Exception

Exception이란?

Exception은 Error의 일종이며 프로그램이 수행 시 또는 컴파일 시에 불능 상태를 만든다.

예외처리란?

Exception이 발생할 것을 대비하여 미리 예측해 이를 소스 상에서 제어하고 처리하도록 만드는 것.
갑작스러운 Exception이 발생하여도 시스템 및 프로그램이 불능 상태가 되지 않고 정상 실행 상태를 유지할 수 있다.

Exception vs Error

Exception	Error
프로그램적으로 처리 가능 프로그램에 에러가 없고, 문법적으로 문제가 없음에도 발생 프로그램의 실행 도중 발생 일반적으로 실행 중 외부적인 요인에 의해 발생 프로그램을 비정상적으로 종료시키지만, 미리 예측하여 처리 가능	JVM에 의존하여 처리 Error 클래스는 try~catch 문으로 처리 불가 프로그램 내에서 해결이 불가능한 치명적인 예외 상황을 알리는 예외 클래스의 정의에 사용 시스템 레벨에서 프로그램에 심각한 문제를 야기하여 프로그램을 종료 미리 예측하여 처리할 수 없는 것이 대부분
런타임 에러 파일 읽기/쓰기 오류 네트워크 전송 오류	컴파일 에러 문법의 오류

자바에서의 예외 처리 방법

1. try - catch - finally

```
try{
            //예외가 발생될만한 코드
}catch(FileNotFoundException e){
            //FileNotFoundException이 발생했다면
}catch(IOException e){
            //IOException이 발생했다면
}catch(Exception e){
            // Exception이 발생했다면
}finally{
```

```
            // 예외 발생 여부와 상관 없이 무조건 실행
}
```

try 블록	예외가 발생할 수 있는 코드를 작성한다.
catch(“예외 종류") 블록	“예외 종류"가 발생했을 때 처리하는 동작을 명시한다. 맨 처음 catch 블록에서 잡히지 않은 예외라면 다음 catch의 예외를 검사 상속 관계에 있는 예외 중 부모가 앞선 catch 문에 위치해야 한다.
finally 블록	예외의 발생 여부와 상관 없이 항상 수행되어야할 코드 작성 임시 파일의 삭제, 할당된 객체의 free 등이 이루어진다.

```
try{
      //.. 중략 ..//
} catch (Exception e){
      //컴파일 오류 발생
} catch (IOException e){

}
```

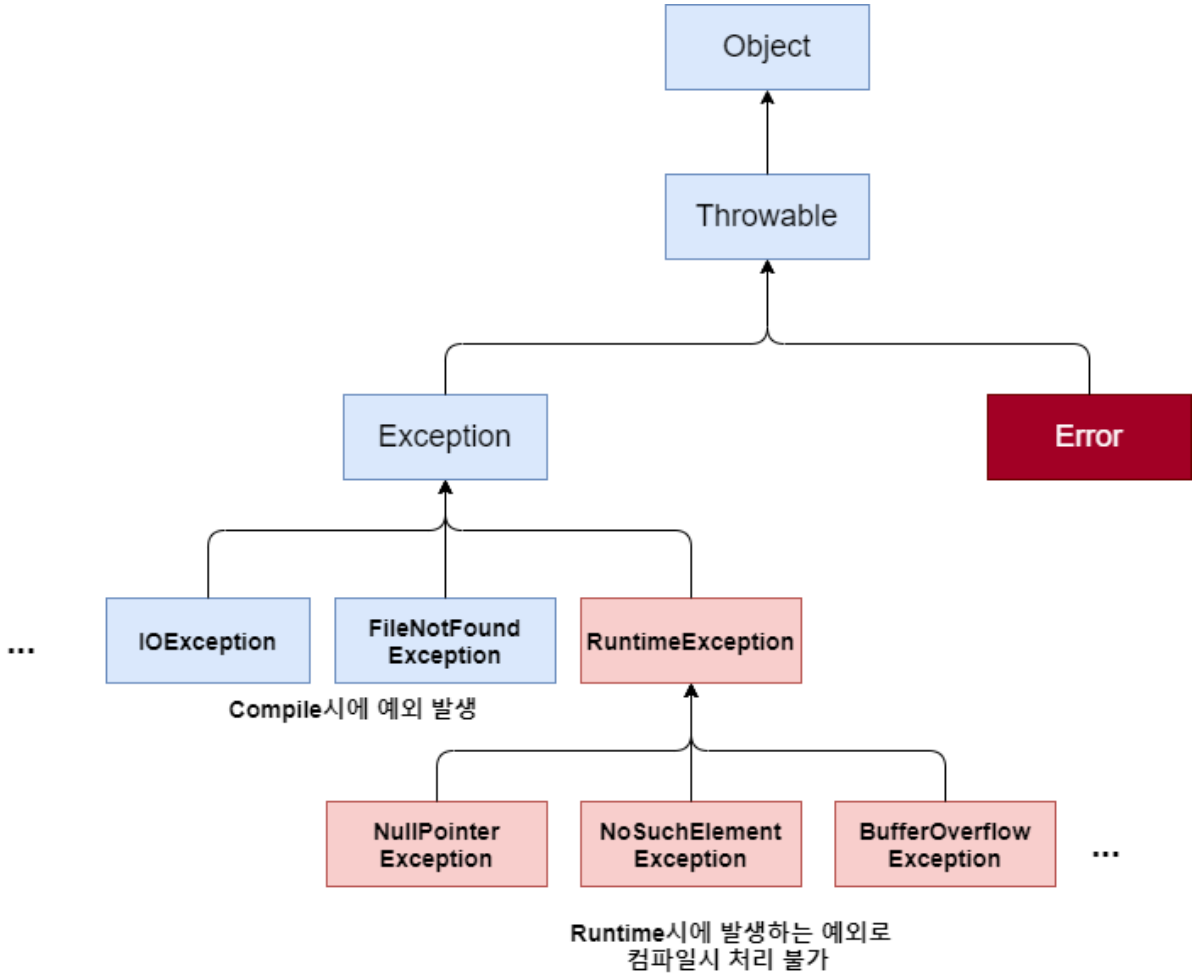
2. throw, throws

```
public static void divide(int a,int b) throws ArithmeticException {
      if(b==0) throw new ArithmeticException("0으로 나눌 수 없습니다.");
      int c=a/b;
      System.out.println(c);
}
public static void main(String[] ar){
      int a=10;
      int b=0;

      divide(a,b);
}
```

