Java

상속

상속

정의

- 상속(inheritance)
  - 이미 작성된 클래스(부모 클래스)를 이어받아서 새로운 클래스(자식 클래스)를 생성하는 기법
  - 기존의 코드를 재활용하기 위한 기법
  - 단일 상속 만을 허용한다.
  - 상속받고자 하는 클래스의 이름을 키워드 extends와 함께 사용한다

```
class Child extends Parent{
    //멤버
}
```

- 생성자와 초기화 블록은 상속되지 않는다
  - 멤버만 상속

상속

정의

- 조상 클래스
  - 부모클래스, 상위 클래스, 기반 클래스
- 자손 클래스
  - 자식 클래스, 하위 클래스, 파생 클래스
- 자손 클래스의 멤버 개수는 조상 클래스보다 항상 같거나 많다.
- 클래스 간의 관계에서 형제 관계는 허용하지 않는다.

상속

정의

- 상속 예

```
class Parent{
        int age;
}

class Child extends Parent{
}

class Child2  extends Parent{
}

class GrandChild extends Child{
}
```

클래스 이름0	클래스의 멤버
Parent	age
Child	age
Child2	age
GrandChiild	age

오버라이딩(Overriding)

정의

- 조상 클래스로부터 상속 받은 메서드의 내용을 변경할 수 있다.

- 자손 클래스에서 오버라이딩 조건
  - 조상 클래스의 메서드와 이름이 같아야 한다
  - 조상 클래스의 메서드와 매개 변수가 같아야 한다.
  - 조상 클래스와 메서드와 리턴 타입이 같아야 한다.

- 접근 제어자와 예외 조건
  - 접근 제어자는 조상 클래스의 메서드 보다 좁은 범위로 변경 할 수 없다
  - 조상 클래스의 메서드 보다 많은 수의 예외를 선언 할 수 없다.
  - 인스턴스 메서드를 static메서드로 또는 그 반대로 변경할 수 없다

오버라이딩(Overriding)

super

- 자손 클래스에서 조상 클래스로 부터 상속받은 멤버를 참조하는데 사용되는 참조 변수
  - 상속받은 멤버와 자신의 클래스에 정의된 멤버의 이름이 같을 때 super를 사용해서 구별
  - super대신 this를 사용 할 수 있다
  - static메서드(클래스 메서드)는 인스턴스와 관련이 없다
  - static메서드에서는 사용 할 수 없고 인스턴스에서만 사용 가능

오버라이딩(Overriding)

super()

- 조상클래스의 생성자
  - 자손 클래스의 인스턴스가 조상 클래스의 멤버를 사용하기 위해서는 초기화 작업이 먼저 이루어 져야 한다.

- 자손 클래스에서 조상 클래스의 생성자를 호출
  - this()는 같은 클래스의 다른 생성자를 호출
  - super()는 조상 클래스의 생성자를 호출

- Object클래스를 제외한 모든 클래스의 생성자 첫 줄에는 생성자를 호출 해야 한다.
  - 컴파일러가 자동적으로 'super();'를 생성자의 첫 줄에 삽입한다

제어자(modifier)

- 클래스, 변수 또는 메서드의 선언부에 함께 사용되어 부가적인 의미를 부여한다.
  - 하나의 대상에 대해서 여러 제어자를 사용할 수 있다
  - 제어자들 간의 순서는 관계 없지만 접근 제어자를 왼쪽에 놓는 것이 일반적

- 접근 제어자
  - public, protected, default, private
  - 네 가지 중 하나만 선택해서 사용 할 수 있다

- 그 외 제어자
  - static, final, abstract, native, transient, synchronized, volatile, strictfp

제어자(modifier)

static

- static이 붙은 멤버 변수와 메서드, 그리고 초기화 블록은 인스턴스를 생성하지 않고도 사용할 수 있다

- 멤버 변수
  - 모든 인스턴스에 공통적으로 사용되는 클래스 변수가 된다
  - 클래스 변수는 인스턴스를 생성하지 않고도 사용 가능하다
  - 클래스가 메모리에 로드 될 때 생성된다

- 메서드
  - 인스턴스를 생성하지 않고도 호출이 가능하다

- 인스턴스 메서드와 static메서드의 차이
  - static 메서드 내에서 인스턴스 멤버를 사용할 수 없다

제어자(modifier)

final

- 상태와 기능을 변경 할 수 없다

- 클래스
  - 변경될 수 없고, 확장 될 수 없기 때문에 다른 클래스의 조상이 될 수 없다.

