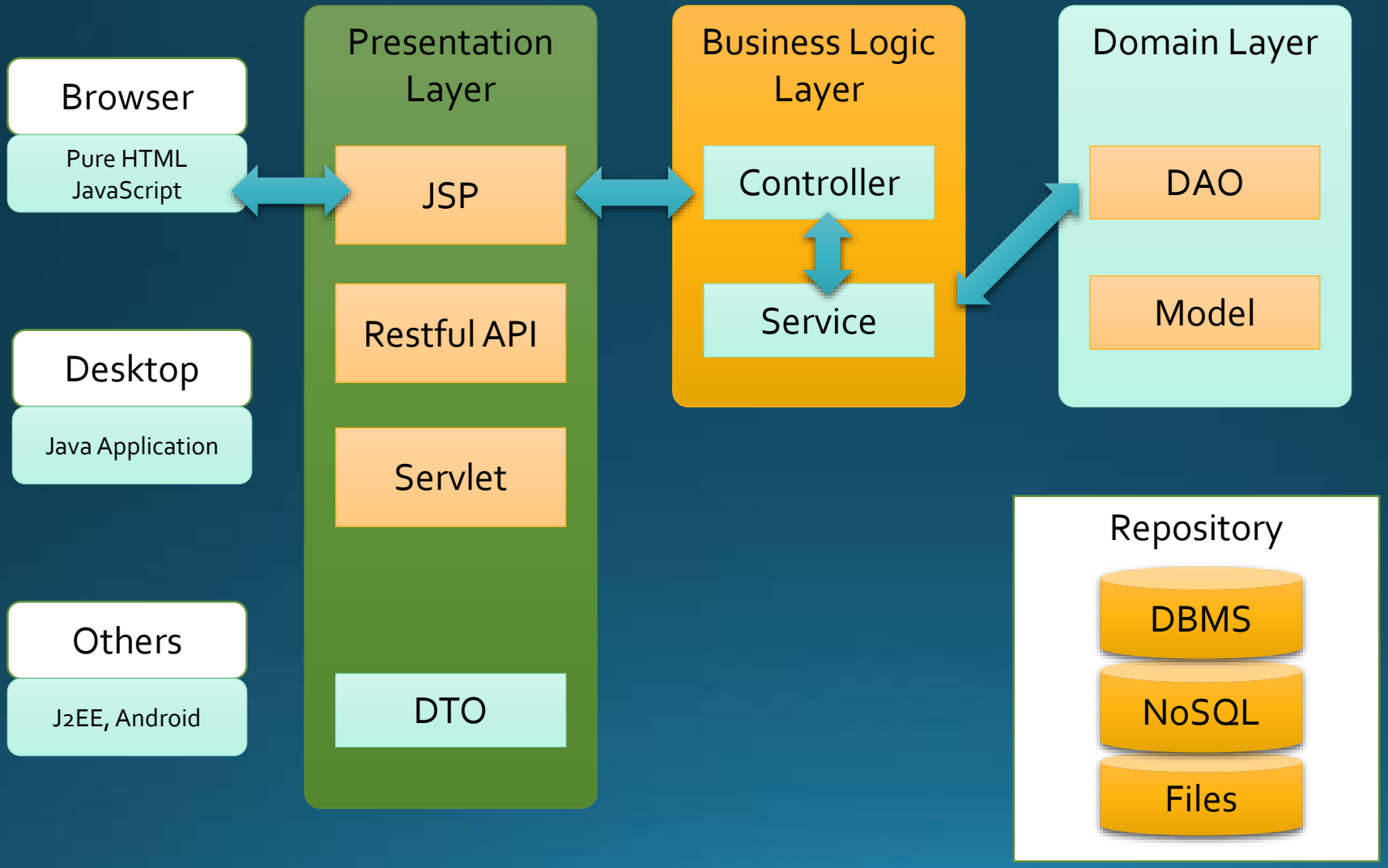
Java Application 구조와 Naming 전략

Java Naming Strategy

목차

- Java Application Architecture
- Java Naming Strategy

Java Application Architecture



Application 구성 규칙

- 각 도메인 별 모듈을 구성한다
  - Web
  - Services
  - Domains
  - Core 또는 Utils
  - 예: 씨앗 프로젝트

```
<modules>
  <module>seaat-core</module>
  <module>seaat-domain-jpa</module>
  <module>seaat-domain-mongodb</module>
  <module>seaat-domain-dto</module>
  <module>seaat-jobserver</module>
  <module>seaat-apiserver</module>
  <module>seaat-webapp</module>
</modules>
```

