# Clean Code

## Chapter 17. 냄새와 휴리스틱
## 부록 A. 동시성 II

# 17

# 냄새와 휴리스틱

# 주석

**부적절한 정보**

- 다른 시스템(소스 코드 관리 시스템 등)에 저장할 정보는 주석으로 부적절하다.
- 일반적으로 작성자, 최종 수정일, SPR(Software Problem Report)와 같은 메타 정보만 주석으로 넣는다.

**쓸모 없는 주석**

- 오래된 주석, 엉뚱한 주석, 잘못된 주석은 재빨리 삭제해야 한다.

**중복된 주석**

- 코드만으로 충분한데 구구절절 설명하는 주석은 중복이다.

# 주석

**성의 없는 주석**

- 주석을 작성할 것이라면 시간을 들여 최대한 멋지게 작성한다.

**주석 처리된 코드**

- 주석 처리된 코드는 코드 관리 시스템이 기억하기 때문에 삭제하라

# 환경

**여러 단계로 빌드해야 한다.**

- 빌드는 간단히 한 단계로 끝나야 한다.
- 한 명령으로 전체를 체크아웃해서 빌드할 수 있어야 한다.

**여러 단계로 테스트해야 한다.**

- 모든 단위 테스트는 한 명령으로 돌려야 한다.

# 함수

## 너무 많은 인수

- 함수에서 인수의 개수는 작을수록 좋다.
- 넷 이상의 인수는 최대한 피한다.

## 출력 인수

- 함수에서 뭔가의 상태를 변경해야 한다면 함수가 속한 객체의 상태를 변경한다.
- 예를 들어, ‘appendFooter(report)’ 대신 ‘report.appendFooter()’을 사용하자

## 플래그 인수

- boolean 인수는 함수가 여러 기능을 수행한다는 증거이다.
- 예를 들어, ‘render(true)’ 대신 ‘renderForSuite()’와 ‘renderForSigleTest()’로 함수를 나눠라

## 죽은 함수

- 아무도 호출하지 않는 함수는 삭제한다.
- 소스 코드 관리 시스템이 모두 기억하므로 걱정할 필요 없다.

# 일반

**한 소스 파일에 여러 언어를 사용한다.**

- 이상적으로 소스 파일 하나에 언어 하나만 사용하는 방식이 가장 좋다.
- 현실적으로는 불가피하지만 노력을 기울여 한 소스 파일에서 언어 수와 범위를 최대한 줄이자

**당연한 동작을 구현하지 않는다.**

- 함수나 클래스는 프로그래머가 당연하게 여길 만한 동작과 기능을 제공해야 한다.

**경계를 올바로 처리하지 않는다.**

- 모든 경계 조건을 찾아내고, 모든 경계 조건을 테스트하라

**안전 절차 무시**

- 컴파일러 경고, IDE 경고 등을 주의 깊게 보자
- 실패하는 테스트 케이스를 나중으로 미루지 말자

# 일반

**중복**

- 똑같은 코드가 여러 차례 나오는 중복은 함수로 교체한다.
- switch/case, if/else 문으로 똑같은 조건을 거듭 확인하는 조건은 다형성(polymorphism)으로 대체하라
- 알고리즘이 유사하나 코드가 서로 다른 경우 Template Method 패턴이나 Strategy 패턴으로 중복을 제거한다.

**추상화 수준이 올바르지 못한다.**

- 세부 구현과 관련한 상수, 변수, 유틸리티 함수는 파생 클래스에서 넣고, 고차원 개념은 기초 클래스에 넣는다.

