Java 디브리핑

2025.05.14

Contents

- 14:00 ~ 14:50　진단 결과 분석
- 14:50 ~ 15:00　쉬는 시간
- 15:00 ~ 16:20　조별 구성, 조별 맞춤형 피드백 / Java Quiz 및 실습 (동시진행)
- 16:20 ~ 16:30　쉬는 시간
- 16:30 ~ 16:50　방향성 제시 및 최적 접근법 제안
- 16:50 ~ 17:00　Wrap up 및 설문조사

Java의 성공

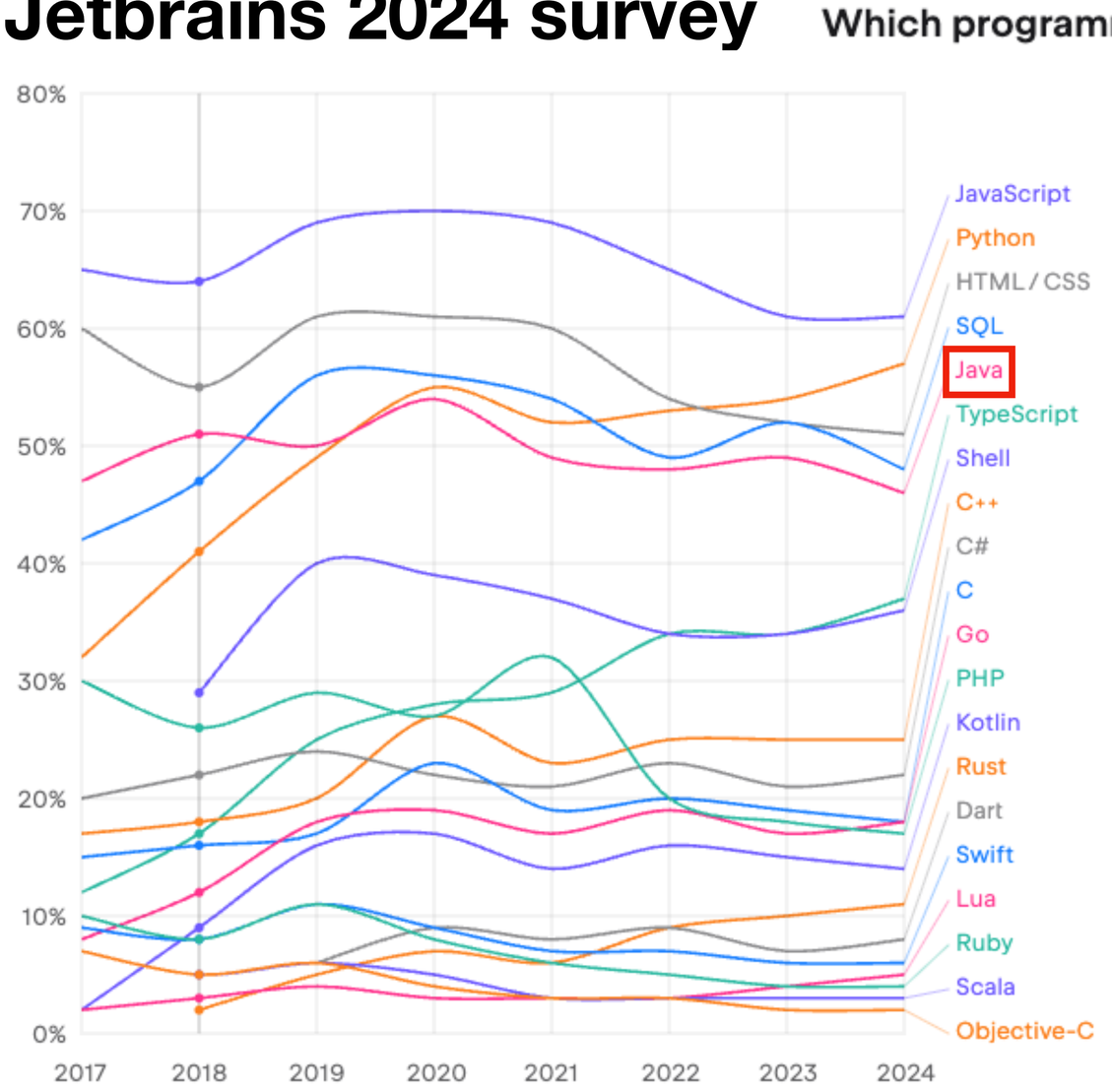
- 원래 타겟: Embedded Device (냉장고, 세탁기 등)
- 웹 시대: Servlet, JSP, Spring, Struts
- 빅데이터 시대: Hadoop Ecosystem, HDFS, MapReduce, Apache 시리즈
- 스마트폰 시대: Android

한국에서의 Java

- 자바 공화국
- 전자정부 표준 프레임워크
- 국비 지원 IT 교육
- 자바만 할줄 알아도 굶어 죽진 않는다(?)
- 자바 코딩하다 막히면 판교 치킨집 아저씨한테 물어본다(?)

