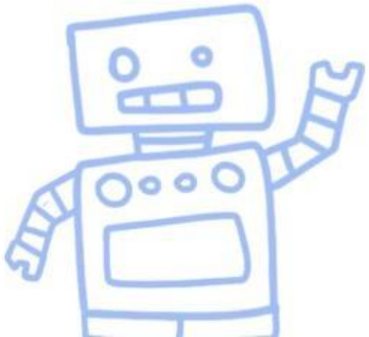
SSAFY JAVA 1차 과목평가

멋찐 서울 11반 CA 윤하



목차(4지선다, 단답형, 서술형)

1. 데이터 타입과 형변환 그리고 연산자

2. 기본 문장 switch (break; 나 parameter로 들어갈 수 있는 데이터 타입)

3. 클래스설계 -생성자 -toString() -eqauls() -제한자 등

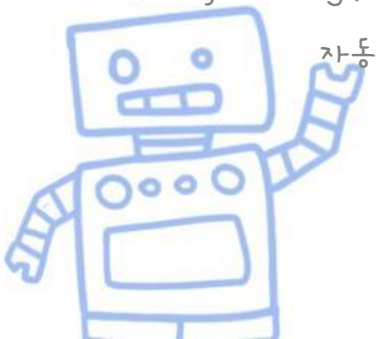
4. 객체생성 -this. -super. 키워드 (생성자에서 호출)

5. 다형성 -오버로딩 -오버라이딩

6. 배열과 배열의 초기화 (배열 선언과 할당, new 이런거?)

7. API -java.lang(디폴트 패키지) -맨날 사용되는거(Object, String) 별도로 import하지 않아도 자동으로 import됨 , -java.util(Collections, Array,Tokenizer)

8. 예외처리 -abstract class, -interface



JAVA란?

OOP is A.P.I.E(Abstraction, Polymorphism, Inheritance, Encapsulation)

추상화 : 현실의 객체를 추상화 해서 클래스 구성

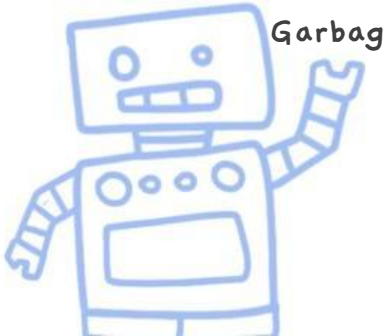
다형성 : 하나의 객체를 여러 가지 타입으로 참조

상속 : 부모 클래스의 자산을 물려받아 자식을 정의(코드 재사용 가능)

캡슐화 : 데이터를 외부에 직접 노출시키지 않고 메서드를 이용해 보호

WORA(Write Once, Run AnyWhere) - JVM (운영체제 종속적이지 않고 플랫폼 독립적)

Garbage Collection - 더 이상 사용하지 않는 메모리를 자동으로 정리



데이터 타입

Primitive Type(기본형 변수)

- 미리 정해진 크기의 Memory size로 표현, 변수 자체에 값 저장
- 대소문자 구분 O, 길이 제한 X, 숫자로 시작 X, 예약어 X, 특수문자는 _와 $만 가능

구분	Type	bit 수	값
논리형	**boolean**		true / false
정수형	**byte**	8	-2^7 ~ 2^7-1 (-128 ~ 127)
	short	16	-2^15 ~ 2^15-1 (-32768 ~ 32767)
	int	32	-2^31 ~ 2^31-1 (-2147483648 ~ 2147483647, 20억 쯤?)
	long	64	-2^63 ~ 2^63-1 (-9223372036854775808 ~ 9223372036854775807)
실수형	**float**	32	`float f = 0.123456789012345678f; // 0.12345679`
	double	64	`double d = 0.1234567890123456789; // 0.12345678901234568`
문자형	**char**	16	\u0000 ~ \uffff (0 ~ 2^16-1)

형 변환

Primitive Type(기본형 변수)

-byte단위는 연산 X (int로 형 변환 필요)

-byte로 데이터 타입 지정 시 메모리 절약, but 연산 실행시 CPU 과부하 가능성 O

따라서 연산을 해야할 경우 int를 사용하는게 효율적.

