

Exception Handling & Collection Framework



목차

1. 예외의 처리
 2. throws 활용
 3. 사용자 정의 예외
 4. List 계열
 5. Set 계열
 6. Map 계열
-

Exception Handling

예외의 처 리

❖ 에러와 예외

- 어떤 원인에 의해 오동작 하거나 비정상적으로 종료되는 경우
- 심각도에 따른 분류
 - ◆ 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**
- exception handling(예외 처리)란
 - ◆ 예외 발생 시 프로그램의 비 정상 종료를 막고 정상적인 실행 상태를 유지하는 것



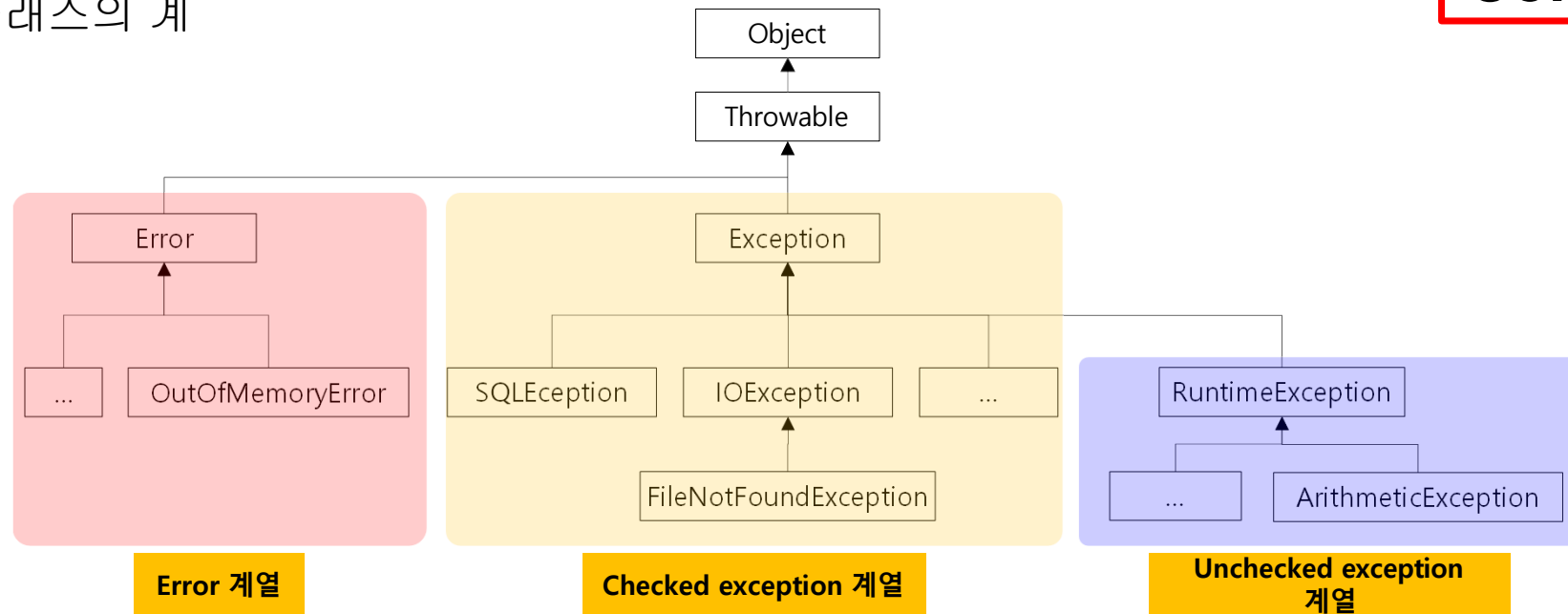
코드 작성 필요



exception handling

Confidential

❖ 예외 클래스의 계층



- checked exception
 - ◆ 예외에 대한 대처 코드가 없으면 컴파일이 진행되지 않음
- unchecked exception (`RuntimeException`의 하위 클래스)
 - ◆ 예외에 대한 대처 코드가 없더라도 컴파일은 진행됨

누가? 컴파일러가

무엇을? 예외 대처 코드가 있는지를



exception handling 기법

Confidential

❖ 예외의 발생

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

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
at ch09.exception.SimpleException.main(SimpleException.java:7)

배열의 범위를 벗어나므로 예외가 발생했고 프로그램이 갑자기 종료되었다.



그게 바로 예외 상황이지!! 컴파일에는 지장이 없었으니까 RuntimeException 계열이다.



exception handling 기법

Confidential

❖ try ~ catch 구문

try {

// 예외가 발생할 수 있는 코드

} catch (XXException e) { // 던진 예외를 받음

// 예외가 발생했을 때 처리할 코드

}

이때 JVM의 동작



예외 발생



new
XXException

받아랏!
throw

```
public static void main(String[] args) {  
    int[] intArray = { 10 };  
    try {  
        System.out.println(intArray[2]);  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println("예외가 발생했지만 처리함: 배열 크기 확인 필요");  
    }  
    System.out.println("프로그램 종료합니다.");  
}
```

예외가 처리되니 프로그램이
정상적으로 종료되는구나.



exception handling 기법

Confidential

❖ Exception 객체의 정보 활용

● Throwable의 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 발생한 메서드가 호출되기까지의 메서드 호출 스택을 출력한다. 디버깅의 수단으로 주로 사용된다.

```
int[] intArray = { 10 };
try {
    System.out.println(intArray[2]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.printf("exception handling: 배열 크기 확인 필요:%s\n", e.getMessage());
    e.printStackTrace();
}
System.out.println("프로그램 종료합니다.");
```

catch에서 아무것도 기록하지 않으면
무슨 일이 일어났는지 알 수가 없겠어.



```
exception handling: 배열 크기 확인 필요:2
java.lang.ArrayIndexOutOfBoundsException: 2
    at ch09.exception.SimpleTryCatch.main(SimpleTryCatch.java:8)
프로그램 종료합니다.
```



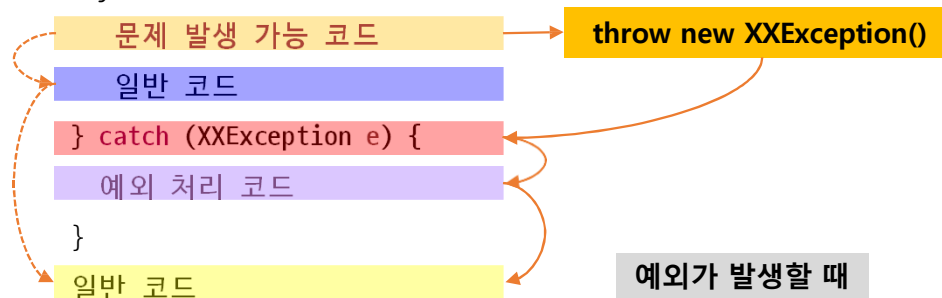
앞선 예제에 대해
exception
handling해보자냥

exception handling 기법

Confidential

❖ try-catch 문에서의 흐름

- 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
예외 발생이 없을 때



❖ try-catch 문에서의 흐름

```
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");  
    } catch (ArithmeticException e) {  
        System.out.println("code 3 - 예외 처리 완료");  
    }  
  
    System.out.println("code 4");  
}
```

- num이 0일 때 출력되는 내용은?
- num이 1일 때 출력되는 내용은?