세계적인 추세

Jetbrains 2024 survey

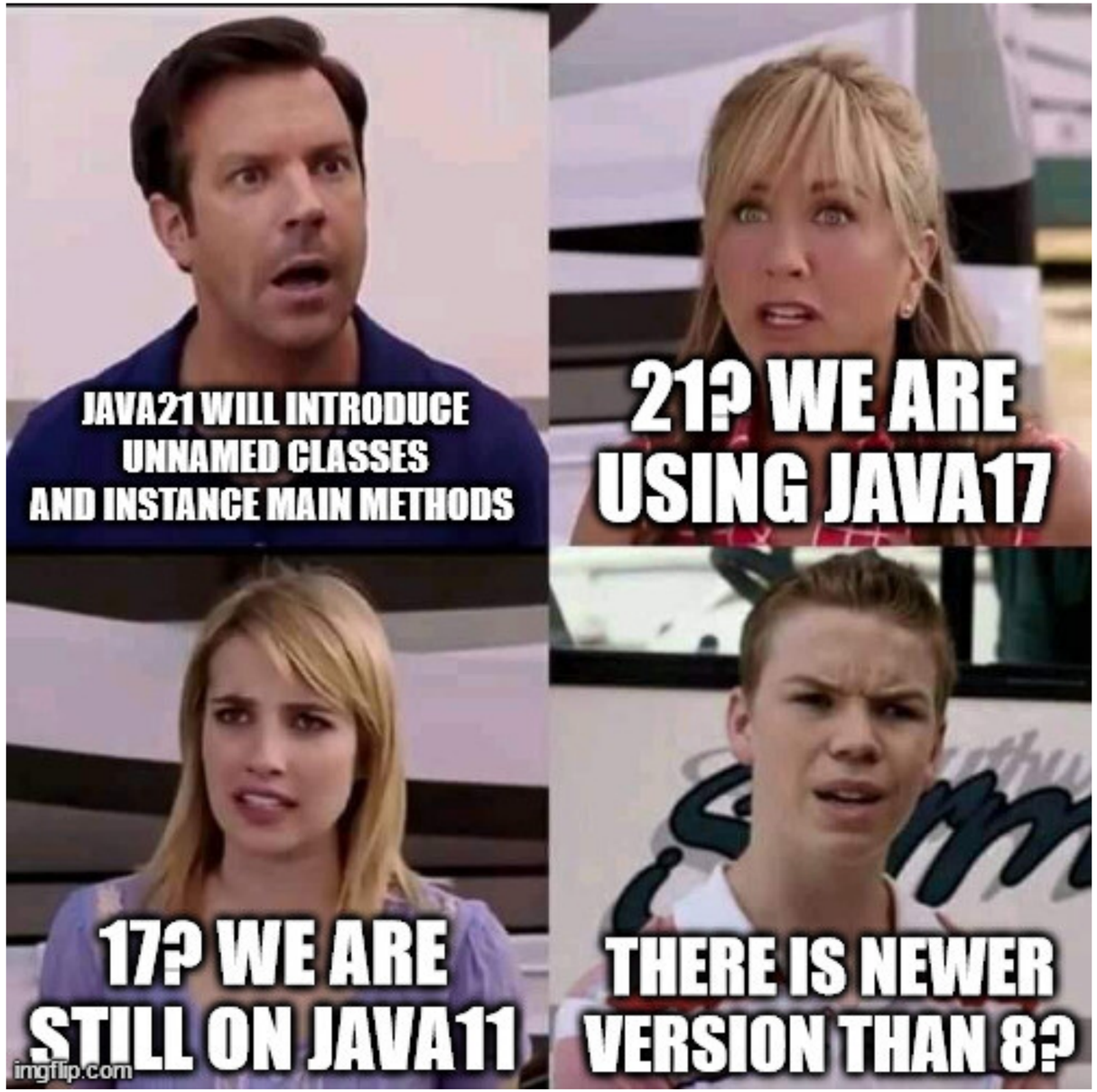


- Rust, Go, TS 등 다양한 언어가 나왔지만 여전히 상위권

출처: https://www.jetbrains.com/lp/devecosystem-2024

자바의 현재

- 엄청난 레거시 시스템
  - Hadoop 진영이 여전히 JAVA 8ㅜ
- Mobile, Server에서의 코틀린의 역습
  - 코틀린으로 프로그래밍
  - 빌드는 Android Dex / JAVA 8
- AI / ML에서는 파이썬 + Kubernetes
  - Hadoop: Cluster 관리를 자바
  - AI / ML: Cluster 관리를 K8s