*정수 계산 시 overflow 주의(overflow는 에러 X), 실수 연산은 정확 X

*int 형 변수에 'A'를 넣으면 65가 들어감. (아래 아스키 코드표 참조)

이진법	십진법	문자
0110000	48	'0'
0110001	49	'1'
0110010	50	'2'
1000001	65	'A'
1100001	97	'a'

형 변환

Reference Type(참조형 변수)

-기본형 데이터 타입 8가지를 제외한 모든 데이터 타입(사용자 정의 타입 포함)

Type Casting(형 변환)

-값의 타입을 다른 타입으로 변환하는 것

-primitive는 primitive끼리, reference는 reference끼리 형 변환 가능

*boolean은 다른 기본 타입과 호환되지 않음

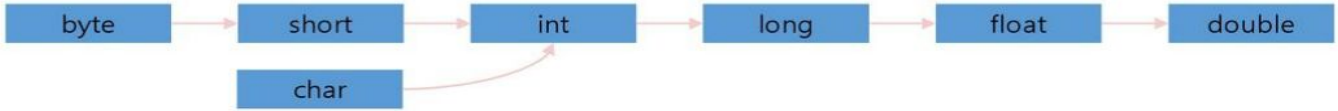
묵시적 형 변환(promotion)

```
byte b = 10;
int i = (int)b;
int i2 = b;
```

명시적 형 변환

```
int i = 300;
byte b = (byte)i;
```

값의 크기, 타입의 크기가 아닌 타입의 표현 범위가 커지는 방향으로 할당할 경우는 묵시적 형변환 발생



명시적 형변환은 값 손실이 발생할 수 있으므로 프로그래머 책임하에 형변환 진행

묵시적 형변환은 자료의 손실 걱정이 없으므로 JVM이 서비스 해줌.

연산자

Operator(연산자) 우선순위

- 이항 연산자는 연산하기 전 피연산자의 타입을 일치시킨다.(큰 타입으로 묵시적 형 변환)

연산기호	결합방향	우선순위
( ), .		
++ -- +(부호) -(부호) ~ ! (type) : 형변환	←	높음
* / %	→	
+ (덧셈) - (뺄셈)	→	
<< >> >>>	→	
< > <= >= instanceof	→	
== !=	→	
&	→	
^	→	
|	→	
&&	→	
||	→	
? :	→	
= *= /= %= += -= <<= >>= >>>= &= ^= |=	←	낮음

연산자 우선순위가 같을 경우
→ 연산 진행 방향에 의해 결정

3 * 4 * 5
1 2

x = y = 3
2 1

Operator(연산자)

-이항 연산자는 연산하기 전 피연산자의 타입을 일치시킨다.(큰 타입으로 묵시적 형 변환)

연산자	연산자 기능	결합 방향
&	두 개의 피연산자가 모두 true인 경우 true Ex) a>0 & b>0	→
|	두 개의 피 연사자가 하나라도 true이면 true Ex) a>0 | b<0	→
!	단항 연산자로 피 연산자의 값이 false이면 true, true이면 false로 변경 Ex) !a	←
^	두 피 연산자가 서로 다를 경우만 true, 같으면 false Ex) true ^ false → true, true ^ true → false	→

연산자	연산자 기능	결합 방향
&&	&와 동일한 의미이나 앞의 피 연산자가 false이면 뒤의 피 연산자를 검사하지 않는다. ex) a>0 && b>0	→
||	|와 동일한 의미이나 앞의 피 연산자가 true이면 뒤의 피 연산자를 검사하지 않는다. ex) a>0 || b<0	→

조건문 Switch

Switch문

-한 번의 조건식 연산만 진행, 이후는 해당하는 case값으로 포인터 이동

-break를 만나지 않으면 적용된 case문 이후의 모든 case 수행

*조건식 연산 결과는 int 형 범위의 정수 값, 또는 문자열, enum

*case 문에는 오로지 리터럴이나 상수만 허용됨. 변수 X

```
switch(조건식){
    case 값1 :
        do something.. // 조건식의 결과가 값1과 같을 경우 실행될 문장
        break;              // switch 문을 벗어난다.
    case 값2 :
        do something.. // 조건식의 결과가 값2와 같을 경우 실행될 문장

     default :
         do something.. // 조건식 결과와 일치하는 case 문이 없을 때 실행될 문장
}
```