❖ 다중 exception handling

- try 블록에서 여러 종류의 예외가 발생할 경우

- ◆ 하나의 try 블록에 여러 개의 catch 블록 추가 가능

- 예외 종류별로 catch 블록 구성

```
try {  
    // exception이 발생할 만한 코드  
} catch (XXException e) {  
    // XXException 발생 시 처리 코드  
} catch (YYException e) {  
    // YYException 발생 시 처리 코드  
} catch (Exception e) {  
    // Exception 발생 시 처리 코드  
}
```

CCException 발생
처리 가능?

```
try {  
    // exception이 발생할 만한 코드  
} catch (Exception e) {  
    // Exception 발생 시 처리 코드  
} catch (YYException e) {  
    // YYException 발생 시 처리 코드  
} catch (XXException e) {  
    // XXException 발생 시 처리 코드  
}
```

만약 이 순서라면?

- 다중 catch 문장 작성 순서 유의 사항

- ◆ JVM이 던진 예외는 catch문장을 찾을 때는 다형성이 적용됨
- ◆ 상위 타입의 예외가 먼저 선언되는 경우 뒤에 등장하는 catch 블록은 동작할 기회가 없음
 - Unreachable catch block for Exception.
- ◆ 상속 관계가 없는 경우는 무관
- ◆ 상속 관계에서는 작은 범위(자식)에서 큰 범위(조상)순으로 정의

exception handling 기법

Confidential

❖ 다중 예외 처리를 이용한 Checked Exception 처리

● 처리하지 않으면 컴파일 불가 : Checked Exception

```
Class.forName("abc.Def"); // ClassNotFoundException
new FileInputStream("Hello.java"); // FileNotFoundException
DriverManager.getConnection("Hello"); // SQLException
public static void main(String[] args) {
    System.out.println("프로그램 정상 종료");
}
```

예외 발생 여부가 중요한게 아니라 예외가 발생했을때 어떻게 할꺼냐가 중요

다시한번 checked의 뜻을 상기시켜보면?



● 예외 처리는 가능한 구체적으로 진행

```
public static void main(String[] args) {
    try {
        Class.forName("abc.Def"); // ClassNotFoundException
        new FileInputStream("Hello.java"); // FileNotFoundException
        DriverManager.getConnection("Hello"); // SQLException
    } catch (ClassNotFoundException e) {
        System.out.printf("클래스를 찾을 수 없습니다.: %s\n", e.getMessage());
    } catch (FileNotFoundException e) {
        System.out.printf("파일을 찾을 수 없습니다.: %s\n", e.getMessage());
    } catch (SQLException e) {
        System.out.printf("DB 접속 실패: %s\n", e.getMessage());
    }
    System.out.println("프로그램 정상 종료");
}
```

기존 예제와 다른 점을 생각하고 처리해보개



❖ 다중 예외 처리를 이용한 Checked Exception 처리

● 발생하는 예외들을 하나로 처리하기

```
try {  
    // 다중 예외 발생 코드  
} catch (Exception e) {  
    System.out.printf("예외 발생: %s\n", e.getMessage());  
}
```

◆ 예외 상황 별 처리가 쉽지 않음

◆ 가급적 예외 상황 별로 처리하는 것을 권장

● 심각하지 않은 예외를 굳이 세분화 해서 처리하는 것도 낭비

◆ ‘|’를 이용해 하나의 catch 구문에서 상속관계가 없는 여러 개의 exception 처리

```
public void exceptionHandling() {  
    try {  
        Class.forName("abc.Def");  
        new FileInputStream("Hello.java");  
        DriverManager.getConnection("Hello");  
    } catch (ClassNotFoundException | FileNotFoundException e) {  
        System.out.printf("자원을 찾을 수 없습니다.: %s\n", e.getMessage());  
    } catch (SQLException e) {  
        System.out.printf("DB 접속 실패: %s\n", e.getMessage());  
    }  
    System.out.println("프로그램 정상 종료");  
}
```

exception handling 기법

Confidential

❖ 다중 exception handling을 이용한 Checked Exception 처리

● 계층을 이루는 예외의 처리

```
public class HierarchyException {  
  
    public static void main(String[] args) {  
        String src = "./.project";  
  
        try {  
            FileInputStream input = new FileInputStream(src);  
            int readData = -1;  
  
            while ((readData = input.read()) != -1) {  
                System.out.print((char) readData);  
            }  
        } catch (FileNotFoundException e) {  
            System.out.printf("읽으려는 파일이 없습니다.: %s\n", e.getMessage());  
        } catch (IOException e) {  
            System.out.printf("파일 읽기에 실패했습니다.: %s\n", e.getMessage());  
        }  
        System.out.println("파일 읽음 완료!");  
    }  
}
```

FileNotFoundException의 상속 관계

```
java.lang.Object  
    java.lang.Throwable  
        java.lang.Exception  
            java.io.IOException  
                java.io.FileNotFoundException
```

상속 관계를 잘
려해서 처리해보고개.



exception handling 기법

Confidential

❖ try ~ catch ~ finally 구문을 이용한 예외 처리

- finally는 예외 발생 여부와 상관 없이 언제나 실행
 - ◆ 중간에 return을 만나는 경우도 finally 블록을 먼저 수행 후 리턴 실행

```
try {  
    // exception이 발생할 만한 코드 - System 자원 사용  
} catch (Exception e) {  
    // XXException 발생 시 처리코드  
} finally {  
    // try block에서 접근했던 System자원의 안전한 원상복구  
}  
  
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");  
}
```

num이 0일 경우와 1일
경우 출력되는 내용은?



exception handling 기법

Confidential

❖ try ~ catch ~ finally 구문을 이용한 예외 처리

● finally를 이용한 자원 정리

```
class InstallApp {  
    void copy() {  
        System.out.println("파일 복사");  
    }  
  
    void install() throws Exception {  
        System.out.println("설치");  
        if (Math.random() > 0.5) {  
            throw new Exception();  
        }  
    }  
  
    void delete() {  
        System.out.println("파일 삭제");  
    }  
}
```

```
InstallApp app = new InstallApp();  
try {  
    app.copy();  
    app.install();  
    app.delete();  
} catch (Exception e) {  
    app.delete();  
    e.printStackTrace();  
}
```



```
InstallApp app = new InstallApp();  
try {  
    app.copy();  
    app.install();  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    app.delete();  
}  
System.out.println("설치 종료");
```

```
try {  
    app.copy();  
    app.install();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
app.delete();
```

이렇게 쓰면 안돼?



설치해보러 갈까냥?