자바의 미래

- 많은 욕을 먹지만 끊임 없이 발전하고 성장하는 언어
  - Stream API / Lambda (Scala)
  - Data Class: Record (Kotlin)
  - Virtual Thread (Golang)
- HotSpot 이외의 다양한 JVM 구현체
  - GraalVM을 이용하면 네이티브 성능 발휘 (AOT)

진단 결과 분석

초급 내용

- Variables: 변수, 초기화, scope
- Data Types: 기본 자료형 / 형변환
- Operators: 산술, 논리, 비교 연산자
- Conditionals: if - else, switch 등 조건 분기
- Loops: for, while 등 반복문
- Strings: 문자열 처리 및 비교
- Arrays: 배열 선언 및 활용
- Methods: 매개변수 전달 및 함수 정의

C style vs Java style

```
for (double clickRate: clickRates) {
      if (clickRate > threshold) count++;
    }
```

vs

```
long count = Arrays.stream(clickRates)
                        .filter(rate -> rate > threshold)
                        .count();
```

```
double rating = Arrays.stream(bookRecord)
                        .average()
                        .orElse(0);
```

C style vs Java style

- 파일 이름 비교 문제

```
for (int i =0; i<projectFiles.length ; i++) {
    if(projectFiles[i].indexOf(projectName)>-1 && projectFiles[i].indexOf("v"+version)>-1)
        System.out.println(projectFiles[i]);
}
```

vs

```
String targetName = projectName + "_v" + version;
ArrayList<String> targetFiles = new ArrayList<>();
for (String projectFile : projectFiles) {
    if (projectFile.startsWith(targetName)) {
        targetFiles.add(projectFile);
    }
}
```

연차별 코드

```java
class HelloWorld
{
    public static void main(String args[])
    {
        // Displays "Hello World!" on the console.
        System.out.println("Hello World!");
    }
}
```

1년차

```java
/**
 * Hello world class
 *
 * Used to display the phrase "Hello World!" in a console.
 *
 * @author Sean
 */
class HelloWorld
{
    /**
     * The phrase to display in the console
     */
    public static final string PHRASE = "Hello World!";

    /**
     * Main method
     *
     * @param args Command line arguments
     * @return void
     */
    public static void main(String args[])
    {
        // Display our phrase in a console.
        System.out.println(PHRASE);
    }
}
```

2년차

연차별 코드

```java
/**
 * Hello world class
 *
 * Used to display the phrase "Hello World!" in a console.
 *
 * @author Sean
 * @license LGPL
 * @version 1.2
 * @see System.out.println
 * @see README
 * @todo Create factory methods
 * @link https://github.com/sean/helloworld
 */
class HelloWorld
{
    /**
     * The default phrase to display in the console
     */
    public static final string PHRASE = "Hello World!";

    /**
     * The phrase to display in the console
     */
    private string hello_world = null;

    /**
     * Constructor
     *
     * @param hw The phrase to display in the console
     */
    public HelloWorld(string hw)
    {
        hello_world = hw;
    }

    /**
     * Display the phrase "Hello World!" in a console
     *
     * @return void
     */
    public void sayPhrase()
    {
        // Display our phrase in a console.
        System.out.println(hello_world);
    }

    /**
     * Main method
     *
     * @param args Command line arguments
     * @return void
     */
    public static void main(String args[])
    {
        HelloWorld hw = new HelloWorld(PHRASE);
        try {
            hw.sayPhrase();
        } catch (Exception e) {
            // Do nothing!
        }
    }
}
```

5년차

```java
/**
 * Used to display the phrase "Hello World!" in a console
 *
 * @author Sean
 * @see README
 */
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello World!");
    }
}
```

10년차

Java object

- Object reference compare
  - a == b

    vs
  - a.equals(b)

String append

- String vs StringBuilder vs StringBuffer
  - String
  - StringBuilder
  - StringBuffer