클래스 설계

equals() 메서드

두 객체가 같은지를 비교하는 메서드

```
public boolean equals(Object obj) {
  return (this == obj);
}
@Override
public boolean equals(Object obj) {
  if (obj != null && obj instanceof Phone) {
    Phone casted = (Phone) obj;
    return number.equals(casted.number);
  }
  return false;
}
```

등가비교 연산자 ==로 두 객체의 주소값 비교

객체의 주소 비교: == 활용

객체의 내용 비교: equals 재정의

toString() 메서드

객체를 문자열로 변경하는 메서드

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

클래스 설계

생성자 Person person1 = new Person();

- 클래스 이름과 동일, 리턴 타입이 없음
- 생성자가 없으면 컴파일러가 기본 생성자 제공(파라미터가 없고 구현부 비어있는 상태)

*파라미터가 있는 생성자를 만들면 컴파일러 기본 생성자 추가 X

toString() 메서드

객체를 문자열로 변경하는 메서드

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
@Override
public String toString() {
    return "SpiderMan [isSpider=" + isSpider + ", name=" + name + "]";
}
```

객체 생성

this.

- 참조 변수로써 객체 자신을 가리킴. -> 객체에 대한 참조이므로 static 영역에서 사용 X
- 로컬 변수와 멤버 변수의 이름이 동일할 경우 멤버 변수임을 명시적으로 나타냄.

```
Person(String name, int age, boolean isHungry){
    this.name = name;
    this.age = age;
    this.isHungry = isHungry;
 }
```

this()

- 메서드와 마찬가지로 생성자도 오버로딩 가능. -> 반드시 첫 줄에서만 호출 가능.
- 한 생성자에서 다른 생성자를 호출할 때 사용.

```
OverloadConstructorPerson(String name, int age) {
  this.name = name;
  this.age = age;
}

OverloadConstructorPerson(String name) {
  this(name, 0); // 첫 번째 라인에서만 사용 가능
}
```

super.

-super를 통해 조상 클래스 멤버 접근. 조상의 메서드 호출로 조상의 코드 재사용.

-로컬 변수와 멤버 변수의 이름이 동일할 경우 멤버 변수임을 명시적으로 나타냄.

super.을 이용해 조상의 메서드 호출로 조상의 코드 재사용

```
void jump() {
  if (isSpider) {
    spider.jump();
  } else {
    System.out.println("뛰기");
  }
}
```

```
void jump() {
  if (isSpider) {
    spider.jump();
  } else {
    super.jump();
  }
}
```

super()

-super()는 조상 클래스의 생성자 호출.

*반드시 자식 클래스 생성자의 맨 첫 줄에서만 호출 가능. (컴파일러가 super() 삽입)

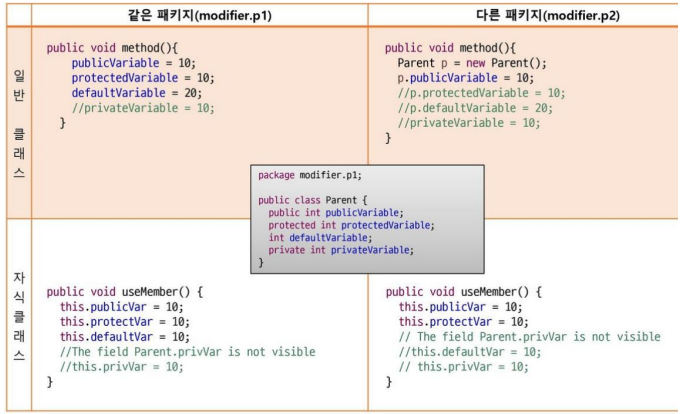
```
class Person2 {
  String name;
  Person2(String name) {
    // super();// --> Object의 기본 생성자 호출
    this.name = name;
  }
}
```

객체 생성

제한자

- 접근 제한자 : public, protected, (default=package), private
- 그 외 제한자 : static, final, abstract, synchronized...

제한자	용도			접근 가능 범위			
	클래스	생성자	멤버	같은 클래스	같은 패키지	다른 패키지의 자손 클래스	전체
public	O	O	O	O	O	O	O
protected		O	O	O	O	O	
package(default)	O	O	O	O	O		
private		O	O	O			



데이터 은닉과 보호

*객체 생성을 제한해야 한다면 Singleton 패턴

*메서드 오버라이드 할 때 해당 메서드 부모의 제한자 범위와 같거나 넓은 범위로만 가능.

- ◆ 외부에서 생성자에 접근 금지 → 생성자의 접근 제한자를 private으로 설정
- ◆ 내부에서는 private에 접근 가능하므로 직접 객체 생성 → 멤버 변수이므로 private 설정
- ◆ 외부에서 private member에 접근 가능한 getter 생성 → setter는 불필요
- ◆ 객체 없이 외부에서 접근할 수 있도록 getter와 변수에 static 추가