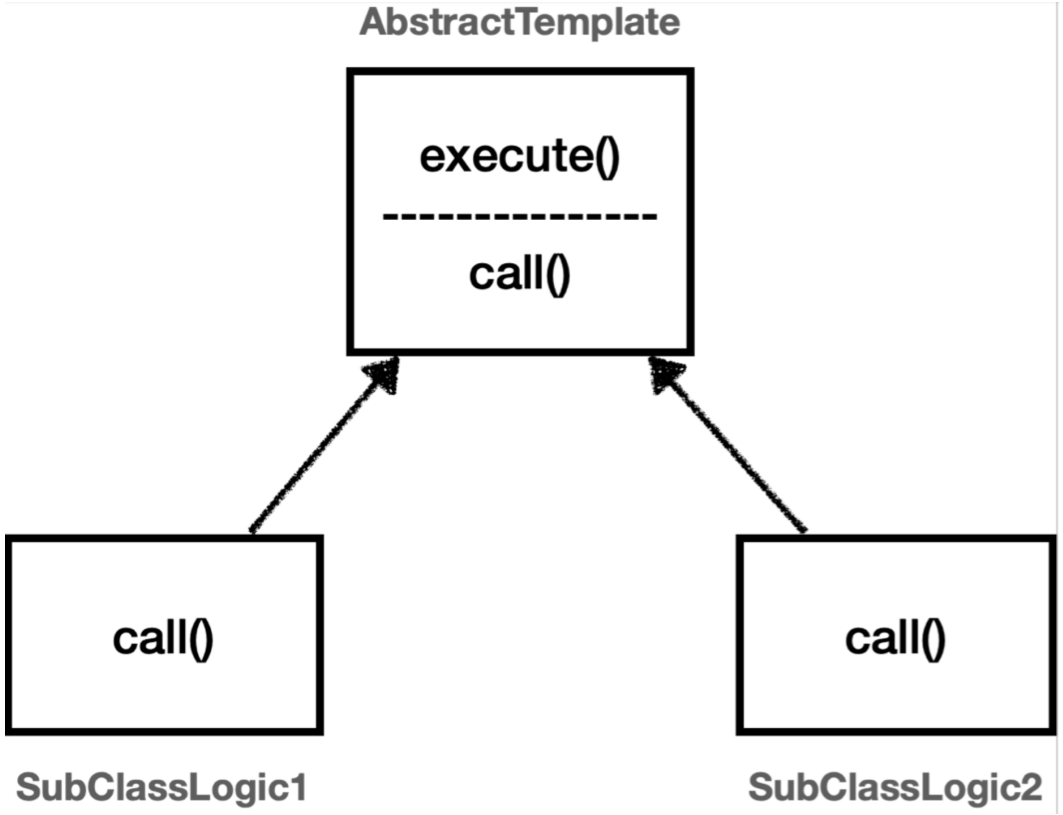
**기초 클래스가 파생 클래스에 의존한다.**

- 일반적으로 기초 클래스는 파생 클래스를 아예 몰라야 마땅하다.

# Template Method Pattern

변하지 않는 것은 추상클래스의 메서드로 선언

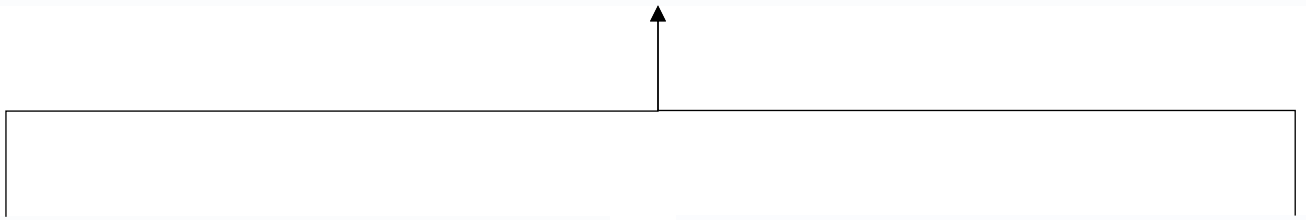
변하는 부분은 추상 메서드로 선언하여 자식 클래스가 오버라이딩




https://redcarrot1.github.io/posts/design_pattern_template_mthod_pattern/

# Template Method Pattern

```java
public abstract class AbstractTemplate {

    public void execute() {
        System.out.println("템플릿 시작");
        //변해야 하는 로직 시작
        logic();
        //변해야 하는 로직 종료
        System.out.println("템플릿 종료");
    }

    protected abstract void logic(); //변경 가능성이 있는 부분은 추상 메소드로 선언한다.
}
```



```java
public class SubClassLogic1 extends AbstractTemplate {
    @Override
    protected void logic() {
        System.out.println("변해야 하는 메서드는 이렇게 오버라이딩으로 사용1.");
    }
}
```

```java
public class SubClassLogic2 extends AbstractTemplate {
    @Override
    protected void logic() {
        System.out.println("변해야 하는 메서드는 이렇게 오버라이딩으로 사용2.");
    }
}
```