String append

- String vs StringBuilder vs StringBuffer
  - String: 불변 (String Pool)
  - StringBuilder: 가변 + Not thread-safe
  - StringBuffer: 가변 + Thread-safe

Spring

- Spring에서 thread unsafe한 클래스를 사용해도 괜찮을까?
  - ex) StringBuilder 사용

Spring

- Spring에서 thread unsafe한 클래스를 사용해도 괜찮을까?
  - ex) StringBuilder 사용
  - 기본적으로는 No, but Request scope을 사용한다면 괜찮다.
    - 매 request 마다 새롭게 생성되니깐

중급 내용

- OOP: 상속, 다형성, 인터페이스
- Standard Libraries: 자바 표준 라이브러리 활용
- Generics: 타입 재활용
- Enum: 열거형 타입
- Collection Framework: 자바 컬렉션 활용
- Exception Handling: 예외처리

전반적인 느낌

- 시간이 조금 부족했을 수도 있을 것 같다.
  - 문제가 길어서 이해하는데에도 시간이 좀 걸릴 것 같다.
  - 역시 개발자는 국어를 잘해야..

OOP

- Encapsulation
  - 객체 상태를 숨김
- Inheritance
  - 상위 클래스 함수 재사용
  - 인터페이스
- Polymorphism
  - Animal a = new Dog()

Polymorphism

- Overloading: add(double, double), add(int, int)
- Overriding:@Override
- Overwriting

Polymorphism

- Overloading: compile time에서 결정이 됨
- Overriding: runtime에서 결정이 됨 --> 주로 얘기하는 Polymorphism
- Overwriting: 기존 값을 덮어씌우는 것

Generic이 필요한 이유?

- Java Collection의 element로 넣기 위해?
  - ArrayList<Dog> list = new ArrayList<Dog>();
- 인터페이스를 사용하면 안되나?
  - ArrayList<Animal> list = new ArrayList<Animal>();
- 차이가?

Enum 타입

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

- Static class field로도 구현 가능 (Sugar 코드라고 생각함)

```
public class DayType {
    public static final DayType MONDAY = new DayType("MONDAY");
    public static final DayType TUESDAY = new DayType("TUESDAY");

    private final String name;

    private DayType(String name) {
        this.name = name;
    }
}

DayType today = DayType.MONDAY;
```

Enum 타입

- Python의 경우

```
>>> from enum import Enum

>>> # class syntax
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # functional syntax
>>> Color = Enum('Color', [('RED', 1), ('GREEN', 2), ('BLUE', 3)])
```

Copy

Collection Framework

```
int[] array = new int[size];
```

vs

```
ArrayList array = new ArrayList<Integer>();
```

Collection Framework

- List: Ordered collections (e.g., ArrayList, LinkedList)
- Set: Unique elements (HashSet, LinkedHashSet, TreeSet)
- Map: Key-value pairs (HashMap, TreeMap, LinkedHashMap)
- Queue / Deque: FIFO/LIFO structures (PriorityQueue, ArrayDeque)

Collection Framework

- 구체화된 class의 차이 비교
  - 특징, performance, 추가, 삭제 Big O
  - ArrayList vs LinkedList
  - HashMap vs TreeMap

Collection Framework

- Iterator

```
it = list.iterator();
while (it.hasNext()) {
    Element e = it.next();
    ....
}
```

- Comparator (or Comparable)

```
Collections.sort(students, new StudentComparator());
```

Collection Framework

- Thread Safety and Concurrency
  - 멀티 쓰레드 환경에서 thread safe한 자바 컬렉션 사용

Exception Handling

- Error, Exception, and Throwable
- Exception type
  - Checked (IOException, SQLException): compile time
  - Unchecked (NullPointerException, IllegalArgumentException): runtime

Exception Handling

- 왜 자바에서는 전부 Exception class 하나만 있고 code로 구분하지 않고
- 각기 다른 Exception class를 만들까?