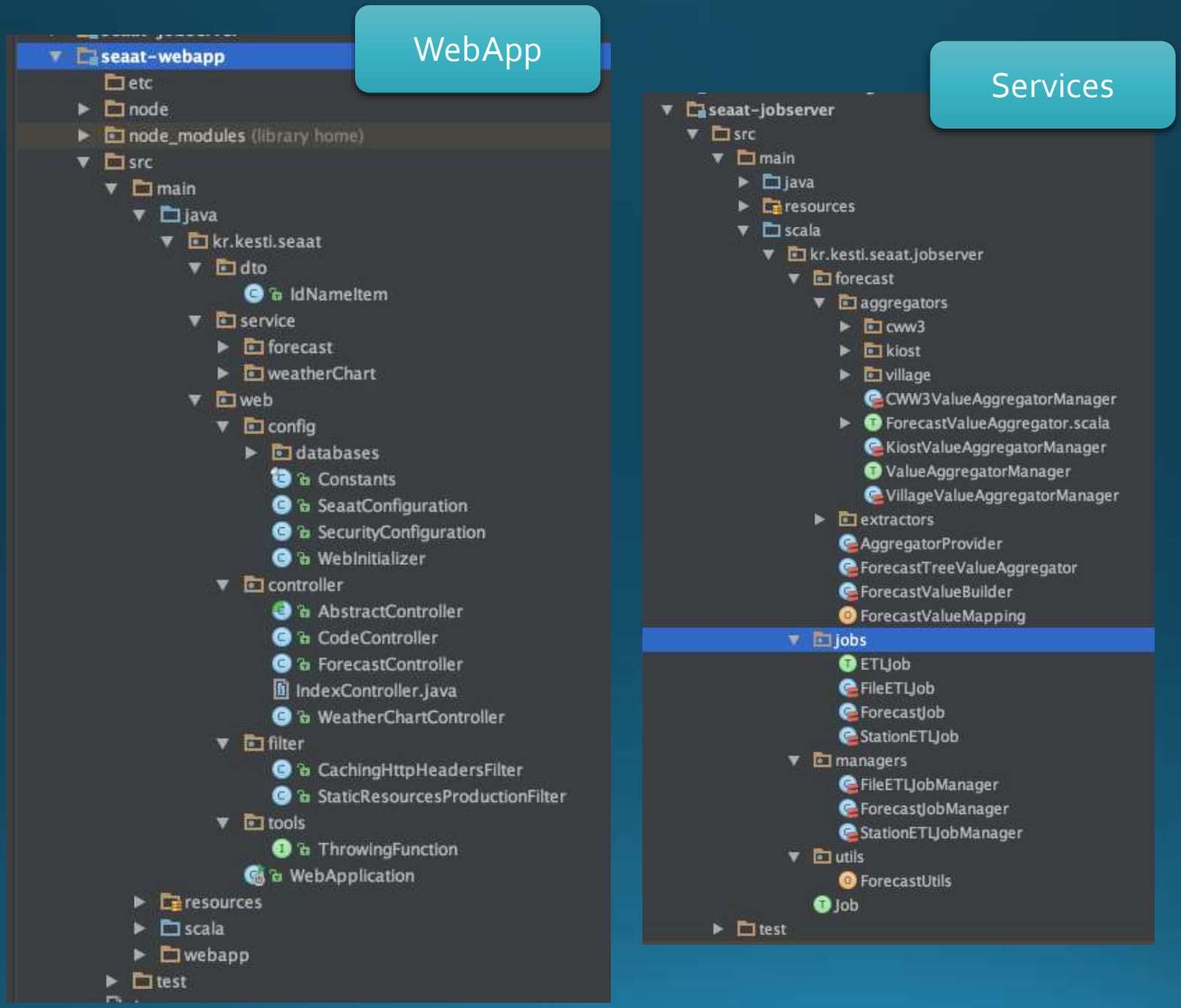
Application 구성 규칙 2

- 모듈 단위로 재활용이 가능하게끔 분리
- 의존성 (Dependency) 을 고려할 것
- 확장성을 고려할 것
  - Background Service 를 독립적으로 실행할 때 Domain Layer 를 Service Layer 와 공유하게 한다

