

# 자바 프로그래밍 기초

전병선

# 자바 프로그래밍 기초

---

1. Hello 프로그램
2. 데이터 타입
3. 연산자
4. 제어문
5. 클래스 기초
6. 상속성
7. 다형성과 인터페이스
8. 클래스 고급
9. 제네릭
10. 예외 처리
11. 내부 클래스와 람다 표현식
12. 컬렉션 프레임워크

# 1. Hello 프로그램

# 1. Hello 프로그램

---

- Java 프로그래밍 환경 갖추기
- Hello 프로그램 작성
- 기본적인 Java 프로그램 구성 요소

# 1. Hello 프로그램

---

- Java 프로그래밍 환경 갖추기
- Hello 프로그램 작성
- 기본적인 Java 프로그램 구성 요소

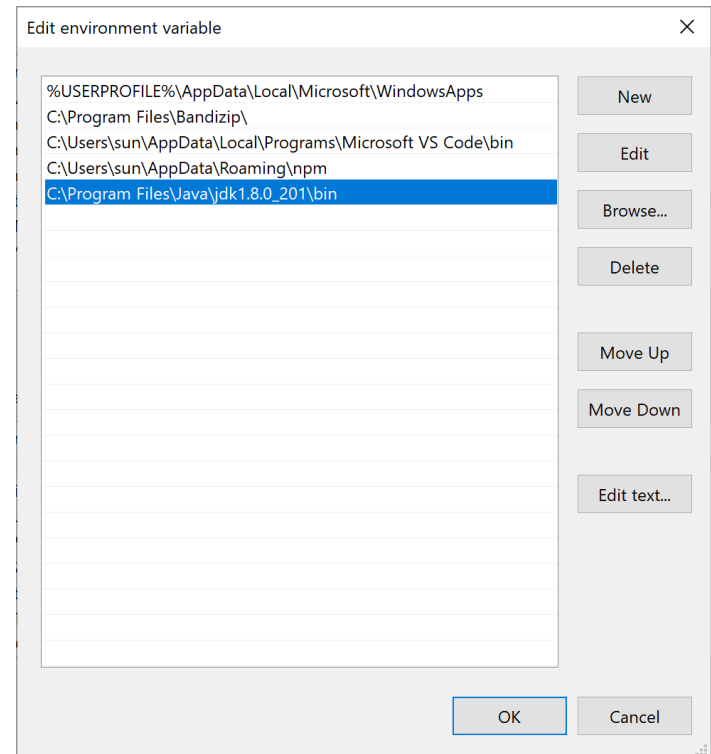
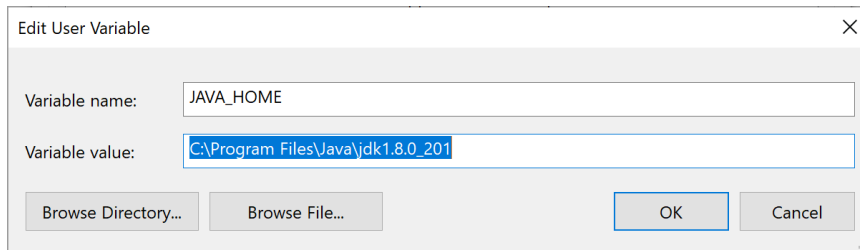
# Java 프로그래밍 환경 갖추기

---

- Java 개발 도구
  - JDK(Java Development Kit)
    - Java 컴파일러
    - Java 가상 머신(VM, Virtual Machine)
  - Eclipse
    - Java 개발 환경

# Java 프로그래밍 환경 갖추기

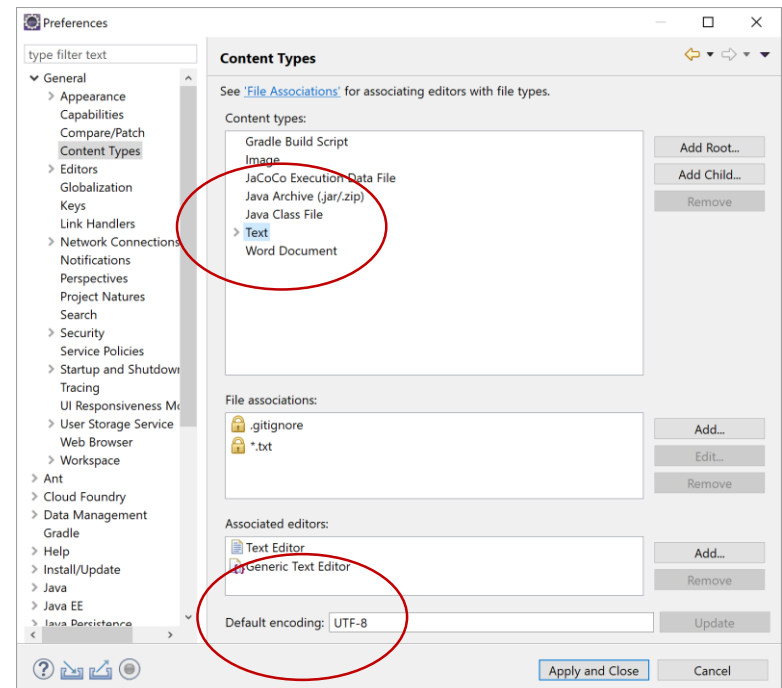
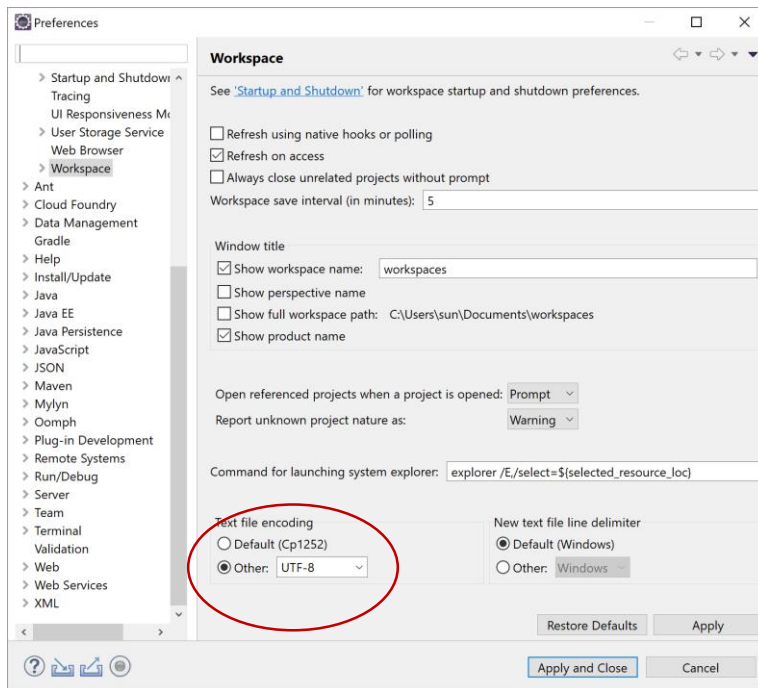
- JDK 다운로드 및 설치
- JDK 환경 설정



- Eclipse 다운로드 및 설치

# Java 프로그래밍 환경 갖추기

- Eclipse 환경 설정





# Hello 프로그램 작성

---

- Hello 프로그램 코드

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 키워드(keyword)
  - 프로그램 안에서 특별한 의미를 부여한 미리 정의된 이름(예약어)
  - 반드시 Java 언어에서 미리 정의한 의미대로만 사용해야 한다
  - class, package, public, static, void 등...

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 클래스(class)
  - 가장 중요한 프로그램 구성 요소
  - Java 프로그램은 하나 이상의 여러 클래스로 구성된다
  - 클래스는 메서드와 필드를 멤버로 포함한다
  - 예 : Hello 클래스

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 메서드(method)
  - 어떤 작업을 수행하는 코드의 묶음
- main() 메서드
  - Java 프로그램에서 main() 메서드를 호출함으로써 프로그램의 실행을 시작한다
  - main() 메서드의 작업이 끝나면 프로그램이 끝난다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- **public**
  - 공개
  - main() 메서드를 외부에서 호출할 수 있다는 의미가 된다
- **static**
  - 정적
  - **static** 키워드가 지정된 멤버는 클래스의 인스턴스 즉, 객체를 생성하지 않고도 사용될 수 있다
- **void**
  - 아무 것도 없음
  - 메서드가 반환할 값이 없다는 것을 의미한다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 매개변수(parameter)
  - 메서드에게 전달할 데이터
  - 데이터 타입과 매개변수명으로 구성
  - main() 메서드는 String 데이터 타입의 배열인 args 매개변수를 갖는다
    - String 데이터 타입 : 문자열 데이터 유형
    - 배열(array) : 같은 데이터 타입의 여러 요소들을 포함하는 데이터 구조

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 문장(statement)
  - 프로그램의 최소 실행 단위(명령문)
  - 모든 문장은 세미콜론(;)으로 끝난다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 명령문 묶음

- 명령문이 여러 문장으로 구성되어 있고 반드시 이들 문장이 묶여져서 하나의 단위로 실행된다면 이들 문장을 중괄호({ })로 묶는다
- 위의 예에서 main() 메서드는 2개의 문장으로 구성되어 있다



# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 토큰(token)
  - 문장을 구성하는 최소 단위
  - 일상 언어의 단어와 유사함
  - 공백문자(white space)로 구분됨
  - 스페이스(space)나 탭(tab) 키. 엔터(enter) 키로 표현되는 문자
  - 예:
    - String
    - msg
    - =
    - "안녕하세요? 첫번째 자바 프로그램입니다!"

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 상수(constant)
  - 항상 그대로 인 값 즉, 변경되지 않는 값을 갖는 토큰
  - 상수의 값은 결코 바뀌지 않으며, 항상 그대로 있는 값이다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 데이터 타입(data type)
  - 데이터의 유형
  - 어떤 종류의 데이터를 표현하는가를 지정한다
  - 상수를 포함하여 모든 값은 데이터 타입을 갖는다
  - 문자열 데이터 타입 : String
  - 숫자 데이터 타입 : int, double
  - 문자 데이터 타입 : char

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 변수(variable)
  - 변하는 수
  - 어떤 값을 임시로 저장해야 할 때 사용
  - 변수에 저장된 값은 컴퓨터 메모리에 저장된다
  - 변수명은 프로그램에서 메모리의 위치에 접근할 수 있게 한다
  - 데이터 타입은 변수에 저장될 데이터의 크기와 데이터를 해석하는 방법을 결정한다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 변수 선언(declaration)
  - 모든 변수는 반드시 선언해야만 사용할 수 있다
  - 선언한다(declare)는 것은 이제 이러 이러한 타입의 변수를 사용하려고 한다는 것을 컴파일러에게 알려주는 것이다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 변수값 저장
  - msg 변수에 "안녕하세요? 첫번째 자바 프로그램입니다!" 라는 문자열을 저장함

```
msg = "당신을 사랑합니다";
```

```
msg = "당신을 미워합니다";
```

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 연산자(operator)
  - 대입 연산자(assignment operator) : 오른쪽에 있는 값을 왼쪽의 변수에 저장함
  - 산술 연산자(arithmetic operator) : 더하기, 빼기, 곱하기, 나누기
  - 비교 연산자(comparison operator) : 같다, 다르다, 크다, 작다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 변수 초기화(initialization)
  - 변수를 선언할 때 값을 저장하는 것



# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 식별자(identifier)
  - 변수명
  - 프로그램에서 유일하게 식별할 수 있는 이름
  - Java 언어는 식별자의 대소문자를 구별한다
    - msg와 Msg는 서로 다른 식별자임
  - 식별자에 한글을 사용할 수도 있지만 관습 상 영문을 사용한다
  - 식별자만 보고도 무엇을 위해 그 식별자를 사용했는지를 이해하기 쉽도록 부여하는 것이 좋다

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.  
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 표준 출력 장치에 정보 출력
  - 표준 출력 장치(standard output device) : 명령 프롬프트 창 또는, 콘솔 창
  - System : 시스템에 접근하기 위해 제공되는 Java 클래스
  - out : 표준 출력 장치에 연결된 출력 스트림(output stream)
  - println() : 문자열 출력

# 기본적인 Java 프로그램 구성 요소

---

```
// 이것은 첫번째 Java 프로그램입니다.
```

```
class Hello {  
    public static void main(String[] args) {  
        String msg = "안녕하세요? 첫번째 자바 프로그램입니다!";  
        System.out.println(msg);  
    }  
}
```

- 주석(comment)
  - 프로그램의 실행이나 컴파일 과정에 전혀 영향을 미치지 않고 소스 코드를 보는 사람에게만 정보를 제공함
  - 주석이 프로그램 실행 코드에 영향을 미치지 않기 때문에 무시하는 경우가 많다
  - 그러나 주석을 잘 작성하는 것이 다른 사람들에게도 도움을 주지만 자기자신에게도 도움이 되는 경우가 더 많다

## 2. 데이터 타입

## 2. 데이터 타입

---

- 상수
- 데이터 타입
- 변수
- 타입 변환
- 상수 변수
- 열거형
- 배열

# 상수

---

- 상수(constant)
  - 항상 그대로인 수
  - 변하지 않는 수

0	// 숫자 상수
'c'	// 문자 상수
"문자열"	// 문자열 상수
true	// 불리안 상수

# 상수

---

- 숫자 상수
  - 정수(integer)
  - 실수(real)
- 문자 상수
- 문자열 상수
- 불리안 상수

# 정수

---

- 표현 방법
  - 10진수

10  
200

- 16진수 : 0x 또는 0X 로 시작

0xA  
0XA

- 8진수 : 0 으로 시작

012

- 2진수 : 0b 또는 0B 로 시작

0b1010  
0B1010



# 정수

---

- 밑줄 문자 사용
  - 천 단위 구분

123_456_789      // 123456789 와 동일
------------------------------------

- 읽기 쉽도록 하기 위해 사용

# 실수

---

- 정수 부분과 소숫점, 소수부로 구성됨
- 표현 방법
  - 표준 표기법

123.456

- 과학 표기법

1.23456E+2

1.23456e+2

# 문자 상수

---

- 유니코드(unicode) 문자
  - 16비트(bit, 2 바이트byte)를 사용하여 하나의 문자를 표현함
  - 홑따옴표로 둘러싸임

‘9’

‘A’

‘가’

# 문자 상수

---

- 이스케이프 시퀀스(escape sequence)
  - 특별한 제어 문자나 인쇄할 수 없는 문자를 표현하는데 사용
  - 역 슬래쉬(\) 다음에 문자 코드를 붙이는 형식으로 표현

이스케이프 시퀀스	문자
'\n'	개행 문자
'\t'	탭 문자
'\"'	홀따옴표 문자
'\''	겹따옴표 문자
'\\'	역슬래쉬 문자

- 윈도우 경로명 표현

```
"C:\wwjavabasic\wwch02\wwMain.java"
```

# 문자열 상수

---

- 문자열(String)
  - 일련의 문자들이 연속되어 있는 문자의 집합
  - 겹따옴표로 둘러싸임

“이것은 문자열입니다.”

- 문자와 문자열 구분
  - 홑따옴표 : 문자
  - 쌍따옴표 : 문자열

'S' // 문자

“S” // 문자열

# 불리안 상수

---

- 참과 거짓 표현

불리안 상수값	의미
true	참
false	거짓

# 데이터 타입

---

- 데이터 타입(data type)
  - 데이터 유형
  - 데이터를 해석하는 방법과 데이터의 크기를 결정함
- Java 데이터 타입
  - 정수
  - 실수
  - 문자
  - 문자열
  - 불리언

# 정수 데이터 타입

- 정수 데이터 타입

데이터 타입	크기	값의 범위
int	4 바이트	-2,147,483,648 ~ 2,147,483,647
short	2 바이트	-32,768 ~ 32,767
long	8 바이트	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
byte	1 바이트	-128 ~ 127

- 기본(default) 정수 상수 타입 : int 데이터 타입
- long 타입 정수 상수 : 뒤에 L 또는 l 을 붙임

```
123          // int 데이터 타입
123L         // long 데이터 타입'
```



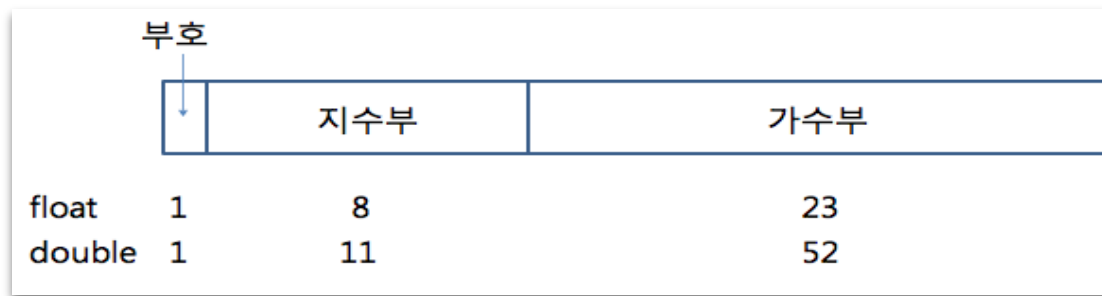
# 실수 데이터 타입

- 실수 데이터 타입

데이터 타입	크기
float	4 바이트
double	8 바이트

- 부동소수점(floating point) 방식

- 가수부 : 유효 숫자 표현(예: 1.23456E2)
- 지수부 : 10의 제곱승으로 소수점의 위치를 나타내는 값(예: 1.23456E2)