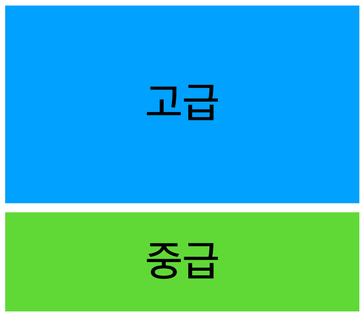
```
try {
  ...
} catch (Exception e) {
  switch(e.cause) { ... }
}
```

Exception Handling

- Checked vs Unchecking exception
- 각자 다른 곳에서 exception 처리 가능
- Exception hierarchy
  - IOException
    - FileNotFoundException, SocketException
- 가독성
    - FileNotFoundException vs 404 code

조별 구성 및 피드백

그룹



A 그룹

B 그룹

Java Quiz

- https://github.com/pepsicolav13/java-debriefing.git
- 평가 X
- 점수 / 채점 X
- 오픈북 / 시간제한 X
- chatGPT 등 AI에 질문 가능
- 개별 그룹 피드백을 위한 자가 평가 문제

그룹 A

- OOP
- Design Pattern
  - 클래스 간의 관계
- Java Collections API

Design Pattern

- Creational Patterns
  - Singleton
  - Factory Method
  - Abstract Factory
  - Builder
  - Prototype
- Structural Patterns
  - Adapter
  - Decorator
  - Composite
  - Proxy
  - Facade
  - Bridge
  - Flyweight
- Behavioral Patterns
  - Strategy
  - Observer
  - Command
  - Iterator
  - Template Method
  - State
  - Chain of Responsibility
  - Mediator
  - Memento
  - Visitor
  - Interpreter

Factory method vs Template method

- Factory method: object 생성을 subclass에서 생성하도록 위임
- Template method: input / output 만 interface로 지정하고 내부속은 subclass에서 작성하도록 위임
  - Framework에서 많이 사용
  - 내부속 로직 --> object 생성 == factory method

Framework: Template Method

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
          http
            .authorizeRequests()
              .antMatchers("/admin").hasAuthority("ROLE_ADMIN")
              .anyRequest().authenticated()
              .and()
           .formLogin()
             .and()
           .httpBasic();
  }
}
```

Factory Method

```
abstract class Dialog {
    abstract Button createButton();  // factory method

    void render() {
        Button btn = createButton();
        btn.onClick();
    }
}

class WindowsDialog extends Dialog {
    @Override
    Button createButton() {
        return new WindowsButton();
    }
}
```

Strategy Pattern

- Spring Controller와 Service의 관계

```
public interface UserService {
    User getUserById(Long id);
}

@Service
public class UserServiceImpl implements UserService {

    @Override
    public User getUserById(Long id) {
       return new User(id, "John Doe");
    }
}
```

```
@RestController
@RequestMapping("/users")
public class UserController {

    // This is the strategy object
    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.getUserById(id);
    }
}
```

Decorator Pattern

```
InputStream in = new BufferedInputStream(
                     new GZIPInputStream(
                         new FileInputStream("data.gz")));
```

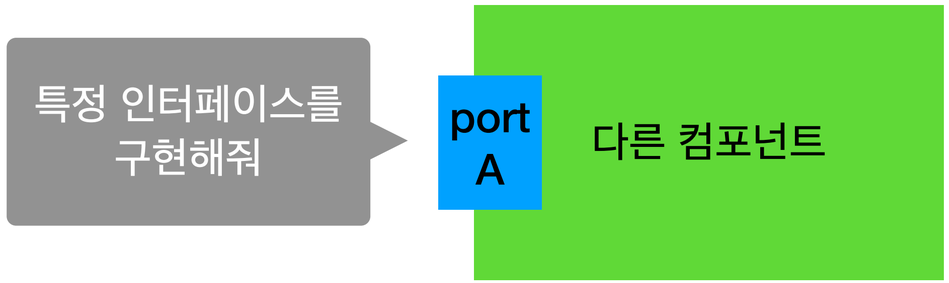
Interface란?

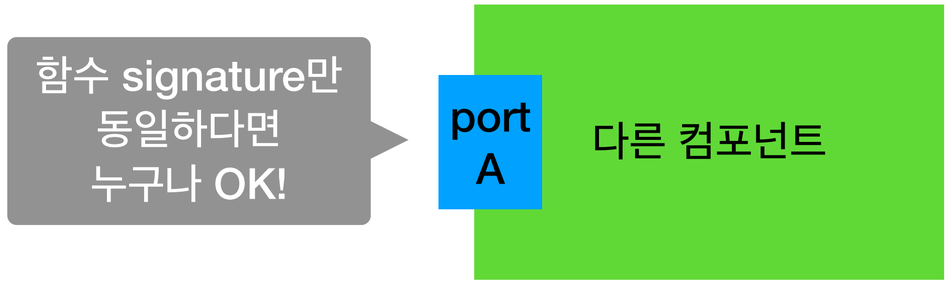
- USB port
  - 구현은 달라도
    - 마우스
    - 프린터
    - Display
    - 충전
  - 연결은 전부 USB 모양으로!

Interface vs Abstract class

- Abstract class는 상속을 목적으로 함.
  - 일부는 super class에서 구현
  - 일부는 sub class에서 구현
- Interface는 다른 컴포넌트와의 연결(인터페이싱)을 목적으로 함.
  - 단지 자바에서는 상속을 통해서 구현
  - 다른 언어에서는 상속을 하지 않고 구현

Interface in Golang (or Python)






