량이 크다.
  - 직렬화된 데이터를 메모리 서버(Redis, Memcached)에 저장하는 경우 메모리 용량이 크지 않기 때문에 `Java 직렬화(Serialization)` 방법이 좋은 선택이 아닐 수 있다. → 용량 크기에 따른 네트워크 비용과 캐시 서버 비용

---

19. 오버로딩, 오버라이딩

`오버로딩(Overloading)`

- 한 클래스 내에서 같은 이름을 가진 메소드를 정의할 수 있다.
- 조건
  - 매개변수의 개수가 다르다.
  - 매개변수의 개수는 같으나 타입이 다르다.

`오버라이딩(Overriding)`

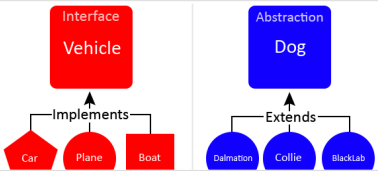
- 부모 클래스에서 정의한 메소드를 자식 클래스에서 재정의 할 수 있다.

20. 추상클래스 vs 인터페이스

자바의 추상클래스(abstract class)와 인터페이스(interface)

자바의 상속 개념을 공부할때 빠지지 않고 등장하는 것이 바로 오늘 배울 추상클래스(abstract class)와 인터페이스(interface)다. 무슨 공통점이 있는지 무슨 차이점이 있는지 말도 애매모호하고 서로 비슷비슷하므로 이 부분만 나오면 한숨이 나오는 이들이 많을 것이다. 오늘 강좌를 통해서 언제나 그랬듯이 여

http://alecture.blogspot.com/2011/05/abstract-class-interface.html



`추상클래스(Abstract Class)`

- 클래스 내 추상 메소드가 하나 이상 포함되거나 abstract으로 정의된 경우
- 클래스 안의 메소드가 단 한개라도 추상 메소드가 있다면 클래스 앞에 `abstract`가 표기되어야 한다.(`abstract`와 `final`을 동시에 표기할 수 없음)
- `추상클래스`로 상속받으면 추상 메소드를 구현하지 않아도 되지만, 일반적인 클래스로 상속받으면 추상 메소드를 구현해야 한다.
- `추상클래스`에서 인터페이스를 구현할 수 있다.(구현을 하지 않아도 됨)
- 생성자를 가질 수 있다.
- `추상클래스`로는 인스턴스를 만들 수 없지만 `추상클래스`를 상속받은 인스턴스를 통해 인스턴스화 할 수 있다. → `AbstractClass ab = new ChildClass()`
- `추상클래스`의 접근 지정자는 어떤 것이든 가능하다.
- `목적` : `상속`(`추상클래스`를 상속해서 기능을 구현하고 확장한다. 이때 `추상클래스`의 구현은 강제한다.)

👩‍💻 `인터페이스(Interface)`

- 모든 메소드가 추상 메소드인 경우(`final`을 붙일 수 없고, 변수들을 `static`이어야 한다. → 일반 변수를 가질 수 없음)
- 내용이 비어있는 메소드들의 형태만 써놓은 것 → 구현하는 클래스에서 해당 메소드의 내용을 구현해야 함
- `인터페이스` 간의 상속이 가능하다.
- `인터페이스`는 생성자를 가질 수 없다.
- `인터페이스`로는 인스턴스를 만들 수 없지만 `인터페이스`를 구현한 인스턴스를 통해 인스턴스화 할 수 있다. → `Interface ab = new ImplementClass()`
- `인터페이스`의 접근 지정자는 `public`, `default`, `abstract`만 가능하다.
- Java8에서는 `default` 키워드를 이용해서 일반 메소드도 구현할 수 있다.
- 구현하는 클래스에서 추상 메소드를 구현하도록 강제한다.
- `목적`: 함수의 구현을 강제하여 구현 객체의 같은 동작을 보장하기 위해 사용한다.