# 문자 데이터 타입

---

- 문자 데이터 타입
  - 유니코드(unicode) 문자 지원

데이터 타입	크기
char	2 바이트

# 문자열 데이터 타입

---

- 문자열 데이터 타입
  - Java 언어 자체에서는 제공하지 않음

데이터 타입	크기
String	Java 표준 라이브러리 제공 클래스

# 불리안 데이터 타입

---

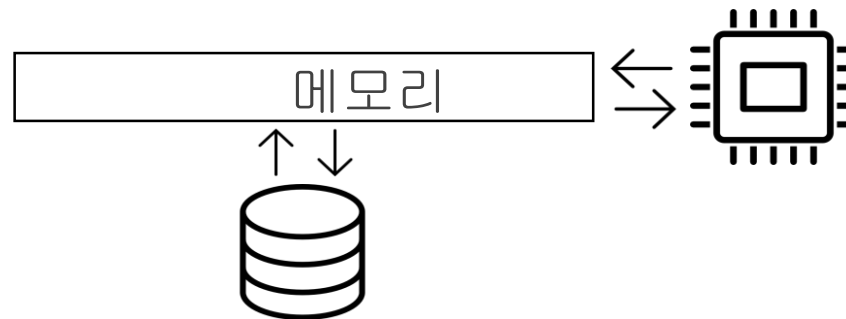
- 불리안 데이터 타입

데이터 타입	크기
boolean	true(참)/false(거짓) 값 표현

# 변수

---

- 변수(variable)
  - 변하는 값
  - 변수명 : 메모리 상의 위치를 나타내는 식별자
  - 데이터 타입 : 데이터의 크기와 해석하는 방법을 지정함



# 변수 선언

---

- 변수 선언(variable declaration)
  - 변수에 지정된 데이터 타입의 크기 만큼 메모리를 할당함
  - 그 위치에 있는 데이터를 데이터 타입에 지정된 방식으로 해석함
  - 모든 변수는 사용하기 전에 선언되어야 한다
- 변수 선언 구문

```
데이터타입 변수명 [ = 초기값 ];
```

- 변수 선언 예

```
int age; // 변수 선언
```

# 변수 사용

---

- 변수에 값 저장 : 대입연산자 사용
  - lvalue(대입연산자 왼쪽 값) : 변수
  - rvalue(대입연산자 오른쪽 값) : 저장할 값

```
age = 42;           // age 변수에 42를 저장한다
```

- 변수에 값 읽기 : 대입연산자 사용
  - lvalue(대입연산자 왼쪽 값) : 읽은 변수의 값을 저장할 다른 변수
  - rvalue(대입연산자 오른쪽 값) : 변수

```
int temp = age;     // age 변수의 값을 읽어 temp 변수에 저장한다
```

# 변수 초기화

---

- 초기화(initialization)
  - 변수 선언과 동시에 값을 저장하는 것
  - 초기화 하지 않은 변수를 읽으면 컴파일러는 에러를 발생시킨다

```
int age = 42;      // 초기화
```



# 변수명 부여 규칙

---

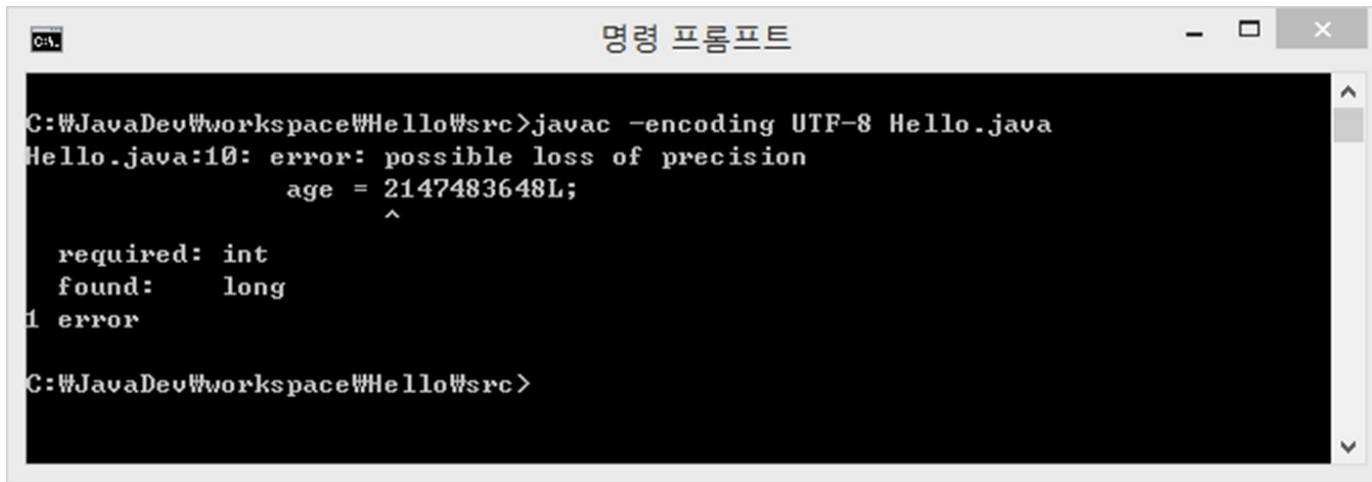
- 문법적 제약 사항
  - 반드시 영문자나 밑줄 문자(\_), 달러 기호(\$)로 시작해야 한다
  - 숫자를 포함할 수 있지만 숫자로 시작할 수 없다
  - 공백 문자를 포함할 수 없다
  - 대소문자를 구별한다
- 좋은 변수명 부여
  - 식별자만 보아도 의미를 이해할 수 있도록 한다
    - 나쁜 변수명의 예 : abc, xyz
    - 좋은 변수명의 예 : age, birthday
  - 여러 단어가 결합되는 경우 각 단어는 대문자로 시작하거나 단어를 밑줄 문자(\_)로 연결한다
    - 예 : phoneNumber, phone\_number
  - 일관성을 가지고 이름을 부여한다

# 타입 변환

---

- 강력한 타입 언어(strongly typed language)
  - 모든 표현식에서 타입이 서로 맞는지 여부를 확인하여 타입이 맞지 않으면 에러를 발생시킴

```
int age;  
age = 2147483648L;           // 에러!!!!
```



```
C:\JavaDev\workspace\Hello\src>javac -encoding UTF-8 Hello.java  
Hello.java:10: error: possible loss of precision  
    age = 2147483648L;  
            ^  
    required: int  
    found:    long  
1 error  
  
C:\JavaDev\workspace\Hello\src>
```

# 타입 변환

---

- 타입 변환(casting)
  - 한 데이터 타입의 값이 다른 데이터 타입의 값으로 변환시키는 것
  - 프로모션(promotion)
    - 데이터의 범위가 작은 데이터 타입의 값을 보다 큰 데이터 범위를 저장할 수 있는 데이터 타입으로 변환하는 것
    - 데이터의 손실이 없음

```
short code = 65;
```

```
int codeValue = code;
```

- 암시적(implicit) 타입 변환이 발생함

# 타입 변환

---

- 타입 변환(casting)
  - 잘라내기(truncation)
    - 데이터의 범위가 큰 데이터 타입의 값을 보다 작은 데이터 타입으로 변환하는 것
    - 데이터 손실 발생
    - 명시적(explicit) 타입 변환이 필요함

```
int orgValue = 32768;  
short convValue = (short)orgValue;
```

# 상수 변수

---

- 직접 상수를 사용할 때의 문제점
  - 상수의 의미를 명확하게 이해하기 어렵다
  - 파이( $\pi$ ) 값의 예 : 3.14159
- 상수 변수 정의
  - `final` 키워드 사용

```
final double PI = 3.14159;
```

- 반드시 초기화 해야 한다
- 상수이기 때문에 상수 변수값을 변경할 수 없다
- 관례적으로 대문자로 이름을 표현한다

# 열거형

---

- 요일을 저장하는 변수의 예

```
int week;
```

- 의도
  - week 변수에 일, 월, 화, 수, 목, 금, 토를 나타내는 0에서부터 6까지의 값을 저장한다
- 문제점
  - week 변수에 0에서 6까지 범위의 값이 아닌 값을 저장하려고 한다면 이것을 막을 수 있는 방법이 없다
  - 각 요일이 어떤 값에 대응되는지 이해하기 어렵다

# 열거형

---

- 열거형(enumeration)
  - 변수가 가질 수 있는 가능한 값을 나열해 놓은 일종의 데이터 타입
  - 열거형 선언
    - enum 키워드 사용

```
enum Week {sunday, monday, tuesday, wednesday, thursday, friday, saturday}
```

- 열거 멤버(enumeration member) : 0 부터 시작하는 정수값이 할당됨
- 클래스 타입으로 선언해야 함

```
class Hello {  
    enum Week {sunday, monday, tuesday, wednesday, thursday, friday, saturday}  
    // 생략...  
}
```

# 열거형

---

- 열거형 변수
  - 일반 데이터 타입의 변수와 동일한 방식으로 선언함

```
Week week;           // 열거형 변수 선언
```

- 열거형 변수 값 저장
  - 열거형의 열거 멤버만 저장할 수 있음

```
week = Week.sunday;
```

- 열거 멤버 이외 값을 저장할 수 없음

```
week = 10;           // 에러!!!
```



# 배열

---

- 1월부터 12월까지의 서울 평균 기온을 저장하는 예

1월	2월	3월	4월	5월	6월	7월	8월	9월	10월	11월	12월
-4	-1	4	11	17	21	24	25	20	13	6	-1

- 혹시...

```
int january = 0;  
int february = -1;  
int march = 4;  
int april = 11;  
int may = 17;  
int june = 21;  
int july = 24;  
int august = 25;  
int september = 20;  
int october = 13;  
int november = 6;  
int december = -1;
```

# 배열

---

- 배열(array)
  - 데이터 타입의 여러 요소element들을 포함하는 데이터 구조
- 배열 선언
  - [] 연산자 사용

```
int [] aveTemp;
```

- 또는,

```
int aveTemp [];
```

# 배열

---

- 배열 생성
  - new 연산자 사용

```
int [] aveTemp = new int[12];
```

- 또는,

```
int aveTemp [] = new int[12];
```

- 배열 초기화

```
int [] aveTemp = { -4, -1, 4, 11, 17, 21, 24, 25, 20, 13, 6, -1 };
```

# 배열

---

- 배열 요소 접근

```
int [] aveTemp = new int[12];  
aveTemp[0] = -4;      // 1월 평균 기온  
aveTemp[1] = -1;      // 2월 평균 기온  
aveTemp[2] = 4;       // 3월 평균 기온  
aveTemp[3] = 11;      // 4월 평균 기온  
aveTemp[4] = 17;      // 5월 평균 기온  
aveTemp[5] = 21;      // 6월 평균 기온  
aveTemp[6] = 24;      // 7월 평균 기온  
aveTemp[7] = 25;      // 8월 평균 기온  
aveTemp[8] = 20;      // 9월 평균 기온  
aveTemp[9] = 13;      // 10월 평균 기온  
aveTemp[10] = 6;      // 11월 평균 기온  
aveTemp[11] = -1;     // 12월 평균 기온
```

# 실습

---

- 나이를 저장할 수 있는 변수를 선언한다.
- 자신의 나이를 변수에 저장한다.
- 변수를 출력하여 확인한다.
- 나이 변수에 허용되는 더 큰 타입을 저장하고 어떤 일이 발생하는 지 알아본다.
- 무지개 색상을 나타내는 열거형을 선언한다.
- 열거형 변수를 선언하고 색상을 저장한다.
- 열거형 변수의 값을 출력하여 확인한다.
- 웹 타임에서 강의하는 과목 전체를 출력한다.
  - 힌트1: lectures 배열을 사용한다
  - 힌트2:

```
for(String lecture : lectures)
    System.out.println(lecture);
```

### 3. 연산자

### 3. 연산자

---

- 연산자
- 산술 연산자
- 증감 연산자
- 비교 연산자
- 논리 연산자
- 대입 연산자
- 연산자 우선순위

# 연산자

---

- 연산자(operator)
  - 피연산자(operand)에 어떤 특정한 기능을 수행하도록 정의된 기호
  - 분류
    - 산술 연산자(arithmetic operator)
    - 비교 연산자(comparison operator)
    - 논리 연산자(logical operator)
    - 대입 연산자(assignment operator)
  - 피연산자 개수에 따른 분류
    - 단항 연산자(unary operator)
    - 이항 연산자(binary operator)
    - 삼항 연산자(ternary operator)



# 산술 연산자

- 산술연산자 : 두 피연산자의 값을 산술 연산

연산자	기능
+	더하기
-	빼기
*	곱하기
/	나누기
%	나머지

- 피연산자 : 숫자 타입

```
int i1 = 20;  
int i2 = 6;  
int i3 = i1 % i2;      // 결과 : i3 = 2
```

- 연산 결과 : int 또는 double

```
short s1 = 10;  
short s2 = 100;  
short s3 = (short)(s1 + s2);
```

# 산술 연산자

---

- 문자열 결합
  - + 연산자 사용

```
String str = “첫번째 문자열” + “두번째 문자열”;
```

- 단항 - 연산자
  - 양수를 음수로, 음수를 양수로 변환

```
int i = 100;  
i = -i;           // i == -100
```

# 증감 연산자

- 증감 연산자 : 피연산자의 값을 증가 또는 감소

연산자	기능
++	1씩 증가
--	1씩 감소

```
int i = 100;  
++i;
```

```
i = i + 1;
```

```
int i = 100;  
--i;
```

```
i = i - 1;
```

- 돌발 퀴즈!
  - 다음 문장이 실행된 후 i1, i2, r1, r2 값은?

```
int i1 = 10;  
int i2 = 10;  
int r1 = 2 * ++i1;  
int r2 = 2 * i2++;
```

# 증감 연산자

---

- 정답

```
i1 = 11  
i2 = 11  
r1 = 22  
r2 = 20
```



- 증감 연산자의 위치

- 피연산자 앞에 오는 경우 : 피연산자를 증감시키고 다른 연산에 참여함

```
int r1 = 2 * ++i1;
```

- 피연산자 뒤에 오는 경우 : 피연산자를 다른 연산에 먼저 참여시키고 증감시킴

```
int r2 = 2 * i2++;
```

# 비교 연산자

---

- 비교 연산자 : 두 연산자의 값을 비교

연산자	기능
<	작다
>	크다
<=	작거나 같다
>=	크거나 같다
==	같다
!=	같지 않다

- 연산 결과 : boolean 타입의 true(참) 또는 false(거짓)

```
int x = 10;  
int y = 100;  
boolean b = x == y;           // b = false
```

# 비교 연산자

---

- 비교 삼항 연산자

표현식1 ? 표현식2 : 표현식3;

- 표현식1의 참/거짓 평가
- 참인 경우 표현식2 연산 수행
- 거짓인 경우 표현식3 연산 수행

```
int x = 3;  
int y = 4;  
int max = x > y ? x : y;
```

- 돌발 퀴즈!

- 다음 문장이 실행된 후 max, x, y 의 값은?

```
int x = 3;  
int y = 4;  
int max = x > y ? ++x : ++y;
```

# 비교 연산자

---

- 정답

```
max = 5  
x = 3  
y = 5
```



- 연산에 참여하지 않는 표현식 코드는 실행되지 않음
- 문자열 비교
  - == 연산자 사용

```
boolean b = “첫번째 문자열” == “두번째 문자열”;
```

- 또는,

```
String s1 = “당신을 사랑합니다”;  
String s2 = “당신을 사랑합니다”;  
boolean b = s1 == s2;
```

# 논리 연산자

- 논리 연산자 : 비교 연산 표현식을 묶어 조건을 결정함

연산자	기능
!	논리 부정(NOT)
&&	논리곱(AND)
	논리합(OR)

- ! 연산자

!	결과
true	false
false	true

```
boolean t = true;  
boolean f = !t;           // f == false
```



# 논리 연산자

- 논리 연산자 : 비교 연산 표현식을 묶어 조건을 결정함
  - && 연산자

&&	true	false
true	true	false
false	false	false

```
int a = 10, b = 20;  
int x = 100, y = 200;  
boolean result = ( a > b ) && ( x < y );           // result == false
```

- || 연산자

	true	false
true	true	true
false	true	false

```
int a = 10, b = 20;  
int x = 100, y = 200;  
boolean result = ( a > b ) || ( x < y );           // result == true
```

# 논리 연산자

---

- 돌발 퀴즈!
  - 다음 문장이 실행된 후 result, a, b, x, y 의 값은?

```
int a = 10, b = 20;  
int x = 100, y = 200;  
boolean result = ( ++a > ++b ) && ( ++x < ++y );
```

# 논리 연산자

---

- 정답

```
result = false
```

```
a = 11
```

```
b = 21
```

```
x = 100
```

```
y = 200
```



- 연산 중 결과가 확정적이면 더 이상 연산을 수행하지 않음

# 대입 연산자

---

- 대입 연산자

연산자	기능
=	대입
+=	더한 후 대입
-=	뺀 후 대입
*=	곱한 후 대입
/=	나눈 후 대입

```
int x = 10;  
x *= 2;      // x = 20, x = x * 2 와 동일
```

# 연산자 우선순위

- 연산자 우선 순위
  - 다른 연산자에 대해 먼저 연산되는 순서

!	++	—			
*	/	%			
+	-				
==	!=	<	>	<=	>=
&&					
?:					
=	*=	/=	%=	+=	-=

- 우선 순위 변경 : () 사용

```
a = (b + c) * ++d;
```

# 실습

---

- 다음 연산의 결과 i1, i2, i3의 값을 출력한다.

```
int i1 = 20;  
int i2 = 6;  
int i3 = i1 % i2;
```

- 실제로 다음 코드를 작성하여 문제점이 무엇이고, 어떻게 해결해야 하는지 알아본다.

```
short s1 = 10;  
short s2 = 100;  
short s3 = s1 + s2;
```

- 다음 코드를 작성하여 결과를 출력한다.

```
int i1 = 10;  
int i2 = 10;  
int r1 = 2 * ++i1;  
int r2 = 2 * i2++;
```

# 실습

---

- 다음 문장이 실행된 후 result, a, b, x, y 의 값을 출력한다.

```
int a = 10, b = 20;  
int x = 100, y = 200;  
boolean result = ( ++a > ++b ) && ( ++x < ++y );
```