```
class SingletonClass{
  private static SingletonClass instance = new SingletonClass();
  private SingletonClass() {}

  public static SingletonClass getInstance() {
    return instance;
  }
```

다형성

오버로딩(overloading)

-메서드 이름은 동일, 파라미터의 개수 또는 순서, 타입이 달라야 함. (리턴 타입 의미 X)

*파라미터가 같으면 중복 선언 오류

-중복 코드에 대한 효율적 관리 가능

```
public void println(int x)
public void println(char x)
public void println(String x)
```

오버라이딩(overriding)

-조상 클래스에 정의된 메서드를 자식 클래스에서 적합하게 수정하는 것

*조건 : 메서드 이름이 같아야 함. 매개변수의 개수, 타입, 순서가 같아야 함.
리턴타입이 같아야함. 접근 제한자는 부모 보다 범위가 넓거나 같아야 함. 조상보다 더 큰 예외를 던질 수 없음.

```
public class Person {
  void jump(){
    System.out.println("두 다리로 힘껏 점프");
  }
}
```

물려받은 jump(). 성능이 좋지 않다.

탐나는 jump()를 가지고 있다.

```
public class Spider {
  void jump(){
    System.out.println("키 * 1000만큼 엄청난 점프");
  }
}
```

다형성

참조형 객체의 형 변환

● 작은 집(child)에서 큰 집 (super) 으로 → 묵시적 케스팅

```
byte b = 10;
int i = b;
```

```
Phone phone = new Phone();
Object obj = phone;
```

◆ 자손 타입의 객체를 조상 타입으로 참조 : 형변환 생략 가능

- 왜냐면 조상의 모든 내용이 자식에 있기 때문에 걱정할 필요가 없다.

● 큰집(super)에서 작은 집 (child) 으로 → 명시적 케스팅

```
int i = 10;
byte b = (byte)i;
```

```
Phone phone = new SmartPhone();
SmartPhone sPhone = (SmartPhone)phone;
```

◆ 조상 타입을 자손 타입으로 참조 : 형변환 생략 불가

조상을 무작정 자손 타입으로 바꿀 수는 없음 -> instanceof 연산자 사용

*instanceof : 실제 메모리에 있는 객체가 특정 클래스 타입인지 boolean으로 리턴

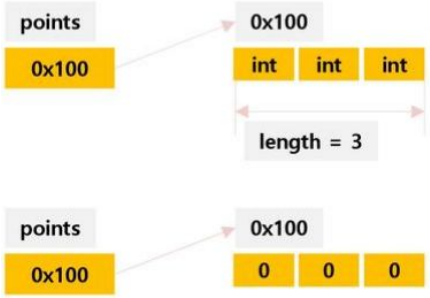
```
Person person = new Person();