throws를 사용하여 예외를 해당 구현 코드에서 처리하지 않고, 그 구현 코드를 호출해 사용하는 부분에서 예외 처리를 할 수 있다.
divide() 메소드는 a와 b를 나눈 후에 출력하는 역할을 한다. 나누기 부분에서 b가 0일때, 예외가 발생할 수 있다.
이를 위해 divide() 메소드에 ArithmeticException을 추가해 divide() 안에서 발생한 예외를 main에 던져줌으로 예외 처리를 할 수 있다.
throws된 ArithmeticException과 같은 Exception을 받은 main에서는 try-catch를 이용해 예외 처리를 할 수 있다.

예외 계층 구조



RuntimeException vs non RuntimeException

RuntimeException : unCheckedException이라고도 부른다. 예외 처리를 강제하지 않기 때문에 개발자 스스로의 판단으로 예외 처리 코드를 작성해야한다.
non RuntimeException : CheckedException이라고 부른다. 반드시 예외처리를 해야하며, 하지 않을 시 컴파일 시점에서 예외 처리를 하라는 에러가 발생한다.

Custom Exception

4가지 Best Practice

1. Always Provide a Benefit(항상 혜택을 제공하라)

커스텀 예외의 의도는 자바의 표준 예외들로 표현할 수 없는 정보나 기능을 제공하는 것이다.

만약 위의 의도가 없다면, JDK가 제공하는 예외들과 비교했을 때 커스텀 예외는 어떠한 혜택도 제공하지 못 한다.

2. Follow the Naming Convention(네이밍 컨벤션을 따라가라)

JDK가 제공하는 예외 클래스들의 이름은 모두 'Exception'으로 끝난다.
일반적인 네이밍 규칙으로 자바 생태계 전체에 적용되는 규칙이다.

3. Provide Javadoc Comments For Your Exception Class(예외 클래스에 대한 Java doc 주석 제공)

기본적으로 API의 모든 클래스, 멤버변수, 생성자들에 대해서는 문서화하는 것이 일반적인 Best Practice다.
문서화 되지 않은 API들은 매우 사용하기가 어렵다.
클라이언트와 직접 관련된 메소드가 예외를 던지면 그 예외는 바로 API의 일부가 된다.
이는 잘 만들어진 Java doc이 필요하다는 의미이다.
커스텀 예외의 java doc에는 예외가 발생할 수도 있는 상황과 예외의 일반적인 의미를 기술한다. 이는 다른 개발자들이 해당 API를 이해하도록 하고 예외 상황을 피하도록 돕기 위함이다.

4. Provider Constructor That Sets the Cause

Exception과 RuntimeException은 에외의 원인을 기술하고 있는 Throwable을 받을 수 있는 생성자 메소드를 제공한다. 생성자를 통해 예외의 원인을 인자로 받는 것이 좋다.

```
public void wrapException(String input) throws MyBusinessException {
    try {
        // do something
    } catch (NumberFormatException e) {
        throw new MyBusinessException("A message that describes the
error.", e, ErrorCode.INVALID_PORT_CONFIGURATION);
    }
}

public class MyBusinessException extends Exception {
    public MyBusinessException(String message, Throwable cause,
ErrorCode code) {
            super(message, cause);
            this.code = code;
        }
        ...
}
```

4가지 Best Practice를 적용시킨 CheckedException

```
public class CustomCheckedExceptionEx extends Exception {

    public CustomCheckedExceptionEx() {
```

```
    }

    public CustomCheckedExceptionEx(String message) {
        super(message);
    }

    public CustomCheckedExceptionEx(String message, Throwable cause) {
        super(message, cause);
    }

    public CustomCheckedExceptionEx(Throwable cause) {
        super(cause);
    }

    public CustomCheckedExceptionEx(String message, Throwable cause,
boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}

class CustomCheckedExceptionTest {
    public static void main(String[] args) throws
CustomCheckedExceptionEx {
        try {
            // doSomething
        } catch (Exception e) {
            throw new CustomCheckedExceptionEx("커스텀 체크드 예외
발생!!", e);
        }
    }
}
```

4가지 Best Practice를 적용한 UnCheckedException

```
public class CustomUncheckedException extends RuntimeException {

    public CustomUncheckedException() {
    }

    public CustomUncheckedException(String message) {
        super(message);
    }

    public CustomUncheckedException(String message, Throwable cause) {
        super(message, cause);
    }
```

```
    public CustomUncheckedException(Throwable cause) {
        super(cause);
    }

    public CustomUncheckedException(String message, Throwable cause,
boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}

class CustomUncheckedExceptionTest {
    public static void main(String[] args) {
        try {
            // doSomething
        } catch (RuntimeException e) {
            throw new CustomUncheckedException("커스텀 언체크드 예외
발생!!", e);
        }
    }
}
```

멀티스레드

Thread Class vs Runnable Interface

Thread Class를 이용한 Thread 생성

```
public class WorkerThread extends Thread {
  @Override
  public void run() {
      //스레드가 실행할 코드
  }
}

Thread thread = new Thread() {
  public void run() {
      //스레드가 실행할 코드
  }
}

thread.start()
```