- 다음 표현식의 결과를 출력한다.

```
String str = “첫번째 문자열” + “두번째 문자열”;
```

- 다음 표현식의 결과를 출력한다.

```
String s1 = “당신을 사랑합니다”;  
String s2 = “당신을 사랑합니다”;  
boolean b = s1 == s2;
```

## 4. 제어문



## 4. 제어문

---

- 명령문과 코드 블록
- 조건문
- 반복문

# 명령문과 코드 블록

---

- 문장 또는 명령문(statement)
  - 명령을 수행하는 단위
  - 세미콜론(;)으로 끝난다
  - 하나의 명령문이 여러 행에 걸쳐 기술될 수 있으며, 한 행에 여러 명령문이 올 수 있다
- 코드 블록(code block) 또는 복합문(compound statement)
  - 여러 개의 명령문이 결합되어 논리적으로 하나의 명령을 수행하도록 구성됨
  - 중괄호({ }) 안에 포함됨
  - 코드 블록 안에 변수를 선언할 수 있으며, 또 다른 코드 블록을 포함시킬 수 있다

# 명령문과 코드 블록

---

- 코드 블록(code block) 또는 복합문(compound statement)

```
void main(String[] args) {  
    // main 코드 블록  
    int a;  
    ...  
    {  
        // 내부 코드 블록  
        int b;  
        ...  
        a = 10;           // 코드 블록 외부의 변수에 접근  
        b = 10;           // 같은 코드 블록 안의 변수에 접근  
    }  
    a = 100;              // 같은 코드 블록 안의 변수에 접근  
    b = 100;              // 에러! - 포함된 코드 블록 안의 변수에는 접근할 수 없음  
}
```

# 조건문

---

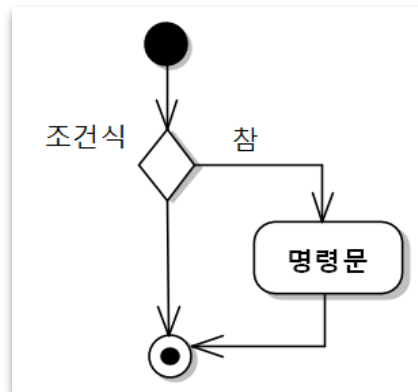
- 조건의 참과 거짓 여부에 따라 어떤 작업을 할 것인지를 결정할 때 사용
  - if 문
  - switch...case 문

# if 문

- if 문

if (조건식)

명령문;



```
if (age > 18)
    login = true;
```

```
if (age > 18) {
    login = true;
    scriptExecution = true;
}
```

# if 문

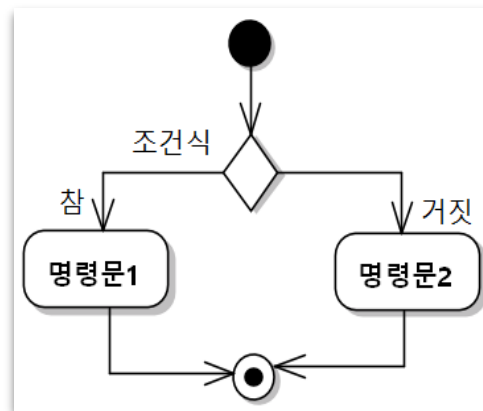
- if ... else ... 문

if (조건식)

명령문1;           // 조건식이 참인 경우에 실행됨

else

명령문2;           // 조건식이 거짓인 경우에 실행됨



# if 문

---

- if ... else ... 문

```
if (age > 18) {  
    login = true;  
    scriptExecution = true;  
}  
else {  
    login = false;  
    scriptExecution = false;  
}
```

# if 문

---

- if ... else if ... else ... 문

```
if (조건식1) {  
    // 조건식1이 참인 경우에 실행됨  
    명령문1;  
}  
else if (조건식2) {  
    // 조건식2이 참인 경우에 실행됨  
    명령문2;  
}  
...  
else if (조건식n) {  
    // 조건식n이 참인 경우에 실행됨  
    명령문n;  
}  
else {  
    // 그밖의 경우에 실행됨  
    명령문z;  
}
```



# if 문

---

- if ... else if ... else ... 문

```
if (score > 90)
    result = “아주 훌륭합니다”;
else if (score > 80)
    result = “참 잘했어요”;
else if (score > 70)
    result = “좋습니다”;
else
    result = “분발하세요”;
```

# switch ... case 문

---

- switch ... case 문

```
switch (조건변수) {  
  case 값1 :  
    // 조건변수 값이 값1 과 동일한 경우에 실행  
    명령문1;  
    break;  
  case 값2 :  
    // 조건변수 값이 값2 과 동일한 경우에 실행  
    명령문2;  
    break;  
    ...  
  case 값n :  
    // 조건변수 값이 값n 과 동일한 경우에 실행  
    명령문n;  
    break;  
  default :  
    // 나머지  
    명령문z;  
}
```

# switch ... case 문

---

- switch ... case 문
  - 다음 구문과 동일함

```
if (조건변수 == 값1) {  
    // 조건변수 값이 값1 과 동일한 경우에 실행  
    명령문1;  
}  
else if (조건변수 == 값2) {  
    // 조건변수 값이 값2 과 동일한 경우에 실행  
    명령문2;  
}  
...  
else if (조건변수 == 값n) {  
    // 조건변수 값이 값n 과 동일한 경우에 실행  
    명령문n;  
}  
else {  
    // 나머지  
    명령문z;  
}
```

# switch ... case 문

---

- switch ... case 문

```
switch(rate) {  
  case 1 :  
    result = “아주 훌륭합니다”;  
    break;  
  case 2 :  
    result = “참 잘했어요”;  
    break;  
  case 3 :  
    result = “좋습니다”;  
    break;  
  default :  
    result = “분발하세요”;  
    break;  
}
```

# switch ... case 문

---

- switch ... case 문
  - 범위를 지정하는 구문을 지원하지 않음

```
switch(score) {  
case 91 to 100 :      // 에러! - 비교값의 범위 지정 불가능  
...  
}
```

- 굳이 범위를 지정해야 한다면... 그러나 적합하지 않음

```
switch(score) {  
case 91 :  
    // 생략...  
case 99 :  
case 100 :  
    rate = 1;  
    break;  
...  
}
```

# switch ... case 문

---

- switch ... case 문 조건 변수 타입
  - byte, short, int, char, 열거형

```
int rate;  
switch(rate) {  
...  
}
```

- JDK 7 부터 String 타입 지원

```
String s;  
switch(s) {  
case “문자열1” :  
    result = 1;  
    break;  
case “문자열2” :  
    result = 2;  
    break;  
// 생략...  
}
```

# 반복문

---

- 특정한 명령문을 반복하여 실행해야 하는 경우에 사용
  - while 문
  - for 문

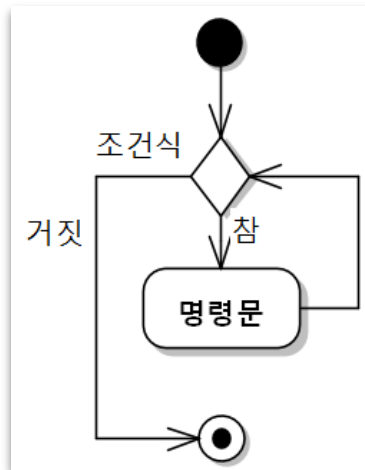
# while 문

---

- while 문

while (조건식)

명령문;





# while 문

---

- while 문

```
Scanner scanner = new Scanner(System.in);
int value;
System.out.println("10 보다 큰 숫자를 입력하세요.");
value = scanner.nextInt();          // 숫자값을 입력 받음
while (value <= 10) {
    System.out.println("잘못 입력하셨습니다. 10 보다 큰 숫자를 입력하세요.");
    value = scanner.nextInt(); // 숫자값을 입력 받음
}
System.out.printf("감사합니다. %d 을 입력하셨습니다.", value);
```

- break 문 : 반복 탈출

```
while (value <= 10) {
    if (value % 2 == 0)
        break;

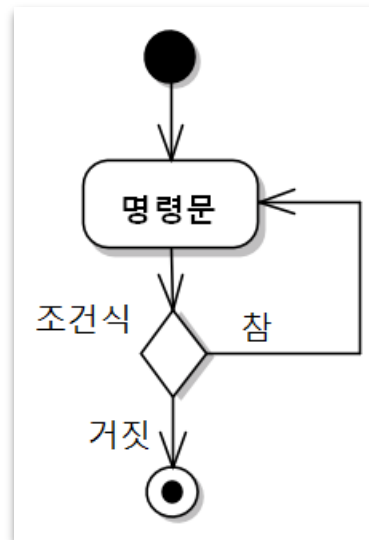
    System.out.println("잘못 입력하셨습니다. 10 보다 큰 숫자를 입력하세요.");
    value = scanner.nextInt(); // 숫자값을 입력 받음
}
```

# while 문

---

- do...while 문

```
do  
    명령문;  
while (조건);
```



# while 문

---

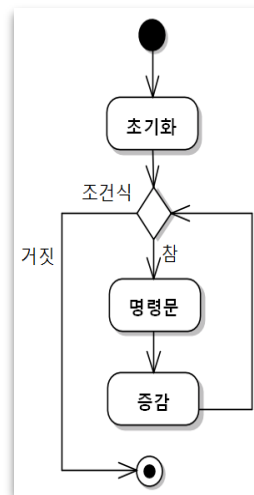
- do...while 문

```
int value;
do {
    System.out.println("10 보다 큰 숫자를 입력하세요.");
    value = scanner.nextInt();      // 숫자값을 입력 받음
} while (value <= 10);
System.out.printf("감사합니다. %d 을 입력하셨습니다.", value);
```

# for 문

- for 문

for ( 초기화; 조건식; 증감 )  
명령문



```
for (int i = 0; i < 5; ++i)
    System.out.println("당신을 사랑합니다!");
```

# for 문

---

- continue 문 : 다시 반복 실행

```
for ( int i = 0; i < 5; ++i ) {  
    if ( i % 2 == 0 )  
        continue;  
    System.out.printf("%d 번째 : 당신을 사랑합니다!", i);  
}
```

# for 문

---

- 집합(collection) for 문

```
for ( 타입 요소변수명 : 집합변수)
```

```
int aveTemp [] = { -4, -1, 4, 11, 17, 21, 24, 25, 20, 13, 6, -1 };  
for ( int element : aveTemp )  
    System.out.println("값 : " + element);
```

- 또는,

```
int aveTemp [] = { -4, -1, 4, 11, 17, 21, 24, 25, 20, 13, 6, -1 };  
for (int i = 0; i < aveTemp.length; ++i)  
    System.out.println("값 : " + aveTemp[i]);
```

# 실습

---

- 점수를 입력받아서
  - 90점 이상이면 1등급
  - 80점 이상, 90점 미만이면 2등급
  - 70점 이상, 80점 미만이면 3등급
  - 그 이하이면 4등급을 부여하고,
  - 1등급이면 "아주 훌륭합니다."
  - 2등급이면 "참 잘했습니다."
  - 3등급이면 "좋습니다."
  - 4등급이면 "분발하세요."를 출력한다.
  - 점수를 입력받을 때 다음 구문을 사용한다.

```
Scanner scanner = new Scanner(System.in);  
int value;  
value = scanner.nextInt();           // 점수를 입력 받음
```

- 서울의 평균 기온을 구하여 출력한다.

## 5. 클래스 기초



## 5. 클래스 기초

---

- 객체지향 개념의 이해
- 클래스 선언
- 접근 지정자
- 필드 정의
- 메서드 정의와 호출
- 인스턴스 생성과 생성자
- this 키워드
- 클래스 멤버 접근
- 객체 소멸과 소멸자

# 객체지향 개념의 이해

---

- 객체지향 기본 개념
  - 추상화(abstraction)
  - 캡슐화(encapsulation)
  - 모듈성(modularity)
  - 계층성(hierarchy)
  - 클래스(class)와 객체(object)
  - 메시지 보내기(sending message)

# 객체지향 개념의 이해

---

- 추상화(abstraction)
  - 보다 중요하고 필수적인 사항에 집중하는 것
  - 무엇이 중요하고 덜 중요한가는 관점과 경험에 따라 다르다
  - 데이터 추상화(data abstraction)
  - 절차적 추상화(procedure abstraction)

# 객체지향 개념의 이해

---

- 캡슐화(encapsulation)
  - 중요하고 세부적인 구현 방법에 대한 자세한 사항을 꼭꼭 숨겨놓은 것
  - 데이터 감추기(data hiding)
  - 인터페이스를 통해 외부에 기능을 노출
  - 캡슐화는 내부 데이터의 구조나 구현 방법을 감출 수 있기 때문에 인터페이스가 변경되지 않는 한 필요에 따라 내부 데이터 구조나 구현 방법을 자유롭게 변경시킬 수 있게 되는 이점을 제공한다

# 객체지향 개념의 이해

---

- 모듈성(modularity)
  - 크고 복잡한 것을 좀 더 작고 관리할 수 있는 조각으로 나누어, 이들 조각을 독립적으로 개발할 수 있게 한다
  - 서브 프로그램 또는 모듈
  - 객체

# 객체지향 개념의 이해

---

- 계층성(hierarchy)
  - 보편적인 것을 상위에 두고 특수한 것을 하위에 두는 것
  - 상위 개념 : 자동차
  - 하위 개념 : 승용차, 승합차, 화물차

# 객체지향 개념의 이해

---

- 추상화와 클래스



- 도로 위를 달리는 자동차라고 하는 실제적인 사물 즉, 객체를 관념적으로 머릿속에 그린 이미지 즉, 모형 또는 템플릿
- 자동차의 추상적인 표현
- 추상화란 중요하고 필수적인 사항을 중심으로 그 이미지나 관념을 표현한 것
- 자동차 설계서
- 클래스란 "객체에 대한 추상화 작업의 결과"이다

# 객체지향 개념의 이해

---

- 객체(object)
  - 애플리케이션에서 명확한 한계와 의미가 있는 사물이나 개념 또는 추상화로서, 특성과 행위, 그리고 정체성을 하나의 단위로 포함하여 캡슐화된 것
    - 물리적인 사물
    - 개념적인 것
    - 한계와 의미가 명확해야 한다
  - 특성과 행위를 하나의 단위로 포함한다
    - 특성 – 부품
    - 행위 – 운전하는 방법
  - 정체성을 가진다
    - 객체를 구별시키는 고유한 특징
    - 차량 번호



# 객체지향 개념의 이해

---

- 인스턴스와 객체



- 인스턴스(instance)란 클래스에 정의된 사항을 모두 충족하는 하나의 경우이다
- 클래스의 인스턴스는 객체와 동의어이다
- 클래스는 객체를 생성하는 수단이 된다

# 객체지향 개념의 이해

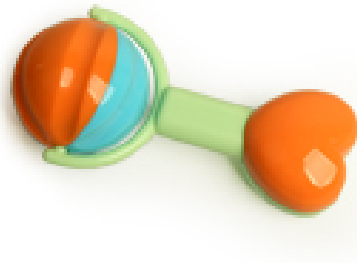
---

- 메시지 보내기(sending message)
  - 상대 객체가 대상 객체에 명령하는 것
  - 주체(主體) – 작용을 하는 쪽
  - 객체(客體) – 작용의 대상이 되는 쪽
  - 객체는 혼자 존재할 수 없다
    - 객체 사이의 관계(relationship)

# 객체지향 개념의 이해

---

- 딸랑이의 예
  - 객체지향적인 접근 방법이 가장 자연스러운 이유는 사람에게서는 어려서부터 객체라고 하는 개념이 형성되기 때문이다



# 클래스 선언

---

- 클래스 선언

```
class 클래스명 {  
    // 멤버  
}
```

- 클래스명과 동일한 .java 확장자를 갖는 파일에 정의

```
// Date.java 파일  
class Date {  
  
}
```

- 클래스 멤버
  - 생성자(constructor)
  - 소멸자(destructor)
  - 필드(field)
  - 메서드(method)

# 클래스 선언

---

- 클래스 멤버

```
// Date.java 파일
class Date {
    // 생성자
    public Date() {
    }
    public Date(int yy, int mm, int dd) {
    }
    // 소멸자
    public void finalize() {
    }
    // 메서드
    public void setDate(int yy, int mm, int dd) {
    }
    public int getYear() {
    }
    public int getMonth() {
    }
    public int getDay() {
    }
    public void displayDate() {
    }
    // 필드
    private int year;
    private int month;
    private int day;
}
```

# 접근 지정자

---

- 접근 지정자(access specifier)
  - 해당 클래스의 인스턴스 즉, 객체의 멤버를 다른 외부 객체에서 접근하려고 할 때 접근이 허용되는 지 여부를 지정함
  - public 접근 지정자
    - 공개 멤버
    - 멤버가 외부에 공개되어 있어 어디에서라도 멤버에 접근할 수 있게 한다
    - 일반적으로 메서드에 지정함
  - private 접근 지정자
    - 비공개 멤버
    - 멤버를 비공개로 지정하여 외부 객체에서 멤버에 접근할 수 없게 한다
    - 일반적으로 필드에 지정함

# 필드 정의

---

- 필드(field)
  - 클래스의 객체의 특성(attribute)
  - 클래스 내부 변수

```
// Date.java 파일
class Date {
    // 생략...
    private int year;      // 년
    private int month;     // 월
    private int day;       // 일
}
```

# 필드 정의

---

- 필드 초기화

```
class Date {  
    // 생략...  
    private int year = 1990;        // 1990년  
    private int month = 1;          // 1월  
    private int day = 1;             // 1일  
}
```

- 필드 기본 값(default value) : 초기화 하지 않는 경우에 저장되는 값
  - 숫자 필드 : 0
  - 문자 필드 : null
  - 불리언 필드 : false
- 필드 초기화의 이점
  - 유효한 데이터를 갖게 한다
  - 초기화하지 않은 경우 0년 0월 0일은 유효한 날짜가 아니다
  - 초기화한 1990년 1월1일은 유효한 날짜다