if (person instanceof SpiderMan) {
  SpiderMan sman = (SpiderMan) person;
}
```

배열 선언

배열의 생성과 초기화

생성 : new data_type[length]

- new int [3]; int타입의 자료 3개를 저장할 수 있는 배열을 메모리에 생성
- points = new int [3]: 생성된 배열을 points라는 변수에 할당
- points는 메모리에 있는 배열을 가리키는 reference 타입 변수



*배열은 한 번 생성하면 크기 변경 불가! 개별요소도 변경은 가능하지만 삭제는 불가!

초기화 : 배열 생성과 동시에 자료형에 대한 default 초기화 진행

자료형	기본값	비고
boolean	false	
char	'\u0000'	공백문자
byte, short, int	0	
long	0L	
float	0.0f	
double	0.0	
참조형 변수	null	아무것도 참조하지 않음

생성과 동시에 할당한 값으로 초기화

- int [] b = {1, 3, 5, 6, 8};
- int [] c = new int [ ]{1, 3, 5, 6, 8};

선언 후 생성 시 초기화 주의

- int [] points;
  points = {1, 3, 5, 6, 8}; // 컴파일 오류
- int [] points;
  points = new int []{1, 3, 5, 6, 8}; // 선언할 때는 배열의 크기를 알 수 없을 때

자료 파업 중!

알아서 찾아보세요^^

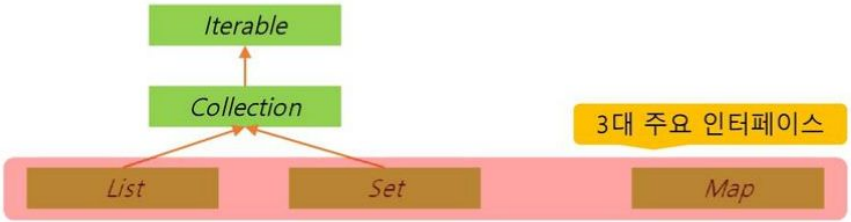
hint : 디폴트 패키지(Object, String)

JAVA API (java.util)

java.util 패키지

◆ 다수의 데이터를 쉽게 처리하는 방법 제공 → DB 처럼 CRDU 기능 중요

collection framework 핵심 interface



interface	특징
List	순서가 있는 데이터의 집합. 순서가 있으니까 데이터의 중복을 허락 ex) 일렬로 줄 서기 ArrayList, LinkedList,
Set	순서를 유지하지 않는 데이터의 집합. 순서가 없어서 같은 데이터를 구별할 수 없음 → 중복 허락 하지 않음 ex) 알파벳이 한 종류 씩 있는 주머니 HashSet, TreeSet...
Map	key와 value의 쌍으로 데이터를 관리하는 집합. 순서는 없고 key의 중복 불가, value는 중복 가능 ex) 속성 – 값, 지역번호-지역 HashMap, TreeMap

예외 처리(Abstract)

추상 클래스(abstract class)

*abstract 클래스는 상속 전용의 클래스

자손 클래스에서 반드시 재정의해서 사용되기 때문에 조상의 구현이 무의미한 메서드

- 메서드의 선언부만 남기고 구현부는 세미콜른(;) 으로 대체
- 구현부가 없다는 의미로 abstract 키워드를 메서드 선언부에 추가
- 객체를 생성할 수 없는 클래스라는 의미로 클래스 선언부에 abstract를 추가한다.

```
abstract class  Vehicle {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)%n", curX, curY);
  }

  public abstract void addFuel();
}
```

*클래스에 구현부가 없는 메서드가 있으므로 객체 생성 불가. (자식 참조는 가능)

```
// Vehicle v = new Vehicle(); // abstract 클래스는 객체를 생성할 수 없다.

Vehicle v = new DieselSUV();  // 자식을 참조하는 것은 문제 없음
```

*조상 클래스에서 상속받은 abstract 메서드를 재정의 하지 않을 경우

-자식 클래스는 abstract 클래스로 선언되어야 함.

*추상 클래스를 사용하는 이유

-구현의 강제를 통해 프로그램 안정성 향상 및 모듈화로 인한 코드의 재사용성 증가

-interface에 있는 메서드 중 구현할 수 있는 메서드를 구현해 개발의 편의 지원

예외 처리(Interface)

인터페이스(interface) 상속

*모든 멤버 변수는 public static final, 모든 메서드는 public abstract이며 생략 가능

클래스와 마찬가지로 인터페이스도 extends를 이용해 상속이 가능

클래스와 다른 점은 인터페이스는 다중 상속이 가능