Runnable Interface를 이용한 Thread 생성

```
// Runnable 구현 객체를 생성한 후, 이것을 매개값으로 Thread 생성자를 호출
Runnable task = new Task();

Thread thread = new Thread(task);

// Thread 생성자를 호출할 때 Runnable 익명 객체를 매개값으로 사용할 수 있다.
이 방법이 더 많이 사용된다.
Thread thread = new Thread( new Runnable() {
  public void run() {
    쓰레드가 실행할 코드;
  }
});

// 람다식을 매개값으로 사용하는 방법
Thread thread = new Thread( () -> {
    쓰레드가 실행할 코드;
});
```

Thread Class vs Runnable Interface

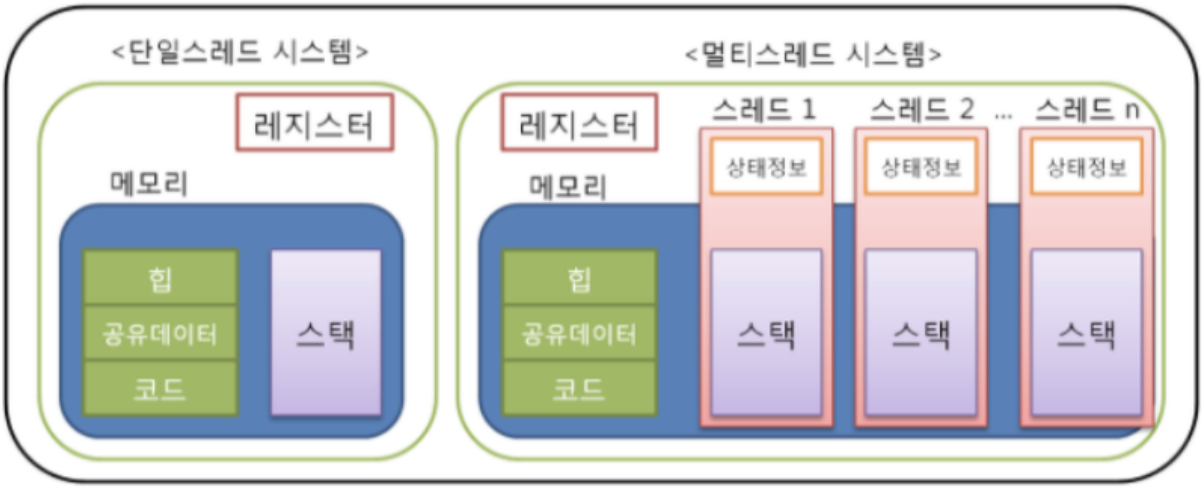
Thread Class를 상속하여 Thread를 구현할 경우 상속에 제한을 받는다.
주로 Runnable Interface를 사용하여 구현하고 필요에 따라 부모 클래스를 상속 받아 구현한다.
Thread Class 또한 Runnable interface의 구현 클래스이므로 run 메소드를 오버라이딩해야 한다.

Multi-Thread vs Multi-Process

Thread : Process 내에서 실제로 작업을 수행하는 주체
Process : 프로그램이 메모리에 적재되어 사용할 수 있는 상태, 실행 중인 프로그램

Multi-Thread

- 하나의 프로세스에 여러 스레드로 자원을 공유하며 작업을 나누어 수행



장점

- 스택과 상태정보를 제외한 메모리 공유를 통한 시스템 자원 소모 감소(자원의 효율성 증대)
- 프로세스를 생성하여 자원을 할당하는 시스템 콜이 줄어 자원을 효율적으로 관리
- 시스템 처리율 향상(처리 비용 감소)
- 스레드 간 데이터를 주고 받는 것이 간단하고 빠름(heap 영역 공유)
- Context Switching이 빠름(캐시 메모리를 비울 필요가 없음)

단점

- 자원을 공유하기에 동기화 문제 발생(병목 현상, 데드락 등)
- 디버깅이 어려움
- 하나의 스레드에 문제가 생길 경우 전체 프로세스 영향
- 단일 프로세스 시스템의 경우 효과를 기대하기 어려움

Multi-Process

- 다수의 프로세서(CPU)가 협력적으로 하나 이상의 작업(Task)를 동시에 처리(병렬 처리)
- 각 프로세스 간 메모리 구분이 필요하거나 독립된 주소 공간을 가져야 할 경우 사용

장점

- 독립된 구조로 안전성이 높음
- 프로세스 중 하나에 문제가 생겨도 다른 프로세스에 영향을 주지 않음
- 다수의 프로세스가 처리되어야 할 때 동일한 데이터를 사용하고, 해당 데이터를 하나의 디스크에 두고 모든 프로세서(CPU)가 공유하면 비용 절감의 효과 기대

단점

- 독립된 메모리 영역을 가지기 때문에 작업량이 많을 수록 Context Switching이 많이 일어나 오버헤드 발생 -> 성능 저하
- Context Switching 과정에서 캐시 메모리 초기화 등 무거운 작업이 진행되고 시간이 소모 되는 등 오버헤드 발생

Context Switching?

- CPU는 한번에 하나의 프로세스만 실행 가능
- CPU에서 여러 프로세스를 돌아가면서 작업을 처리하는 과정
- 동작 중인 프로세스가 대기하면서 해당 프로세스의 상태(Context)를 보관하고, 대기하고 있던 다음 순서의 프로세스가 동작하면서 이전에 보관했던 프로세스의 상태를 복구하는 작업

Multi-Thread vs Multi-Process

- 멀티 스레드는 멀티 프로세스보다 작은 메모리 공간을 차지하고 Context Switching이 빠르지만, 동기화 문제와 하나의 스레드 장애로 전체 스레드가 영향을 받을 위험을 가짐
- 멀티 프로세스는 하나의 프로세스가 장애가 발생해도 다른 프로세스에 영향을 주지 않아 안정성이 높지만, 멀티 스레드보다 많은 메모리 공간과 CPU 시간을 차지

동시성과 병렬성

동시성, Concurrency

멀티 작업을 위해 하나의 코어에서 멀티 스레드가 번갈아 가며 실행하는 성질

병렬성, Parallelsim

멀티 작업을 위해 멀티 코어에서 개별 스레드를 동시에 실행하는 성질

※ 싱클 코어에서의 멀티 쓰레드 작업은 병렬적으로 실행되는 것처럼 보이지만, 사실은 번갈아가며 실행하는 동시성 작업이다. 번갈아 실행하는 것이 워낙 빨라서 병렬성으로 보일 뿐이다.