# Template Method Pattern

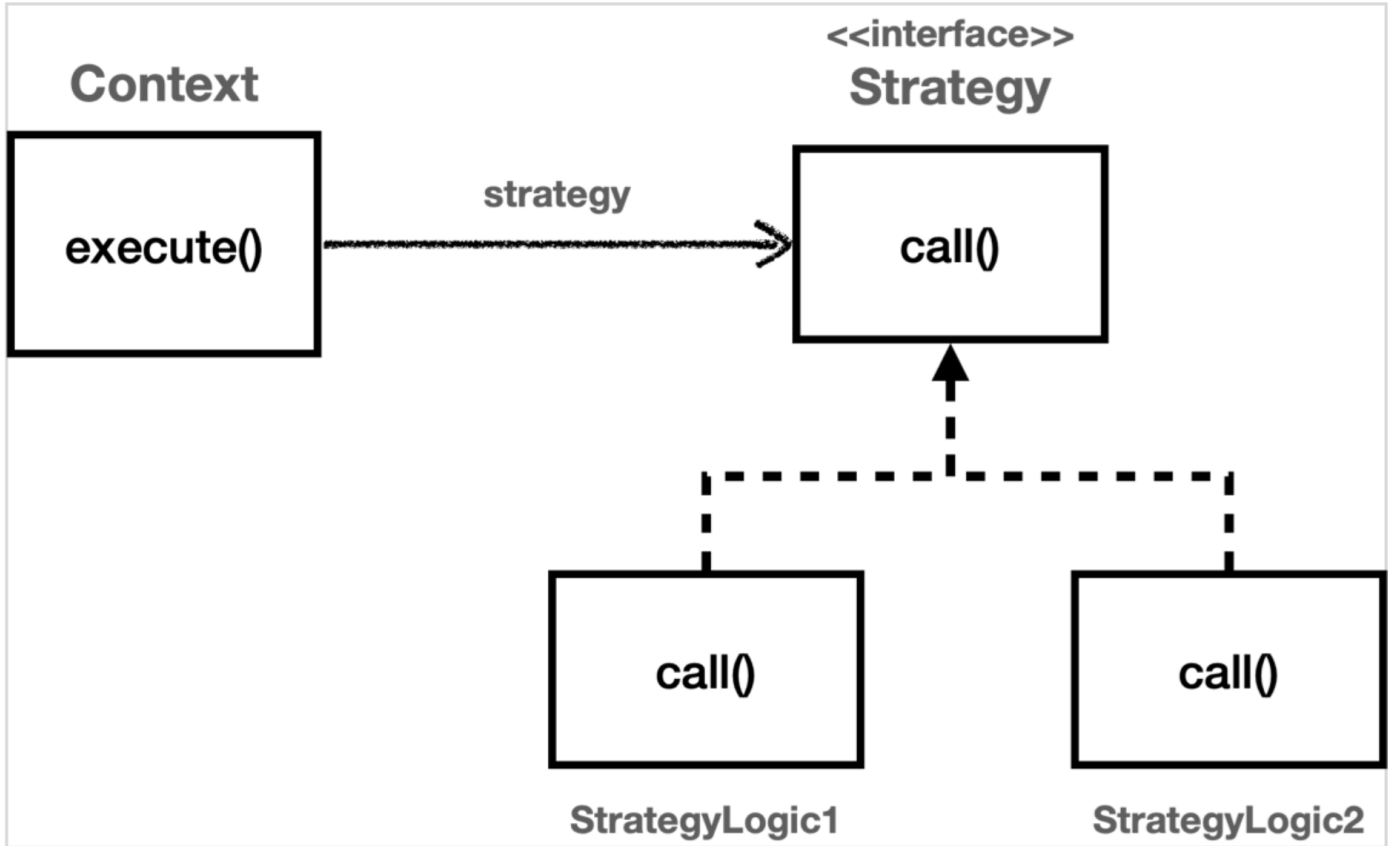
```java
public class templateMethod1 extends AbstractTemplate {
    public static void main(String[] args) {
        AbstractTemplate template1 = new SubClassLogic1();
        template1.execute();

        System.out.println();

        AbstractTemplate template2 = new SubClassLogic2();
        template2.execute();
    }
}
```

```
//출력
템플릿 시작
변해야 하는 메서드는 이렇게 오버라이딩으로 사용1.
템플릿 종료

템플릿 시작
변해야 하는 메서드는 이렇게 오버라이딩으로 사용2.
템플릿 종료
```