- 메서드
  - 오버라이딩을 통해 재 정의 될 수 없다

- 멤버 변수 / 지역 변수
  - 값을 변경 할 수 없는 상수가 된다

제어자(modifier)

final

- 생성자를 이용한 final멤버 변수 초기화
  - final이 붙은 인스턴스의 경우 생성자에서 초기화 할 수 있다

제어자(modifier)

abstract

- 추상 메서드를 선언하는데 사용
  - 메서드의 선언부만 선언하고 구현하지 않는다

- 클래스
  - 클래스 내에 추상 메서드가 선언 되어 있음을 의미한다

- 메서드
  - 선언부만 작성하고 구현부는 작성하지 않는 추상 메서드 임을 알린다.

제어자(modifier)

abstract

- 추상 메서드 선언

```
abstract class AbsEx {
        abstract void getArea();
}
```

추상클래스(abstract class)

정의

- 추상화
  - 구체적 개념에서 공통된 성질을 뽑아 일반적인 개념으로 접근
  - 클래스간의 공통점을 찾아내서 공통의 조상을 만드는 작업

- 추상 클래스
  - 추상 메서드를 포함 하고 있는 클래스
  - 추상 클래스로 인스턴스를 생성할 수 없다
  - 상속을 통해서 자손 클래스에 의해서만 완성 될 수 있다

- 추상 클래스 선언
  - Abstract class 클래스 이름{
  - }

추상클래스(abstract class)

정의

- 추상 메서드
  - 선언부만 작성하고 구현부는 작성하지 않은 채로 남겨둔 메서드
  - 실제 내용은 상속받은 클래스가 구현한다
  - 추상 클래스로부터 상속 받는 자손 클래스는 오버라이딩을 통해 상속 받은 추상 메서드를 모두 구현해 주어야 한다.
  - 상속 받은 추상메서드중 하나라도 구현하지 않는다면, 자손 클래스 역시 추상클래스로 지정해 주어야 한다.

- 추상 메서드 선언
  - Abstract 리턴타입 메서드이름();
  - 구현부가 없기 때문에 {}대신 문장의 끝을 알리는 ';'을 적어 준다

인터페이스(interface)

정의

- 멤버 메서드 또는 멤버 변수를 구성원으로 가질 수 없는 추상 클래스
  - 추상 메서드와 상수 만을 멤버로 가질 수 있다

인터페이스(interface)

인터페이스 작성

- 클래스를 작성하는 것과 같지만, 키워드로 class 대신 interface를 사용한다.

```
Interface 인터페이스이름{
        public static final 타입 상수이름 = 값;
        public abstract 메서드이름(매개변수 목록);
}
```

- 모든 멤버 변수는 public static final이어야 하며, 생략 할 수 있다
- 모든 메서드는 public abstract이어야 하며, 이를 생략 할 수 있다
- 생략된 제어자는 컴파일러가 자동적으로 추가해 준다

인터페이스(interface)

인터페이스 상속

- 인터페이스는 인터페이스로부터만 상속 받을 수 있다
  - 다중 상속이 가능하다

```
Interface Movable{
        void move(int x, int y);
}

Interface Attackable{
        void attack(unit u);
}

Interface Fightable extends Movable, Attackable{}
```

인터페이스(interface)

인터페이스 구현

- 인터페이스 자체로는 인스턴스를 생성할 수 없다
  - 자신에게 정의된 추상 메서드를 구현 해주는 클래스를 작성해야 한다.
  - 상속을 받는 클래스를 작성할 때 implements 키워드를 사용한다.

```
Class 클래스 implements 인터페이스이름{
    //인터페이스에 정의된 추상메서드를 구현
}
```

내부 클래스(inner class)

정의

- 내부 클래스
  - 클래스 안에 선언된 클래스
  - 두 클래스의 멤버들 간에 쉽게 접근 할 수 있고, 외부로 부터 클래스 접근을 차단 할 수 있다.