예외 처리 기법

Confidential

❖ try ~ catch ~ finally 구문을 이용한 예외 처리

- 주요 목적: try 블록에서 사용한 리소스 반납



abc.txt

```
FileInputStream input = new FileInputStream("abc.txt")
```



file descriptor
- 열린 파일 목록 관리

하지만 무한정 파일을 열수 없기 때문에 OS별로 열 수 있는 파일의 개수가 정해져 있지

다 썼다면 반납이 필요하겠군요!



빌려간 자원을 꼭 반납하개!



- 생성한 시스템 자원을 반납하지 않으면 장래 resource leak 발생 가능 → close 처리

예외 처리 기법

Confidential

❖ try ~ catch ~ finally 구문을 이용한 예외 처리

● finally에서 close를 통한 자원 반납

```
public void useStream() {  
    FileInputStream fileInput = null;  
    try {  
        fileInput = new FileInputStream("abc.txt");  
        fileInput.read();  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        if (fileInput != null) {  
            try {  
                fileInput.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

checked exception 발생 가능

자원 반납

● 지저분할 수밖에 없는 finally 블록

- ◆ close 메서드 자체가 IOException 유발 가능
- ◆ FileInputStream 생성자에서 IOException 발생 시 fileInput은 null인 상황

❖ try-with-resources

- JDK 1.7 이상에서 리소스의 자동 close 처리

```
try(리소스_타입1 res1 = 초기화; 리소스_타입2 res2 = 초기화; ...){  
    // 예외 발생 코드  
}catch(Exception e) {  
    // exception handling 코드  
}
```

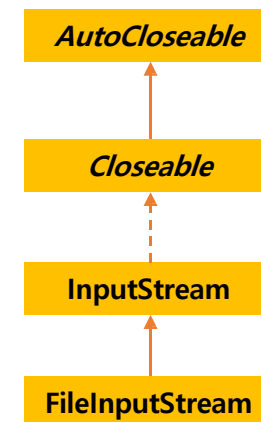
- try 선언문에 선언된 객체들에 대해 자동 close 호출(finally 역할)

- ◆ 단 해당 객체들이 AutoCloseable interface를 구현할 것

- 각종 I/O stream, socket, connection ...

- ◆ 해당 객체는 try 블록에서 다시 할당될 수 없음

```
public void useStreamNewStyle() {  
    try (FileInputStream fileInput = new FileInputStream("abc.txt")) {  
        System.out.println("FileInputStream이 생성되었습니다.");  
        fileInput.read();  
    } catch (IOException e) {  
        System.out.println("파일 처리 실패");  
    }  
}
```



함께가요 미래로!
Enabling People

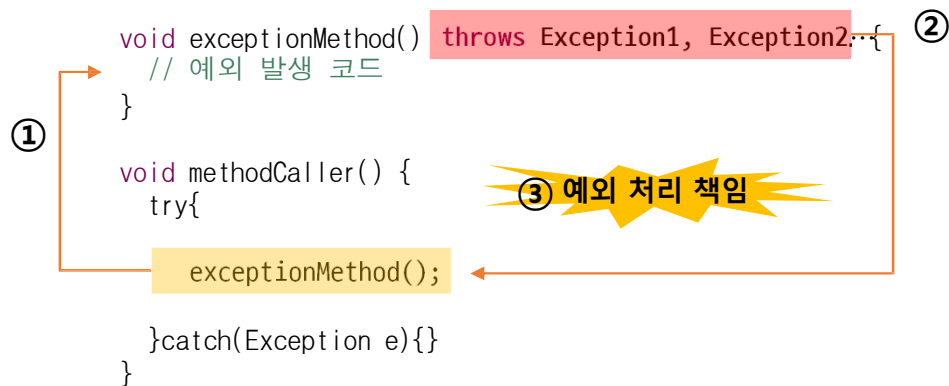
throws 활
공

throws 키워드를 통한 처리 위임

Confidential

❖ throws 키워드를 통한 처리 위임

- method에서 처리해야 할 하나 이상의 예외를 **호출한 곳으로** 전달(처리 위임)
- ◆ 예외가 없어지는 것이 아니라 단순히 전달됨
- ◆ 예외를 전달받은 메서드는 다시 예외 처리의 책임 발생



- ◆ 처리하려는 예외의 조상 타입으로 throws 처리 가능

throws 키워드를 통한 처리 위

Confidential

❖ checked exception과 throws

```
public static void main(String[] args) {
    CheckedThrowsTest et = new CheckedThrowsTest();
    try {
        et.method1();
    } catch (ClassNotFoundException e) {
        System.out.printf("exception handling: %s\n", e.getMessage());
    }
    System.out.println("프로그램 종료");
}

public void method1() throws ClassNotFoundException {
    method2();
}

public void method2() throws ClassNotFoundException {
    Class.forName("Some Class");
}
```

ClassNotFoundException 발생

- checked exception은 반드시 try~catch 또는 throws 필요
- 필요한 곳에서 try~catch 처리

throws 키워드를 통한 처리 위

Confidential

❖ runtime exception과 throws

```
public static void main(String[] args) {  
    RuntimeThrowsTest et = new RuntimeThrowsTest();  
    try {  
        et.method1();  
    } catch (ArithmeticException e) {  
        System.out.printf("예외 처리: %s\n", e.getMessage());  
    }  
    System.out.println("프로그램 종료");  
}
```

```
public void method1() {  
    method2();  
}
```

```
public void method2() {
```

```
    int i = 1/0;
```

```
}
```

ArithmeticException 발생

- runtime exception은 throws 하지 않아도 전달되지만
- 하지만 결국은 try~catch로 처리해야 함



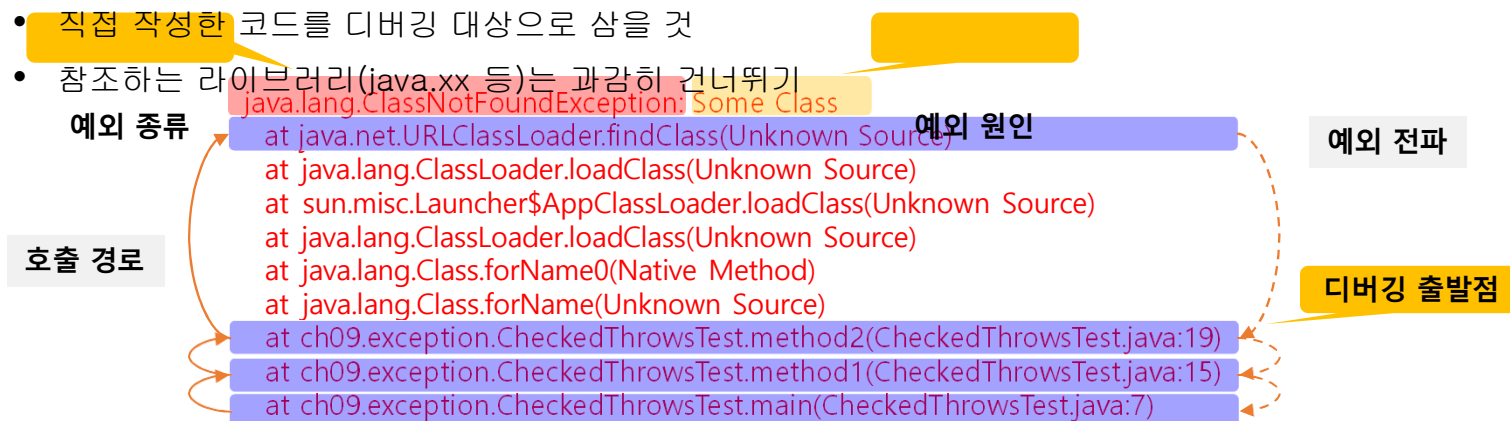
예외 전파를
확인해보자냥.

throws 키워드를 통한 처리 위

Confidential

❖ 로그 분석과 예외의 추적

- Throwable의 printStackTrace는 메서드 호출 스택 정보 조회 가능
 - ◆ 최초 호출 메서드에서부터 예외 발생 메서드까지의 스택 정보 출력
- 꼭 확인해야 할 정보
 - ◆ 어떤 예외인가? - 예외 종류
 - ◆ 예외 객체의 메시지는 무엇인가? - 예외 원인
 - ◆ 어디서 발생했는가? - 디버깅 출발점



throws 키워드를 통한 처리 위

Confidential

❖ throws의 목적과 API 활

• FileInputStream