# Strategy Pattern

일반 클래스(Context)에 변하지 않는 부분을, 변하는 부분은 인터페이스(Strategy)에 선언

사용시 원하는 로직의 구현체를 만든다.




https://redcarrot1.github.io/posts/design_pattern_strategy_pattern/

# Strategy Pattern

```java
public class Context {
    private Strategy strategy;

    public ContextV1(Strategy strategy) {
        this.strategy = strategy;
    }

    public void execute(){
        log.info("템플릿 시작");

        //변해야 하는 로직 시작
        strategy.call(); //위임
        //변해야 하는 로직 끝

        log.info("템플릿 종료");
    }
}
```

```java
public interface Strategy {
    void call();
}
```

```java
public class StrategyLogic1 implements Strategy{
    @Override
    public void call() {
        log.info("비즈니스 로직1 실행");
    }
}
```

```java
public class StrategyLogic2 implements Strategy{
    @Override
    public void call() {
        log.info("비즈니스 로직2 실행");
    }
}
```

# Strategy Pattern

```
void strategyV1() {
    Strategy strategyLogic1 = new StrategyLogic1();
    ContextV1 context1 = new ContextV1(strategyLogic1);
    context1.execute();

    Strategy strategyLogic2 = new StrategyLogic2();
    ContextV1 context2 = new ContextV1(strategyLogic2);
    context2.execute();
}

//출력
템플릿 시작
비즈니스 로직1 실행
템플릿 종료

템플릿 시작
비즈니스 로직2 실행
템플릿 종료
```

# Strategy Pattern Ver 2. 콜백 패턴

수정 전

```
void strategyV1() {
    Strategy strategyLogic1 = new StrategyLogic1();
    ContextV1 context1 = new ContextV1(strategyLogic1);
    context1.execute();

    Strategy strategyLogic2 = new StrategyLogic2();
    ContextV1 context2 = new ContextV1(strategyLogic2);
    context2.execute();
}
```

```
public class Context {
    public void execute(Strategy strategy){
        log.info("템플릿 시작");

        //변해야 하는 로직 시작
        strategy.call(); //위임
        //변해야 하는 로직 끝

        log.info("템플릿 종료");
    }

}
```