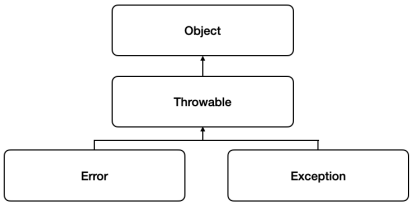
---

21. error, exception

Java의 Error와 Exception 그리고 예외처리 전략

우아한테크코스(이하 우테코) 레벨 1 복습을 위한 글입니다. 자바는 오류가 발생하거나 발생할 여지가 있는 여러 상황에 대한 예외를 만들어두었습니다. 우리에게 익숙한 IllegalArgumentException을 비롯해 원인을 찾기 힘든 NullPointerException도 있습니다. 또한 파일 처리를 하다 보면 발생하기도 하는

https://toneyparky.tistory.com/40



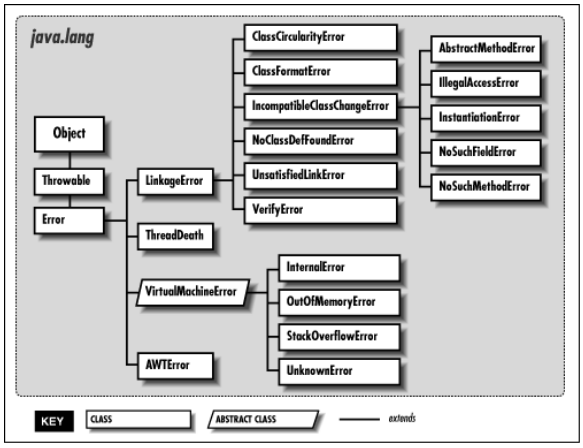
👩‍💻 `오류(Error)`

- 시스템이 종료되어야 할 수준의 상황과 같이 수습할 수 없는 심각한 문제 → 미리 예측하여 방지할 수 없다.

👩‍💻 `예외(Exception)`

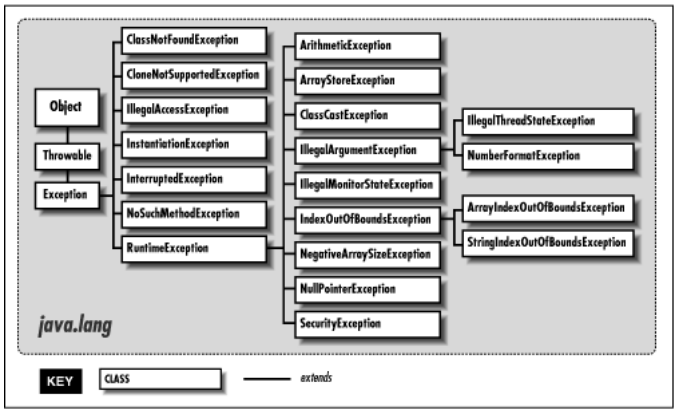
- 개발자가 구현한 로직에서 발생한 실수나 사용자의 영향에 의해 발생 → 미리 예측하여 방지할 수 있다.(`예외처리`, `Exception Handler`)

21.1.Error



- `StackOverflowError`: 호출의 깊이가 깊어지거나 재귀가 지속되어 stack overflow 발생 시 던져지는 오류
  - 간접적인 예방: 재귀 사용 유의, 가시적인 `loop` 사용
- `OutOfMemoryError`: `JVM`이 할당된 메모리의 부족으로 더이상 객체를 할당할 수 없을 때 던져지는 오류 혹은 `Garbage Collector`에 의해 추가적인 메모리가 확보되지 못하는 상황
  - 간접적인 예방: memory leak 차단, heap 크기 늘리기

21.2. Exception



- `NullPointerException`: 객체가 필요한데 응용프로그램이 `null`을 사용하려고 시도할 경우
- `IllegalArgumentException`: 메서드가 허가하지 않거나 부적절한 argument를 받았을 경우

21.3. Checked Exception, unchecked Exception

- `Checked Exception (Compile Exception)`: `Exception`을 바로 상속받고 `컴파일 시점`에 예외를 catch하는지 정적으로 확인한다.
  → `컴파일 에러`
  - `트랜잭션 Rollback`이 안된다.

```
public class InvalidPositionException extends Exception {
  public InvalidPositionException(String message) {
    super(message);
  }
}
```

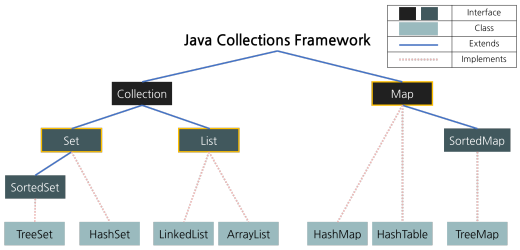
```
public class Position {
  public Position(String position) throws InvalidPositionException {
    if (position.length() != 2) {
      throw new InvalidPositionException(position + "은 위치 값 형식에 맞지 않습니다.");
    }
    ...
```

```
  }
}
```

- `Unchecked Exception` : `RuntimeException`을 상속받고 `컴파일 시점`에 예외를 catch하는지 확인하지 않는다. → `컴파일 시점`에 예외가 발생하는지 확인할 수 없음
  - `트랜잭션 Rollback`이 된다.