```
interface Fightable{
  int fire();
}
interface Transformable{
  void changeShape(boolean isHeroMode);
}
public interface Heroable extends Fightable, Transformable{
  void upgrade();
}
```

인터페이스 구현과 객체 참조

클래스에서 implements 키워드를 사용해서 interface 구현

implements 한 클래스는

◆ 모든 abstract 메서드를 override해서 구현하거나

◆ 구현하지 않을 경우 abstract 클래스로 표시해야 함

여러 개의 interface implements 가능

인터페이스의 필요성

- 구현의 강제로 표준화 처리(abstract 메서드 사용)
- 손쉬운 모듈 교체 지원 -> 독립적 프로그래밍 가능 -> 개발 기간 단축
- 서로 상속 관계가 없는 클래스들에게 인터페이스를 통한 관계 부여로 다형성 확장

예외 처리(Interface)

Default Method

- 인터페이스에 선언된 구현부가 있는 일반 메서드

메서드 선언부에 default modifier 추가 후 메서드 구현부 작성

- 접근 제한자는 public으로 한정됨(생략 가능)

```
interface DefaultMethodInterface {
  void abstractMethod();

  default void defaultMethod() {
    System.out.println("이것은 기본 메서드입니다.");
  }
}
```

Default Method의 필요성

- 기존 interface 기반 동작 라이브러리의 interface에 추가해야 하는 기능 발생
- 기존 방식이라면 모든 구현체들이 추가되는 메서드를 override 해야 함.
- Default 메서드는 abstract가 아니므로 반드시 구현할 필요가 없어짐. (override 불필요)

예외 처리(Interface)

- default method의 충돌
  - ◆ JDK 1.7 이하의 java에서는 interface method에 구현부가 없으므로 충돌이 없었음
  - ◆ 1.8 부터 default method가 생기면서 동일한 이름을 갖는 구현부가 있는 메서드가 충돌

method 우선 순위

- super class의 method 우선 : super class가 구체적인 메서드를 갖는 경우 default method는 무시됨
- interface간의 충돌 : 하나의 interface에서 default method를 제공하고 다른 interface에서도 같은 이름의 메서드(default 유무와 무관)가 있을 때 sub class는 반드시 override 해서 충돌 해결!!

Static Method

- 인터페이스에 선언된 static 메서드

일반 static 메서드와 마찬가지로 별도의 객체가 필요 없음

구현체 클래스 없이 바로 인터페이스 이름으로 메서드에 접근해서 사용 가능

```
package ch08.inter.method;

interface StaticMethodInterface{
  static void staticMethod() {
    System.out.println("Static 메서드");
  }
}