# 필드 정의

---

- 비공개(private) 필드 이점
  - 데이터 유효성을 보장할 수 있다
  - 내부 데이터 구조를 쉽게 바꿀 수 있다

# 메서드 정의와 호출

---

- 메서드(method)
  - 특정한 작업을 수행하기 위한 명령문의 그룹

```
접근지정자 반환타입 메서드명 ( 매개변수 목록) {  
    // 메서드 몸체  
}
```

- 접근지정자 : 일반적으로 public
- 반환타입 : 메서드가 반환하는 값의 데이터 타입
  - 반환할 값이 없는 경우 : void
- 메서드명 : 메서드의 이름
- 매개변수 목록 : 메서드가 호출될 때 전달되는 값 즉, 인수(argument)들의 데이터 타입과 매개변수 명이 나열됨
- 메서드 몸체 : 메서드가 무엇을 하는 지를 정의한 명령문

# 메서드 정의와 호출

---

- 메서드(method)

```
public void setDate(int yy, int mm, int dd) {  
    int [] days = {0, 31, 28, 31, 30, 31, 30,  
                   31, 31, 30, 31, 30, 31};  
  
    year = max(1990, yy);  
    month = max(1, mm);  
    month = min(month, 12);  
    // 윤년 계산은 생략함  
    day = max(1, dd);  
    day = min(day, days[month]);  
}
```

# 메서드 정의와 호출

---

- 내부 비공개(private) 메서드

```
private int max(int x, int y) {  
    if(x > y)  
        return x;  
    return y;  
}  
  
private int min(int x, int y) {  
    if(x < y)  
        return x;  
    return y;  
}
```

- return 문


- 메서드에서 호출한 측에 값을 반환할 때 사용
- 메서드의 반환 타입과 동일한 데이터 타입 또는 암시적 변환이 가능한 데이터 타입
- 반환할 값이 없는 경우에는 사용하지 않음

# 메서드 정의와 호출

- 메서드 호출(method call)
  - 프로그램이 메서드를 호출하면 프로그램 제어(program control)는 호출된 메서드로 넘어간다
  - 호출된 메서드가 정의된 작업을 수행하고 반환하거나 메서드의 실행이 끝나면 프로그램 제어는 메서드를 호출한 프로그램으로 되돌아온다
  - 메서드를 호출할 때는 메서드에 정의된 매개변수 순서대로 지정된 데이터 타입의 값 즉, 인수를 괄호 안에 지정하면 된다
  - 메서드가 반환값을 반환한다면 = 연산자를 사용하여 반환값을 저장한다

```
public void setDate(int yy, int mm, int dd) {  
    ...  
    month = max(1, mm);  
    ...  
}
```

```
private int max(int x, int y) {  
    if(x > y)  
        return x;  
    return y;  
}
```



# 인스턴스 생성과 생성자

---

- 참조 변수(reference variable) 선언

```
Date currentDate;
```

- 초기값(default value) : 널(null)

```
Date currentDate;
```

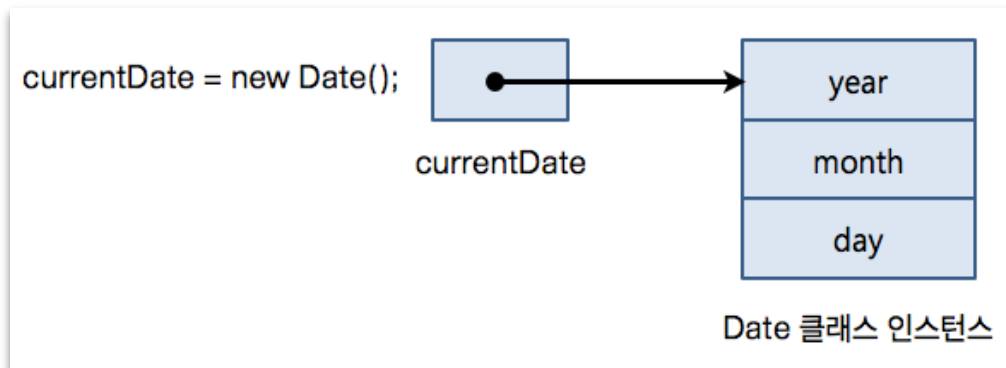
null

currentDate

# 인스턴스 생성과 생성자

- 인스턴스 생성
  - 클래스 인스턴스(instance of class) : 객체(object)
  - new 연산자

```
currentDate = new Date();
```



```
Date currentDate = new Date();
```

# 인스턴스 생성과 생성자

---

- 생성자(constructor)
  - 클래스의 인스턴스(객체)가 생성될 때마다 자동적으로 호출되는 특별한 종류의 메서드

```
// 생성자
public Date() {
    year = 1990;
    month = 1;
    day = 1;
}
```

- 반환 타입이 없음(void 조차 안됨)



# 인스턴스 생성과 생성자

---

- 디폴트 생성자(default constructor)
  - 매개변수가 없는 생성자
  - 어떠한 생성자도 정의하지 않으면 컴파일러가 자동적으로 생성함
  - 필드를 기본값(default value)으로 초기화함

```
Date myDate = new Date(); // Date() 생성자 호출
```

- 매개변수를 갖는 생성자 정의

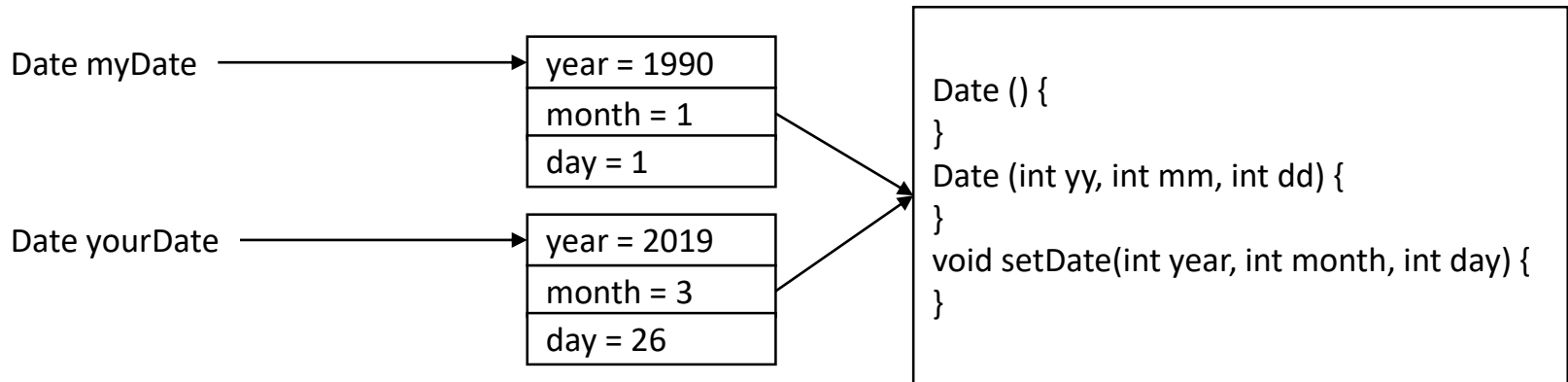
```
// 생성자  
public Date(int yy, int mm, int dd) {  
    setDate(yy, mm, dd);  
}
```

```
Date myDate = new Date(1800, 20, 90); // Date(int yy, int mm, int dd) 생성자 호출
```

# this 키워드

- 어느 인스턴스가 메서드를 호출했나?

```
Date myDate = new Date();  
myDate.setDate(1990, 1, 1);  
Date yourDate = new Date(2019, 3, 26);  
yourDate.setDate(2019, 3, 26);
```



# this 키워드

---

- this 키워드
  - 메서드가 호출된 인스턴스를 가리키는 참조(reference) 정보를 가짐

```
Date() {  
    this.year = 1990;  
    this.month = 1;  
    this.day = 1;  
}
```

- 매개변수의 이름과 필드의 이름이 동일한 경우에 필드를 참조하기 위해서는 this 키워드를 사용해야 함

```
void setDate(int year, int month, int day) {  
    int [] days = {0, 31, 28, 31, 30, 31, 30,  
                   31, 31, 30, 31, 30, 31};  
    this.year = max(1990, year);  
    this.month = max(1, month);  
    this.month = min(this.month, 12);  
    // 윤년 계산은 생략함  
    this.day = max(1, day);  
    this.day = min(this.day, days[month]);  
}
```

# 클래스 멤버 접근

---

- 클래스 메서드 호출
  - . 연산자 사용

```
public static void main(String [] args)
{
    Date currentDate = new Date();
    Date myDate = new Date(1800, 20, 90);
    currentDate.setDate(2012, 2, 29);
    currentDate.displayDate();
    myDate.displayDate();
}
```

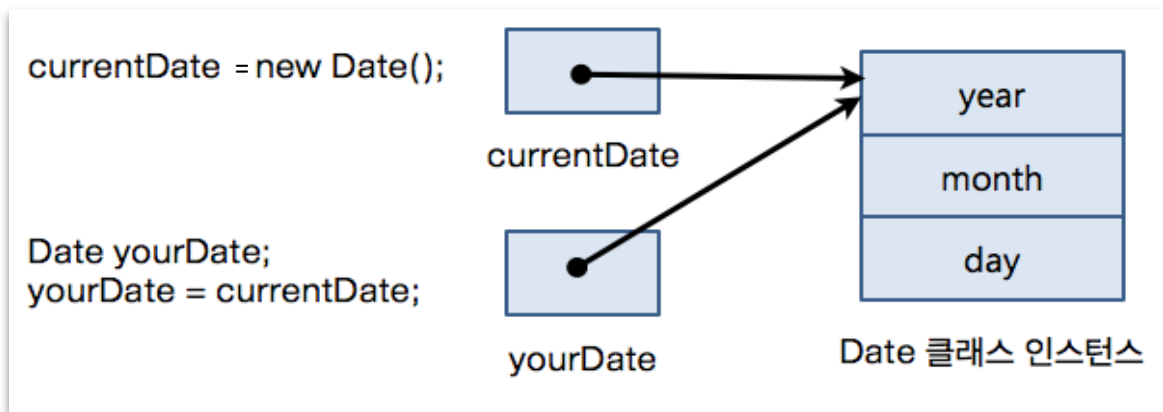
- 메시지 보내기(sending message)
- 초기화되지 않은 참조 변수에 대한 메서드 호출 => 에러!

```
Date yourDate;
yourDate.displayDate(); // 에러! 참조 변수가 초기화되지 않음
```

# 클래스 멤버 접근

- 참조 변수 대입

```
yourDate = currentDate;
```



```
yourDate.displayDate(); // currentDate.displayDate() 와 동일함
```

# 객체 소멸과 소멸자

---

- 객체 소멸
  - 가비지 컬렉터(garbage collector, 쓰레기 수집상)이 더 이상 참조되지 않는 클래스 인스턴스 즉, 객체들을 수집하여 자동적으로 소멸시킴
- 소멸자(destructor)
  - 객체가 호출되는 특별한 메서드
  - 객체가 메모리 블록에서 해제되기 전에 필요한 여러가지 뒷처리를 할 목적으로 사용됨
  - finalize() 메서드

```
protected void finalize() {  
    // 소멸자 구현 코드  
}
```

# 실습

---

- 날짜를 추상화하는 클래스를 정의한다.
  - 년,월,일 필드를 선언한다
  - 날짜 객체가 생성될 때 유효한 값을 갖도록 생성자를 정의한다
  - 특정한 날짜로 날짜 객체를 생성할 수 있도록 한다. 유효한 날짜를 지정할 수 있도록 한다. 유효하지 않은 날짜이면 디폴트 값으로 대체한다
  - 각 필드에 대하여 get/set 메서드를 정의한다. 유효한 날짜를 지정할 수 있도록 한다. 유효하지 않은 날짜이면 디폴트 값으로 대체한다
  - 날짜를 출력하는 메서드를 정의한다
- 날짜 클래스를 사용한다
  - 매개변수가 지정되지 않은 객체를 생성한다
  - 자신의 생일이 지정된 객체를 생성한다
  - 오늘 날짜가 지정된 객체를 생성한다
  - 각 객체를 출력한다
  - 매개변수가 지정되지 않은 객체의 참조 변수에 오늘 날짜가 지정된 객체의 참조 변수 값을 저장한다
  - 매개변수가 지정되지 않은 객체를 출력한다
  - 매개변수가 지정되지 않은 객체의 날짜를 변경시킨다
  - 오늘 날짜가 지정된 객체를 출력한다

# 실습

---

- 사원을 추상화하는 클래스를 정의한다
  - 어떤 정보를 포함해야 할까?
  - 어떤 생성자가 필요할까?
  - 이 클래스를 유용하게 사용하게 하기 위해서는 어떤 메서드가 필요할까?
- 사원 클래스를 사용하는 코드를 작성한다
  - 사원 클래스를 유용하게 사용하기 위해서 어떤 것들이 필요한가?
  - 추가할 필드가 있는가?
  - 추가할 메서드가 있는가?
  - 변경해야 할 메서드가 있는가?
  - 생성자는 어떤가?



## 6. 상속성

## 6. 상속성

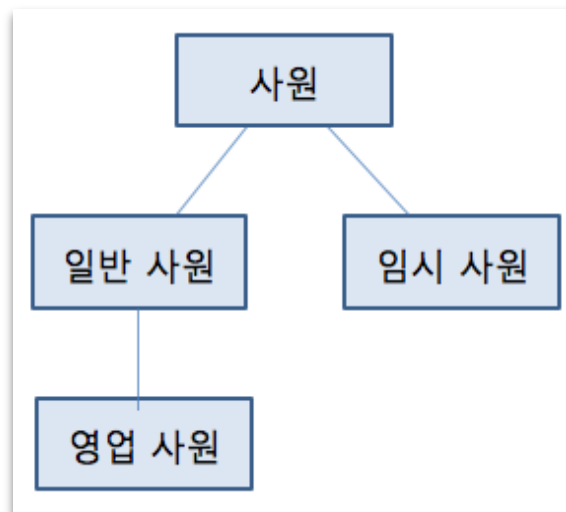
---

- 상속성
- 서브 클래스 정의
- 서브 클래스의 인스턴스 생성
- 수퍼 클래스 멤버 접근
- 수퍼 클래스 메서드 재정의
- 수퍼 클래스와 서브 클래스 사이 변환
- 상속성, 그 이상
- Object 클래스

# 상속성

---

- 상속성(inheritance)
  - 객체나 클래스가 부모와 자식과의 관계를 갖는 것
  - 트리 구조와 같은 계층적인 구조를 형성함
  - 부모 클래스(parent class) : 수퍼 클래스(super class), 기초 클래스(base class)
  - 자식 클래스(child class) : 서브 클래스(sub class), 파생 클래스(derived class)
  - 서브 클래스는 수퍼 클래스로부터 특성과 행위 그리고 수퍼 클래스의 관계를 모두 상속받는다



# 상속성

---

- 서브 클래스
  - 서브 클래스 정의 : extends 키워드 사용

```
class 서브클래스명 extends 슈퍼클래스명 {  
    // 메서드와 필드 정의  
}
```

- 일반 사원 클래스 예 :

```
// RegularEmployee.java 파일  
// 일반 사원 클래스  
class RegularEmployee extends Employee {  
    public RegularEmployee(String name, String address,  
                           String telno, Date joindate, double salary) {  
    }  
    public double payCheck() {           // 급여 계산  
    }  
    private double salary;              // 급여  
}
```

# 서브 클래스 정의

---

- 수퍼 클래스
  - 사원 클래스 예 :

```
// Employee.java 파일
// 사원 클래스
class Employee {
    public Employee(String name, String address,
                    String telno, Date joindate) {

    }
    public void displayEmployee() { // 사원 정보 표시
    }
    private String name;           // 사원명
    private String address;         // 주소
    private String telno;           // 전화번호
    private Date joindate;          // 입사일
}
```

# 서브 클래스의 인스턴스 생성

---

- 인스턴스 생성 순서
  - 클래스 계층도의 가장 상위에 있는 수퍼 클래스의 인스턴스 생성
  - 클래스 계층도를 따라 내려오면서 수퍼 클래스의 인스턴스 생성
  - 서브 클래스의 인스턴스 생성
- 수퍼 클래스 생성자 호출
  - `super` 키워드 사용

```
super ( 인수 목록 );
```

- 일반 사원 클래스 예 :

```
public RegularEmployee (String name, String address,  
                        String telno, Date joindate, double salary) {  
    super(name, address, telno, joindate); // 수퍼 클래스 생성자 호출  
    salary = salary;  
}
```

# 서브 클래스의 인스턴스 생성

---

- 일반 사원 클래스 인스턴스 생성 예 :

```
Date joindate = new Date(2012,1,1);
RegularEmployee re = new RegularEmployee("전병선", "서울시", "123-4567",
                                           joindate, 10000000000);

re.displayEmployee();           // 슈퍼 클래스 메서드 호출
double salary = re.payCheck();  // 서브 클래스 메서드 호출
System.out.printf("급여액 : %f 원\n", salary);
```

- 슈퍼 클래스 타입 참조 변수 사용

```
Employee e = new RegularEmployee("전병선", "서울시", "123-4567",
                                   joindate, 10000000000);

e.displayEmployee();           // 슈퍼 클래스 메서드 호출
```

- 슈퍼 클래스 메서드만 호출할 수 있음

# 수퍼 클래스 멤버 접근

---

- 자식에게 용돈 지급 방법
  - 자식이 부모에게 용돈을 달라고 해야만 지급한다
    - 서브 클래스는 수퍼 클래스의 공개 메서드를 호출한다
  - 부모가 용돈함을 제공한다
    - 수퍼 클래스는 보호(protected) 멤버를 제공한다
- 보호(protected) 멤버
  - 서브 클래스에서 공개(public) 멤버 처럼 마음대로 접근하여 사용할 수 있다
  - 클래스 외부에서는 비공개(private) 멤버와 같이 접근할 수 없다



# 수퍼 클래스 멤버 접근

- 일반 사원 클래스의 보호 멤버 정의

```
// 일반 사원 클래스
class RegularEmployee extends Employee {
    public RegularEmployee(String name, String address,
                           String telno, Date joindate, double salary) { }

    public double payCheck() { }    // 급여 계산
    protected double salary;      // 급여
}
```