---

22. Java collections



- Collections
  - Set
    - 데이터의 중복을 허용하지 않는 데이터 집합
    - 순서를 보장하지 않음
    - value의 중복을 허용하지 않음
  - List
    - 순서가 있는 데이터 집합
    - 데이터를 중복해서 포함시킬 수 있음
- Map
  - Key와 Value의 형태로 이루어진 데이터 집합
  - 순서를 보장하지 않음
  - Key는 중복이 허용되지 않고, Value는 중복을 허용함

22.1. Map

- HashMap
  - 내부 hash값에 따라 키순서가 정해지므로 특정 규칙없이 출력된다.
  - key와 value에 null값을 허용 한다.
  - 비동기 처리
  - 동작 방식
    - key-value 쌍의 데이터를 저장하고 key값은 중복을 허용하지 않는다.
    - 동일하지 않은 두 객체가 같은 bucket에 들어가려고 하는 경우를 Collision이라고 하는데 성능에 영향을 미치므로 어떤 Hash 함수를 사용하는 가에 성능이 결정된다.
    - 공간이 없을 때 수행 방식
      - 리스트로 넣는 방식
      - 사이즈를 늘리는 방식
        - new HashMap<>(4, 0.75f) -> 사이즈가 4이고 load factor가 0.75세팅
        - 4 * 0.75 = 3 이므로 3개의 요소가 채워지는 순간 사이즈를 2배 늘려서 8개로 만든다.
- LinkedHashMap

    - 입력 순서대로 출력된다.
    - 비동기 처리
- TreeMap
    - 내부적으로 레드-블랙 트리(이진 탐색 트리) 로 저장된다.
    - Null값 비허용
    - 키값이 기본적으로 오름차순 정렬 되어 출력된다.
    - Compartor 구현으로 정렬 방법을 지정할 수 있다.
- ConCurrentHashMap
    - key,value에 null값 비허용
    - 쓰기작업에서만 동기 처리
- HashTable
    - key,value에 null값 비허용
    - 모든 작업에 동기 처리

22.2. Set

- HashSet
    - 저장 순서를 유지하지 않는 데이터의 집합
    - Null 저장 가능
    - 해시 알고리즘을 사용하여 검색속도가 매우 빠르다.
    - 내부적으로 HashMap 인스턴스를 이용하여 요소를 저장한다.
- LinkedHashSet
    - 저장 순서를 유지하는 HashSet
- TreeSet
    - 데이터가 정렬된 상태로 저장되는 이진 탐색 트리의 형태로 요소를 저장한다.
    - Null 저장 불가능
    - 레드 블랙 트리(이진탐색트리) 로 구현되어 있다.
    - Compartor 구현으로 정렬 방법을 지정할 수 있다.

22.3. List

- ArrayList
    - 내부적으로 배열을 사용하는 자료구조로 메모리가 연속적으로 배치된다.
    - 배열과 달리 메모리 할당이 동적이다.
    - 데이터 삽입, 삭제 시 해당 데이터 이후 모든 데이터가 복사되므로 빈번한 삭제, 삽입이 일어나는 경우에는 부적합하다.
    - 검색의 경우는 인덱스의 데이터를 가져오면 되므로 빠르다.
    - **재할당 시 크기의 절반씩 증가한다.**
- LinkedList
    - 양방향 포인터 구조로 각 노드가 데이터와 포인터를 가지고 한 줄로 연결되어 있는 방식으로 데이터를 저장하는 자료구조
    - 데이터의 삽입, 삭제 시 해당 노드의 주소지만 바꾸면 되므로 삽입, 삭제가 빈번한 데이터에 적합하다.
    - 메모리가 불연속적이다.
    - 데이터 검색 시 처음부터 순회하므로 검색에는 부적합하다.
    - 스택, 큐, 양방향 큐를 만들기 위한 용도로 사용한다.