수정 후

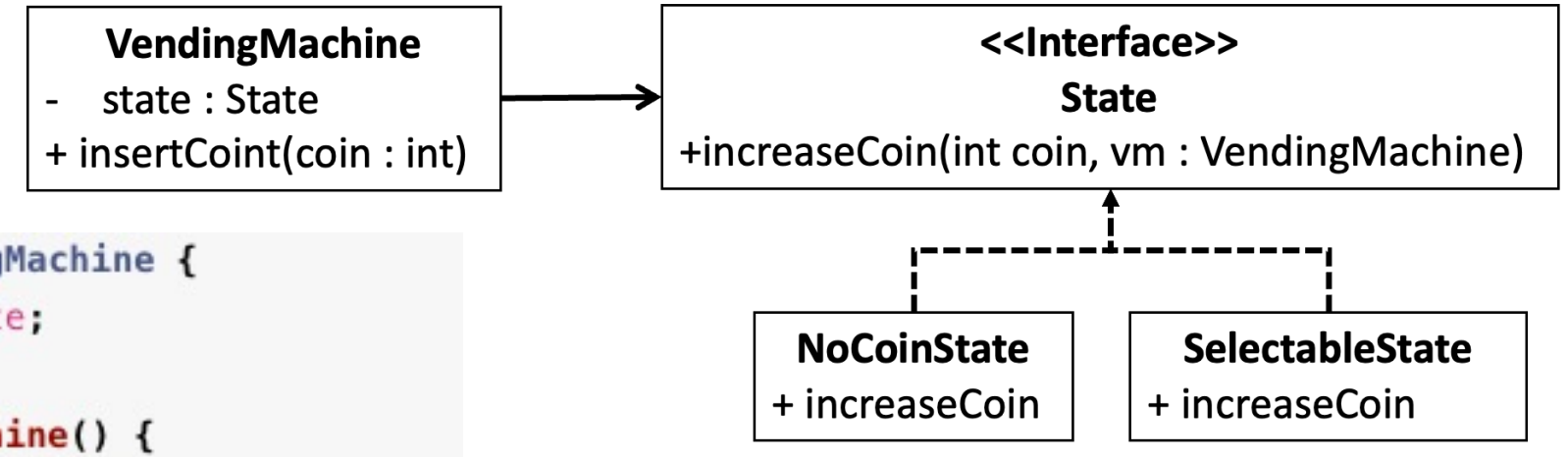
```
void strategyV1() {
    ContextV2 context = new ContextV2();
    context.execute(new StrategyLogic1());
    context.execute(new StrategyLogic2());
}
```

# State Pattern

```
public class VendingMachine {
  public static enum State { NOCOIN, SELECTABLE }

  private State state = State.NOCOIN;

  public void insertCoin(int coin) {
    switch (state) {
      case NOCOIN:
        // Do Something
      case SELECTABLE:
        // Do Something
    }
  }

  public void changeState(State state) {
    this.state = state;
  }
}
```

https://redcarrot1.github.io/posts/design_pattern_state_pattern/

# State Pattern




```java
public class VendingMachine {
  private State state;

  public VendingMachine() {
    state = new NoCoinState();
  }

  public void insertCoin(int coin) {
    // 상태 객체에게 코드 구현을 위임
    state.increaseCoin(coin, this);
  }

  public void changeState(State state) {
    this.state = state;
  }
}
```

```java
public interface State {
  public void increaseCoin(int coin, VendingMachine vm);
}


public class NoCoinState implements State {
  @Override
  public void increaseCoin(int coin, VendingMachine vm) {
    // Do Something

    // Change State
    vm.changeState(new SelectableState());
  }
}
```

# 일반

**과도한 정보**

- 잘 정의된 인터페이스는 많은 함수를 제공하지 않고, 결합도가 낮다.
- 클래스가 제공하는 메서드 수는 작을수록 좋고, 인스턴스 분수 수도 작을수록 좋다.
- 함수가 아는 변수 수도 작을수록 좋다.

**죽은 코드**

- 불가능한 조건을 확인하는 if 문과 throw 문이 없는 try 문에서 catch 블록은 죽은 코드이다.

**수직 분리**

- 변수와 함수는 사용되는 위치에 가깝게 정의한다.
- 지역 변수는 처음으로 사용하기 직전에 선언하며 수직으로 가까운 곳에 위치해야 한다.
- private 함수는 처음으로 호출한 직후에 정의한다.