동기화

싱글 스레드 프로그램에서는 한개의 스레드가 객체를 독차지해서 사용하면 되지만, 멀티 스레드 프로그램에서는 스레드들이 객체를 공유해서 작업해야 하는 경우가 발생한다.
이 경우, 스레드 A를 사용하던 객체가 스레드 B에 의해 상태가 변경될 수 있기 때문에 스레드 A가 의도했던 것과는 다른 결과를 산출할 수 있다.

임계 영역(critical section)

멀티 스레드 프로그램에서 단 하나의 스레드만 실행할 수 있는 코드 영역
JAVA에서는 synchronized 메소드와 블록을 제공한다.

동기화 메소드

동기화 메소드는 메소드 전체 내용이 임계 영역이므로 스레드가 동기화 메소드를 실행하는 즉시 객체에는 잠금이 일어나고, 스레드가 동기화 메소드를 종료하면 잠금이 풀린다,

```
public synchronized void method() {
  임계 영역; //단 하나의 스레드만 실행
}
```

동기화 블록

동기화 블록의 외부 코드들은 여러 스레드가 동시에 실행할 수 있지만, 동기화 블록의 내부 코드는 임계 영역이므로 한번에 한 스레드만 실행할 수 있고 다른 스레드는 실행할 수 없다.
만약 동기화 스메소드와 동기화 블록이 여러개 있을 경우, 스레드가 이들 중 하나를 실행할 때 다른 스레드는 해당 메소드는 물론이고 다른 동기화 메소드 및 블록도 실행할 수 없다.
단, 일반 메소드는 실행이 가능하다.

```
public void method () {
  //여러 스레드가 실행 가능한 영역
  ...

  synchronized(공유객체) {
    임계 영역; //단 하나의 스레드만 실행
  }

  //여러 스레드가 실행 가능한 영역
  ...
}
```

데드락

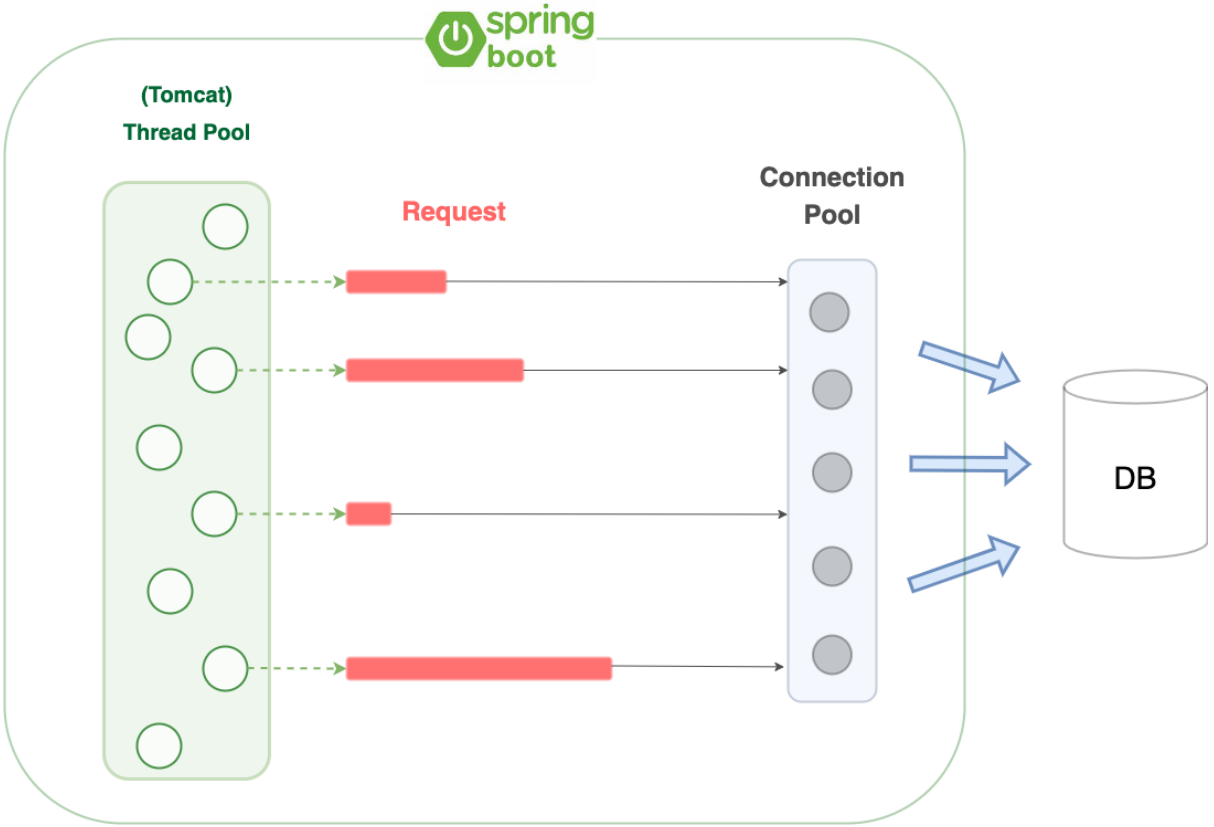
멀티 스레드 프로그램에서 동기화를 통해 락을 획득하여 동일한 자원을 여러 곳에서 사용하지 못 하도록 하였을 때, 서로 다른 스레드가 서로가 가지고 있는 락이 해제되기를 기다리는 상태가 생길 수 있으며 이러한 상태를 교착상태(Deadlock)라 한다.
교착 상태는 어떤 작업도 실행되지 못하고 서로 상대방의 작업이 끝나기만 바라는 무한정 대기 상태이다.

