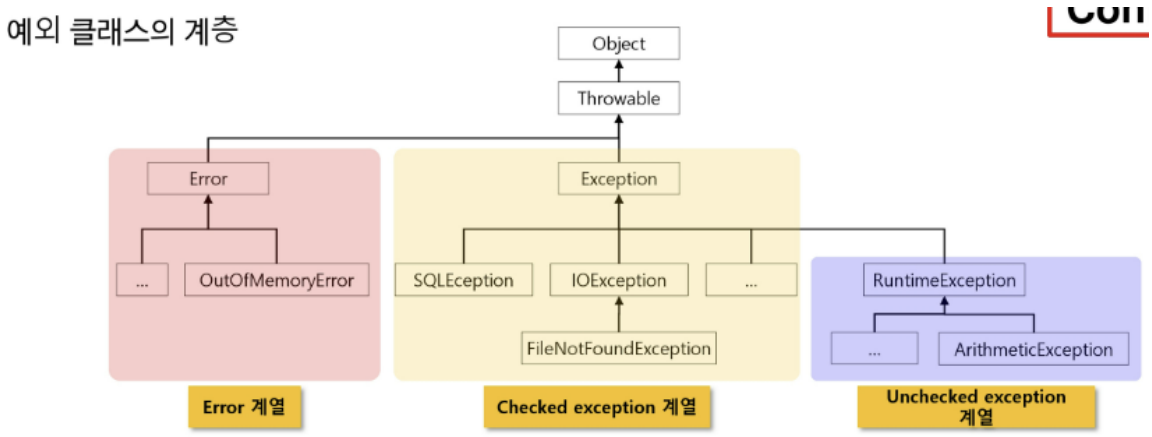
2023/01/25

Exception Handling

- Error
  - 발생하면 복구할 수 없는 상황
- Exception



- check 여부 : 예외에 대한 대처코드가 없는지 체크, **예외 발생 시 어떻게 할 것인가?** 가 중요
- `Checked` Exception : 예외에 대한 대처코드 없으면 컴파일 진행 X
  - 예외처리 코드를 반드시 동반하게 함
  - 문제가 무엇인지 컴파일러가 아는 상황
- `Unchecked` Exception : 예외에 대한 대처코드 없어도 컴파일 진행 O
  - 예외처리 코드가 있든 없든 상관 없음
  - 컴파일러가 몰랐던 상황

```
public class SimpleException {
    public static void main(String[] args) {
        int[] intArray = { 10 };
        System.out.println(intArray[2]);
        System.out.println("프로그램 종료합니다.");
```

```
    }
}
// Runtime Error (컴파일할 때는 문제가 없지만 런타임에 ArrayIndexOutOf 예외)
```

- try-catch 문의 흐름
  1. try 블록에서 예외 발생 시 **JVM 이 해당 Exception 클래스의 객체 생성 후 throw**
  2. 던져진 exception 처리할 수 있는 catch 블록에서 받고 처리
- 다중 catch 문장 작성 시 유의사항
  - 상위 타입 예외가 먼저 선언되면, JVM이 던진 예외는 catch 문을 찾을 때 **다형성이 적용되므로** 뒤에 등장하는 catch 블록이 동작할 기회 없음

    → 따라서 자식 예외부터 잡아주기
- 가급적 **예외상황별로 처리**하는 것이 권장되지만, 심각하지 않은 예외를 굳이 세분화하는 것도 낭비, `|` 연산자 사용해서 catch 문에서 **상속관계 없는** 여러 개의 exception 을 처리

```
try {
  Class.forName("abc.Def");
  ...
} catch (ClassNotFoundException | FileNotFoundException e){
  ...
}
```

- try ~ finally
  - try 블록에서 자원을 획득하고, 문제를 처리하고 싶지 않을 때 항상 실행하고 싶은 게 있을 때 `finally` 를 사용
- try ~ catch ~ finally 구문을 이용한 예외처리
  - 중간에 return 을 만나도 **finally 블록을 먼저 수행 후** 리턴 실행

```
public static void main(String[] args) {
  int num = new Random().nextInt(2);
  try {
    System.out.println("code 1, num: " + num);
    int i = 1 / num;
    System.out.println("code 2 - 예외 없음");
    return;
  } catch (ArithmeticException e) {
    System.out.println("code 3 - exception handling 완료");
  } finally {
    System.out.println("code 4 - 언제나 실행");
  }
  System.out.println("code 5");
}
```

- num 이 0인 경우 : code 1 / code 3 / code 4 / code 5
- num 이 1인 경우 : code 1 / code 2 / code 4 ~~/ code 5~~

  → finally 문장 실행 후 다시 돌아와서 return; 하므로 code 5 는 출력 안됨

- try - with - resources
  - JDK 1.7 이상에서 리소스의 자동 close 처리

    → try 문에 선언된 객체들에 대해 **자동 close 호출 (finally 역할)**

    → 단, 해당 객체들이 AutoCloseable 인터페이스를 구현하고 있어야 함

```
try (리소스 타입1 res1 = 초기화; 리소스 타입2 res2 = 초기화; ...){
} catch(Exception e){
}
```

throws

```
void exceptionMethod() throws Exception1, Exception2 {
  // 예외발생 코드
}

void methodCaller(){
  try {
    exceptionMethod();
  }
  catch (Exception e){
  }
}
```

- methodCaller() 에서 exceptionMethod()를 호출한다.
- exceptionMethod() 에서는 Exception1 또는 Exception2 예외를 던질 수 있음
- 만약 예외가 발생하면 exceptionMethod() 를 호출한 methodCaller() 에게 전달됨 (처리가 위임된다)

- 예외는 없어지는게 아니라 단순히 전달됨 !
  예외 전달받은 메소드는 다시 예외처리 책임 발생

throws = 너 나 호출하면 나 이런 예외 발생시킬지도 몰라 ~

- checked exception 은 반드시 try-catch, 또는 throws 필요
- unchecked exception 은 throws 하지 않아도 전달되지만 결국 try-catch 로 처리 필요

throw = 실행코드 내에서 진짜 예외 던질 때 사용

1. 한번 예외를 잡아서 다시 던질 때
2. 사용자 정의 예외일 때

- 메소드 재정의와 throws
  - 메소드 재정의 시 `조상 클래스 메소드가 던지는 예외`보다, **부모예외 던질 수 없다**

```
class Parent {
  void methodB()
    throws ClassNotFoundException {}
  }
}

public class OverridingTest extends Parent {
  @Override
  void methodB() throws Exception { // 잘못됨
  }
}
```

- 사용자 정의 예외, Custom Exception
  - 대부분 Exception, 또는 RuntimeException 클래스 상속받아 작성
  - 어떤 예외를 상속받느냐에 따라 checked, unchecked exception 만들 수 있음
  - 호출자에게 **정상적이지 않은 상황을 리턴 값 대신 알리고 싶을 때** 사용
  - 예외를 표현하고 싶은 상황을 만들고, 예외를 던져야 하는 상황이 오면 내가 직접 예외를 만들어 던진다.
    - 예기치 못한 상황을 만들어서 적절하게 예외를 던진다