# 일반

**일관성 부족**
- 예를 들어, 한 함수에서 `response`라는 변수에 `HttpServletResponse` 인스턴스를 저장했다면, 다른 함수에서도 일관성 있게 동일한 변수 이름을 사용한다.

**잡동사니**
- 아무도 사용하지 않는 변수와 함수, 정보를 제공하지 못하는 주석은 제거해라

**인위적 결합**
- 서로 무관한 개념을 인위적으로 결합하지 않는다.
- 예를 들어, 일반적인 `enum`은 특정 클래스에 속할 이유가 없고, 범용 `static` 함수도 특정 클래스에 속할 이유가 없다.

# 일반

## 기능 욕심

- 메서드가 다른 객체의 getter, setter를 사용해 그 객체 내용을 조작한다면 클래스 범위를 욕심내는 탓이다.
- 하지만 때로는 어쩔 수 없는 경우도 생긴다.

## 선택자 인수

- boolean, enum, int 인수로 함수 동작을 제어하는 것을 바람직하지 않다.
- 대신 새로운 함수를 만드는 편이 좋다.

## 모호한 의도

- 코드를 짤 때는 의도를 최대한 분명히 밝힌다.

# 일반

**잘못 지운 책임**

- 코드는 독자가 자연스럽게 기대할 위치에 배치한다.

**부적절한 static 함수**

- 함수를 재정의할 가능성이 전혀 없는 경우, 메서드를 소유하는 객체에서 가져오는 정보가 아닌 경우에 static 함수를 사용한다.
- 일반적으로 인스턴스 함수가 더 좋으므로 조금이라도 의심스럽다면 인스턴스 함수로 정의한다.

**서술적 변수**

- 계산을 몇 단계로 나누고 중간값에 좋은 변수 이름만 붙여도 읽기 쉬운 모듈로 바뀐다.

# 일반

```
Matcher match = hedaerPattern.matcher(line);
if(match.find()) {
  headers.put(match.group(1).toLowerCase(), match.group(2));
}
```

```
Matcher match = hedaerPattern.matcher(line);
if(match.find()){
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

# 일반

**이름과 기능이 일치하는 함수**

- 'Date newDate = date.add(5)'는 5시간, 5주, 5일 중에 무엇을 더하는 함수인 지 알 수 없다.

**알고리즘을 이해하라**

- 구현이 끝났다고 선언하기 전에 함수가 돌아가는 방식을 확실히 이해하는지 확인하라

**논리적 의존성은 물리적으로 드러내라**

- 의존하는 모든 정보를 명시적으로 요청하는편이 좋다.

**if/else 혹은 switch/case 문보다 다형성을 사용하라**

**표준 표기법을 따르라**

- 팀이 정한 표준은 팀원들 모두가 따라야 한다.

# 일반

## 매직 숫자는 명명된 상수로 교체하라

## 정확하라

- 코드에서 뭔가를 결정할 때는 정확히 결정한다.
- 예를 들어, null을 반환할 가능성이 있는지, 조회 결과의 개수가 예측값과 동일한지, 동시성으로 인한 문제가 있는지 등을 정확히 파악해야 한다.

## 관례보다 구조를 사용하라

- 설계 결정을 강제할 때는 규칙보다 관례, 특히 구조 자체를 강제하면 더 좋다.
- 예를 들면, 추상 메서드를 사용하여 반드시 모두 구현하도록 강제할 수 있다.

# 일반

## 조건을 캡슐화하라

- 부울 논리는 조건의 의도를 분명히 밝히는 함수로 표현하라
- 예를 들어, ‘if (timer.hasExpired() && !timer.isRecurrent())’보다
  ‘if (shouldBeDeleted(timer))’가 더 좋다.

## 부정 조건은 피하라

- 부정 조건은 긍정 조건보다 이해하기 어려우므로, 가능하면 긍정 조건을 표현하라
- 예를 들어, ‘if (!buffer.shouldNotCompat())’보다
  ‘if (buffer.shouldCompat())’가 더 좋다.

# 일반

## 함수는 한 가지만 해야 한다.