- 영업 사원 클래스에서 보호 멤버에 접근

```
// 영업 사원 클래스
class SalesEmployee extends RegularEmployee {
    // 생략...
    public double payCheck() { }    // 급여 계산
        return salary + (sales * commission);
    }

    private double sales;           // 영업실적
    private double commission;     // 영업수당
}
```

# 수퍼 클래스 멤버 접근

- super 키워드의 또 다른 사용
  - 수퍼 클래스 멤버의 이름과 서브 클래스 멤버의 이름이 동일한 경우, 서브 클래스에서 수퍼 클래스 멤버에 접근할 때

```
// 영업 사원 클래스
class SalesEmployee extends RegularEmployee {
    // 생략...
    public double payCheck() {           // 급여 계산
        return super.payCheck() + (sales * commission);
    }
}
```

```
// 영업 사원 클래스
class SalesEmployee extends RegularEmployee {
    // 생략...
    public double payCheck() {           // 급여 계산
        return super.salary + (sales * commission);
    }
}
```

# 수퍼 클래스 메서드 재정의

---

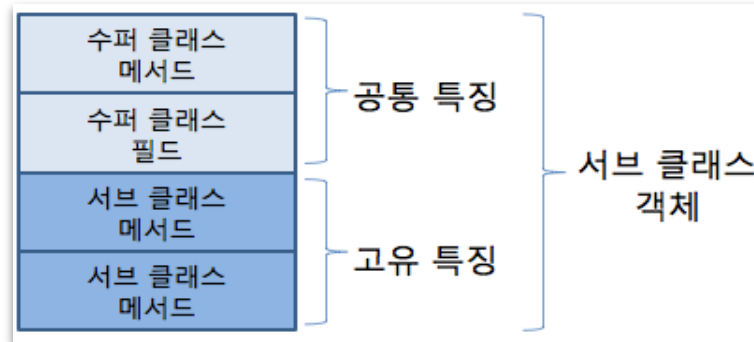
- 메서드 재정의(method overriding)
  - 서브 클래스에서 수퍼 클래스의 메서드를 재정의하여 기능을 변경하는 것

```
// 일반 사원 클래스
class RegularEmployee extends Employee {
    public double payCheck() {    // 급여 계산
        return salary;
    }
    protected double salary;      // 급여
}

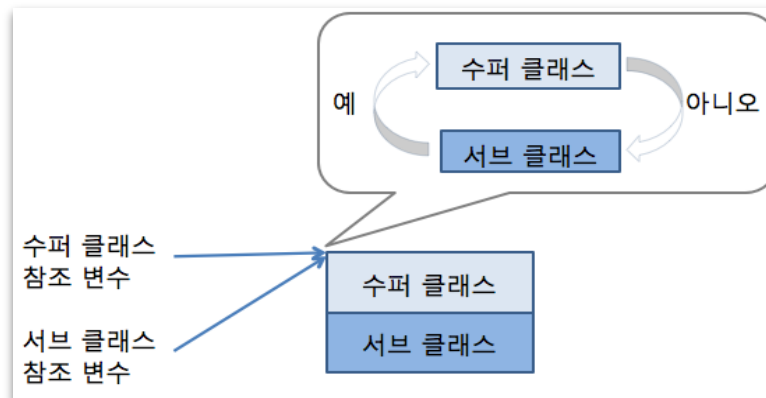
// 영업 사원 클래스
class SalesEmployee extends RegularEmployee {
    public double payCheck() {    // 급여 계산
        return super.salary + (sales * commission);
    }
    // 생략...
}
```

# 수퍼 클래스와 서브 클래스 사이 변환

- 서브 클래스 객체



- 수퍼 클래스와 서브 클래스 타입 변환



# 수퍼 클래스와 서브 클래스 사이 변환

---

- 부서 클래스 예 :

```
// Department.java 파일
// 부서 클래스
class Department {
    public Department() {
        employees = new Employee[10];
        headCount = 0;
    }
    public void addEmployee(Employee e) {
        if (headCount < 10)
            employees[headCount++] = e;
    }
    public void display() {
        for(int i = 0; i < headCount; ++i) {
            employees[i].displayEmployee();
        }
    }
    private int headCount;
    private Employee [] employees;
}
```

# 수퍼 클래스와 서브 클래스 사이 변환

- 부서 클래스 사용 예 :

```
public static void main(String [] args)
{
    Date date1 = new Date(2012, 8, 1);
    Date date2 = new Date(2000, 8, 2);
    RegularEmployee regEmp = new RegularEmployee("김일", "서울시",
                                                    "123-4567", date1, 500000);
    SalesEmployee saleEmp = new SalesEmployee("김이", "인천시",
                                                "234-5678", date2, 200000, 0.2);
    saleEmp.setSales(30000);                      // 영업 실적을 저장한다

    Department dept = new Department(); // 부서를 생성한다
    dept.addEmployee(regEmp);                // 일반사원을 추가한다
    dept.addEmployee(saleEmp);               // 영업사원을 추가한다
    dept.display();                          // 사원 정보를 출력한다
}
```

# 상속성, 그 이상

---

- 사원 급여액 출력 기능 추가
  - 사원 클래스

```
// Employee.java 파일
// 사원 클래스
class Employee {
    public double payCheck() {    // 사원 급여 계산
        return 0;
    }
    // 생략...
}
```

- 부서 클래스

```
// Department.java 파일
public void display() {
    for(int i = 0; i < headCount; ++i) {
        employees[i].displayEmployee();
        double pay = employees[i].payCheck();
        System.out.printf("급여액 : %f\n", pay);
    }
}
```

# 상속성, 그 이상

---

- 돌발 퀴즈!
  - 프로그램 실행 결과는?  
1번)

사원명 : 김일, 주소 : 서울시, 전화번호 : 123-4567, 입사일 : 2012-8-1

급여액 : 0.0

사원명 : 김이, 주소 : 인천시, 전화번호 : 234-5678, 입사일 : 2000-8-2

급여액 : 0.0

2번)

사원명 : 김일, 주소 : 서울시, 전화번호 : 123-4567, 입사일 : 2012-8-1

급여액 : 500000

사원명 : 김이, 주소 : 인천시, 전화번호 : 234-5678, 입사일 : 2000-8-2

급여액 : 206000



# Object 클래스

---

- Object 클래스 = 모든 Java 클래스의 수퍼 클래스

메서드	기능
Object clone()	객체를 복제하여 똑같은 새로운 객체를 생성한다
boolean equals()	다른 객체와의 동일성을 비교한다
void finalize()	소멸자
Class<?> getClass()	실행 시에 객체의 클래스를 반환한다
int hashCode()	객체의 해시 코드를 반환한다
void notify()	객체를 기다리는 단일 스레드를 깨운다
void notifyAll()	객체를 기다리는 모든 스레드를 깨운다
String toString()	객체를 표시하는 문자열을 반환한다
void wait()	다른 스레드의 실행을 기다린다
void wait(long milliseconds)	
void wait(long milliseconds, int nanoseconds)	

# Object 클래스

---

- equals() 메서드
  - 두 객체의 객체를 비교하여 같으면 true, 다르면 false를 반환함
  - 사용 예 :

```
Employee e = new RegularEmployee("전병선", "서울시", "123-4567",  
                                   joindate, 10000000000);  
  
Object obj1 = e;  
Object obj2 = e;  
boolean b = obj1.equals(obj2);           // b == true
```

# Object 클래스

---

- equals() 메서드 재정의
  - 사원 클래스 예 :

```
// 사원 클래스
class Employee {
    public boolean equals(Object object) {
        Employee e = (Employee)object;
        if(name == e.name &&
            address == e.address &&
            telno == e.telno)
            return true;
        else
            return false;
    }
    // 생략...
}
```

# Object 클래스

---

- equals() 메서드 재정의
  - 사용 예 :

```
RegularEmployee o1 = new RegularEmployee("전병선", "서울시", "123-4567",  
                                           joindate1, 10000000000);  
RegularEmployee o2 = new RegularEmployee("전병선", "서울시", "123-4567",  
                                           joindate2, 20000000000);  
  
boolean b = o1.equals(o2);           // b == true
```

# Object 클래스

---

- toString() 메서드
  - 객체를 표시하는 문자열을 반환함

```
employee.RegularEmployee@64cfe0
```

# Object 클래스

---

- toString() 메서드 재정의
  - Date 클래스 예 :

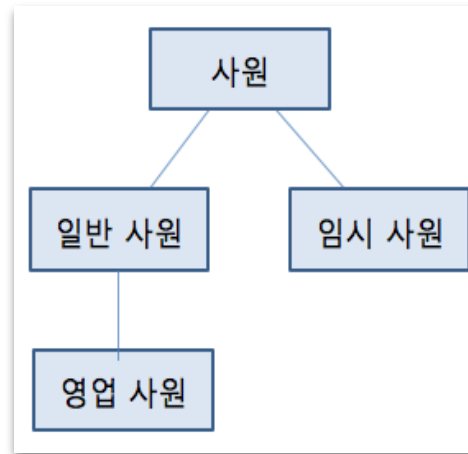
```
// Date 클래스
class Date {
    public String toString() {
        return String.format("%d-%d-%d", year, month, day);
    }
    // 생략...
}
```

- 사용 예 :

```
// Employee.java 파일
public void displayEmployee(){    // 사원 정보 표시
    System.out.printf("이름 : %s, 주소 : %s, 전화번호 : %s, 입사일 : %s\n",
        name, address, telno, joindate );
}
```

# 실습

- 다음 그림과 같은 계층도를 표현하는 클래스를 구현한다.



- 직원은 이름과 전화번호 등의 일반 정보를 관리한다.
- 일반직원은 급여를 받는다.
- 임시직원은 시간 당 수당을 받는다.
- 영업직원은 급여 외에도 영업 실적에 따른 수당을 추가로 받는다.
- 계층도를 사용하는 코드를 구현한다.
  - 각 2명씩의 직원 객체를 생성한다.
  - 직원 일반정보를 출력한다.
  - 직원 급여정보를 출력한다.

# 실습

---

- 부서 클래스를 구현한다.
  - 부서는 최대 10명의 사원이 있다.
  - 앞에서 생성된 사원 객체를 부서에 추가한다.
  - 부서에 포함된 모든 사원의 일반 정보를 출력한다.
  - 부서에 포함된 모든 사원의 급여 정보를 출력한다.
- 두 객체를 비교하는 코드를 작성한다.
  - Object 클래스의 equals() 메서드를 사용한다.
  - 동일한 이름과 전화번호를 갖는 일반사원과 영업사원 객체를 생성하고 비교하고 그 결과를 확인한다.
  - 이름과 전화번호가 같은 객체는 동일한 객체로 간주할 수 있도록 코드를 작성한다.



## 7. 다형성과 인터페이스

## 7. 다형성과 인터페이스

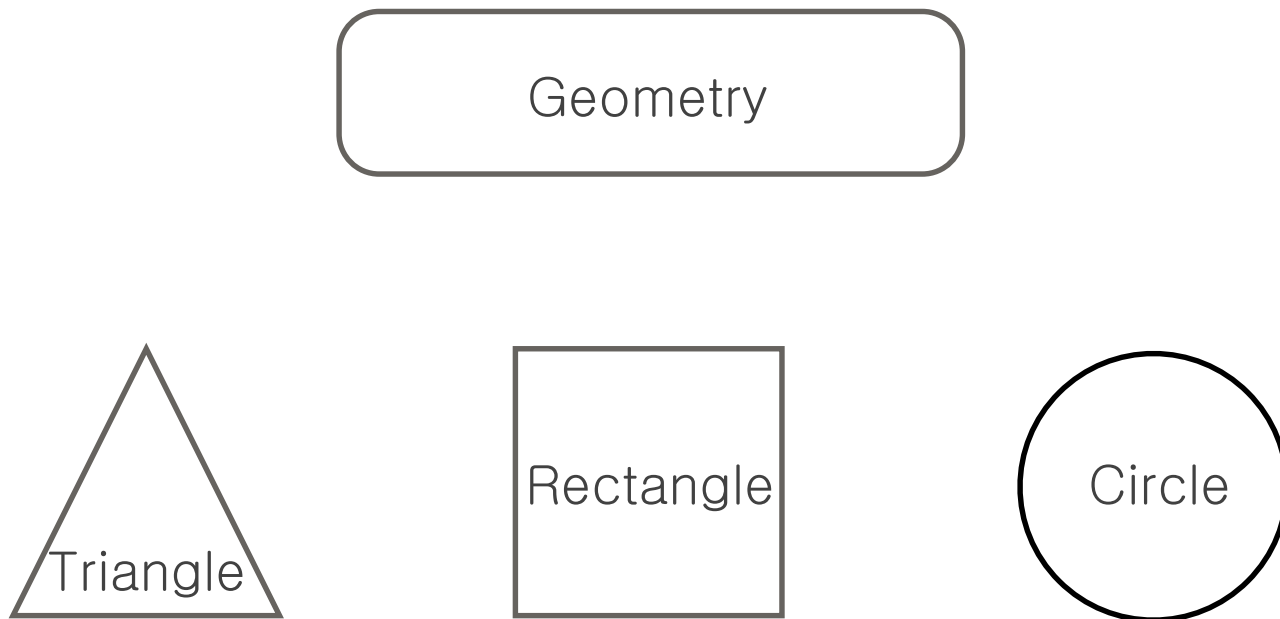
---

- 다형성
- 가상 메서드
- 동적 바인딩
- 추상 클래스
- 인터페이스 정의
- 인터페이스 구현 클래스 정의
- 인터페이스 구현 클래스 활용
- 인터페이스와 다형성
- 인터페이스와 다중 상속

# 다형성과 인터페이스

---

- 다형성(polymorphism)
  - poly(많음) + morph(형태) : 여러 형태를 띠는 것
  - 같은 메시지에 대해 객체가 서로 다르게 반응하는 것
- 도형 클래스 예



# 다형성

---

- Geometry 클래스

```
class Geometry {  
    public Geometry(int x, int y) {  
    }  
    public void draw() {  
    }  
    protected int x;           // x 좌표  
    protected int y;           // y 좌표  
}
```

# 다형성

---

- Triangle 클래스

```
class Triangle extends Geometry {  
    public Triangle(int x, int y, int x1, int y1, int x2, int y2) {  
    }  
    public void draw() {                // 삼각형을 그린다  
    }  
    private int x1;                    // 꼭지점1 x 좌표  
    private int y1;                    // 꼭지점1 y 좌표  
    private int x2;                    // 꼭지점2 x 좌표  
    private int y2;                    // 꼭지점2 y 좌표  
}
```

# 다형성

---

- Rectangle 클래스

```
class Rectangle extends Geometry {  
    public Rectangle(int x, int y, int width, int height) {  
    }  
    public void draw() {           // 사각형을 그린다  
    }  
    private int width;           // 넓이  
    private int height;          // 높이  
}
```

# 다형성

---

- Circle 클래스

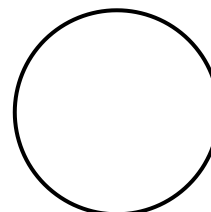
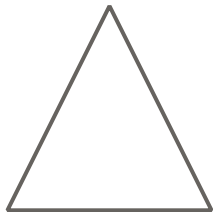
```
class Circle extends Geometry {  
    public Circle(int x, int y, int radius) {  
    }  
    public void draw() {           // 원형을 그린다  
    }  
    private int radius;           // 반지름  
}
```

# 다형성

---

- Geometry 클래스 사용

```
Triangle tri = new Triangle(10, 20, 11, 21, 12, 22);    // 삼각형
Rectangle rect = new Rectangle(20, 30, 100, 200);        // 사각형
Circle cir = new Circle(30, 40, 300);                  // 원형
Geometry [] geos = new Geometry[3];
geos[0] = tri;
geos[1] = rect;
geos[2] = cir;
for(Geometry geo : geos)
    geo.draw();                                         // 도형 그리기
```





# 가상 메서드

---

- 가상 메서드(virtual method)
  - 서브 클래스에서 재정의(overriding)될 것으로 기대되는 메서드
  - Java 언어에서는 특별히 지정하지 않아도 슈퍼 클래스의 메서드를 서브 클래스에서 재정의하는 경우에 자동적으로 가상 메서드가 됨
    - Geometry 클래스의 draw() 메서드
    - Employee 클래스의 payCheck() 메서드

# 동적 바인딩

---

- 정적 바인딩(static binding)
  - 초기 바인딩(early binding)
  - 컴파일 시에 어떤 메서드를 호출되는 지 결정됨
  - 일반 메서드
- 동적 바인딩(dynamic binding)
  - 지연 바인딩(late binding)
  - 실행 시에 어떤 메서드를 호출되는 지 결정됨
  - 가상 메서드

# 추상 클래스

---

- Geometry 클래스의 문제점과 해결 방안
  - 문제점
    - draw() 메서드가 구현 코드를 포함하고 있음
    - 인스턴스를 생성할 수 없음
  - 해결 방안
    - 추상 클래스로 정의함

# 추상 클래스

---

- 추상 메서드(abstract method)
  - 서브 클래스에서 반드시 재정의(overriding)되어야 하는 메서드
  - 구현 코드를 포함하지 않음
- 추상 클래스(abstract class)
  - 추상 메서드를 포함하는 클래스
  - 인스턴스를 생성할 수 없음
  - 파생 클래스에서 추상 메서드를 재정의하지 않으면 해당 클래스도 추상 클래스가 됨

# 추상 클래스

---

- 추상 클래스 정의
  - abstract 키워드 사용

```
// 도형 클래스
```

```
abstract class Geometry {
```

```
    public Geometry(int x, int y) {
```

```
    }
```

```
    public abstract void draw();
```

```
    protected int x;
```

```
    protected int y;
```

```
}
```

```
// 추상 클래스
```

```
// 추상 메서드
```