발생 조건

- 상호 배제(Mutual Exclusion) : 한 자원에 대해 여러 스레드가 동시 접근 불가
- 점유와 대기(Hold and Wait) : 자원을 가지고 있는 상태에서 다른 스레드가 사용하고 있는 자원의 반납을 기다림
- 비선점(Non-Preemptive) : 다른 스레드의 자원을 실행 중간에 강제로 가져올 수 없음
- 환형대기(Circle Wait) : 각 스레드가 순환적으로 다음 스레드가 요구하는 자원을 가짐

위 4가지의 조건을 모두 충족할 경우 교착 상태 발생.
즉, 4가지 중 하나라도 충족하지 않을 경우 교착 상태는 발생하지 않는다(교착 상태의 예방)

Spring의 Thread 관리



내장 Tomcat에 thread pool을 만들고, Request가 들어오면 해당 Thread가 해당 요청을 담당해서 로직을 처리
스레드는 Connection Pool에서 유휴 상태인 DB Connection이 존재하면 사용하고, 생성해야 한다면 생성한다.

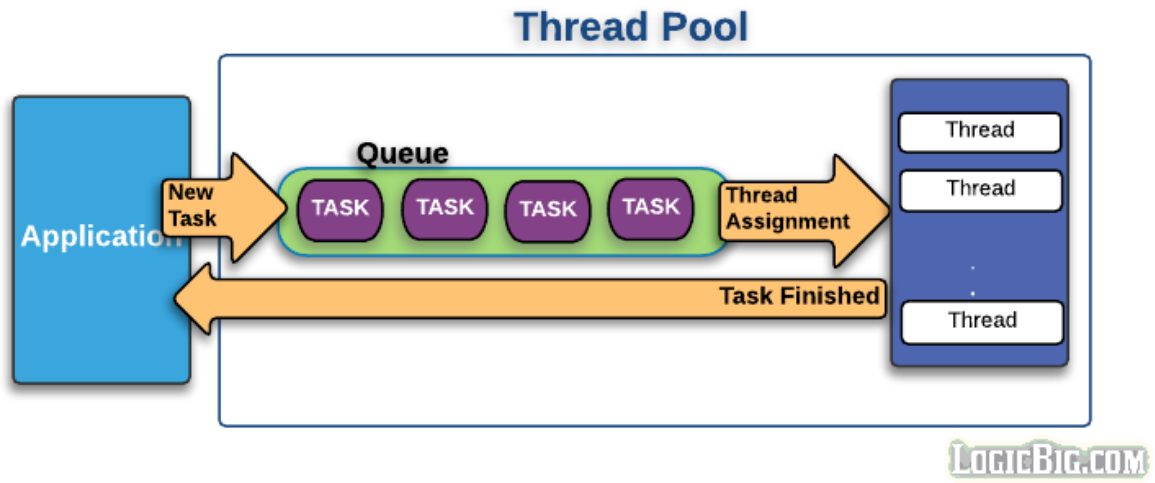
마지막으로 Connection을 이용해 DB에 접근

내장 Tomcat 설정

```yaml
# application.yml (적어놓은 값은 default)
server:
  tomcat:
    threads:
      max: 200 # 생성할 수 있는 thread의 총 개수
      min-spare: 10 # 항상 활성화 되어있는(idle) thread의 개수
    max-connections: 8192 # 수립가능한 connection의 총 개수
    accept-count: 100 # 작업큐의 사이즈
    connection-timeout: 20000 # timeout 판단 기준 시간, 20초
  port: 8080 # 서버를 띄울 포트번호
```

만약 설정을 하지 않는다면, SpringBoot AutoConfiguration에서 정의한 디폴트 값을 주입. 해당 디폴트 값은 org.springframework.boot.autoconfigure.web.ServerProperties 에서 확인 가능