```
class MyComponent implements A {
  public String methodA() {}
  public String methodB() {}
}
```

Java

```
class MyComponent {
  public String methodA() {}
  public String methodB() {}
}
```

Golang

Python

Duck Typing

- 오리처럼 울고 오리처럼 뒤뚱뒤뚱 걷는다면 그건 오리야!
- Python(dynamic), Golang(static)는 함수 signature만 맞는다면 인터페이싱이 된다.
  - Python에서 상속은 주로 부모의 함수를 재사용
  - Framework에서 명시적으로 어떤 함수를 재작성해야 하는지 보여주기 위해서 사용

Inner class

- Static inner class
  - Namespace 처럼 사용
    - Utility.ToolA / Utility.ToolB
  - 주로 builder pattern에서 XXclass.Builder 로 많이 활용
- Non-static inner class
  - Event Callback
  - GUI 코딩할 때, 많이 사용 (Android, Swing 등)

Is-a vs Has-a 관계

- Is-a: Inheritance (상속) 관계
  - Animal & Dog
- Has-a: Composition (합성) 관계
  - Car & Engine
  - Spring에서 Controller & Service 관계
  - Design Pattern: Strategy Pattern

Generics

- Type parameter
  - public T getData() { return data; }
  - 특정 type 제한이 없음
- Bounded parameter
  - public <T extends InterfaceA> T getData() { return data; }
  - 특정 인터페이스를 꼭 구현해야 함

Generics

- 그냥 Interface를 사용하면 안되나?
  - public <T extends InterfaceA> T getData() { return data; }
  - public InterfaceA getData() { return data; }

Generics

- 그냥 Interface를 사용하면 안되나?
  - public <T extends InterfaceA> T getData() { return data; }
  - public InterfaceA getData() { return data; }
  - Interface로 반환하면 super class method 밖에 사용할 수 없다.

```
InterfaceA a = getData();
a.super();
```

```
SubClassB b = getData<SubClassB>();
b.methodOnlyInB();
```

Generics & Java Collections

- 인터페이스 구현 시, 많이 사용

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

```
public class Student implements Comparable<Student> {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Implement the compareTo method
    @Override
    public int compareTo(Student other) {
        return Integer.compare(this.age, other.age);
    }
}
```

```
// Main method to test sorting
public static void main(String[] args) {
    List<Student> students = new ArrayList<>();
    students.add(new Student("Alice", 22));
    students.add(new Student("Bob", 20));
    students.add(new Student("Charlie", 25));

    Collections.sort(students); // uses compareTo()
}
```

Java Collections

- HashMap Element class
  - equals();
  - hashCode();
- for each loop 대신 iterator를 사용하는 이유?
  - remove() while looping
  - records를 노출하고 싶지 않을 때

그룹 B

- Java I/O: 파일, 스트림 데이터 등 입출력 처리
- JDBC: DB 연결 및 SQL 실행
- Stream API and Lambda Expressions: 함수형 스타일 코딩
- Thread: 동시 작업 및 경쟁 상태 제어
- Network: 소켓, 웹 등 네트워킹

Buffered IO

- Buffered vs UnBuffered IO 차이
  - BufferedReader / BufferedWriter
  - BufferedInputStream / BufferedOutputStream

Buffered IO

- Buffered vs UnBuffered IO 차이
  - BufferedReader / BufferedWriter
  - BufferedInputStream / BufferedOutputStream
  - class 내부적으로 buffer를 가지고 있냐 / 없냐
    - system call 호출 횟수의 차이

Multi Thread programming

- Multi-tasking
  - concurrency vs parallelism
- Race condition
- Critical section
  - Lock, Semaphore
- volatile variable
- Atomic operation
  - synchronized
  - AtomicInteger class 등
- Thread safe class / ThreadLocal class

Race condition

```
public class RaceExample {
    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter++;  // not thread-safe
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final counter: " + counter);
    }
}
```

Multi Thread programming

- Runnable vs Thread 차이
  - 무엇을 vs 어떻게?

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running");
    }
}

new MyThread().start();
```