# 인터페이스 정의

---

- 인터페이스(interface)
  - 클래스가 제공하는 서비스를 명시하는데 사용되는 행위의 집합
  - 추상 메서드로만 구성된 추상 클래스

```
interface 인터페이스명 {  
    //    인터페이스 멤버 선언  
}
```

- 인터페이스의 모든 메서드는 공개(public) 메서드임

# 인터페이스 정의

---

- 맞춤법 검사기 예
  - 맞춤법을 검사할 언어를 선택하는 옵션
  - 맞춤법 검사할 단어 목록 저장
  - 맞춤법 검사 실행

```
// SpellingChecker.java 파일
interface SpellingChecker {
    String [] check();
    void setWords(String [] words);
    void setLanguage(int lang);
}
```

# 인터페이스 구현 클래스 정의

---

- 인터페이스 구현 클래스
  - implements 키워드 사용

```
class 파생클래스명 implements 인터페이스명 {  
    // 인터페이스 멤버 구현  
}
```

- 맞춤법 검사기 구현 클래스
  - 인터페이스 메서드 재정의

```
// SpellingCheckerImpl.java 파일  
class SpellingCheckerImpl implements SpellingChecker {  
    public String[] check() {  
    }  
    public void setWords(String[] words) {  
    }  
    public void setLanguage(int lang) {  
    }  
}
```



# 인터페이스 구현 클래스 정의

---

- 인터페이스 구현 클래스
  - 맞춤법 검사기 구현 클래스
  - 필드 멤버

```
// SpellingCheckerImpl.java 파일
class SpellingCheckerImpl implements SpellingChecker {
    // 생략...
    private String [] words;
    private int language;
    private String [] wordsKorean = { "사랑", "평화", "행복" };
    private String [] wordsEnglish = { "love", "peace", "happiness" };
}
```

# 인터페이스 구현 클래스 정의

---

- 인터페이스 구현 클래스 사용
  - 클래스의 인스턴스를 생성한다

```
SpellingCheckerImpl oSpellingchecker= new SpellingCheckerImpl();
```

- 생성된 인스턴스로부터 인터페이스를 구한다

```
SpellingChecker spellingchecker = oSpellingchecker;
```

- 인터페이스를 통하여 메서드를 호출한다

```
spellingchecker.check();
```

# 인터페이스와 다형성

---

- 맞춤법 검사기 예에서 배운 것
  - 맞춤법 검사기가 SpellingChecker를 구현하고 있다면 문서편집기는 어떤 맞춤법 검사기도 사용할 수 있게 된다
  - 맞춤법 검사기는 문서편집기에 끼워넣을 수 있는 부품이 된다
- 인터페이스와 다형성
  - 컴포넌트 기반 개발(CBD, component-based development)의 기반이 됨
  - 이미 하드웨어 분야는 컴포넌트를 활용하고 있음
    - 조립식 컴퓨터
    - 오디오 컴포넌트 시스템
    - 조선 산업 분야 ...
- 인터페이스 사용의 이점
  - 사용 측 코드의 변경 없이 다른 클래스로 대체할 수 있다
  - 사용 측 코드의 변경 없이 클래스의 내부 구현 방법을 손쉽게 변경할 수 있다

# 인터페이스와 다중 상속

---

- 인터페이스 다중 상속(multiple inheritance)
  - 클래스가 여러 인터페이스를 구현하는 것
- 맞춤법 검사기에 사전 찾기 기능 추가 예
  - Dictionary 인터페이스

```
public interface Dictionary {  
    String find(String word);  
}
```

- 맞춤법 검사기 클래스

```
// SpellingCheckerImpl.java 파일  
class SpellingCheckerImpl implements SpellingChecker, Dictionary {  
    // SpellingChecker 인터페이스  
    // Dictionary 인터페이스  
}
```

# 인터페이스와 다중 상속

- 맞춤법 검사기에 사전 찾기 기능 추가 예
  - Dictionary 인터페이스 사용

```
// WordProcessor.java 파일
class WordProcessor {
    public static void main(String[] args) {
        // SpellingCheckerImpl 클래스 인스턴스 생성
        SpellingCheckerImpl spellingchecker = new SpellingCheckerImpl();
        // SpellingChecker 인터페이스 사용
        SpellingChecker sc = spellingchecker;
        sc.check(...)
        // Dictionary 인터페이스 사용
        Dictionary dic = spellingchecker;
        result = dic.find("사랑");
        System.out.println(result);
        // 생략...
    }
}
```

# 실습

---

- 도형 클래스 구현
  - 3개 이상의 다른 유형의 도형 클래스를 정의한다. 모든 각 도형 클래스는 도형을 그리는 메서드를 포함한다.
  - 이들 도형 클래스로부터 공통적인 Geometry 클래스를 정의한다. 각 도형 클래스에 Geometry 클래스를 슈퍼 클래스로 지정한다.
  - 각 도형 클래스의 객체를 생성한다.
  - Geometry 배열에 각 도형 클래스 객체를 저장한다.
  - Geometry 배열의 각 요소에 대하여 도형을 그리는 메서드를 호출하여 도형을 표현한다
- 도형 클래스와 사원 클래스
  - 도형 클래스와 사원 클래스의 공통적인 문제는 무엇인가?
  - 이 문제를 해결하도록 코드를 변경시킨다.

# 실습

---

- 맞춤법 검사기 구현
  - SpellingChecker 인터페이스를 정의한다.
    - 언어 선택 기능
    - 검사할 단어 지정 기능
    - 맞춤법 검사 기능
  - SpellingChecker 를 구현하는 클래스를 정의한다.
    - 표준 단어 목록을 정의한다
    - 인터페이스 메서드를 정의한다
- 워드프로세스 구현
  - 맞춤법 검사기를 사용하는 코드를 구현한다

## 8. 클래스 고급



## 8. 클래스 고급

---

- 메서드 인수 전달 방식
- 메서드 오버로딩
- 가변 인수
- final 키워드
- 정적 멤버
- 패키지
- 어노테이션

# 메서드 인수 전달 방식

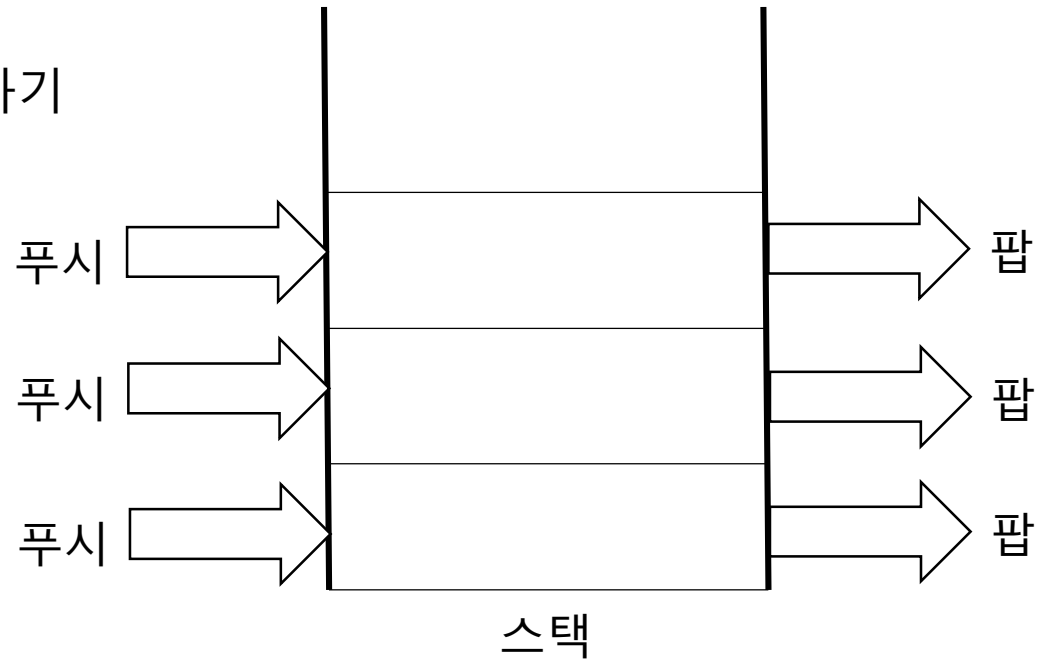
---

- 인수 전달 방식
  - 값으로 호출(call-by-value)
  - 참조로 호출(call-by-reference)
- 스택(stack)
  - LIFO(Last-In-First-Out) 구조를 갖는 메모리 영역
    - 마지막으로 들어온 것이 제일 처음에 나가게 된다(예: 접시)
  - 푸시(push) : 스택에 메모리를 할당하는 것
  - 팝(pop) : 스택에 할당된 메모리를 해제하는 것
  - 메서드에 전달되는 인수와 지역 변수(local variable)는 스택 메모리에 저장됨

# 메서드 인수 전달 방식

---

- 푸시(push)
  - 스택에 저장하기
- 팝(pop)
  - 스택에서 제거하기



# 메서드 인수 전달 방식

---

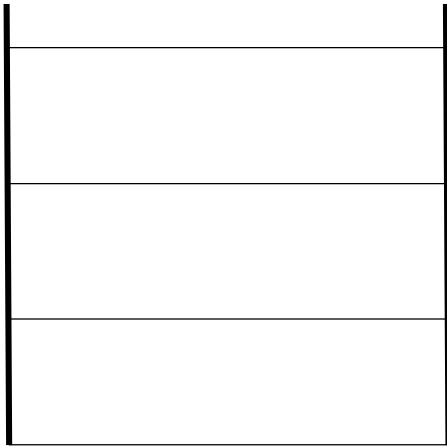
- 값으로 호출 방식

```
void method(int value) {  
    value = 100;  
}  
  
void foo() {  
    int i = 0;  
    method(i);           // 값으로 호출  
    System.out.println(i); // i == 0  
}
```

# 메서드 인수 전달 방식

---

- 값으로 호출 방식



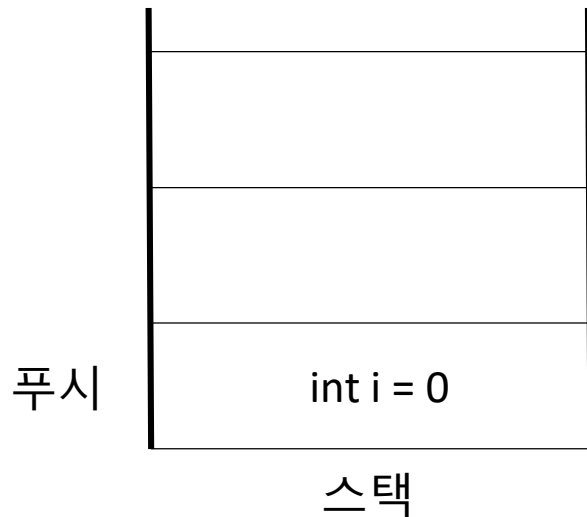
스택

```
void foo() {  
    int i = 0;  
    method(i);           // 값으로 호출  
    System.out.println(i); // i == 0  
}
```

# 메서드 인수 전달 방식

---

- 값으로 호출 방식

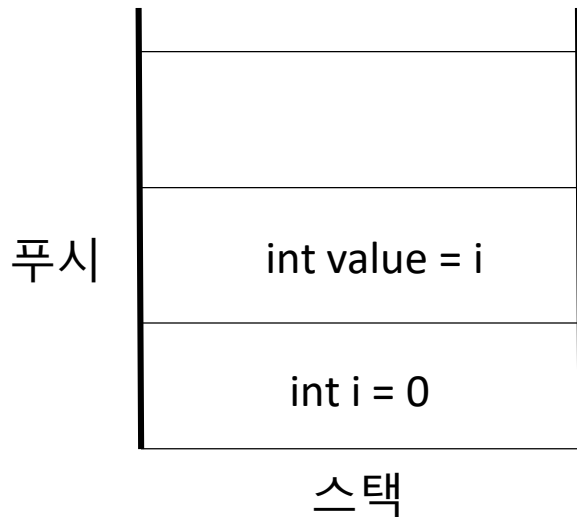


```
void foo() {  
    int i = 0;  
    method(i);           // 값으로 호출  
    System.out.println(i); // i == 0  
}
```

# 메서드 인수 전달 방식

---

- 값으로 호출 방식

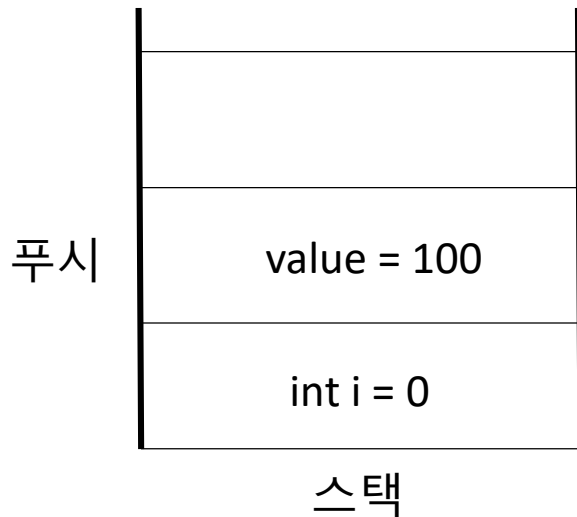


```
void foo() {  
    int i = 0;  
    method(i);           // 값으로 호출  
    System.out.println(i); // i == 0  
}  
  
void method(int value) {  
    value = 100;  
}
```

# 메서드 인수 전달 방식

---

- 값으로 호출 방식



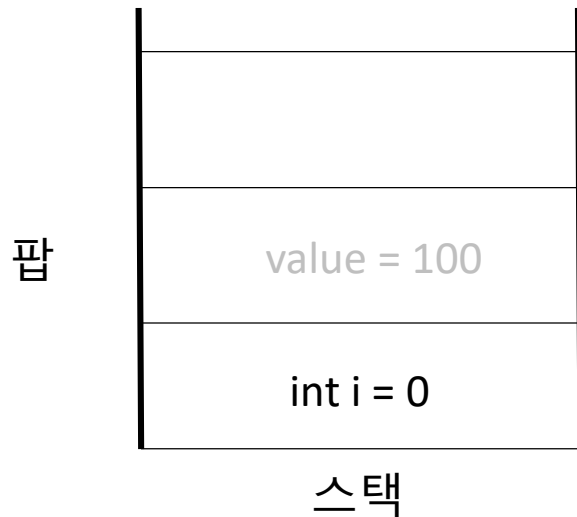
```
void foo() {  
    int i = 0;  
    method(i);           // 값으로 호출  
    System.out.println(i); // i == 0  
}  
  
void method(int value) {  
    value = 100;  
}
```



# 메서드 인수 전달 방식

---

- 값으로 호출 방식

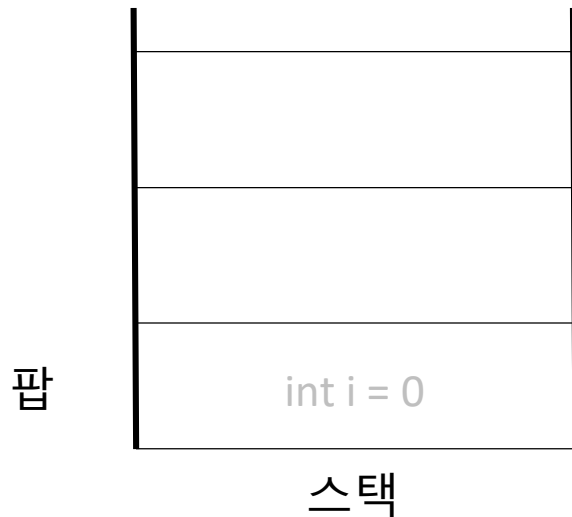


```
void foo() {  
    int i = 0;  
    method(i);           // 값으로 호출  
    System.out.println(i); // i == 0  
}  
  
void method(int value) {  
    value = 100;  
}
```

# 메서드 인수 전달 방식

---

- 값으로 호출 방식



```
void foo() {  
    int i = 0;  
    method(i);           // 값으로 호출  
    System.out.println(i); // i == 0  
}  
  
void method(int value) {  
    value = 100;  
}
```

# 메서드 인수 전달 방식

---

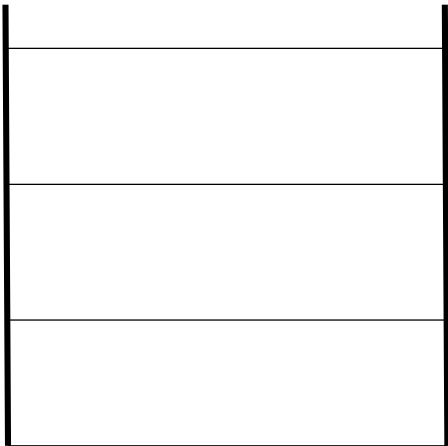
- 참조로 호출 방식

```
void method(MyClass r) {  
    r.setValue(100);  
}  
  
void foo() {  
    MyClass o = new MyClass();  
    method(o);                // 참조로 호출  
    int value = o.getValue();  // value == 100  
}
```