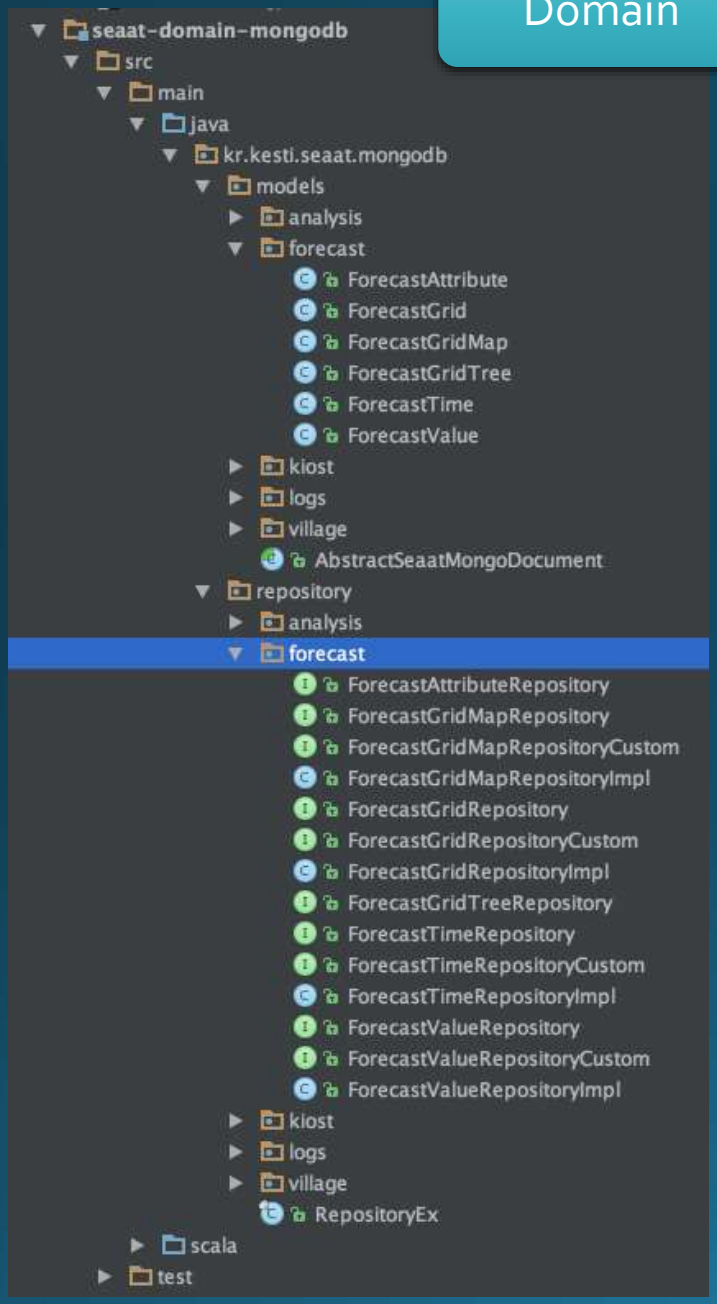
Package naming 규칙

- Java Convension을 따른다.
  - 국가 / 회사 / 프로젝트 명이 먼저 정의되어야 한다.
  - 예: kr.co.kesti.seaat, kr.co.kesti.argogis
  - 단 회사의 경우 굳이 co 를 넣을 필요 없을 때에는 생략한다.
    kr.kesti.seaat, kr.kesti.argogis