Thread Pool



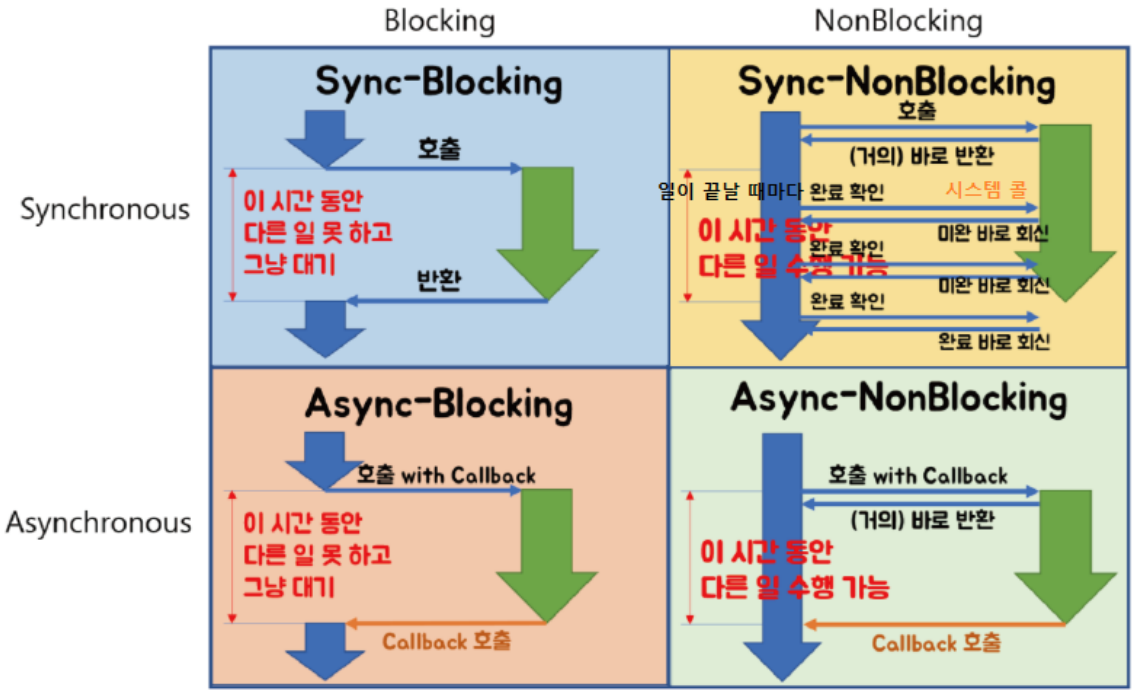
Thread Pool flow

1. 첫 작업이 들어오면, core size 만큼의 스레드를 생성
2. 유저 요청(Connection, Server Socket에서 accept한 소켓 객체)이 들어올 때마다 작업 큐(queue)에 담아둔다.
3. core size의 스레드 중, 유휴 상태(idle)인 스레드가 있다면 작업 큐에서 작업을 꺼내 스레드에 작업을 할당하여 처리
    a. 만약 유휴 상태인 스레드가 없다면, 작업은 작업 큐에서 대기
    b. 작업 큐가 가득 찬다면, 스레드를 새로 생성
    c. 3번 과정을 반복하다 스레드 최대 사이즈에 도달하고 작업 큐도 가득 차게 되면, 추가 요청에 대해선 connection-refused 오류 반환
4. 작업이 완료되면 스레드는 다시 유휴 상태로 돌아감

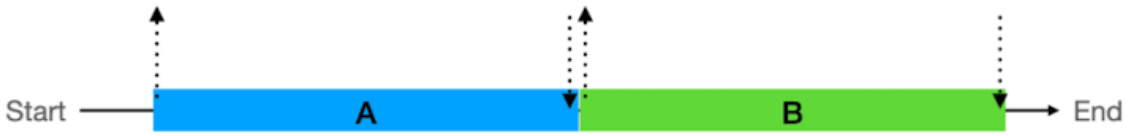
a. 작업 큐가 비어있고 core size 이상의 스레드가 생성되어 있다면 스레드 destroy

- Thread Pool 전략 참고

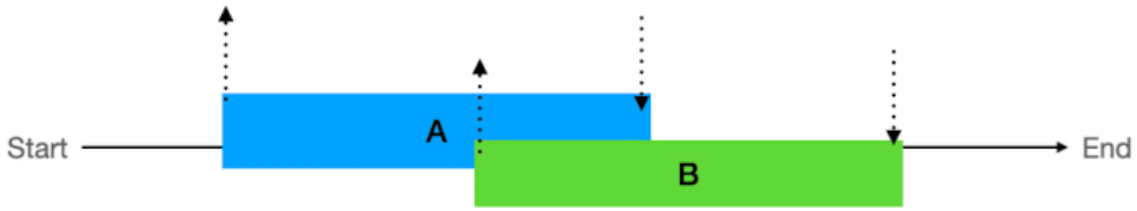
Async vs Sync & Blocking vs Non-Blocking



- Blocking
  - 직접 제어할 수 없는 작업이 끝날 때까지 기다려야 하는 경우
  - 호출된 함수에서 I/O 작업 등을 요청했을 경우 I/O 작업이 처리되기 전까지 다음 작업 불가
- Non-Blocking
  - 직접 제어할 수 없는 작업이 완료되기 전에 제어권을 넘겨주는 경우
  - 호출된 함수에서 I/O 작업 등을 요청했을 경우 I/O 작업의 처리 여부와 관계 없이 다음 작업 가능
- Sync : 작업 A의 종료 시간과 작업 B의 시작 시간이 같으면 동기



- Async : 작업 A와 B가 서로의 시작, 종료 시간과는 관계 없이 별도의 수행 시작/종료 시간을 찾는다.



Async & Blocking 예제

node.js + MySql : MySql Driver가 Blocking 방식으로 작동

Sync & Async vs Blocking & Non-Blocking

Sync & Async : 프로세스의 수행 순서 보장
Blocking & Non-Blocking : 프로세스의 유휴 상태

Spring의 비동기

@Async 어노테이션 사용

- @EnableAsync로 비동기 기능 활성화
- 비동기로 동작을 원하는 public 메소드에 @Async 어노테이션 적용

Enum

Enum 정의

```
// Enum 정의
// enum 열거체이름 { 상수1이름, 상수2이름, ... }

enum Rainbow { RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET }

// Enum의 상숫값 정의 및 추가
// 열거체이름.상수이름

Rainbow.RED

enum Rainbow {
    RED(3), ORANGE(10), YELLOW(21), GREEN(5), BLUE(1), INDIGO(-1),
VIOLET(-11);

    private final int value;
    Rainbow(int value) { this.value = value; }
    public int getValue() { return value; }
}
```

Enum Method

values() method

해당 열거체의 모든 상수를 저장한 배열을 생성하여 반환

```
enum Rainbow { RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET }

public class Enum01 {
    public static void main(String[] args) {
        Rainbow[] arr = Rainbow.values();
        for (Rainbow rb : arr) {
            System.out.println(rb);
        }
    }
}

/********************
실행 결과
RED
ORANGE
```

```
YELLOW
GREEN
BLUE
INDIGO
VIOLET
********************/
```

valueOf() method

전달된 문자열과 일치하는 해당 열거체의 상수 반환

```
enum Rainbow { RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET }

public class Enum02 {
    public static void main(String[] args) {
        Rainbow rb = Rainbow.valueOf("GREEN");
        System.out.println(rb);
    }
}

// 실행 결과
// GREEN
```

java.lang.Enum

Enum 클래스는 모든 자바 열거체의 공통 조상 클래스

Method	설명
protected void finalize()	해당 Enum 클래스가 final 메소드를 가질 수 없게 됨
String name()	해당 열거체 상수의 이름을 반환
int ordinal()	해당 열거체 상수가 열거체 정의에서 정의된 순서(0부터 시작)를 반환

EnumSet

EnumSet이란?