    - 양옆의 정보만을 갖고 있기 때문에 순차적으로 검색을 진행하여 검색 속도가 느리다.
- Vector
    - 내부에서 자동으로 동기처리가 일어난다.
    - 성능이 좋지 않아 잘 사용하지 않는다.
    - 재할당 시 크기의 두 배로 증가한다.
- Stack
    - new 키워드로 직접 사용 가능
    - Vector를 상속받아 동기 처리
- 배열 vs List
    - 배열(Array)
        - 메모리 상에 연속적으로 데이터를 저장하는 자료구조
        - 크기가 고정적이라 선언 시 지정한 크기를 변경할 수 없다.(Immutable)

---

23. String, StringBuilder, StringBuffer

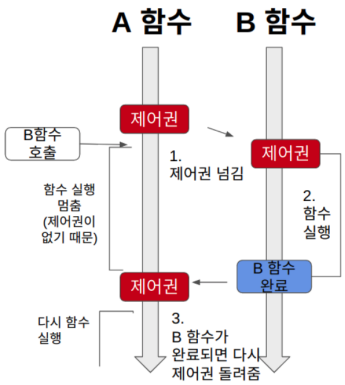
- String
    - 새로운 값을 할당할 때마다 새로운 클래스에 대한 객체가 생성
    - String에 저장되는 문자열은 private final char[] 형태이므로 변경할 수 없다.
    - String + String + String..
        - 가비지 컬렉터가 호출되기 전까지 생성된 String 객체들은 Heap에 머물기 때문에 메모리 관리에서 치명적이다.
- StringBuilder
    - 메모리에 append하는 방식으로 클래스에 대한 객체를 생성하지 않는다.
    - 비동기 처리
- StringBuffer
    - 메모리에 append하는 방식으로 클래스에 대한 객체를 생성하지 않는다.
    - 동기 처리
- String new vs “”
    - new 는 계속 새로운 객체를 생성해내는 반면에 ""의 경우는 이미 존재하는 String 값이라면 같은 래퍼런스를 참조한다.

---

25. Blocking vs non-blocking

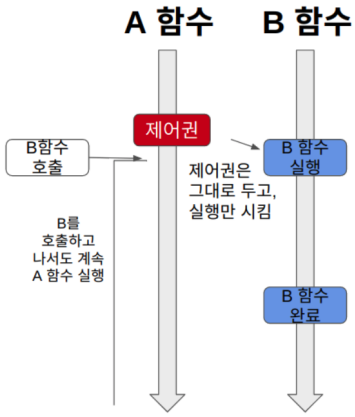
- 두 가지의 차이점은 다른 주체가 작업할 때 자신이 코드를 실행할 제어권이 있는지 없는지로 판단할 수 있다.

25.1. Blocking



- Blocking 자신의 작업을 진행하다가 다른 주체의 작업이 시작되면 제어권을 다른 주체로 넘긴다. 따라서 자신은 제어권이 없기 때문에 실행할 수 없고, 다른 주체가 실행을 완료하고 제어권을 돌려줄 때까지 아무 작업도 할 수 없다.

25.2. Non-Blocking



- Non-Blocking은 다른 주체의 작업에 관련없이 자신이 제어권을 갖고 있다. 따라서, 자신은 계속 작업을 수행할 수 있다.

---

26. sync vs async

- 두 가지의 차이점은 호출하는 함수가 호출되는 함수의 작업 완료 여부를 신경쓰는지 여부에 차이가 있다.

26.1. Sync(동기)

- 동기는 함수 A가 B를 호출한 뒤, B의 결과값이 나오면 해당 결과값을 가지고 바로 처리하는 방식이다.

26.2. Async(비동기)

- 비동기는 함수 A가 B를 호출한 뒤, B의 결과값에 큰 비중을 두지 않고 결과가 나오면 처리를 할 수도 있고 안 할 수도 있다.

---

27. 리플렉션

- 런타임 상황에서 메모리에 올라간 클래스나 메서드등의 정의를 동적으로 찾아서 조작할 수 있는 기술
- 컴파일 시간이 아닐 실행 시간에 동적으로 특정 클래스의 정보를 객체화를 통해 분석 및 추출해낼 수 있는 프로그래밍 기법이다.
- 자바에서 이미 로딩이 완료된 클래스에서 또 다른 클래스를 동적으로 로딩하여 생성자, 필드, 메서드 등을 사용할 수 있는 방식이다.