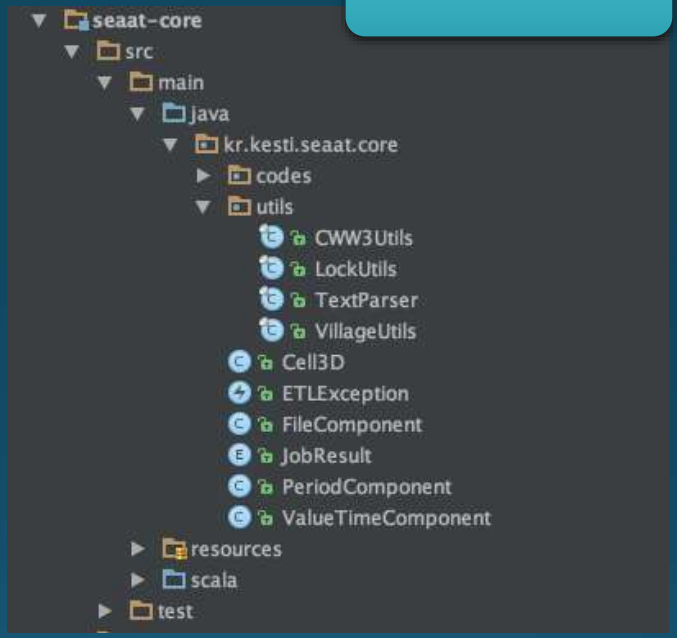
- 하위 package 명은 layer 에 따라 명명한다.
  - Web Application Controller 가 정의되는 곳은
    kr.kesti.seaat.web.controllers
  - Domain 관련 정의는
    kr.kesti.seaat.domain.models (모델 – ValueObject)
    kr.kesti.seaat.domain.daos (DAO 들 – repository 라고도 함)


WebApp
Services
seaat-webapp
etc
node
node_modules (library home)
src
main
java
kr.kesti.seaat
dto
IdNameItem
service
forecast
weatherChart
web
config
databases
Constants
SeaatConfiguration
SecurityConfiguration
WebInitializer
controller
AbstractController
CodeController
ForecastController
IndexController.java
WeatherChartController
filter
CachingHttpHeadersFilter
StaticResourcesProductionFilter
tools
ThrowingFunction
WebApplication
resources
scala
webapp
test
seaat-jobserver
src
main
java
resources
scala
kr.kesti.seaat.jobserver
forecast
aggregators
cww3
kiost
village
CWW3ValueAggregatorManager
ForecastValueAggregator.scala
KiostValueAggregatorManager
ValueAggregatorManager
VillageValueAggregatorManager
extractors
AggregatorProvider
ForecastTreeValueAggregator
ForecastValueBuilder
ForecastValueMapping
jobs
ETLJob
FileETLJob
ForecastJob
StationETLJob
managers
FileETLJobManager
ForecastJobManager
StationETLJobManager
utils
ForecastUtils
Job
test

Domain


seaat-domain-mongodb
src
main
java
kr.kesti.seaat.mongodb
models
analysis
forecast
ForecastAttribute
ForecastGrid
ForecastGridMap
ForecastGridTree
ForecastTime
ForecastValue
kiost
logs
village
AbstractSeaatMongoDocument
repository
analysis
forecast
ForecastAttributeRepository
ForecastGridMapRepository
ForecastGridMapRepositoryCustom
ForecastGridMapRepositoryImpl
ForecastGridRepository
ForecastGridRepositoryCustom
ForecastGridRepositoryImpl
ForecastGridTreeRepository
ForecastTimeRepository
ForecastTimeRepositoryCustom
ForecastTimeRepositoryImpl
ForecastValueRepository
ForecastValueRepositoryCustom
ForecastValueRepositoryImpl
kiost
logs
village
RepositoryEx
scala
test

Core


seaat-core
src
main
java
kr.kesti.seaat.core
codes
utils
CWW3Utils
LockUtils
TextParser
VillageUtils
Cell3D
ETLException
FileComponent
JobResult
PeriodComponent
ValueTimeComponent
resources
scala
test

Class 명명 규칙

- 각 Layer 의 기능을 접미사로 둔다. (PascalCase)
  - Controller: UserController, MapController
  - Service: UserService, MapService
  - DAO: UserDao, MapDao (UserRepository, MapRepository)
  - Model : User, Map (VO 를 붙일 필요 없다)
  - DTO : UserDTO, MapDTO …
  - Utility classes : ~Utils, ~Tool, ~s, ~Ex (StringUtils, Strings, StringEx)