Enum 클래스로 작동하기 위해 특화된 Set 컬렉션(데이터를 중복 저장할 수 없고, 순서가 보장되지 않는 자료구조)
Set Interface를 구현하고 AbstractSet을 상속
추상 클래스이며, 인스턴스 생성을 위한 다양한 정적 팩토리 메소드가 정의
JDK에서는 RegularEnumSet, JumboEnumSet 2가지의 EnumSet 구현체 제공

RegularEnumSet

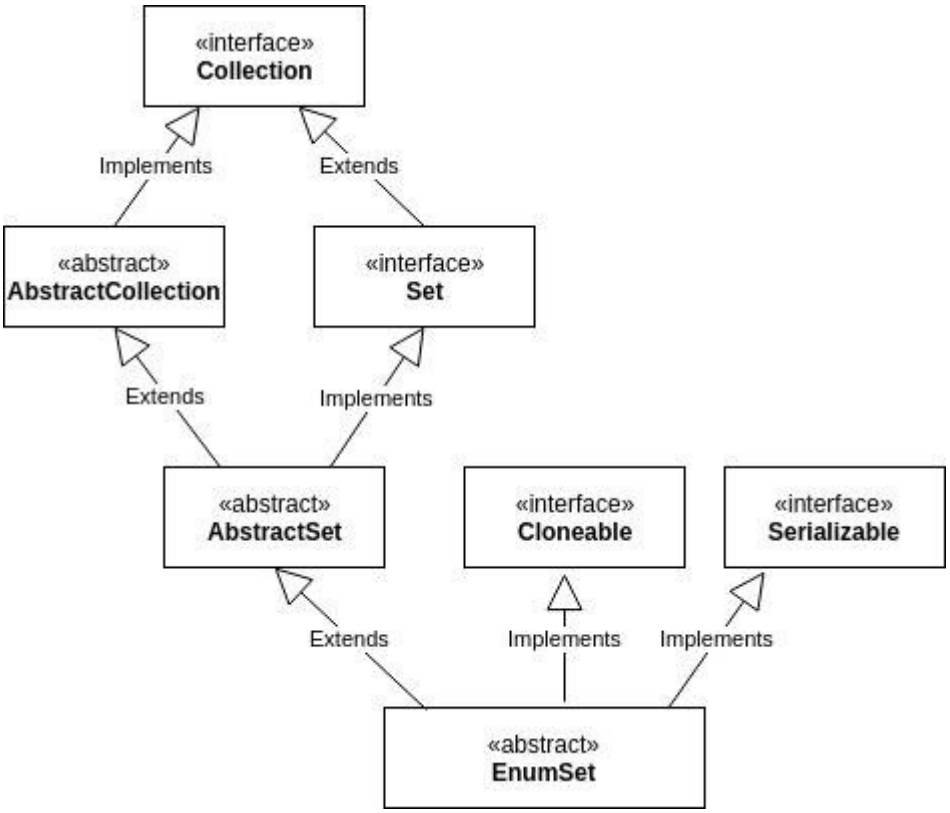
비트 벡터를 표현하기 위해 단일 long 자료형 사용.
long의 각 비트는 Enum 값을 나타낸다.
열거형의 i 번째 값은 i 번째 비트에 저장되므로 값이 있는지 여부를 쉽게 알 수 있다.
long이 64 비트의 데이터이기 때문에 64개의 원소를 저장할 수 있다.

JumboEnumSet

long 요소의 배열을 비트 벡터로 사용한다.
64개 이상의 원소를 저장한다.
RegularEnumSet과 비슷하게 작동하지만, 저장된 배열 인덱스를 찾기 위해 몇가지 추가 계산을 수행

장점

모든 메소드가 산술 비트 연산을 사용하여 구현되므로 일반적인 연산이 매우 빠름
HashSet과 같은 다른 Set 구현체와 비교했을 때, 데이터가 예상 가능한 순서로 저장되어 있고, 각 계산을 하는데 하나의 비트만이 필요하므로 더 빠르다.
HashSet처럼 데이터를 저장할 버킷을 찾는데 hashcode를 계산할 필요가 없다.
비트 벡터의 특성상 더 작은 메모리를 사용



고려 사항

1. 열거형 값만 포함할 수 있으며, 모든 값은 동일한 열거형이어야 한다.
2. null을 추가할 수 없다.

3. Thread에 안전하지 않으므로, 필요할 경우 외부에서 동기화한다.
4. 복사본에 fail-safe iterator(장애 발생시 작업을 중단하지 않음)를 사용하여 컬렉션을 순회할 때, 컬렉션이 수정되어도 ConcurrentModificationException이 발생하지 않는다.