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Running");
    }
}

new Thread(new MyRunnable()).start();
```

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.submit(new MyRunnable());
```

Thread Pool

- ThreadPoolExecutor
  - Managing fixed number of threads
  - Thread creation / reuse / termination

Thread-safe Collections

- Not thread-safe
  - ArrayList, HashMap
- Thread-safe
  - Vector, HashTable, ConcurrentHashMap

Modern Multi threading

- Parallel streams: Stream API
- CompletableFuture: Async style
- Reactive programming: Stream + Async + Observer pattern
- Virtual threads: Green Thread

Lambda expression

- 콜백 함수로 활용

```
// Anonymous inner class
button.setOnClickListener(new OnClickListener()
{
    @Override
    public void onClick() {
       System.out.println("Button clicked");
    }
});
```

```
// Lambda Expression
button.setOnClickListener(() ->
    System.out.println("Button clicked")
);
```

Lambda expression

- Stream API

```
List<String> words = Arrays.asList("hello", "java", "lambda");

// Using lambda expression with map()
List<String> uppercased = words.stream()
                              .map(word -> word.toUpperCase())  // lambda here
                              .collect(Collectors.toList());
```

- Lazy evaluation

```
log.debug(expensiveOps());
```

```
log.debug(() -> expensiveOps());
```

Stream API

- Lazy evaluation

```
String find = Stream.of("a", "b", "c")
       .filter(s -> s.startsWith("b"))
       .findFirst();
// stops at "b", doesn't scan "c"
```

- Readability

```
String find = "";
for (String s: list) {
    if (s.startsWith("b"))
        find = s;
```

Stream API

- Parallelism

```
list.parallelStream().map(...).collect(...);
```

- 특히 Stream API를 사용하면 Thread safe에 대해서 걱정할 필요가 없다!(비교적)

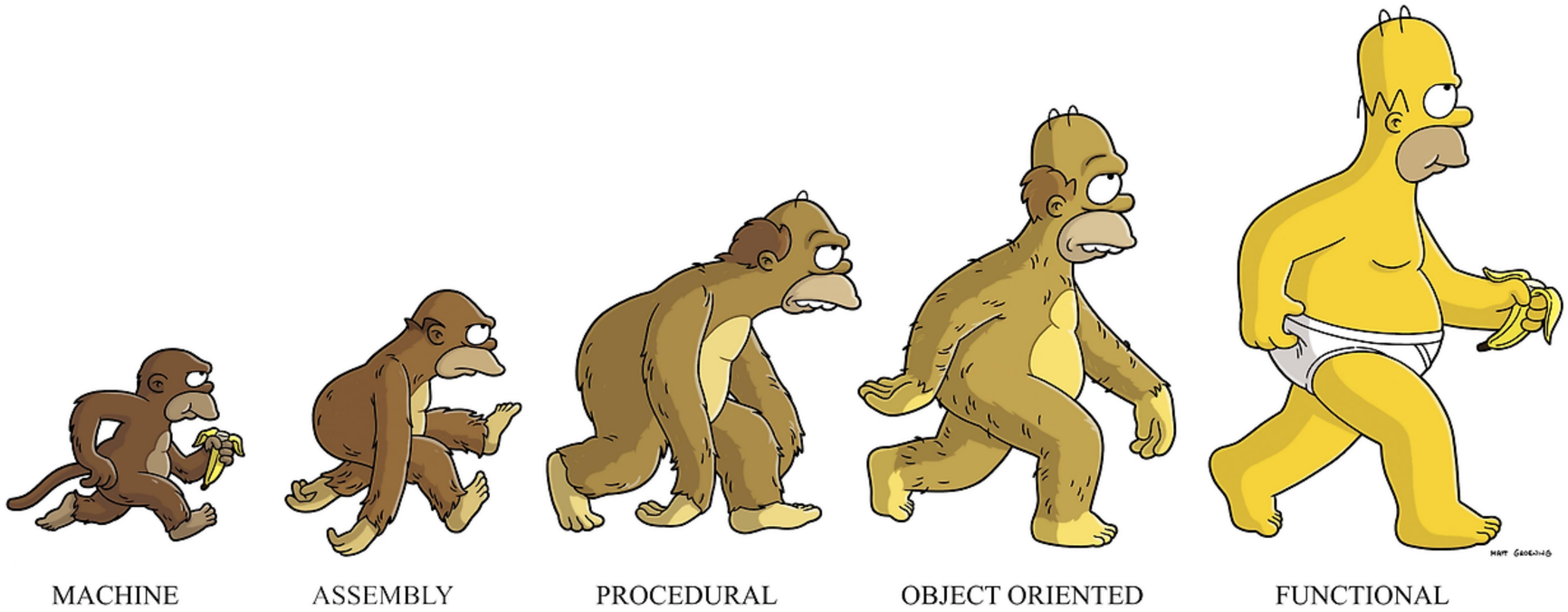
```
List<String> list = List.of("a", "b", "c");
list.parallelStream().map(String::toUpperCase)
                     .forEach(System.out::println);
```