```
public FileInputStream(String name) throws FileNotFoundException
```

Creates a `FileInputStream` by opening a connection to an actual file, the file named by the path name `name` in the file system. A new `FileDescriptor` object is created to represent this file connection.

First, if there is a security manager, `FileInputStream` checks that the caller has permission to open the file named by the `name` argument as its argument. If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading then a `FileNotFoundException` is thrown.

Parameters:

`name` - the system-dependent file name.

Throws:

[FileNotFoundException](#) - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

[SecurityException](#) - if a security manager exists and its `checkRead` method denies read access to the file.

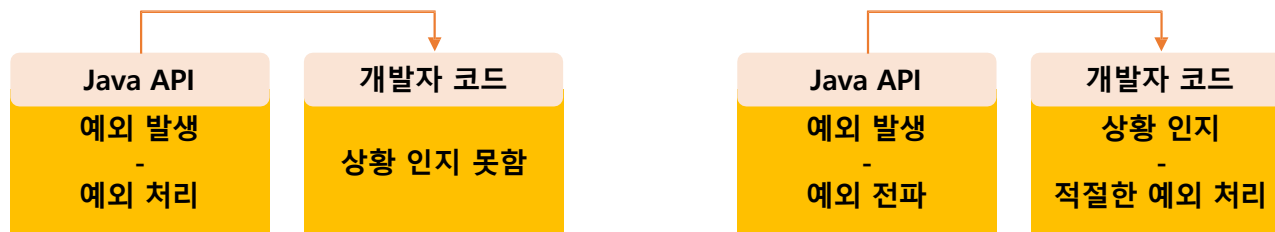
checked exception

runtime exception

오라클 개발자들이 예외 처리 방법을 몰랐나?



- API가 제공하는 많은 메서드들은 사전에 예외가 발생할 수 있음을 선언부에 명시하고 프로그래머가 그 예외에 대처 하도록 강요함



throws 키워드를 통한 처리 위

Confidential

❖ 메서드 재정의와 throws

- 메서드 재정의 시 조상클래스 메서드가 던지는 예외보다 부모예외를 던질 수 없다.
- ◆ 부모가 치지 않은 사고를 자식이 칠 수 없다.

```
class Parent{  
    void methodA() throws IOException{}  
    void methodB() throws ClassNotFoundException{}  
}  
public class OverridingTest extends Parent {  
  
    @Override  
    void methodA() throws FileNotFoundException {  
  
    }  
  
    @Override  
    void methodB() throws Exception {  
  
    }  
}
```

어디가 잘못된
곳일까?



사용자 정의 예 외

❖ 사용자 정의 예외

- API에 정의된 exception이외에 필요에 따라 사용자 정의 예외 클래스 작성
- 대부분 Exception 또는 RuntimeException 클래스를 상속받아 작성
 - ◆ checked exception 활용: 명시적 예외 처리 또는 throws 필요
 - 코드는 복잡해지지만 처리 누락 등 오류 발생 가능성은 줄어듦
 - ◆ runtime exception 활용: 묵시적 예외 처리 가능
 - 코드가 간결해지지만 예외 처리 누락 가능성 발생
- 사용자 정의 예외를 만들어 처리하는 장점
 - ◆ 객체의 활용 - 필요한 추가정보, 기능 활용 가능
 - ◆ 코드의 재사용 - 동일한 상황에서 예외 객체 재사용 가능
 - ◆ throws 메커니즘의 이용 - 중간 호출 단계에서 return 불필요

사용자 정의 예외

Confidential

❖ 사용자 정의 예외 작성

```
public class UserExceptionTest {  
    private static String[] fruits = {"사과", "오렌지", "토마토"};  
    public static void main(String[] args) {  
        boolean result = getFruit1("사과");  
        if(!result) {  
            System.out.println("사과는 없습니다.");  
        }  
        result = getFruit1("사과");  
        if(!result) {  
            System.out.println("사과는 없습니다.");  
        }  
        System.out.println("참고 관리 끝~");  
    }  
    private static boolean getFruit1(String name) {  
        for (int i = 0; i < fruits.length; i++) {  
            if (fruits[i] != null && fruits[i].equals(name)) {  
                fruits[i] = null;  
                return true;  
            }  
        }  
        return false;  
    }  
}
```



과일이 없는 상태를
예외로 만들어보자냥!

사용자 정의 예외

Confidential

❖ 사용자 정의 예외 작성

```
class FruitNotFoundException extends Exception {  
    public FruitNotFoundException(String name) {  
        super(name + "에 해당하는 과일은 없습니다.");  
    }  
}  
  
public class CustomExceptionTest {  
    private static String[] fruits = {"사과", "오렌지", "토마토"};  
    public static void main(String[] args) {  
        try {  
            getFruit2("오렌지");  
            getFruit2("오렌지");  
        } catch (FruitNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
  
    private static void getFruit2(String name) throws FruitNotFoundException {  
        for (int i = 0; i < fruits.length; i++) {  
            if (fruits[i] != null && fruits[i].equals(name)) {  
                fruits[i] = null;  
                return;  
            }  
        }  
        throw new FruitNotFoundException(name);  
    }  
}
```

사용자 정의 예외 활용의 장점?

객체의 활용, 코드의 재사용, 전파 메커니즘



배열의 빈 칸에 과일을 추가하는데 놓을 곳이 없을 경우 예외로 처리해보자냥.

당신의 생각은?

❖ 아래 코드에 대한 여러분의 생각은?