```
public void pay() {
    for (Employee e : employees) {
      if (e.isPayday()) {
        Money pay = e.calculatePay();
        e.deliverPay(pay);
      }
    }
  }
```

```
public void pay() {
    for (Employee e : employees) {
      payIfNecessary(e);
    }
  }

  private void payIfNecessary(Employee e) {
    if (e.isPayday()) {
      calculateAndDeliverPay(e);
    }
  }

  private void calculateAndDeliverPay(Employee e) {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
  }
```

# 일반

## 숨겨진 시간적인 결합

```
public class MoogDirver {
    Gradient gradient;
    List<Spline> splines;

    public void dive(String reason) {
      saturateGradient();
      reticulateSplines();
      diveForMoog(reason);
    }
  }
```

```
public class MoogDirver {
    Gradient gradient;
    List<Spline> splines;

    public void dive(String reason) {
      Gradient gradient = saturateGradient();
      List<Spline> splines = reticulateSplines(gradient);
      diveForMoog(splines, reason);
    }
  }
```

# 일반

**일관성을 유지하라**

**경계 조건을 캡슐화하라**
- 경계 조건은 한 곳에서 별도로 처리한다.

**함수는 추상화 수준을 한 단계만 내려가야 한다.**
- 또한 함수 내 모든 문장은 추상화 수준이 동일해야 한다.

**설정 정보는 최상위 단계에 둬라**

**추이적 탐색을 피하라**
- 디미터의 법칙 (‘a.getB().getC().doSomethig()’)
- 클래스에서 사용하는 모듈은 해당 클래스에 필요한 서비스를 모두 제공해야 한다.
- 원하는 메서드를 찾느라 객체 그래프를 따라 시스템을 탐색할 필요가 없어야 한다.

# 자바

**긴 import 목록을 피하고 와일드카드를 사용하라**

**상수는 상속하지 않는다.**
- 대신 static import를 사용하라

**상수 대 Enum**
- 자바 5부터 enum을 제공하므로 적극 활용하자

# 이름

**서술적인 이름을 사용하라**
- 이름을 성급하지 정하지 않고, 서술적인 이름을 신중하게 고른다.

**적절한 추상화 수준에서 이름을 선택하라**
- 구현을 드러내는 이름은 피하라
- 작업 대상 클래스나 함수가 위치하는 추상화 수준을 반영하는 이름을 선택하라

**가능하다면 표준 명명법을 사용하라**
- 디자인 패턴, toString 등은 가능하면 관례를 따르는 편이 좋다.

# 이름

**명확한 이름**
- 함수나 변수의 목적을 명확히 밝히는 이름을 선택한다.

**긴 범위는 긴 이름을 사용하라**
- 이름의 길이는 범위 길이에 비례해야 한다.