- 장점
  - 런타임 시점에 사용할 instance를 선택하고 동작시킬 수 있는 유연한 기능을 제공한다.
- 단점
  - **컴파일 시점이 아니라 런타임 시점에서 오류를 잡기 때문에 컴파일 시점에 확인할 수 없다.**
  - 접근 제어자로 캡슐화된 필드, 메서드에 접근 가능해지므로 기존 동작을 무시하고 깨뜨리는 행위가 가능해진다.
  - 위와 같은 단점 때문에 피할 수 있다면 사용을 자제하는 것이 좋다.
- 사용처
  - 런타임 시점에 다른 클래스를 동적으로 로딩하여 접근할 때 사용
  - 클래스와 멤버 필드, 메서드 등에 관한 정보를 얻어야 할 때 사용

---

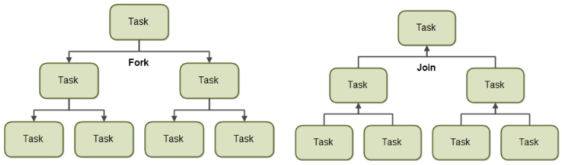
28. stream

- java 8 에서 추가된 API
- 컬렉션 타입의 데이터를 Stream 메서드로 내부 반복을 통해 정렬, 필터링 등이 가능
- 특징
  - parallel 메서드 제공을 통해 병렬처리 가 가능
    - 각 스레드가 개별 큐를 가지고 있으며, 놀고 있는 스레드가 있으면 일하는 스레드의 작업을 가져와 수행
  - 데이터를 변경하지 않는다(Immutable)
    - 원본데이터로부터 데이터를 읽기만 할 뿐, 원본데이터 자체를 변경하지 않는다.
  - 작업을 내부 반복으로 처리하므로 불필요한 코딩을 줄일 수 있다.
  - 최종 연산 후 stream이 닫히므로 일회용 이다.
- 구조
  - Stream 생성
  - 중간 연산
    - 데이터를 가공하는 과정에 해당한다.
    - 필터링: filter, distinct
    - 변환: map, flatMap
    - 제한: limit, skip
    - 정렬: sorted
    - 연산결과확인: peek
  - 최종 연산
    - Stream 안의 데이터를 모아 반환하는 역할을 한다.
    - 출력: foreach
    - 소모: reduce
    - 검색: findFirst, findAny
    - 검사: anyMatch, allMatch, noneMatch
    - 통계: count, min, max
    - 연산: sum, savage
    - 수집: collect
- 중간 연산 작업은 바로 실행되는 것이 아니라 종결 처리의 실행이 필요할 때서야 비로소 중간 처리가 실행된다.(Lazy Evalutation)
- ParallelStream

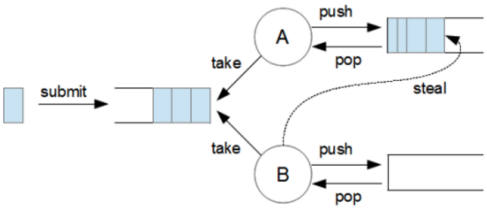
- 개발자가 직접 스레드 혹은 스레드 풀을 생성하거나 관리할 필요 없이 parallelStream, parallel()만 사용하면 알아서 내부적으로 common ForkJoinPool 을 사용하여 작접들을 분할하고 병렬적으로 처리한다.
  - forkJoinPool은 ExecutorService의 구현체로 각 스레드별 개별 큐를 가지고 스레드에 아무런 task가 없으면 다른 스레드의 task를 가져와 처리하여 최적화 성능을 낼 수 있다는 특징이 있다.
- 내부적으로 스레드 풀을 만들어서 작업을 병렬화시킨다.
- 중요한 점은 parallelStream 각각 스레드 풀을 만드는게 아니라 별도의 설정이 없다면 하나의 스레드 풀을 모든 parallelStream이 공유한다.
- ParallelStream은 중간 연산에서 순서가 보장되지 않기 때문에 중간 연산에서 순서에 관계없는 연산의 경우에만 사용한다.
- Parallel Stream은 작업을 분할하기 위해 Spliterator의 trySplit()을 사용하게 되는데, 이 분할되는 작업의 단위가 균등하게 나누어져야 하며, 나누어지는 작업에 대한 비용이 높지 않아야 순차적 방식보다 효율적으로 이뤄질 수 있다. array, arrayList와 같이 정확한 전체 사이즈를 알 수 있는 경우에는 분할 처리가 빠르고 비용이 적게 들지만, LinkedList의 경우라면 별다른 효과를 찾기 어렵다.

---

29. Fork, Join, Pool



- Java7에서 새로 지원하는 fork-join 풀을 기본적으로 큰 업무를 작은 업무로 나눠 배분하여, 일을 한 후에 일을 취합하는 형태이다. 분할 정복 알고리즘과 비슷하다고 보면 된다. 자바에서 풀을 관리하는 ThreadPoolExecutor와 마찬가지로 ForJoinPool도 내부에 inbound queue 라는 편지함이 하나 있다. 그걸 두고 싸우느라 시간을 낭비하는 것을 방지하기 위해 ForkJoinPool은 스레드 개별 큐를 만들었다.