```
public void method(String[] args, String name, int divisor) {  
    try {  
        System.out.println(args[0]);  
  
        System.out.println(name.length());  
  
        System.out.println(1 / divisor);  
    } catch (Exception e) {  
        // 췌!!!  
    }  
}
```

```
public void method2(String[] args, String name, int divisor) {  
    if (args != null && args.length > 1) {  
        System.out.println(args[0]);  
    }  
  
    if (name != null) {  
        System.out.println(name.length());  
    }  
  
    if (divisor != 0) {  
        System.out.println(1 / divisor);  
    }  
}
```

Confidential

많은 경우 시스템의
RuntimeException은 처리의 대상이기
보다는 디버깅의 대상이다.



Collection Framework

List 계 열

❖ 자료 구조

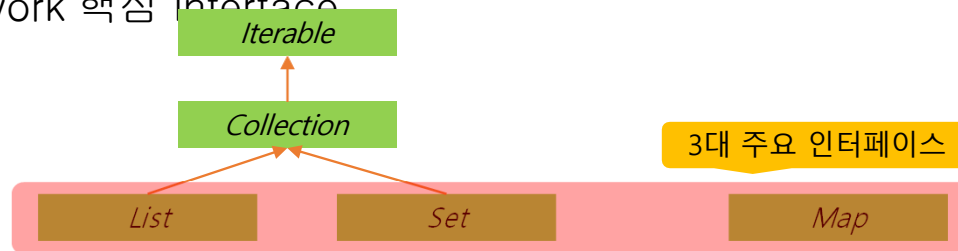
- 위키에 따르면
- **자료구조(data structure)**는 컴퓨터 과학에서 효율적인 접근 및 수정을 가능케 하는 자료의 조직, 관리, 저장을 의미한다. 더 정확히 말해, 자료 구조는 데이터 값의 모임, 또 데이터 간의 관계, 그리고 데이터에 적용할 수 있는 함수나 명령을 의미한다.

❖ 배열

- 가장 기본적인 자료 구조
- homogeneous collection: 동일한 데이터 타입만 관리 가능
 - ◆ 타입이 다른 객체를 관리하기 위해서는 매번 다른 배열 필요
- Polymorphism
 - ◆ Object를 이용하면 모든 객체 참조 가능 → Collection Framework
 - ◆ 담을 때는 편리하지만 빼낼 때는 Object로만 가져올 수 있음
 - ◆ 런타임에 실제 객체의 타입 확인 후 사용해야 하는 번거로움
- Generic을 이용한 타입 한정
 - ◆ 컴파일 타임에 저장하려는 타입 제한 → 형변환의 번거로움 제거

❖ Collection Framework

- java.util 패키지
 - ◆ 다수의 데이터를 쉽게 처리하는 방법 제공 → DB 처럼 CRUD 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

❖ Collection interface

분류	Collection
추가	add(E e), addAll(Collection<? extends E> c)
조회	contains(Object o), c containsAll(Collection<?> c), equals(),
	isEmpty(), iterator(), size(),
삭제	clear(), removeAll(Collection<?> c), retainAll(Collection<?> c),
수정	
기타	toArray()

Collection을 사용할 줄
 Colle 것은 특성에 맞게
 안다 추가 / 수정 / 삭
 자료 제 / 수 있다는 이
 조회할 야기

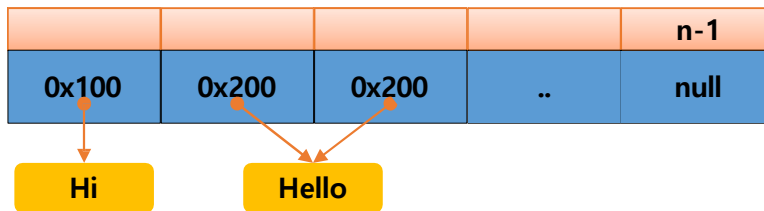


Collection Framework – List

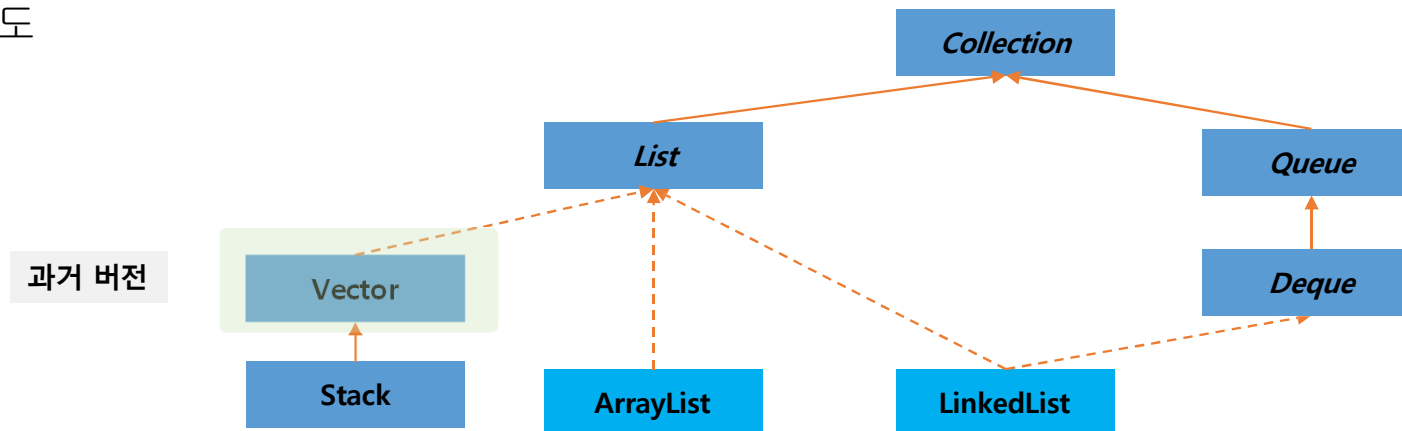
Confidential

❖ 특징

- 순서가 있는 데이터의 집합
- 순서가 있으므로 데이터의 중복을 허락



- 관련 클래스 관계도



Collection Framework - List

Confidential

❖ 주요 메서드

분류	Collection	List
추가	add(E e), addAll(Collection<? extends E> c)	add(int index, E element), addAll(int index, Collection<? extends E> c)
조회	contains(Object o), containsAll(Collection<?> c), equals(), isEmpty(), iterator(), size(),	get(int index), indexOf(Object o), lastIndexOf(Object o), listIterator(),
삭제	clear(), removeAll(Collection<?> c), retainAll(Collection<?> c),	remove(int index)
수정		set(int index, E element)
기타	toArray()	subList(int fromIndex, int toIndex)

List를 이용해서
구이름을 관리해보려고.



❖ 주요 메서드

```
List<String> friends = new ArrayList<>();  
  
public void createTest() {  
    friends.add("홍길동");  
    friends.add("홍길동");    // 동일 데이터 추가  
    friends.add("장길산");  
    friends.add("임꺽정");  
    friends.add(0, "이몽룡"); // 끼워넣기  
    System.out.println("추가 후 내용 출력: " + friends);  
}
```

내부적으로 배열이라며..
크기는??



Collection Framework - List

Confidential

❖ 소스 분석

● constructor