사용법

```
// enum 선언
public enum Color {
    RED, YELLOW, GREEN, BLUE, BLACK, WHITE
}

// allOf()를 사용하여 모든 요소를 포함하는 EnumSet 생성
EnumSet<Color> set = EnumSet.allOf(Color.class);

// noneOf()를 사용하여 빈 Color 컬렉션을 갖는 EnumSet 생성
EnumSet<Color> set = EnumSet.noneOf(Color.class);

// of()를 사용하여 들어갈 요소를 직접 입력하여 EnumSet을 생성
EnumSet<Color> set = EnumSet.of(Color.YELLOW, Color.BLUE;

// complementOf()를 사용하여 원하는 요소를 제거하고 EnumSet 생성
EnumSet<Color> set =
EnumSet.complementOf(EnumSet.of(Color.BLACK,Color.BLUE));

// copyOf()를 사용하여 다른 EnumSet의 모든 요소를 복사하여 EnumSet 생성
EnumSet.copyOf(EnumSet.of(Color.BLACK, Color.WHITE));

// add()를 사용하여 EnumSet에 요소 추가
EnumSet<Color> set = EnumSet.of(Color.YELLOW, Color.BLUE;
    set.add(Color.RED);

// contains()를 사용하여 특정 요소가 EnumSet()에 포함되어 있는지 확인
EnumSet<Color> set = EnumSet.of(Color.YELLOW, Color.BLUE);
set.add(Color.RED);
boolean isContain = set.contains(Color.RED);

if(isContain)
    System.out.println("빨간색 포함");
// 출력 결과
// 빨간색 포함

// remove()를 사용하여 EnumSet에서 특정 요소 제거
set.remove(Color.RED);
```

Enum & JPA

@Enumerated(value = EnumType.STRING)

Enum값의 index가 아닌 텍스트로 DB에 저장하기 위해 사용

```
import javax.persistence.EnumType;

...
public class Order {
	...

	@Enumerated(value = EnumType.STRING)
	// 또는 @Enumerated(EnumType.STRING)
	@Column(name = "menu")
	private Menu menu;

	...
}
```

Stream

Stream이란

데이터 처리 연산을 지원하도록 소스에서 추출된 연속된 요소

Stream을 사용하지 않았을 때

```
// 빨간색 사과 필터링
List<Apple> redApples = forEach(appleList, (Apple apple) ->
apple.getColor().equals("RED"));

// 무게 순서대로 정렬
redApples.sort(Comparator.comparing(Apple::getWeight));

// 사과 고유번호 출력
List<Integer> redHeavyAppleUid = new ArrayList<>();
for (Apple apple : redApples)
    redHeavyAppleUid.add(apple.getUidNum());
```

Stream을 사용했을 때

```
List<Integer> redHeavyAppleUid = appleList.stream()
         // 병렬 처리를 한다면
         // appleList.parallelStream()
        .filter(apple -> apple.getColor().equals("RED"))
         // 빨간색 사과 필터링
        .sorted(Comparator.comparing(Apple::getWeight))
         // 무게 순서대로 정렬
        .map(Apple::getUidNum).collect(Collectors.toList());
         // 사과 고유번호 출력
```

연속된 요소(Sequence of element)

컬렉션 자료구조와 마찬가지로 스트림은 특정 요소 형식으로 이루어진 연속된 값 집합의 인터페이스를 제공
컬렉션에서는 시간, 공간의 복잡성과 관련된 요소 저장 및 연산이 이루어진다면
스트림에서는 filter, sorted, map과 같은 표현 계산식으로 이루어져 있음
컬렉션의 주제는 데이터, 스트림의 주제는 계산

소스(Source)

컬렉션, 배열, I/O 자원 등의 소스로부터 데이터를 소비하고 정렬된 컬렉션으로 스트림을 생성하면 정렬이 그대로 유지
즉, 리스트로 스트림을 만들면 스트림의 요소는 리스트의 요소와 같은 순서를 유지

데이터 처리 연산

함수형 프로그래밍에서 지원하는 연산과 데이터베이스의 SQL 질의형과 비슷한 연산을 처리
filter, sort, map, match 등으로 데이터를 조작할 수 있고 순차적 혹은 병렬로 실행 가능

중간 연산 : 파이프라인으로 연결할 수 있는 연산들

filter나 map 같은 중간 연산은 다른 스트림을 반환하기 때문에 여러개의 중간 연산을 연결하여 질의를 만들 수 있음.
최종 연산을 실행하기 전까지는 아무 연산도 수행하지 않는다.

최종 연산 : 파이프라인 연산의 결과를 출력

List, Integer, void 등 다양한 형태로 출력

```
List<String> highCaloriesFoodName = foodList.stream()
        .filter(food -> food.getCalories() > 400)     // 중간연산
        .map(Food::getName)                           // 중간연산
        .limit(3)                                     // 중간연산
        .collect(Collectors.toList());                // 최종연산
```

Stream의 특징

파이프라이닝(Pipelining)

스트림 연산들은 서로 연결하여 큰 파이프 라인을 구성할 수 있도록 스트림 자신을 반환한다.
데이터 소스에 적용하는 데이터베이스 질의문과 비슷하다.

내부 반복

반복자를 이용하여 명시적으로 반복하는 컬렉션과 다르게 스트림은 내부 반복 기능을 제공

Stream vs Collections

1. 데이터 계산 시점
    - 스트림 : 요청할 때만 요소를 계산하는 고정된 자료구조
    - 컬렉션 : 모든 요소는 컬렉션에 추가하기 전에 계산
2. 반복의 일회성

- 스트림 : Consumer 개념을 쓰기 때문에 한번 소비한 요소에 대해 접근 할 수 없음
- 컬렉션 : 같은 소스에 대하여 여러번 반복 처리 가능

```
Stream<Food> s = foodList.stream();
s.forEach(System.out::println); // 정상
s.forEach(System.out::println); // IllegalStateException 발생
```

위 코드를 실행할 시, “stream has already been operated upon or closed” 에러 발생

3. 외부 반복, 내부 반복
    - 스트림 : 라이브러리를 사용하는 내부 반복

```
List<String> foodNameList = foodList.stream()
        .map(Food::getName)
        .collect(Collectors.toList());
```

- 컬렉션 : 사용자가 반복문을 명시해야하는 외부 반복

```
List<String> foodNameList = new ArrayList<>();
for(Food food : foodList){
    foodNameList.add(food.getName());
}
```