내부 클래스(inner class)

내부 클래스의 종류

- 인스턴스 클래스
  - 외부 클래스의 멤버 변수 선언 위치에 선언
  - 외부 클래스의 인스턴스 멤버처럼 접근 할 수 있다
- 스태틱 클래스
  - 외부 클래스의 멤버 변수 선언 위치에 선언
  - 외부 클래스의 static멤버 처럼 접근 할 수 있다
- 지역 클래스
  - 외부 클래스의 메서드나 초기화 블록 안에 선언
  - 선언된 영역 내부에서만 사용될 수 있다
- 익명 클래스
  - 클래스의 선언과 객체의 생성을 동시에 하는 이름 없는 클래스

내부 클래스(inner class)

익명 클래스

- 익명 클래스(Anonymous Class)
  - 이름이 없는 클래스이며, 한 번만 사용할 목적으로 주로 사용
  - 클래스의 선언과 객체의 생성을 동시에 하기 때문에 하나의 객체 만을 생성할 수 있다
  - 보통 인터페이스나 추상 클래스를 구현하거나, 기존 클래스를 확장하여 특정 메서드만 오버라이딩할 때 사용
  - 단 하나의 클래스를 상속 받거나 하나의 인터페이스만 구현 할 수 있다
  - 익명 클래스는 new 키워드를 사용하여 즉시 생성된다.
  - Lambda 표현식이 가능할 경우, Java 8 이상에서는 익명 클래스 대신 주로 Lambda를 사용
- 익명 클래스의 일반적인 사용
  - 인터페이스 구현
  - 추상 클래스 구현
  - 기존 클래스 확장 및 메서드 오버라이딩

내부 클래스(inner class)

익명 클래스

- 이름이 없기 때문에 생성자를 가질 수 없다
  - 조상 클래스의 이름이나 구현하고자 하는 인터페이스의 이름을 사용해서 정의
  - 단 하나의 클래스를 상속 받거나 하나의 인터페이스만 구현 할 수 있다

```
new 조상클래스이름() {
        //멤버 선언
}

new 구현된 인터페이스이름(){
        //멤버 선언
}
```

다형성

정의

- 다형성
  - 한 타입의 참조 변수로 여러 타입의 객체를 참조 할 수 있다
  - 개념적으로 동일한 작업을 하는 멤버 함수들에 똑같은 이름을 부여할 수 있으므로 코드가 더 간단해진다
  - 조상 클래스 타입의 참조 변수로 자손 클래스의 인스턴스를 참조 할 수 있다

- 상속 관계에 있는 클래스 사이에서 형 변환이 가능하다.
  - 자손 타입의 참조 변수를 조상 타입의 참조 변수로 형 변환 가능
  - 조상 타입의 참조 변수를 자손 타입의 참조 변수로 형 변환 가능
  - 간접 적인 상속 관계의 경우에도 형 변환이 가능하다

다형성

참조변수 형 변환

- 업 캐스팅
  - 자손 타입의 참조 변수를 조상 타입으로 형 변환
  - 자손 타입 -> 조상 타입 : 형 변환 생략 가능

- 다운 캐스팅
  - 조상 타입의 참조 변수를 자손 타입으로 형 변환
  - 자손 타입 <- 조상 타입 : 형 변환 생략 불 가능

- 참조 변수의 타입이 참조 변수가 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의 개수를 결정한다
  - 같은 타입의 인스턴스라도 참조 변수의 타입에 따라 사용할 수 있는 멤버의 개수가 달라진다
  - 존재하지 않는 멤버를 사용할 수 없다
  - 모든 참조 변수는 4 byte의 주소 값을 갖는다.