public class StaticMethodTest {
  public static void main(String[] args) {
    StaticMethodInterface.staticMethod();
  }
}
```

예외 처리(Exception)

에러와 예외(Error & Exception)

*심각도에 따라 에러와 예외로 분류됨

Error

- 메모리 부족, stack overflow와 같이 일단 발생하면 복구할 수 없는 상황
- 프로그램의 비 정상적 종료를 막을 수 없음 → 디버깅 필요

Exception

- 읽으려는 파일이 없거나 네트워크 연결이 안 되는 등 수습될 수 있는 비교적 상태가 약한 것들
- 프로그램 코드에 의해 수습될 수 있는 상황

예외 처리(Exception Handling)란?

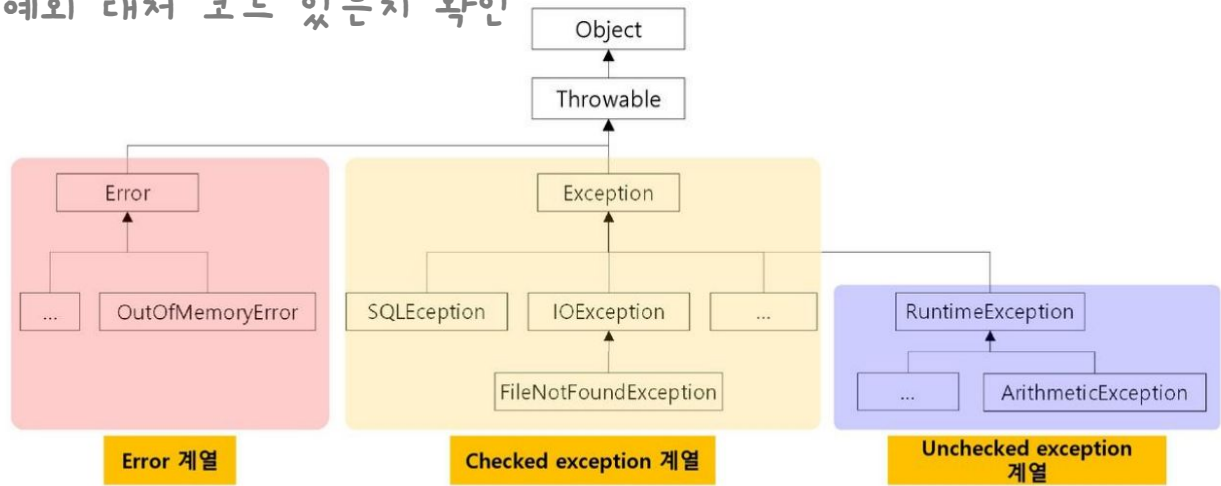
-예외 발생 시 프로그램의 비정상 종료를 막고 정상적인 실행 상태를 유지하는 것

*예외 감지 및 예외 발생 시 동작할 코드 작성 필요!

예외 처리(Exception)

예외 클래스 계층

*컴파일러가 예외 대처 코드 있는지 확인



*Checked Exception

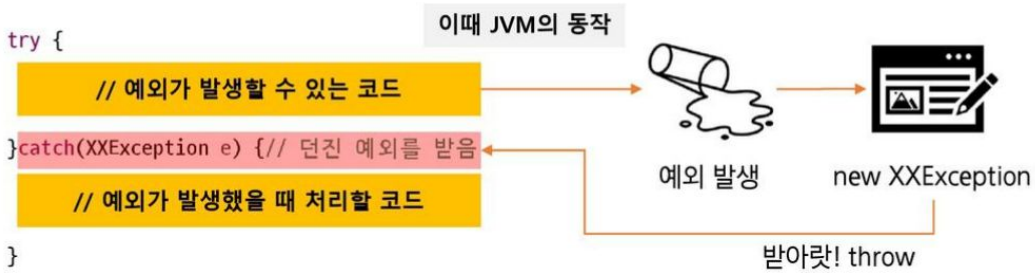
-예외에 대한 대처 코드 없으면 컴파일 진행 X -> 반드시 try~catch 또는 throws 필요

*Unchecked Exception(RuntimeException의 하위 클래스)

-예외 대처 코드가 없어도 컴파일은 진행 -> throws 안해도 되지만 try~catch 처리 필요

예외 처리(Exception)

try~catch 구문



try 블록에서 예외가 발생하면

◆ JVM이 해당 Exception 클래스의 객체 생성 후 던짐(throw)

- throw new XXException()

◆ 던져진 exception 을 처리할 수 있는 catch 블록에서 받은 후 처리

- 적당한 catch 블록을 만나지 못하면 예외처리는 실패

◆ 정상적으로 처리되면 try-catch 블록을 벗어나 다음 문장 진행

try 블록에서 어떠한 예외도 발생하지 않은 경우

◆ catch 문을 거치지 않고 try-catch 블록의 다음 흐름 문장을 실행

*throws 키워드로 처리 위임

- 예외를 호출한 곳으로 전달(처리 위임)
- 예외 삭제 X, 단순 전달 O
- 전달 받은 곳에서 예외 처리 해야 함.

예외 처리(Exception)

finally 구문

- 생성한 시스템 자원을 반납하지 않으면 추후 resource leak 발생 가능 -> close 처리
- finally 구문을 이용해 try 블록에서 사용한 리소스 반납
- 예외 발생 여부과 상관없이 항상 실행 (중간에 return을 만나도 우선 수행 후 return)

Throwable 주요 메서드

메서드	설명
public String getMessage()	발생된 예외에 대한 구체적인 메시지를 반환한다.
public Throwable getCause()	예외의 원인이 되는 Throwable 객체 또는 null을 반환한다.
public void printStackTrace()	예외가 발생된 메서드가 호출되기까지의 메서드 호출 스택을 출력한다. 디버깅의 수단으로 주로 사용된다.

*예외 발생 시 꼭 확인해야 할 정보

- 예외 종류, 예외 원인, 디버깅 출발점

감사합니다~ 얏호!