```
public ArrayList(int initialCapacity) {  
    if (initialCapacity > 0) {  
        this.elementData = new Object[initialCapacity];  
    } else if (initialCapacity == 0) {  
        this.elementData = EMPTY_ELEMENTDATA;  
    } else {  
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);  
    }  
}  
  
public ArrayList() {  
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA; // {}  
}
```

내부적으로 Object []에 저장

애초에 크기를 안다면 미리 지정하는 것도 좋겠다.



● add → ensureCapacityInternal → ensureExplicitCapacity → grow

```
private void grow(int minCapacity) {  
    // overflow-conscious code  
    int oldCapacity = elementData.length;  
    int newCapacity = oldCapacity + (oldCapacity >> 1);  
    if (newCapacity - minCapacity < 0)  
        newCapacity = minCapacity;  
    if (newCapacity - MAX_ARRAY_SIZE > 0)  
        // minCapacity is usually close to size, so this is a win:  
        newCapacity = Integer.MAX_VALUE;  
    elementData = Arrays.copyOf(elementData, newCapacity);  
}
```

칸이 부족하면 알아서 늘여주고

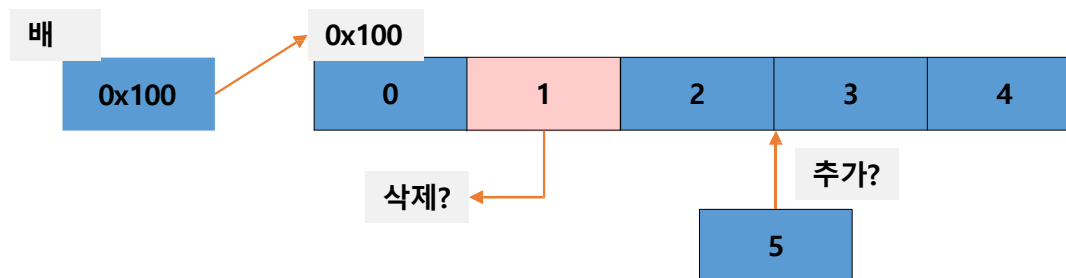
❖ 배열과 ArrayList

● 배열의 장점

- ◆ 가장 기본적인 형태의 자료 구조로 간단하며 사용이 쉬움
- ◆ 접근 속도가 빠름

● 배열의 단점

- ◆ 크기를 변경할 수 없어 추가 데이터를 위해 새로운 배열을 만들고 복사해야 함
- ◆ 비 순차적 데이터의 추가, 삭제에 많은 시간이 걸림

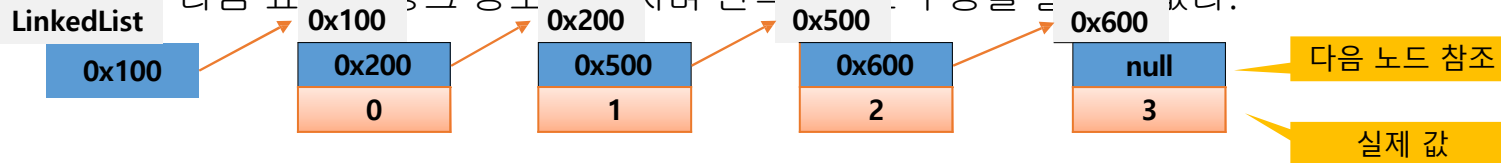


- 배열을 사용하는 ArrayList도 태생적으로 배열의 장-단점을 그대로 가져감

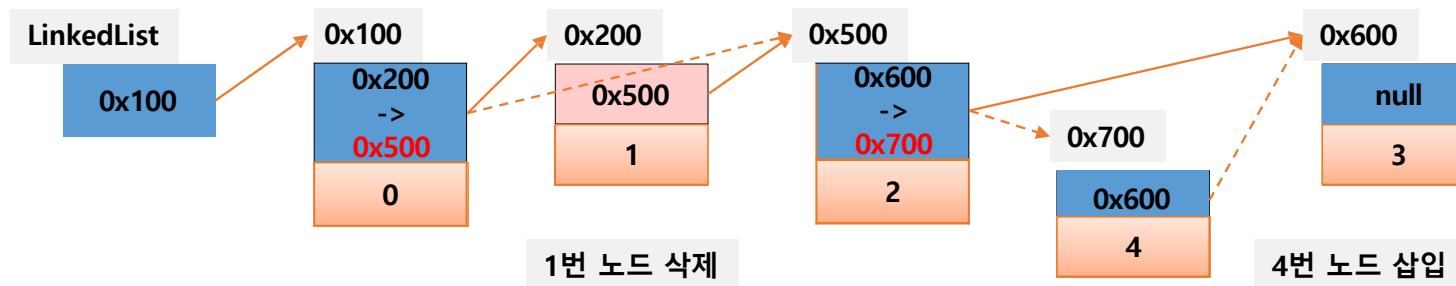
❖ LinkedList

- 각 요소를 Node로 정의하고 Node는 다음 요소의 참조 값과 데이터로 구성됨

◆ 각 요소가 다음 요소의 링크 정보를 가지며 연속적으로 구성될 필요가 없다.



- 데이터 삭제 및 추가



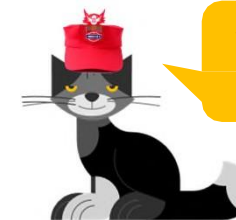
❖ LinkedList 와 ArrayList의 용

구분	순차 추가/수정/삭제 100만건 추가 시	비 순차 추가/수정/삭제 10만건 추가 시	조회 10만 건 조회 시
ArrayList	빠름	느림	빠름
	22,108,711	31,444,190,835	6,428,554
LinkedList	느림	빠름	느림
	84,540,898	5,900,334	20,895,705,164

단위: 나노초

● 결론

- ◆ 특정 클래스가 좋고 나쁨이 아니라 용도에 적합하게 사용
- ◆ ~~해당~~ ^{데이터}의 데이터를 가지고 사용할 경우는 큰 차이가 없음
- ◆ 정적인 데이터 활용, 단순한 데이터 조회용 : ArrayList
- ◆ 동적인 데이터 추가, 삭제가 많은 작업 : LinkedList



기존 예제를 LinkedList로 바꿔서 테스트 하는데 20초 준다냥.

❖ 자료 삭제 시 주의 사항

● index를 이용한 for 문

```
List<Integer> nums = new ArrayList<>();
Random rand = new Random();

for(int i=0; i<10; i++) {
    nums.add(rand.nextInt(20));
}
System.out.println("전체: "+nums);

for(int i=0; i<nums.size(); i++) {
    if(nums.get(i)%2==0) {
        nums.remove(i);
        i--;
    }
}
System.out.println("짝수 삭제 후: "+nums);
```

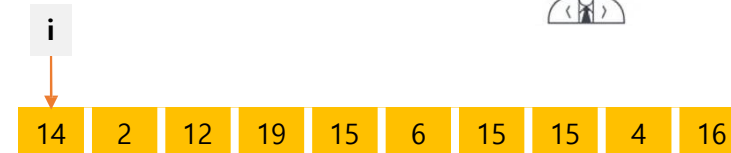
◆ 요소가 삭제되면 size가 줄어들기 때문에 index 차감 필요

◆ 거꾸로 접근하면 자연스럽게 해결

```
for(int i=nums.size()-1; i>=0; i--) {
    if(nums.get(i)%2==1) {
        nums.remove(i);
    }
}
```

전체: [14, 2, 12, 19, 15, 6, 15, 15, 4, 16]
짝수 삭제 후: [2, 19, 15, 15, 15, 16]

왜 남아있을까?



역발상이 필요해!



❖ 자료 삭제 시 주의 사항

- forEach 문장은 read only!!