다형성

참조변수 형 변환

- Instanceof 연산자
  - 참조 변수가 참조하고 있는 인스턴스의 실제 타입을 알아보기 위한 연산자
  - 왼쪽에는 참조 변수를, 오른쪽에는 클래스 명을 피 연산자로 위치시킨다
  - Boolean값을 리턴 한다

다형성

참조변수와 인스턴스

- 멤버가 조상 클래스와 자손 클래스에 중복으로 정의 됐을 때
  - 조상 타입의 참조 변수로 자손 인스턴스를 참조하는 경우와
  - 자손 타입의 참조 변수로 자손 인스턴스를 참조하는 경우 다른 결과를 얻는다

- 메서드의 경우
  - 항상 실제 인스턴스의 메서드가 호출된다
  - Static메서드는 참조 변수의 타입에 영향을 받는다

- 멤버 변수의 경우
  - 참조 변수의 타입에 따라 달라진다
  - 조상타입의참조 변수를 사용 했을 때는 조상 클래스에 선언된 멤버 변수가 사용된다.
  - 자손 타입의 참조 변수를 사용 했을 때는 자손 클래스에 선언된 멤버 변수가 사용된다.

다형성

매개 변수의 다형성

- 메서드의 매개 변수에도 참조 변수의 다형적인 특징이 적용된다.
  - 메서드의 매개 변수로 자손 타입의 참조 변수이면,어느 것이나 매개변수로 받아 들일 수 있다

다형성

객체 배열

- 조상 타입의 참조 변수 배열을 사용하면 공통의 조상을 가진 서로 다른 종류의 객체를 배열로 묶어서 다룰 수 있다

```
Product p1 = new Tv();
Product p2 = new Computer();
Product p3 = new Audio();


Product p[] = new Product[3];
p[0] = new Tv();
p[1] = new Computer();
p[2] = new Audio();
```

예외 처리

정의

- 프로그램 오류
  - 프로그램이 실행 중 오작동을 하거나 비 정상적으로 종료시키는 어떤 원인

- 컴파일 에러 (compile-time error)
  - 컴파일 할 때 발생하는 에러

- 런타임 에러 (runtime error)
  - 프로그램 실행 도중에 발생하는 에러
  - 컴파일이 성공했다 하더라도, 실행 도중 잘못된 결과를 얻거나 잘못된 연산을 수행 할 수 있다

예외 처리

런타임 에러 분류

- 런타임 에러 분류
  - 에러 : 일단 발생하면 복구 할 수 없는 심각한 오류
  - 예외 : 발생하더라도 수습될 수 있는 비교적 덜 심각한 오류

예외 처리

예외 처리 정의

- 예외 처리
  - 프로그램 실행 시 발생할 수 있는 예기치 못한 예외의 발생에 대비한 루틴을 작성하는 것

- 예외 처리 목적
  - 예외의 발생으로 인한 프로그램의 비 정상 종료를 막고, 정상적인 실행 상태를 유지할 수 있어야 한다

예외처리

Try-catch

- 예외처리를 하기 위해 try-catch문을 사용한다

```
Try {
      //예외가 발생할 가능성이 있는 문장
}catch(Exception e1){
      //Exception1이 발생 했을 경우 ,이를 처리하기 위한 문장
} catch ( Exception e2){
      //Exception2가 발생 했을 경우 ,이를 처리하기 위한 문장
}
…
} catch ( Exception eN){
      //ExceptionN이 발생 했을 경우 ,이를 처리하기 위한 문장
}
```

-하나의 try블럭 다음에 여러 종류의 예외를 처리할 수 있도록 하나 이상의 catch 블럭이 올 수 있으며, 이 중 발생한 예외의 종류와 일치하는 단 한 개의 catch 블럭만 수행된다.

- Try 블럭이나 catch 블럭은 {}를 생략할 수 없다

- 발생한 예외의 종류와 일치하는 catch 블럭이 없으면 예외는 처리되지 않는다.