**인코딩을 피하라**

**이름으로 부수 효과(side-effect)를 설명하라**
- 함수, 변수, 클래스가 하는 일을 모두 기술하는 이름을 사용한다.

# 테스트

불충분한 테스트
커버리지 도구를 사용하라
사소한 테스트를 건너뛰지 마라
무시한 테스트는 모호함을 뜻한다
경계 조건을 테스트하라
버그 주변은 철저히 테스트하라
실패 패턴을 살펴라
테스트 커버리지 패턴을 살펴라
테스트는 빨라야 한다.

# 부록A

## 동시성 II

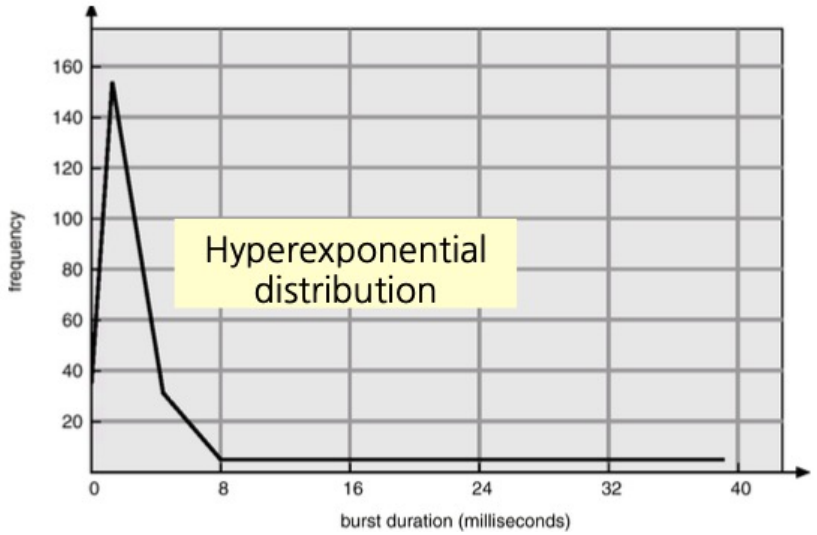
# 프로세스 작업

I/O Bound : 소켓 사용, 데이터베이스 연결, 가상 메모리 스와핑 등

동시성(concurrent)이 성능을 높여줄 수 있다.
따라서 쓰레드를 추가하여 성능 향상을 이룰 수 있다.

CPU Bound : 수치 계산, 정규 표현식 처리, 가비지 컬렉션 등

새로운 하드웨어를 추가해 성능을 높여야 한다.
쓰레드를 늘린다고 무조건 빨라지지 않는다.(CPU 사이클은 한계가 있기 때문이다.)



# 다중 쓰레드 프로그램

다중 쓰레드 프로그램을 깨끗하게 유지하려면 잘 통제된 곳으로
쓰레드 관리를 모아야 한다.

쓰레드를 관리하는 코드는 쓰레드만 관리해야 한다.(SRP)

동시성은 그 자체가 복잡한 문제이므로 SRP가 특히 더 중요하다.

# Race Condition

바이트 코드 N개를 쓰레드 T개로 실행할 때, 발생 가능한 순열 수 $\frac{(NT)!}{N!^T}$

$\underbrace{11\cdots1}_{N개}$ $22\cdots2$ $33\cdots3$ $\cdots$ $TT\cdots T$

자바 코드 ‘return ++id’ 는 바이트 코드 8개로 이루어짐
쓰레드 2개가 동시 실행한다면, 발생 가능한 순열은 12,870개
long 타입이라면, 발생 가능한 순열은 2,704,156개

자바 : synchronized 키워드, Executor 프레임워크, AtomicInteger 클래스

# DeadLock

**발생 조건 4가지**

**1. 상호 배제(Mutual exclusion)**
- 여러 쓰레드가 동시에 사용하지 못하며, 개수가 제한적이다.
- 데이터베이스 연결, 쓰기용 파일 열기, 레코드 락, 세마포어 등의 자원

**2. 잠금 & 대기(Lock & Wait)**
- 일단 쓰레드가 자원을 점유하면 필요한 나머지 자원까지 모두 점유해 작업을 마칠 때까지 이미 점유한 자원을 내놓지 않는다.

**3. 선점 불가(No Preemption)**
- 한 쓰레드가 다른 쓰레드로부터 자원을 빼앗지 못한다.

**4. 순환 대기(Circular Wait)**
- 각 쓰레드가 상대방이 가진 자원을 대기한다.

# DeadLock Prevent

**1. 상호 배제(Mutual exclusion) – 어렵다.**
- 동시에 사용가능한 자원을 사용하거나, 쓰레드 수 이상으로 자원 수를 늘린다.

**2. 잠금 & 대기(Lock & Wait)**
- 각 자원을 점유하기 전에 점유할 수 있는지 확인한다.
- 만약 어느 하나라도 점유하지 못한다면 지금까지 점유한 자원을 몽땅 내놓고 처음부터 다시 시작한다. (Starvation, Livelock 문제 발생)

**3. 선점 불가(No Preemption)**
- 필요한 자원이 잠겼다면 자원을 소유한 쓰레드에게 풀어달라 요청한다.

**4. 순환 대기(Circular Wait) – 가장 흔한 전략**
- 모든 쓰레드에 공통적으로 자원 획득 순서를 정해둔다.
- 자원을 할당하는 순서와 사용하는 순서가 다를수도 있다.(필요한 이상으로 오랫동안 점유)