- 왼쪽에서 업무를 submit하면 하나의 inbound queue에 누적되고 그걸 A와 B 스레드가 가져다가 일을 처리한다. A와 B는 각자 큐가 있으며, 자신의 큐에 업무가 없으면 상대방의 큐에서 업무를 가져와서 처리한다. 최대한 노는 스레드가 없게 하기 위한 알고리즘이다.

---

30. lambda

- 자바 8에서 등장
- 메서드를 하나의 식으로 표현하는 익명 함수
- 인터페이스 내에 한 개의 추상 메서드만 정의되어있는 함수형(Function) 인터페이스에 사용 가능
- 장점
  - 기존에 익명함수로 작성하던 코드를 줄일 수 있음
  - 가독성 증가
  - 병렬 프로그래밍이 용이하다.

---

31. optional

- T orElse(T other)
  - 반환할 값을 그대로 받는다.
  - 무조건 인스턴스화 된다.
- T orElseGet(Supplier<? extends T> other)
  - Supplier로 랩핑된 값을 인자로 받는다.
  - 함수를 전달받아 바로 값을 가져오지 않고 필요할 때(Lazy) 값을 가져온다.
  - Optional 안의 값이 null일 경우에만 함수가 실행되면서 인스턴스화 된다.

---

32. Java 8 → Java 11

32.1. Java 8

- optional
- stream
- lambda
- localDateTime
  - 기존 Date의 문제점
    - 불변 객체가 아님
    - 헷갈리는 월 지정(1월을 0으로 표현)
    - 일관성 없는 요일 상수 (어디서는 일요일이 0 어디서는 1)
    - Date와 Calendar 객체의 역할 분담(Date만으로 부족해서 왔다갔다 해야함)
    - 상수 필드 남용
- default 메서드
  - 인터페이스는 메서드 정의만 가능하고 구현은 불가능했는데 default 메서드 개념이 생기면서 인터페이스에 구현된 메서드도 추가가 가능해졌다.

32.2. Java 11

- default GC가 Paralle GC에서 G1GC로 변경
- strip(), stripLeading(), stripTrailing(), isBlank(), repeat(n) 과 같은 String 클래스에 새로운 메서드 추가
- writeString, readString, isSameFile 과 같은 File관련 새로운 메서드 추가
- 컬렉션의 toArray() 메서드를 오버로딩하는 메서드가 추가되어 원하는 타입의 배열을 선택하여 반환할 수 있게 되었다.
  - sampleList.toArray(String[]::new)
- Predicate 인터페이스에 부정을 나타내는 not() 메서드 추가
- 람다에서 로컬 변수 var이 사용 가능
- Javac를 통해 컴파일하지 않고도 바로 java 파일을 실행할 수 있게 되었다.

---

33. 함수형 프로그래밍

- 함수형 프로그래밍은 거의 모든 것을 순수 함수로 나누어 문제를 해결하는 기법으로, 작은 문제를 해결하기 위한 함수를 작성하여 가독성을 높이고 유지보수를 용이하게 해준다.Robert C.Martin은 함수형 프로그래밍을 대입문이 없는 프로그래밍 이라고 정의하였다.명령형 프로그래밍에서는 메소드를 호출하면 상황에 따라 내부의 값이 바뀔 수 있다. 즉, 우리가 개발한 함수 내에서 선언된 변수의 메모리에 할당된 값이 바뀌는 등의 변화가 발생할 수 있다.하지만 함수형 프로그래밍에서는 대입문이 없기 때문에 메모리에 한 번 할당된 값은 새로운 값으로 변할 수 없다.

- 순수함수 : 같은 인자에 대해서 항상 같은 값을 반환하고 외부의 어떤 상태도 바꾸지 않는 함수
- 일급객체 : 함수의 인자로 전달할 수 있고, 함수의 반환값으로 사용할 수 있고, 변수에 담을 수 있는 객체
- 일급함수 : 함수가 일급객체면 일급함수라고 하고 일급함수에 이름이 없다면 람다식이라고 한다.