# 메서드 인수 전달 방식

---

- 참조로 호출 방식

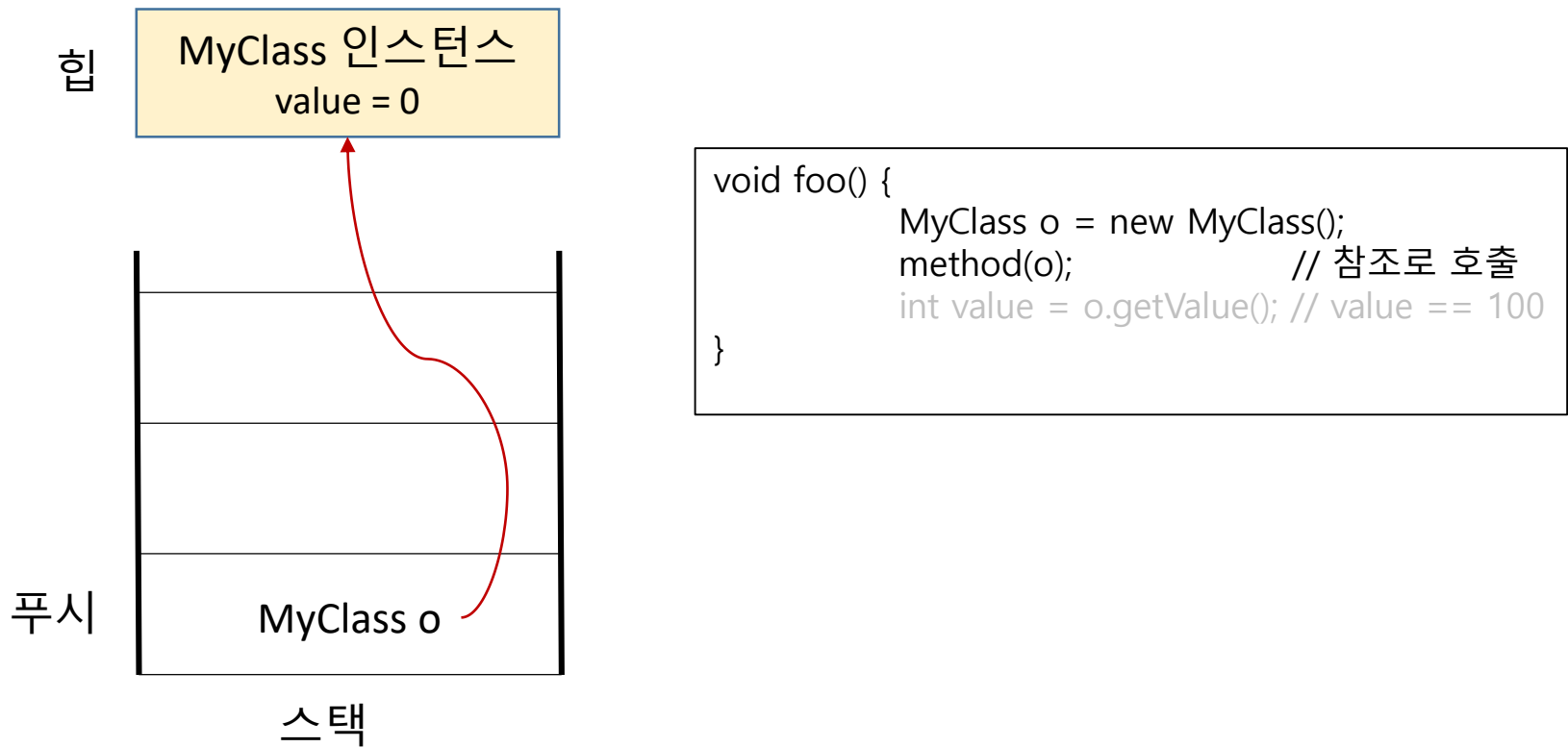


스택

```
void foo() {  
    MyClass o = new MyClass();  
    method(o);           // 참조로 호출  
    int value = o.getValue(); // value == 100  
}
```

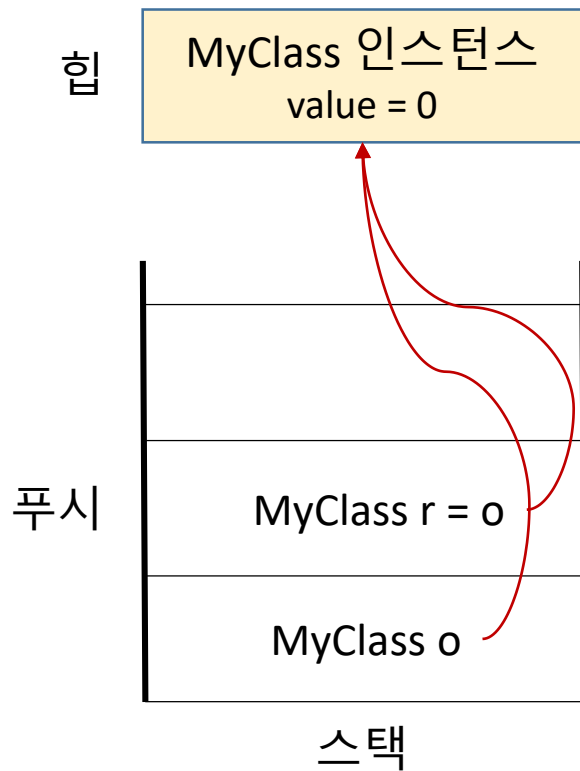
# 메서드 인수 전달 방식

- 참조로 호출 방식



# 메서드 인수 전달 방식

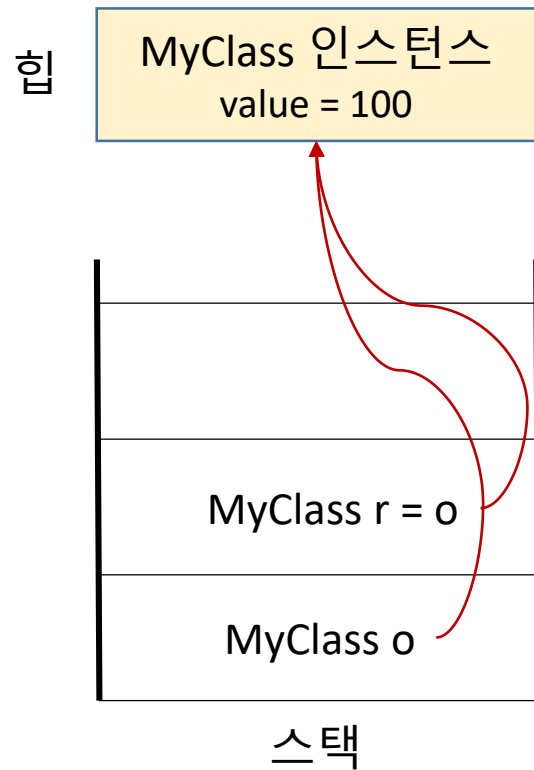
- 참조로 호출 방식



```
void foo() {  
    MyClass o = new MyClass();  
    method(o);           // 참조로 호출  
    int value = o.getValue(); // value == 100  
}  
  
void method(MyClass r) {  
    r.setValue(100);  
}
```

# 메서드 인수 전달 방식

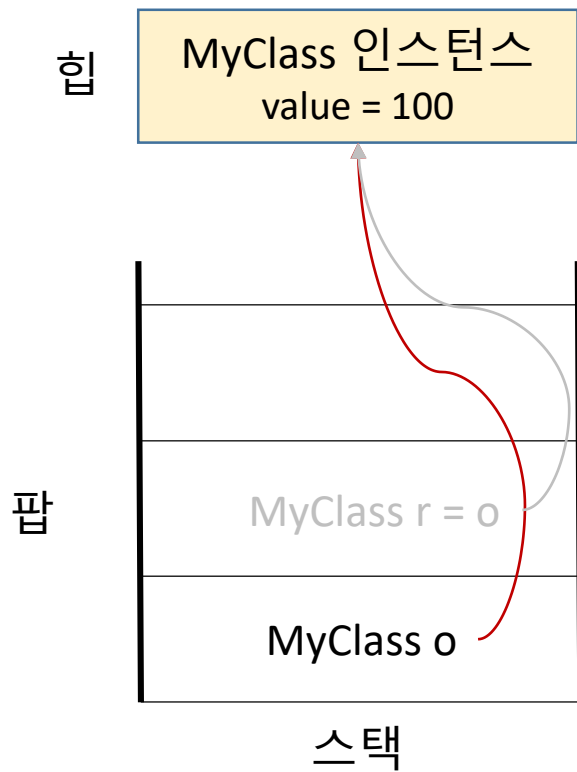
- 참조로 호출 방식



```
void foo() {  
    MyClass o = new MyClass();  
    method(o);           // 참조로 호출  
    int value = o.getValue(); // value == 100  
}  
  
void method(MyClass r) {  
    r.setValue(100);  
}
```

# 메서드 인수 전달 방식

- 참조로 호출 방식

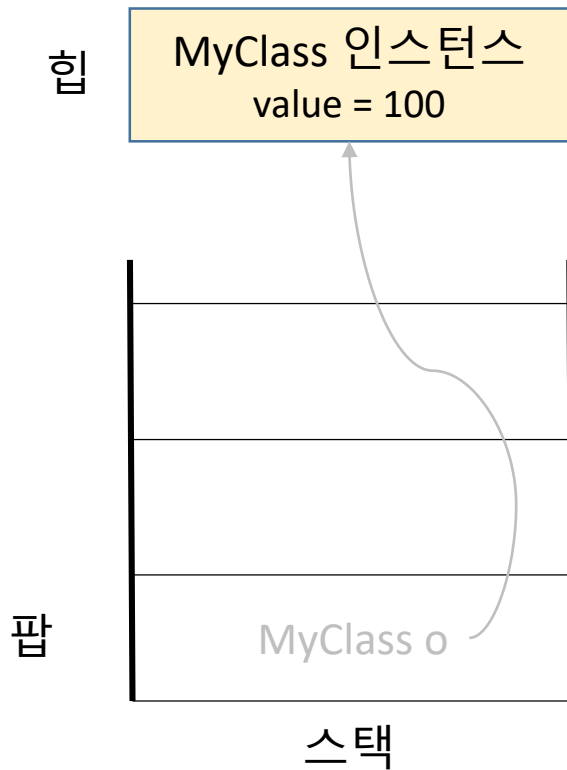


```
void foo() {  
    MyClass o = new MyClass();  
    method(o);           // 참조로 호출  
    int value = o.getValue(); // value == 100  
}  
  
void method(MyClass r) {  
    r.setValue(100);  
}
```



# 메서드 인수 전달 방식

- 참조로 호출 방식



```
void foo() {  
    MyClass o = new MyClass();  
    method(o);           // 참조로 호출  
    int value = o.getValue(); // value == 100  
}  
  
void method(MyClass r) {  
    r.setValue(100);  
}
```

# 메서드 오버로딩

---

- 메서드 오버로딩(method overloading)
  - 매개변수의 타입과 개수, 반환 타입이 다르지만 이름이 같은 여러 메서드를 정의할 수 있는 기능
  - 필요한 사례 :

```
int getIntMul(int x) {  
    return x * x;  
}  
  
float getFltMul(float x) {  
    return x * x;  
}  
  
double getDbIMul(double x) {  
    return x * x;  
}
```

```
int i = 30;  
float f = 30.2;  
double d = 33.3;  
  
i = getIntMul(i);  
f = getFltMul(f);  
d = getDbIMul(d);
```

# 메서드 오버로딩

---

- 메서드 오버로딩(method overloading)
  - 메서드 오버로딩 적용 사례 #1

```
int getMul(int x) {  
    return x * x;  
}  
float getMul(float x) {  
    return x * x;  
}  
double getMul(double x) {  
    return x * x;  
}
```

```
int i = 30;  
float f = 30.2;  
double d = 33.3;  
  
i = getMul(i);        // getMul(int x) 호출  
f = getMul(f);        // getMul(float f) 호출  
d = getMul(d);        // getMul(double d) 호출
```

# 메서드 오버로딩

---

- 메서드 오버로딩(method overloading)
  - 메서드 오버로딩 적용 사례 #2

```
int add(int a, int b) {  
    return a+b;  
}  
int add(int a, int b, int c) {  
    return a+b+c;  
}  
double add(double a, double b) {  
    return a+b;  
}
```

```
int result1, result2;  
double result3;  
result1 = add(100, 200);  
result2 = add(10, 20, 30);  
result3 = add(3.14195, 2.54);
```

# 메서드 오버로딩

---

- 메서드 오버로딩(method overloading)
  - 메서드 오버로딩 적용 사례 #3 - 생성자 오버로딩

```
// 생성자
public Date() {
    year = 1990;
    month = 1;
    day = 1;
}
public Date(int yy, int mm, int dd) {
    setDate(yy, mm, dd);
}
```

```
// Date() 생성자 호출
Date myDate = new Date();

// Date(int yy, int mm, int dd) 생성자 호출
Date myDate = new Date(1800, 20, 90);
```

# 메서드 오버로딩

---

- 메서드 오버로딩(method overloading)
  - 메서드 오버로딩 적용 나쁜 사례
    - 완전히 관련이 없는 메서드

```
void getHome() { }           // 커서를 화면의 원점(0, 0)으로 이동시킨다
String getHome(String name) { } // 매개변수의 name 에 지정된 이름의 주소를 구한
```

다

- 매개변수는 같고 반환 값만 다른 경우 => 에러!

```
int search(String key) { }      // 에러!
String search(String name) { }  // 에러!
```

# 가변 인수

---

- 가변 인수(variable-length argument)
  - 메서드 매개변수의 인수 개수가 가변적인 것
  - String 클래스의 format() 메서드 예 :

```
String format(String format, Object ... args)
```

- 사용 예 :

```
System.out.printf("이름 : %s, 주소 : %s, 전화번호 : %s, 입사일 : %s\n",  
                name, address, telno, joindate );  
String.format("이름 : %s, 주소 : %s, 전화번호 : %s, name, address, telno );
```

# 가변 인수

---

- 가변 인수 정의
  - 말줄임표( ... ) 사용

```
void foo(int ... v)
```

- 가변 인수는 내부적으로 배열로 처리함

```
void foo(int ... v) {  
    System.out.print("인수 개수 : " + v.length + "개 인수값 :");  
    for( int x : v)  
        System.out.print(x + ", ");  
    System.out.println();  
}
```

- 가변 인수 사용

```
foo(10, 20, 30, 40, 50);  
foo(100, 200);  
foo();
```



# 가변 인수

- 가변 인수 정의 규칙
  - 항상 마지막에 위치한다

```
foo(int a, int b, int ... v)
foo(int a, int ... v, int b)    // 에러!
```

- 메서드는 반드시 하나의 가변 인수 만 가질 수 있다

```
foo(int a, int b, int ... v)
foo(int a, double b, int ... va, double ... vb)    // 에러!
```

- 가변 인수를 갖는 메서드도 오버로딩할 수 있다

```
foo(int ... v)
foo(boolean ... v)
foo(String msg, int ... v)
```

- 오버로딩 가변 인수 사용 예

```
foo(10, 20, 30);           // foo(int ... v)
foo(true, false);          // foo(boolean ... v)
foo("가변인수 : ", 100, 200); // foo(String msg, int ... v)
```

# 가변 인수

---

- 가변 인수 오버로딩 사용 시 모호성 발생
  - 사례 #1 :

```
foo();           // 에러!
```

- 사례 #2 :

```
foo(int ... v)  
foo(int i, int ... v)
```

의 경우, 다음 코드 호출 시

```
foo(100);         // 에러!  
foo(100, 200);    // 에러!
```

# final 키워드

---

- 상수 필드(constant field)
  - final 키워드가 지정된 필드
  - 읽기 전용(read-only)
  - 반드시 초기화되어야 하며, 클래스의 메서드에서 값을 변경시킬 수 없음

```
class Geometry {  
    // 생략...  
    protected final int TRIANGLE = 0;           // 삼각형  
    protected final int RECTANGLE = 1;          // 사각형  
    protected final int CIRCLE = 2;             // 원형  
}
```

```
class Triangle extends Geometry {  
    // 생략...  
    public int getGeoType() {                    // 상수 필드 읽기  
        return TRIANGLE;  
    }  
    public void setGeoType(int type) {  
        TRIANGLE = 5;                           // 에러!!!  
    }  
}
```

# final 키워드

---

- 상수 객체(constant object)
  - final 키워드가 지정된 참조 변수
  - 읽기 전용(read-only)
  - 반드시 초기화되어야 하며, 다른 객체로 변경시킬 수 없음

```
final Date myBirthDay = new Date(1960, 9, 9);
```

- 객체 자체의 변경은 가능함

```
myBirthDay.setDate(2019, 8, 15);    // 날짜 값 변경 OK!
```

# final 키워드

---

- 인터페이스 상수 필드
  - 인터페이스에 정의된 상수 필드

```
public interface ISpellingChecker {  
    String [] check();  
    void setWords(String [] words);  
    void setLanguage(int lang);  
    // 상수 필드  
    int KOREAN = 0;  
    int ENGLISH = 1;  
}
```

# final 키워드

---

- 최종 메서드(final method)
  - 서브 클래스에서 재정의 할 수 없음

```
// 일반 사원 클래스
class RegularEmployee extends Employee {
    // 생략...
    public final double getSalary() {          // 최종final 메서드
        return salary;
    }
    protected double salary;
}
```

```
// 영업 사원 클래스
class SalesEmployee extends RegularEmployee {
    // 생략...
    public double getSalary() {                // 에러!!!!
    }
}
```

# final 키워드

---

- 최종 클래스(final class)
  - 서브 클래스를 파생시킬 수 없음

```
// 영업 사원 클래스
```

```
final class SalesEmployee extends RegularEmployee {           // 최종final 클래스
    // 생략...
}
```

```
class SalesManager extends SalesEmployee {                    // 에러!!
    // 생략...
}
```