```
public class ForEachTest {  
    public static void main(String[] args) {  
        List<Integer> nums = new ArrayList<>();  
        for(int i=0; i<10; i++) {  
            nums.add(i);  
        }  
        System.out.println("시작: "+nums);  
        for(Integer num: nums) {  
            if(num%2==0) {  
                // nums.add(num*num);  
                nums.remove(num);  
            }  
        }  
    }  
}
```

시작: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Exception in thread "main" java.util.ConcurrentModificationException
at
java.util.ArrayList\$Itr.checkForComodification(ArrayList.java:909)
at java.util.ArrayList\$Itr.next(ArrayList.java:859)
at
com.ssafy.day5.collection.ForEachTest.main(ForEachTest.java:19)

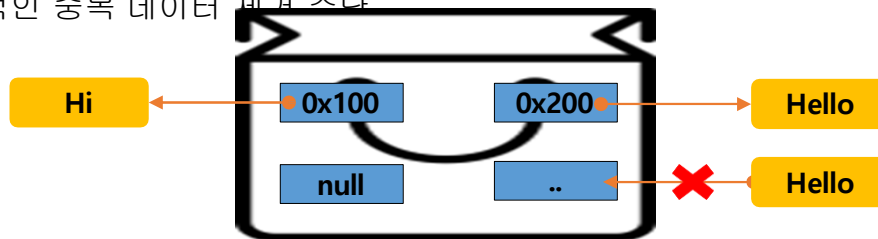
함께가요 미래로!
Enabling People

Set 계
열

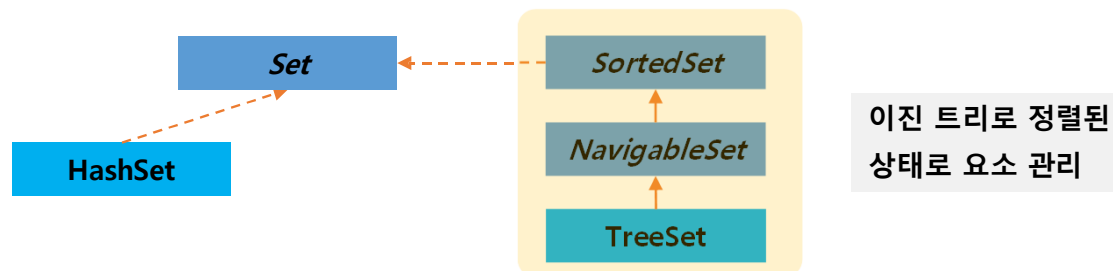
❖ Set interface

● 특징

- ◆ 순서 없이 주머니에 구슬(데이터)을 넣는 형태
- ◆ 순서가 없으므로 데이터를 구별할 index가 없어 중복이 허용되지 않는다.
 - 효율적인 중복 데이터 제거 수단



● 관련 클래스 관계도



Collection Framework - Set

Confidential

❖ Set API 활용

```
public class SetTest {  
    Set<Object> hset = new HashSet<Object>();  
    private void addMethod() {  
        hset.add(new Integer(1));  
        hset.add("Hello");  
        hset.add("Hello");    // 동일한 데이터 추가 확인  
        hset.add(1);          // 기본형은 wrapper를 통해 추가  
        System.out.println("데이터 추가 결과: " + hset);  
    }  
    private void retrieveMethod() {  
        System.out.println("데이터 개수: " + hset.size());  
  
        for (Object sobj : hset) {  
            System.out.println("데이터 조회: " + sobj);  
        }  
    }  
    private void removeMethod() {  
        hset.remove("Hello"); System.out.println("데이터 삭제 결과: " + hset);  
    }  
    public static void main(String[] args) {  
        SetTest test = new SetTest();  
        test.addMethod();  
        test.retrieveMethod();  
        test.removeMethod();  
    }  
}
```

데이터 추가 결과: [1, Hello]
데이터 개수: 2
데이터 조회: 1 데이터 조회: Hello
데이터 삭제 결과: [1]

동일한 데이터가 없어!



넣은 순서대로 출력되지도 않았어!!

update는 왜 없죠?



Collection Framework - Set

Confidential

❖ 동일한 데이터의 기준

● SmartPhone도 넣어보자.

```
class SmartPhone {  
    String number;  
    public SmartPhone(String number) {  
        this.number = number;  
    }  
  
    public String toString() {  
        return "전화 번호: " + number;  
    }  
}
```

번호 같은
전화는 없지?



```
hset.add(new SmartPhone("010-111-2222"));  
hset.add(new SmartPhone("010-111-2222"));
```

데이터 조회: 1 데
이터 조회: Hello
데이터 조회: 전화 번호: 010-111-2222
데이터 조회: 전화 번호: 010-111-2222

● 동일한 데이터의 기준은 객체의 equals()가 true를 리턴하고 hashCode() 값이 같은 것

```
public boolean equals(Object obj) {  
    boolean result = false;  
    if (obj != null && obj instanceof SamePhone) {  
        SamePhone param = (SamePhone) obj;  
        result = (this.number.equals(param.number));  
    }  
    return result;  
}  
  