---

34. 멀티스레드 프로그래밍

- 스레드 생성 방법
  - 방법 1: Thread 클래스를 상속받아서 run을 오버라이드해서 정의한다.
  - 방법 2: Runnable 인터페이스를 구현하여 Thread 생성자에 인자로 넘긴다.
  - 방법 3: Callable 인터페이스를 구현하여 FutureTask에 한번 감싸서 Thread의 인자로 넘긴다.
  - Runnable은 Exception이 없고 리턴값도 없으나 Callable은 리턴값이 있고 Exception을 발생시킬 수 있다.
- 스레드 실행 방법
  - 보통 start() 메서드를 사용해서 호출하는데 start 한다고 해서 바로 실행되는 것은 아니고 실행 대기열에 저장된 후 차례가 오면 실행된다.
  - 정확하게 말하자면, start는 새로운 스레드가 작업을 실행하는데 필요한 새로운 호출 스택을 생성해서 그곳에 run 메서드를 올려둔다. 이후 그곳에서 run 메서드를 호출하고 스레드가 별개의 작업을 수행하게 된다.
- 보통 위와 같이 인터페이스를 구현해서 Thread의 인자로 넘기거나, 상속으로 구현하는 방식은 운영환경에서 프로그램 성능에 영향을 미치기 때문에 사용하지 않는다.운영 시에는 ExecutorService와 Executors를 이용해 스레드풀을 생성 하여 병렬처리한다.앞선 방식은 각기 다른 Thread를 생성해서 작업 처리하고 처리가 완료되면 Thread를 제거하는 작업을 손수 진행해야 하지만 ExecutorService 클래스를 사용하면 손쉽게 처리할 수 있다.ExecutorService에 Task만 지정해주면 알아서 ThreadPool을 이용해 Task를 실행하고 관리한다.Executors는 ExecutorService 객체를 생성하며 다음과 같은 메서드를 제공하여 스레드 풀의 개수 및 종류를 정할 수 있다.

34.1. 스레드 상태

- NEW: 생성되고 아직 start가 호출되지 않은 상태
- RUNNABLE: 실행 중 또는 실행 가능한 상태
- BLOCKED: 동기화 블럭에 의해서 일시정지된 상태(lock이 풀릴 때까지 기다리는 상태)
- WAITING: 다른 스레드가 통지할 때까지 기다리는 상태
  - 메인 스레드에서 별도 스레드를 만들고 별도스레드.join()을 호출하면 메인 스레드는 별도 스레드가 끝날때 까지 해당 위치에서 대기하다가 끝난 후에야 이후 로직을 수행한다.
- TIMED_WAITING: 주어진 시간 동안 기다리는 상태
  - 스레드를 하나 만들고 run메서드에서 sleep을 시키고 메인 스레드에서 해당 스레드의 상태를 getState로 찍어보면 TIMED_WAITING 상태로 찍힌다.
- TERMINATED: 쓰레드의 작업이 종료된 상태

34.2. 스레드 스케줄링

- sleep: 주어진 시간동안 일시 정지
- join: 특정 스레드가 다른 스레드의 완료를 대기
- interrupt: sleep, join에 의해 일시정지 상태인 스레드를 실행 대기상태로 전환하는 것
- yield: 실행 중에 다른 스레드에게 양보하고 실행 대기상태가 되는 것

34.3. 우선순위

- setPriority(1~10): 5가 디폴트로 실행 우선순위를 바꿀 수 있으나 항상 우선순위대로 동작하진 않는다.

34.4. 스레드 그룹

- 서로 관련된 스레드를 그룹으로 묶어서 다루기 위한 것으로 모든 쓰레드는 반드시 하나의 스레드 그룹에 포함되어 있어야 한다.

- 스레드 그룹을 지정하지 않고 생성한 스레드는 main 스레드 그룹에 속한다.
- 자신을 생성한 스레드(부모 스레드)의 그룹과 우선순위를 상속받는다.

---

35. 동기화

- 멀티스레드 프로그래밍에서는 하나의 프로세스를 동시에 여러 쓰레드가 접근하게 된다.즉, 하나의 데이터를 공유해서 작업하게 된다.
- synchronized 키워드는 통해 공유 데이터에 lock을 걸어서 먼저 작업 중이던 스레드가 작업을 완전히 마칠 때까지 다른 쓰레드가 접근해도 공유 데이터가 변경되지 않도록 보호하는 기능을 제공한다.
- synchornized 키워드는 어떤 것을 위한 wait, notify인지 명시적으로 알기 어렵기 때문에 Lock, Condition을 사용하기도 한다.(내부적으로는 synchornized로 구현되어 있다.)
- 스레드 로컬을 사용하면 각 스레드만의 고유 공간을 만들어서 사용할 수 있다.
- java.util.concurrent 패키지에 동기화 컬렉션과 atomic 자료형을 제공한다.(concurrentHashMap, AtomicInteger 등)
  - atomic은 CAS알고리즘을 사용한다. (특정 메모리 위치와 주어진 위치의 value를 비교해 다르면 대체하지 않는다.)