Class 명명 규칙

- Design Patterns 을 구현한 경우, 그 명칭을 활용한다
  - ~Builder (Builder 패턴)
  - ~Decorator (Decorator 패턴)
  - ~Façade (Façade 패턴)
  - ~Provider (Provider 패턴)
  - ~Factory (Factory 패턴)
- Business 업무를 표현하는 방식
  - ~Aggregator (집계)
  - ~Extractor (추출)
  - ~Job (일반적인 작업을 Job, Task, Work 로 많이 표현한다)
  - ~Manager (작업들을 관리하는 기능)

Interface / Class 간 명명규칙

- Java의 대표적인 Interface 명
  - Serializable, Comparable (~ 할수 있다)
  - Iterator, Serializer (~ or - ~ 하는 자)
- Interface 를 구현하는 Class 는
  - 목적구분용 명칭 + Interface 명 + Impl
  - 예: BinarySerializerImpl

- 요즘은 꼭 이 규칙을 쓰지 않는다
  - Interface 의 첫자인 `I` 를 접두사로 써서 Interface 명명
  - ITimePeriod ← TimePeriod ← TimeBlock
  - Job ← ForecastJob (Impl 생략)

메소드 명명 규칙

- 영어식으로 표현 (camelCase)
  - 동사 + 목적어 형식
    - doJob(), executeJob(), makeUser(), newUser() ...
  - 전치사를 적극 활용해라
    - at / on / in / out, from~to, until, since, for, ago ...
    - up / down / on / off / over / under / through / into ...
    - by / with ....
- public 메소드가 호출하는 protected/private 메소드
    - public void execute()
    - protected void doExecute()
    - private void executeInternal()

메소드 명명 규칙

- 메소드 명에서 인자형식 및 반환 형식을 추측
  - public User **findByName**(String name) ;
  - public List<User> **findAllByNameMatch**(String nameToMatch);
  - public boolean **isActiveUser**(User user);
  - can~, exists~, is~ 등은 boolean 을 반환
  - check~, assert~, validate~ 는 void 형이지만, 내부에서 예외를 발생시킬 수 있다.

Field 명명 규칙

- 일반 변수는 camel case
  - private boolean running → public boolean isRunning()
  - private int age → public int getAge()
- 한번 설정되어 바꿀 수 없다면 final 을 지정해라
  - private final String companyName;
- 상수에 해당하는 것은 대문자와 `_` 를 조합
  - public static final int PAGE_SIZE = 10

Enum 명명 규칙

enum 명명 규칙은 기본적으로는 상수 명명규칙과 같다

다만 요즘은 Class 명명규칙을 사용하기도 한다

```
public enum SortDirection {
  /** 올림차순 */
  ASC,
  /** 내림차순 */
  DESC
}
```

```
public enum BinaryStringFormat {
  /** Base 64 Encoding */
  Base64,
  /** Hex decimal encoding */
  HexDecimal
}
```

코드 구현

발생한 예외를 무시할 경우에는 'ignored' 로 무시함을 표현해라

```
try {
  thread.join();
} catch (InterruptedException ignored) {
  log.debug("스레드 종료 대기 중에 interrupted exception 이 발생했습니다.");
}
```

switch 구문에서는 꼭 default: 를 이용하여, 처리되지 않는 case 가 없도록 한다.

```
switch (format) {
  case HexDecimal:
    return byteArrayToHexString(bytes);
  case Base64:
    return byteArrayToBase64String(bytes);
  default:
    throw new NotSupportedException("지원하지 않는 format 입니다. format=" + format);
}
```

코드 구현

static method 만 있는 static class 는 private 생성자를 정의하라

```
@Slf4j
public final class StreamEx {

  private StreamEx() {}

  public static IntArrayList asIntArrayList(@NonNull IntStream intStream) {
    return IntArrayList.newListWith(intStream.toArray());
  }
```

Abstract class 의 생성자는 protected 로 선언해라

상속받는 Concrete class 에서 public constructor 를 정의한다.

```
@Getter
public abstract class AbstractConfigElement {

  protected final Config config;

  protected AbstractConfigElement(@NonNull Config cfg) {
    this.config = cfg;
  }
}
```