- Thread safe하지 않는 상황

```
List<String> list = List.of("a", "b", "c");
List<String> result = new ArrayList<>();
list.parallelStream().map(...).forEach(result::add);
```

Stream API

- Functional programming을 사용한다는 자부심!



방향성 제시
및
최적 접근법 제안

단순히 외우기 보다는 왜?

- Interface와 Abstract class의 차이?
  - 두개 이상의 클래스를 상속하지 못하고
  - Diamond 상속이 가능하고
  - 등등 보다는
- Interface는 다른 컴포넌트와 연결하기 위해
- Abstract class는 상속을 받기 위해

Design Pattern은 만능이 아니다

- 망치를 든 사람에게는 모든 게 못으로 보인다
- 굳이 복잡하게 클래스 나눌 필요가 없다.

Design Pattern은 만능이 아니다

내가 작성하고 있는 코드

```
// Strategy Interface
interface LoggerStrategy {
    void log(String message);
}

// Concrete Strategy
class ConsoleLogger implements LoggerStrategy {
    public void log(String message) {
        System.out.println("[Console] " + message);
    }
}

// Logger Factory
class LoggerFactory {
    public static LoggerStrategy getLogger(String type) {
        if ("console".equals(type)) {
            return new ConsoleLogger();
        } else if ("file".equals(type)) {
            return new FileLogger();
        }
        throw new IllegalArgumentException("Unknown logger
type");
    }
}
```

```
// Singleton Holder
class LoggerService {
    private static LoggerStrategy logger;

    static {
        logger = LoggerFactory.getLogger("console");
    }

    public static LoggerStrategy getLogger() {
        return logger;
    }
}

// Usage
public class App {
    public static void main(String[] args) {
        LoggerService.getLogger().log("Hello, world!");
    }
}
```

Design Pattern은 만능이 아니다

진짜 필요한 코드

```
public class App {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Design Pattern은 만능이 아니다

- 코드를 수정하는 일이 있다고 하더라도 최대한 구체화된 코드를 작성
- 뭔가 복붙하는 느낌이 든다면 고민해보자.
- 코드를 추가/수정할 때마다 특정 파일(class)를 수정해야 하는 경우가 빈번하다면 고민해 보자.
- 상속보다는 합성(composition) 관계를 활용하자 (Strategy, Decorator, Proxy 등)
  - 상속은 대부분의 경우, 인터페이싱을 위한 경우가 많음
  - 하위 클래스에서 뭔가를 결정하게 만드는 경우, 상속을 사용 (Template, factory method)

제안

- 아이러니하게 다른 언어도 경험하면 더 깊게 자바를 이해할 수 있다고 생각함.
  - Python
  - JS
  - Golang
- chatGPT에게 많이 물어보자.
  - What is the difference between A and B?

Framework (Spring) 공부

- Spring: IoC / DI, Patterns (Proxy, Command, Template Method)
- Spring Boot: Convention over configuration (XML)
- Spring Cloud: Micro Service

OS 공부 (Linux)

- 프로세스 scheduling
- Memory 구조
  - page cache
- Synchronization technic
  - mutext, semaphore, spinlock
- System call
- Virtual File System
- Block I/O
- Networking
  - TCP / IP, ethernet, load balancer

JVM 공부

- Class Loader
- Runtime Data Areas
  - Method Area
  - Heap / Stack
  - Program Counter (PC) Register
- Execution Engine
  - Interpreter
  - JIT compiler
- Java Native Interface
- Heap memory optimization: Stop the world

도서

- Effective Java
- Head First Design Pattern
- 토비의 스프링 3.0
- 그림으로 배우는 리눅스 구조

강의

- 인프런 김영한 강사
  - 실전 자바: https://www.inflearn.com/roadmaps/744
  - Spring Core: https://www.inflearn.com/course/스프링-핵심-원리-기본편
  - Spring + JPA: https://www.inflearn.com/roadmaps/149

EOD