public int hashCode() {  
    return number.hashCode();  
}
```

데이터 조회: 1
데이터 조회: 전화 번호: 010-111-2222
데이터 조회: Hello

동일한 번호의
SmartPhone이 등록되지
않도록 처리해보개.



함께가요 미래로!
Enabling People

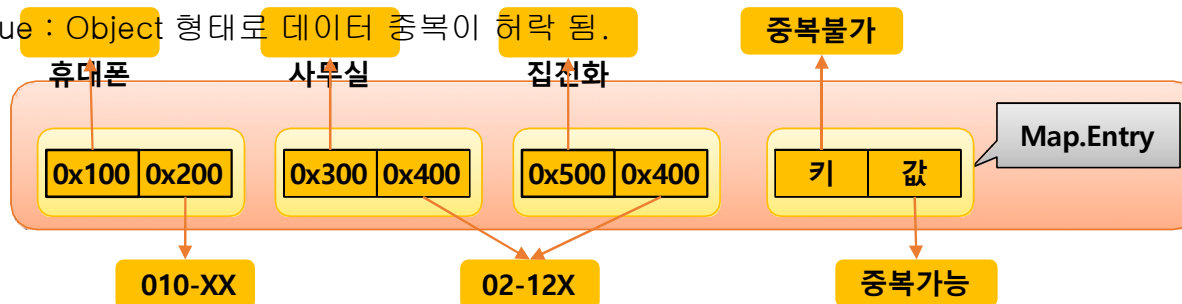
Map 계
영

❖ Map interface

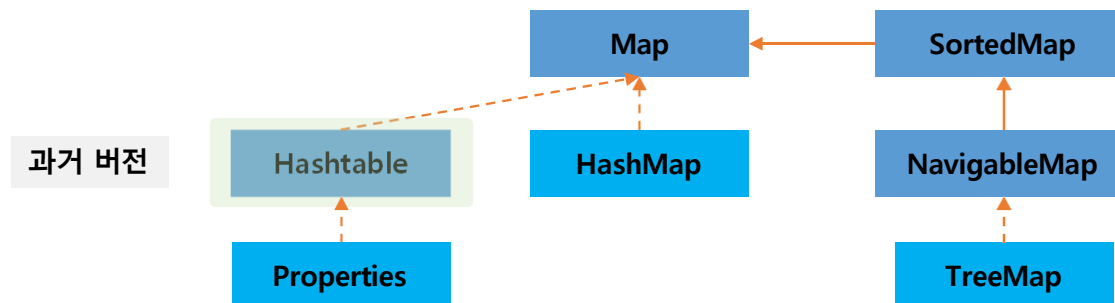
● 특징

◆ Key와 Value를 하나의 Entry로 묶어서 데이터 관리

- Key : Object 형태로 데이터 중복을 허락하지 않음
- Value : Object 형태로 데이터 중복이 허락 됨.



● 관련 클래스 관계도



❖ Map interface의 주요 메서드

분류	Map<K, V>
추가	put(K key, V value) putAll(Map<? extends K, ? extends V> m)
조회	containsKey(Object key) containsValue(Object value) entrySet() keySet() get(Object key) values() size(); isEmpty();
삭제	clear(), remove(Object key)
수정	put(K key, V value) putAll(Map<? extends K, ? extends V> m)

key와 entry가 리
Set으로 관된다니..



Collection Framework - Map

Confidential

❖ Map interface의 주요 메서드

```
Map<String, String> hMap = new HashMap<>();
```

```
private void addMethod() {  
    System.out.println("추가 성공?: " + hMap.put("andy", "1234"));  
    // 동일한 키의 사용 결과는?  
    System.out.println("추가 성공?: " + hMap.put("andy", "4567"));  
  
    hMap.put("kate", "9999");  
  
    // 기존에 해당 키에 대한 값이 없을 때만 추가하기  
    hMap.putIfAbsent("kate", "1234");  
  
    hMap.put("henry", "4567");  
  
    System.out.println("추가 결과: "+hMap);  
}
```

// 동일한 값

따로 업데이트가 필요
없긴한데 조심해야겠어!!



추가 성공?:
null 추가 성공
?: 1234
추가 결과: {henry=4567, kate=9999,
andy=4567}



조회와 삭제도
테스트해보자냥~

❖ 정렬

- 요소를 특정 기준에 대한 내림차순 또는 오름 차순으로 배치하는 것
- 순서를 가지는 Collection들만 정렬 가능
 - ◆ List 계열
 - ◆ Set에서는 SortedSet의 자식 객체
 - ◆ Map에서는 SortedMap의 자식 객체(key 기준)
- Collections의 sort()를 이용한 정렬
 - ◆ sort(List<T> list)
 - 객체가 Comparable을 구현하고 있는 경우 내장 알고리즘을 통해 정렬

```
private List<String> names = Arrays.asList("Hi", "Java", "World", "Welcome");  
public void basicSort() {  
    Collections.sort(names);  
    System.out.println(names); // [Hi, Java, Welcome, World]  
}
```


SmartPhone도
정렬해볼까?



❖ 정렬

● SmartPhone 정렬 처리

```
public void sortPhone() {  
    List<SmartPhone> phones = Arrays.asList(new SmartPhone("010"), new SmartPhone("011"), new SmartPhone("000"));  
    Collections.sort(phones);  
}
```

 The method sort(List<T>) in the type Collections is not applicable for the arguments (List<SmartPhone>)

```
@SuppressWarnings("unchecked")  
public static <T extends Comparable<? super T>> void sort(List<T> list) {  
    list.sort(null);  
}
```

Comparable 을 상속
받은 녀석만 가능해!



SmartPhone은 크기를
비교할 수 없었구나!

● 언제나 정렬 가능한 요소들

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence { ...}  
  
public final class Integer extends Number implements Comparable<Integer> { ...}
```


❖ 정렬

● Comparable interface

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

양수: 자리 바꿈
음수: 자리 유지
0: 동일 위치

반환값의 크기가 문제가 아니라 부호가 문제!!



● SmartPhone 정렬 처리

```
public class SmartPhone implements Comparable<SmartPhone>{  
    String number;  
  
    . . .  
    @Override  
    public int compareTo(SmartPhone o) {  
        // 사전에 비교 가능한 기준이 있다면 그것들을 재사용하기  
        return this.number.compareTo(o.number);  
    }  
}
```

```
public void sortPhone() {  
    List<SmartPhone> phones = Arrays.asList(new SmartPhone("010"), new SmartPhone("011"), new SmartPhone("000"));  
    Collections.sort(phones);  
    System.out.println(phones); // [전화 번호: 000, 전화 번호: 010, 전화 번호: 011]  
}
```

❖ Comparator의 활용

- 객체가 Comparable을 구현하고 있지 않거나 사용자 정의 알고리즘으로 정렬하려는 경우
 - ◆ String을 알파벳 순이 아닌 글자 수 별로 정렬하려면?
- `sort(List<T> list, Comparator<? Super T> c)`

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}  
  
public class StringLengthComparator implements Comparator<String> {  
    @Override  
    public int compare(String o1, String o2) {  
        int len1 = o1.length();  
        int len2 = o2.length();  
        return Integer.compare(len1, len2);  
    }  
}
```

양수: 자리 바꿈
음수: 자리 유지
0: 동일 위치

```
public void stringLengthSort() {  
    Collections.sort(names, new StringLengthComparator());  
    System.out.println(names); // [Hi, Java, World, Welcome]  
}
```

그런데 StringLengthComparator
는 여기 저기서 쓸것 같지는 않아

..



❖ Comparator의 활용

- 1회성 객체 사용 시 anonymous inner class 사용

◆ 클래스 정의, 객체 생성을 한번에 처리

```
// before
Collections.sort(names, new StringLengthComparator());
// after
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return Integer.compare(o1.length(), o2.length());
    }
});
```

◆ 람다 표현식 이용

```
Collections.sort(names, (o1, o2)->{
    return Integer.compare(o1.length(), o2.length());
});
```

문자열 길이의
내림차순으로
정렬해보개.