---

36. Synchronized, lock&condition 동기화

- 동기화 처리하는 블록을 Synchronized로 묶게 되면 해당 블록에는 하나의 스레드만 접근할 수 있다. 만약 하나의 스레드가 들어가서 특정 조건이 만족될 때까지 반복 루프를 돌게 될 경우 빠져나오지 않는 상황이 발생할 수 있다. 이런 상황에서 동기화의 효율을 높이기 위해 wait, notify를 사용한다. 이 경우에는 안에서 wait을 호출하여 객체의 lock을 풀고 루프를 돌고 있는 스레드를 해당 객체의 waiting pool에 넣는다. 그리고 조건을 만족시키도록 하는 로직에서 작업 후 notify를 호출하여 대기 중인 스레드 중 하나를 깨워준다. notify를 통해 스레드를 깨울 때는 랜덤하게 pool에 있는 것 중 하나를 깨우기 때문에 일반적으로는 notifyAll를 사용해서 전체를 깨운다. 하지만 이렇게 로직을 짜다보면 wait과 notify가 어떤 것을 위한 것인지 알 수 없기 때문에 lock, condition이 추가되었고 이를 통해 명확하게 구분할 수 있게 되었다.
- concurrent.locks 패키지는 내부적으로 synchronized를 사용하여 구현되어 있지만 더욱 유연하고 세밀하게 처리하기 위해 사용된다.
  - Lock: 공유 자원에 한 번에 한 쓰레드만 read, write가 수행 가능하도록 제공하는 인터페이스
    - ReentrantLock: Lock의 구현체로 임계 영역의 시작과 종료 지점을 직접 명시할 수 있다.
  - ReadWriteLock: 공유 자원에 여러 개의 스레드가 read 가능하고, write는 한 스레드만 가능한 인터페이스
    - ReentrantReadWriteLock: ReadWriteLock의 구현체
- synchronized는 블록구조를 사용하여 메서드 안에 임계영역의 시작과 끝이 있고 간단하다. Lock은 lock, unlock 메서드로 시작과 끝을 명시하기 때문에 명확하고 임계 영역을 여러 메서드에서 나눠서 작성할 수 있다.

---

37. 디자인 패턴

- 싱글톤 패턴
  - 하나의 객체만 생성하여 공유하는 방식
- 팩토리 패턴
  - 객체간 의존성을 줄이기 위해 객체의 생성과 데이터 주입만 담당하는 Factory class를 정의하여 사용하는 부분에서는 생성된 객체를 가져다 사용하여 의존성을 줄이는 방식
- 템플릿 메서드 패턴
  - 하나의 템플릿 구조를 만들어 두고 메인이 되는 로직은 추상 메서드로 만들고 상속받는 클래스가 구현하는 방식
- 전략 패턴
  - 템플릿 메서드 패턴처럼 구조를 만들어 두어 사용하지만 메인이 되는 로직은 인터페이스를 사용하여 전략을 주입하는 방식

- 옵저버 패턴
  - 정보 수신 시 하나의 객체가 변하면 다른 객체에 객체가 변했다는 사항을 알려주는 패턴

---

38. 강한 결합 vs 느슨한 결합

- 결합도는 의존성의 정도를 나타내며 다른 모듈에 대해 얼마나 많은 정보를 알고 있는지에 대한 척도이다.
- 어떤 모듈이 다른 모듈에 대해 자세한 부분(구현 세부사항)까지 알고 있는 경우를 강한 결합이라고 하고, 필요한 정보(추상화 정도)만 알고 있는 경우 느슨한 결합이라고 한다.

---

39. DDD

- DDD란 비즈니스 도메인 별로 나눠서 설계하는 방식으로 여러 도메인들이 서로 상호작용하도록 도메인을 중심으로 설계하는 것을 말한다.
- DDD의 핵심 목표는 모듈간의 의존성을 최소화하고 응집성은 최대화하는 것이다.
- Presentation, application, domain, infrastructure 계층으로 구분되어 있다.
  - presentation -> application -> domain -> infrastructure
  - 하위 계층으로만 의존함으로서 구조가 복잡하여 발생할 수 있는 순환참조를 막을 수 있다.
  - 단방향으로 구성되어있기에 각 계층별 로직을 쉽게 이해할 수 있다.
  - infrastructure는 domain의 추상화를 구현하는 계층으로 DIP의 활용을 극대화할 수 있다.