예외 처리

Try-catch문 실행 순서

- Try블럭 내에서 예외가 발생한 경우
  - 발생한 예외와 일치하는 catch블럭이 있는지 확인한다
  - 일치하는 catch블럭을 찾게 되면, 그 catch블럭 내의 문장들을 수행하고 전체 try-catch문을 빠져나가서 그 다음 문장을 계속 실행한다.
  - 일치하는 catch블럭을 찾지 못하면 예외는 처리되지 못한다

- Try블럭 내에서 예외가 발생하지 않은 경우
  - Catch블럭을 거치지 않고 전체 try-catch문을 빠져나가서 수행을 계속한다

예외 처리

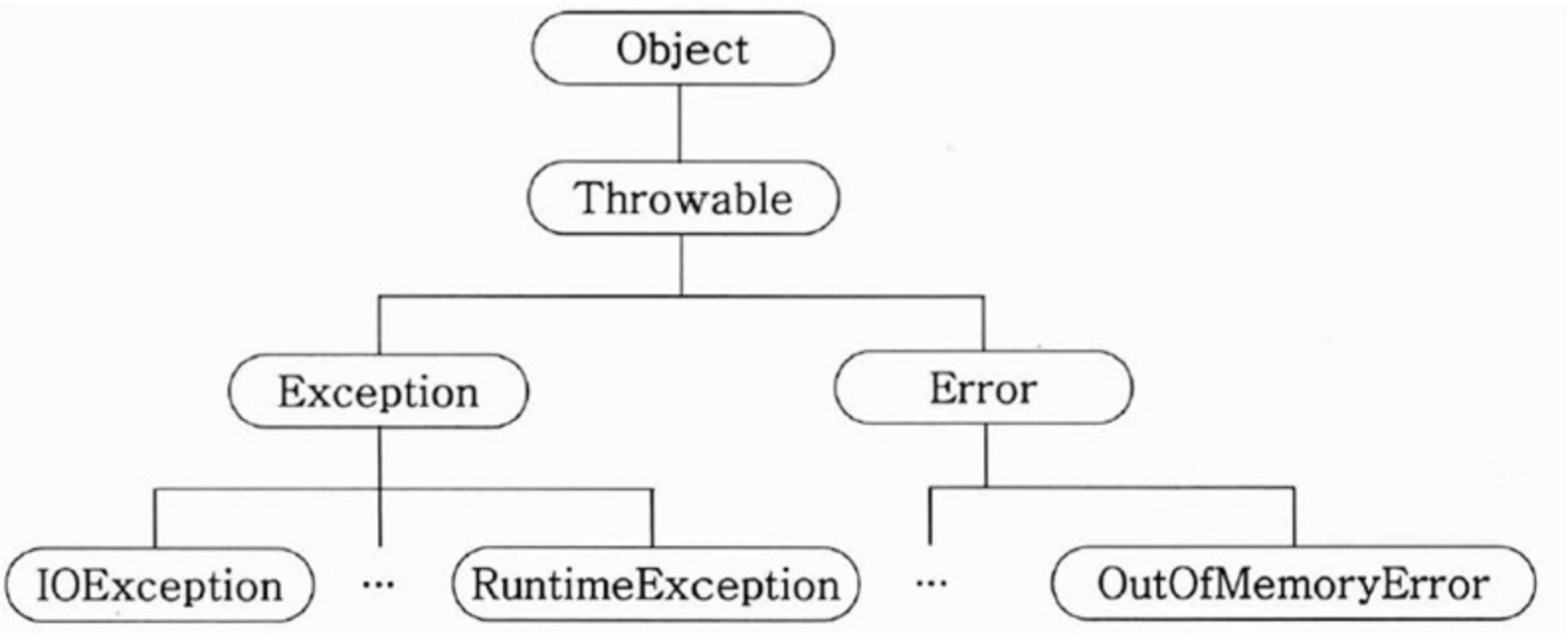
예외 발생 시키기

- 키워드 throw를 사용해서 예외를 발생 시킬 수 있다
  - 연산자 new를 이용해서 발생 시키려는 예외 클래스의 객체를 만든다
    a. Exception e = new Exception("예외발생");
  - 키워드 throw를 이용해서 예외를 발생 시킨다.
    a. throw e;

예외 처리

예외 클래스 계층구조

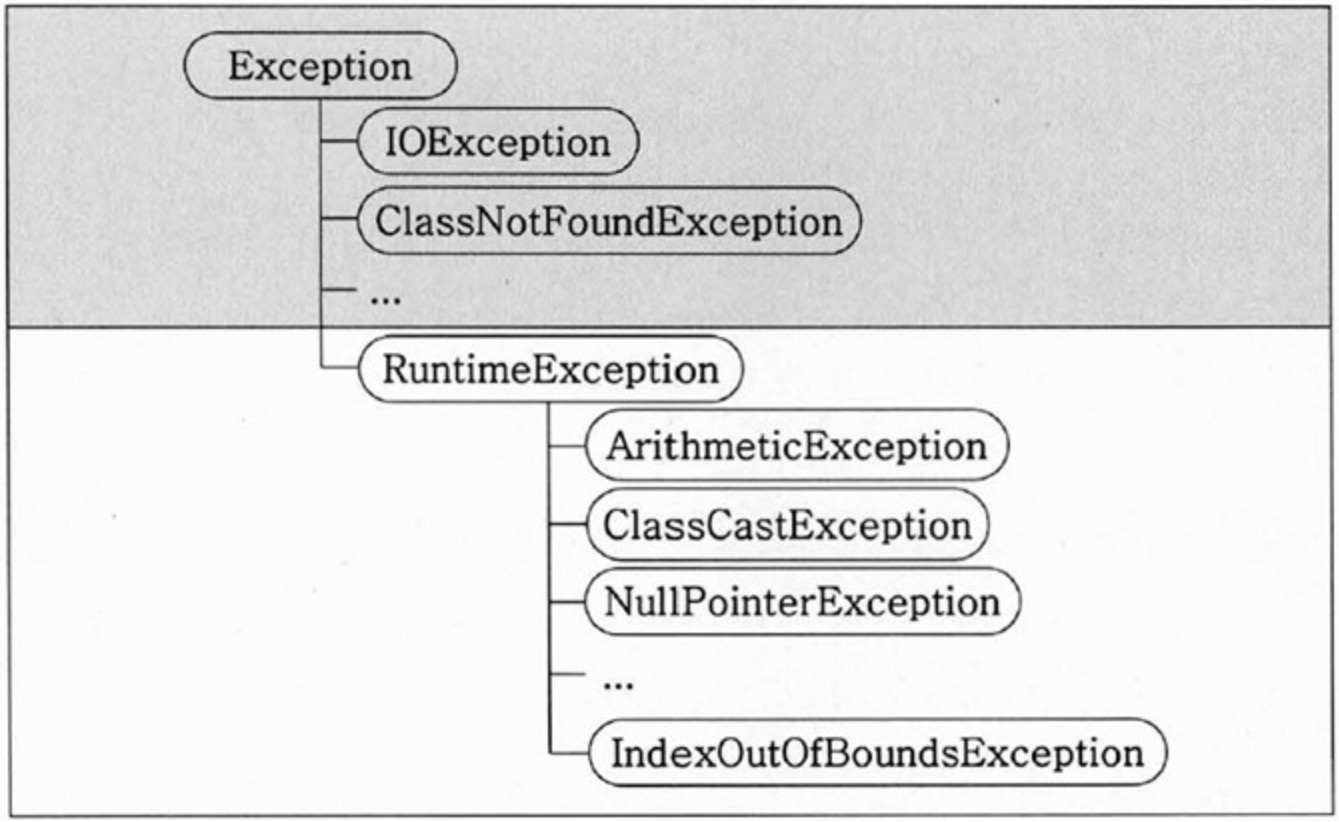
- 예외 클래스 계층도



예외 처리

예외 클래스 계층구조

- Exception클래스와 RuntimeException클래스 계층도



예외 처리

예외 발생과 catch

- catch블럭의 ()안에 처리하고자 하는 예외와 같은 타입의 참조 변수를 선언한다.
- 예외가 발생하면, 발생한 예외에 해당하는 클래스의 인스턴스가 만들어 진다
- 예외가 발생한 문장이 try블럭에 포함되어 있다면, 이 예외를 처리할 수 있는 catch블럭이 있는지 찾아본다
  - catch블럭의 괄호 안에 선언된 참조 변수의 종류와, 생성된 예외 클래스의 인스턴스에 instanceof연산자를 이용해서 결과가 true인 catch블럭을 만날 때 까지 계속 검사한다
  - 데이터 타입이 동일한 catch문을 찾아 instance의 참조 값을 catch문의 참조 변수에 전달
- catch블럭을 찾게 되면 블록에 있는 문장들을 수행하여 예외를 처리
- catch블럭이 하나도 없으면 예외는 처리되지 않는다
- catch블럭의 괄호 안에 Exception클래스 타입의 참조 변수를 선언해 놓으면 어떤 종류의 예외가 발생하더라도 이 catch블럭에 의해서 처리된다.

예외 처리

예외 발생과 catch

- 예외가 발생 했을 때 생성되는 예외 클래스의 인스턴스에는 발생한 예외에 대한 정보가 담겨져 있다
  - catch블럭의 ()에 선언된 참조 변수를 통해 이 인스턴스에 접근 할 수 있다
  - catch블럭의 ()에 선언된 참조 변수는 선언된 catch블럭 내에서만 사용 가능

- printStackTrace()
  - 예외 발생 당시의 호출 스택에 있었던 메서드의 정보와 예외 메시지를 화면에 출력한다

- getMessage()
  - 발생한 예외 클래스의 인스턴스에 저장된 메시지를 얻을 수 있다

예외 처리

finally블럭

- finally블럭은 try-catch문과 함께 예외의 발생여부에 상관없이 실행되어야 할 코드를 포함한다

```
try{
    //예외가 발생한 가능성이 있는 문장들
} catch (Exception e) {
    //예외처리
} finally {
    //예외의 발생여부에 관계없이 항상 수행되어야 하는 문장들
    //finally블럭은 try-catch문의 맨 마지막에 위치해야한다.
}
```

- 예외가 발생한 경우
  - try -> catch -> finally
- 예외가 발생하지 않은 경우
  - try -> finally

예외 처리

메서드에 예외 선언하기

- 메서드 내에서 발생할 가능성이 있는 예외를 메서드의 선언부에 명시하여 이에 대한 처리를 강요 할 수 있다
  - 메서드의 선언부에 키워드 throws를 사용해서 메서드 내에서 발생할 수 있는 예외를 적어 준다
  - 예외가 여러 개 일 경우에는 쉼표로 구분한다
  - 메서드를 사용하기 위해서 어떤 예외들이 처리 되어져야 하는지 정보전달 역할을 한다
  - 일반적으로 RuntimeException클래스 들은 적지 않는다.
  - Exception클래스 예외는 반드시 처리해 주어야 한다.

```
void method() throws Exception1, Exception2, …, ExceptionN{
      //메서드 내용
}
```

예외 처리

Exception re-throwing

- 한 메서드에서 발생할 수 있는 예외가 여럿인 경우, try-catch문과 호출한 메서드에서 처리되도록 할 수 있다

- 하나의 예외에 대해서 예외가 발생한 메서드와 호출한 메서드 양쪽에서 처리하도록 할 수 있다
  - 예외를 처리한 후에 인위적으로 다시 발생
  - try-catch문을 사용해서 예외를 처리해 주고 catch에서 throw문을 사용해서 예외를 다시 발생시켜 호출한 메서드에게 전달한다
  - 예외가 발생할 메서드에서는 try-catch문을 사용해서 예외처리를 해 줌과 동시에 메서드의 선언부에 발생할 예외를 throws에 지정해 줘야 한다