# 정적 멤버

---

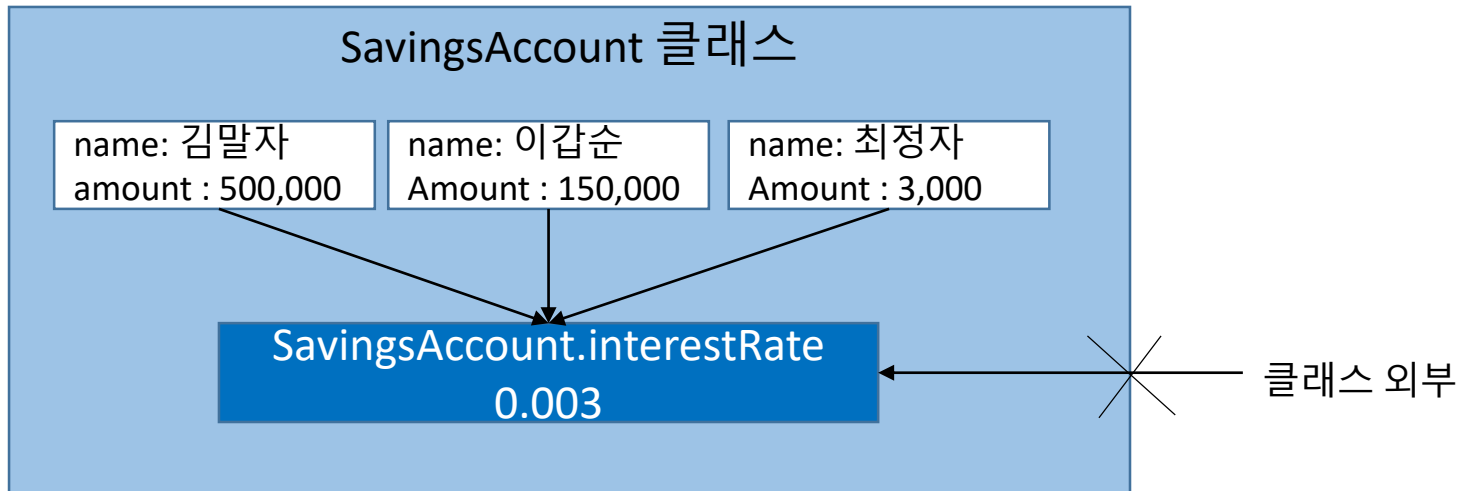
- 예금 계좌 클래스
  - 고객명
  - 예금 잔액
  - 이자율

```
// 은행 계좌 클래스
class SavingsAccount {
    public SavingsAccount() {
    }
    public SavingsAccount(String name, double amount) {
    }
    public void earnInterest() {
    }
    private String name;           // 이름
    private double amount;         // 예금 잔액
    private double interestRate;   // 이자율
}
```



# 정적 멤버

- 이자율 처리



# 정적 멤버

---

- 정적 필드(static field)

- 정적 데이터 영역 안에 저장되므로 프로그램이 실행 중에 계속하여 생명이 유지되며, 전체 프로그램에 대하여 단지 하나의 인스턴스만 갖게 됨
- 해당 클래스에만 국한되어 사용됨
- static 키워드 사용

```
// 은행 계좌 클래스
class SavingsAccount {
    public SavingsAccount() {
    }
    public SavingsAccount(String name, double amount) {
    }
    public void earnInterest() {
    }
    private String name;           // 이름
    private double amount;         // 예금 잔액
    private static double interestRate; // 이자율
}
```

# 정적 멤버

---

- 정적 필드 초기화
  - 필드 선언과 함께 초기화

```
class SavingsAccount {  
    // 생략..  
    private static double interestRate = 0.01;  
}
```

- 정적 코드 블록(static code block)

```
class SavingsAccount {  
    // 생략..  
    private static double interestRate;  
    static {  
        interestRate = 0.01;  
    }  
}
```

# 정적 멤버

---

- 공개 정적 필드 접근

```
class SavingsAccount {  
    // 생략..  
    public static double interestRate = 0.01;  
}
```

- 객체를 통한 접근

```
SavingsAccount myAccount = new SavingsAccount();  
myAccount.interestRate = 0.002;
```

- 클래스를 통한 접근

```
SavingsAccount.interestRate = 0.002;
```

# 정적 멤버

---

- 정적 메서드(static method)
  - static 키워드가 지정된 메서드

```
// 은행 계좌 클래스
class SavingsAccount {
    public static void setInterestRate(double rate) {
        interestRate = rate;
    }
    public static double getInterestRate() {
        return interestRate;
    }
    private static double interestRate = 0.01;
    // 생략...
}
```

# 정적 멤버

---

- 정적 메서드 접근
  - 객체를 통한 접근

```
SavingsAccount myAccount = new SavingsAccount();  
myAccount.setInterestRate(0.002);
```

- 클래스를 통한 접근

```
SavingsAccount.setInterestRate(0.002);
```

# 정적 멤버

---

- 정적 메서드 제약 사항
  - 특정 인스턴스 즉, 객체에 영향을 미치지 않는다
  - this 키워드를 사용할 수 없다
  - 정적 멤버에만 접근할 수 있다
  - 정적 멤버가 아닌 일반 필드와 메서드에는 접근할 수 없다

```
public static void setInterestRate(double rate) {  
    interestRate = rate;  
    amount += interestRate * amount;    // 에러!! 일반 필드에 접근할 수 없음  
    earnInterest();                    // 에러!!! 일반 메서드에 접근할 수 없음  
}
```

# 패키지

---

- 패키지(package)
  - 식별자의 이름을 일정한 영역 안에 그룹화시켜서 중복으로 인해 충돌이 발생하지 않도록 영역을 구분하는데 사용됨
- 패키지 정의
  - package 키워드 사용

`package 패키지명;`

- 클래스가 저장되는 영역을 정의함
- 물리적인 파일 시스템의 디렉터리
- package 문이 생략되면 클래스는 이름이 없는 기본 패키지(default package)에 저장됨



# 패키지

---

- 패키지 계층 구조
  - 패키지 이름을 . 연산자로 구분

```
package mypkg1.mypkg2.mypkg3;
```

- 파일 시스템의 서브 디렉터리에 반영
  - mypkg1mypkg2mypkg3 서브 디렉터리

# 패키지

---

- 패키지에 포함된 클래스 접근
  - 단순 접근

```
employee.Employee emp = new employee.Employee();
```

- 클래스 임포트
  - import 문 사용
  - 패키지 안에 있는 클래스를 임포트한 후에 패키지를 지정하지 않고 해당 클래스에 접근함

```
// Department.java 파일
import employee.Employee;
class Department {
    public void foo() {
        Employee emp = new Employee();
        // 생략..
    }
}
```

# 패키지

- 패키지에 포함된 클래스 접근
  - 패키지 안의 모든 클래스 임포트

```
import employee.*;
```

- 서로 다른 패키지 안의 같은 이름을 갖는 클래스 접근

```
// mypackage1.MyClass 파일
package mypackage1
class MyClass {
    // 생략...
}

// mypackage2.MyClass 파일
package mypackage2
class MyClass {
    // 생략...
}
```

```
mypackage1.MyClass myclass1 = new mypackage1.MyClass();
mypackage2.MyClass myclass2 = new mypackage2.MyClass();
```

# 어노테이션

---

- 어노테이션(annotation)
  - 소스 코드에 추가된 보충 정보, 메타데이터(metadata)
  - 프로그램의 행위를 변경시키거나 의미를 변경시키지 않는다
  - 프로그램을 개발하는 중에 컴파일러 등 다양한 도구에 의해 사용될 수 있다

@어노테이션명

- Java 언어 내장 어노테이션
  - @Override
  - @Deprecated
  - @SuppressWarnings

# 어노테이션

---

- @Override 어노테이션
  - 메서드에 적용
  - 해당 메서드가 수퍼 클래스의 메서드를 재정의(overriding) 하는 것임을 표시함
  - 재정의할 메서드 이름을 잘못 작성했을 때 Java 컴파일러가 컴파일 시 에러를 표시하게 함으로써 오류를 쉽게 찾아낼 수 있게 함

```
class SuperClass {  
    public void overridemethod() {  
    }  
}  
  
class SubClass extends SuperClass {  
    @Override  
    public void overridemethod() {    // 에러!  
    }  
}
```

# 어노테이션

---

- @Deprecated 어노테이션
  - 클래스나 메서드, 필드에 적용
  - 지정된 클래스나 메서드, 필드가 더 이상 사용되지 말아야 함을 표시함

```
class SuperClass {  
    @Deprecated  
    public void oldmethod() {  
    }  
}  
  
void foo() {  
    SuperClass superclass = new SuperClass();  
    superclass.oldmethod();           // 경고!  
}
```

# 어노테이션

---

- @SuppressWarnings 어노테이션
  - 컴파일러가 발생될 특정 경고를 억제하도록 함

```
@SuppressWarnings("deprecation")
void foo() {
    SuperClass superclass = new SuperClass();
    superclass.oldmethod();
}
```

# 실습

---

- 참조로 호출 방식의 예를 구현하고 호출 측의 값이 변경되는지 확인한다.
- 두 숫자의 곱을 반환하는 메서드를 구현한다. 어느 숫자 타입이든 동일한 방식으로 호출할 수 있도록 한다.
- Employee 클래스에 서브 클래스에서 재정의할 수 없는 메서드를 정의한다. 서브 클래스에서 이 메서드를 재정의하려고 할 때 어떤 일이 발생하는 지 확인한다.
- 예금 계좌를 클래스를 구현한다. 이자율은 모든 인스턴스에 대하여 하나만 생성되도록 한다. 이자율은 변경될 수 있으며, 이 경우 모든 인스턴스에 적용된다.
- 새로운 프로젝트를 생성하고, Employee 클래스들을 employee 패키지에 포함시킨다. Employee 클래스를 사용하는 main() 메서드 클래스에서 employee 패키지를 임포트한다.



## 9. 제네릭

## 9. 제네릭

---

- 제네릭
- 제네릭 클래스
- 제네릭 인터페이스
- 제네릭 메서드
- 와일드카드 인수 사용

# 제네릭

---

- 제네릭(generic)
  - 매개변수의 데이터 타입에 기초한 메서드와 클래스를 생성하는 매커니즘
  - 매개변수가 있는 데이터 타입(parameterized type)
  - 제네릭 메서드(generic method)
  - 제네릭 클래스(generic class)

# 제네릭

---

- 스택(stack) 클래스 예 :
  - 데이터 타입에 따라 여러 버전의 스택 클래스가 필요함

```
// int 데이터 타입 스택
class Stack_Integer {
    public Stack_Integer(int size) { }
    public void push(int value) { }
    public int pop() { }
    private int [] v;
}

// double 데이터 타입 스택
class Stack_Double {
    public Stack_Double(int size) { }
    public void push(double value) { }
    public double pop() { }
    private double [] v;
}
```

```
// String 데이터 타입 스택
class Stack_String {
    public Stack_String(int size) { }
    public void push(String value) { }
    public String pop() { }
    private String [] v;
}

// 등등...
```

# 제네릭

---

- 스택(stack) 클래스 예 :
  - Object 타입의 스택 클래스

```
// Object 타입 스택
class Stack {
    public Stack(int size) { }
    public void push(Object value) { }
    public Object pop() { }
    private Object [] v;
}
```

- 활용 코드

```
// int 데이터 타입으로 스택 사용
Stack is = new Stack(10);
is.push(100);
int iv = (int)is.pop();

// String 데이터 타입으로 스택 사용
Stack ss = new Stack(10);
ss.push("문자열");
String sv = (String)ss.pop();
```

# 제네릭

---

- 스택(stack) 클래스 예 :
  - Object 타입의 스택 클래스의 문제점
    - 특정한 데이터 타입으로 변환을 해야 한다

```
// #1 데이터 변환이 필요함  
String sv = (String)ss.pop();
```

- 스택에 어떤 특정한 데이터 타입의 데이터 만을 저장하게 할 수 없다

```
// #2 여러 데이터 타입의 데이터가 저장됨  
Stack stack = new Stack(10);  
stack.push(100);           // int 데이터 타입 데이터 저장  
stack.push(200.123);       // double 데이터 타입 데이터 저장  
stack.push("문자열");      // String 데이터 타입 데이터 저장  
Object o = stack.pop();    // ?? 어떤 데이터 타입으로 변환해야 하지?
```

# 제네릭

---

- 제네릭(generic)의 이점
  - 제네릭을 사용할 때 코드를 작성하기 쉽다
    - 모든 경우의 클래스 또는 메서드를 작성하는 대신에, 일반적인 하나의 클래스 또는 메서드를 작성하면 된다
  - 제네릭을 사용할 때 코드를 이해하기 쉽다
    - 제네릭은 데이터 타입 정보를 추상화하는 직접적인 방법을 제공하기 때문에 코드를 이해하기가 쉽다
  - 제네릭은 데이터 타입을 보장(type-safe)한다
    - 제네릭의 데이터 타입은 컴파일 시에 알려지기 때문에 컴파일러는 에러가 발생하기 전에 타입 검사(type checking)를 수행할 수 있다

# 제네릭 클래스

---

- 구문

```
class 클래스명 < 타입매개변수 > {  
    // 멤버...  
}
```

- 타입 매개변수(type parameter)
  - 객체가 생성될 때 클래스에 적용되는 실제 데이터 타입을 지정하는 플레이스홀더(placeholder)
  - T 를 많이 사용함



# 제네릭 클래스

---

- 스택 제네릭 클래스

```
// Stack.java 파일
// Stack 제네릭 클래스
class Stack < T > {
    public Stack(int size) {
        v = (T []) new Object[size];
        count = 0;
    }
    public void push(T item) {
        v[count++] = item;
    }
    public T pop() {
        return v[--count];
    }
    public int size() {
        return count;
    }
    private T [] v;
    private int count;
}
```

# 제네릭 클래스

- 스택 제네릭 클래스 사용

```
Stack<Integer> si = new Stack<Integer>(10);
```

- 제네릭 타입 매개변수에 사용할 수 있는 타입
  - 클래스 타입
  - 기본 데이터 타입을 사용할 수 없으며, 대응되는 Java 라이브러리 클래스를 사용해야 함

Java 라이브러리 클래스	기본 데이터 타입
Integer	int
Short	short
Long	long
Byte	byte
Float	float
Double	double
Character	char
Boolean	boolean

# 제네릭 클래스

---

- 스택 제네릭 클래스 사용

```
Stack<Character> sc = new Stack<Character>(10);
Stack<Point> sp = new Stack<Point>(5);

sc.push('c');
char c = sc.pop();
sp.push(new Point(10, 20));
Point p = sp.pop();

sc.push(200.23); // 에러! double 데이터 타입을 사용함
sc.push("문자열"); // 에러! String 데이터 타입을 사용함
```

# 제네릭 클래스

---

- 맵(map) 제네릭 클래스
  - 키(key)와 대응되는 값(value)로 구성되는 데이터 구조를 표현

```
// Map.java 파일
// 맵 제네릭 클래스
class Map <T, V> {
    public Map(int size) {
        key = (T [])new Object[size];
        value = (V [])new Object[size];
        count = 0;
    }
    public void push(T k, V v) {
        key[count] = k;
        value[count] = v;
        count++;
    }
}
```

# 제네릭 클래스

---

- 맵(map) 제네릭 클래스

```
public V get(T k) {  
    for(int i = 0; i < count; ++i) {  
        if(key[i] == k)  
            return value[i];  
    }  
    return null;  
}  
  
private T [] key;  
private V [] value;  
private int count;  
}
```

- 맵(map) 제네릭 클래스 사용

```
Map<Integer, String> map = new Map<Integer, String>(5);  
for(int i = 0; i < 5; i++)  
    map.push(i, "문자열"+i);  
String s = map.get(3);      // s == “문자열3”
```

# 제네릭 클래스

- 타입 매개변수의 타입 제한

< T extends 수퍼클래스 >

- 숫자 타입으로 제한

```
class Map <T extends Number, V> {  
    public Map(int size) {  
        key = (T [])new Number[size];  
        value = (V [])new Object[size];  
        count = 0;  
    }  
    // 생략..  
}
```

```
Map<String, String> map = new Map<String, String>(3);    // 에러!!!
```

# 제네릭 인터페이스

---

- 구문

```
interface 인터페이스명 < 타입매개변수 > {  
    // 멤버...  
}
```

- 타입 매개변수(type parameter)
  - T 를 많이 사용함
- 스택 제네릭 인터페이스

```
// IStack.java 파일  
// IStack 제네릭 인터페이스  
interface IStack < T > {  
    void push(T item);  
    T pop();  
    int size();  
}
```

# 제네릭 인터페이스

---

- 스택 제네릭 인터페이스 구현

```
class Stack < T > implements IStack < T > {  
    public void push(T item) {  
        v[count++] = item;  
    }  
    public T pop() {  
        return v[--count];  
    }  
    public int size() {  
        return count;  
    }  
    // 생략...  
}
```



# 제네릭 인터페이스

---

- 스택 제네릭 인터페이스 사용

```
IStack<Integer> si = new Stack<Integer>(10);  
si.push(10);  
int d = si.pop();  
IStack<Point> sp = new Stack<Point>(5);  
sp.push(new Point(10, 20));  
Point p = sp.pop();
```

# 제네릭 메서드

---

- 구문

```
< 타입매개변수 > 메서드명 {  
}
```

- 타입 매개변수(type parameter)
  - T 를 많이 사용함

# 제너릭 메서드

- min() 오버로딩 메서드의 제너릭 버전

```
< T > T min (T x, T y) {  
    // x 와 y 를 비교하여 작은 값을 반환한다  
}
```

- T 타입의 변수에 비교 연산자를 사용할 수 없다
- Comparable 인터페이스를 구현한 타입으로부터 파생되도록 지정해야 한다

```
< T extends Comparable<T>> T min( T x, T y) {  
    return x.compareTo(y) < 0 ? x : y;  
}
```

- min() 제너릭 메서드 사용

```
int i = min(100, 200);           // i == 100  
double d = min(200.1, 300.1);    // d == 200.1  
String s = min("당신을", "사랑합니다"); // s == "당신을"
```

# 와일드카드 인수 사용

- Stack 제네릭 클래스의 compareSize() 메서드 추가
  - 다른 Stack 클래스 객체의 멤버 개수를 비교함

```
class Stack < T > {  
    // 생략...  
    public int size() {  
        return count;  
    }  
    public boolean compareSize(Stack<T> o) {  
        return size() > o.size() ? true : false;  
    }  
}
```

- compareSize() 메서드 사용

```
Stack<Integer> si1 = new Stack<Integer>(10);  
si1.push(100);  
si1.push(200);  
Stack<Integer> si2 = new Stack<Integer>(10);  
si2.push(1000);  
boolean b = si2.compareSize(si1); // b == false
```

# 와일드카드 인수 사용

---

- Stack 제네릭 클래스의 compareSize() 메서드
  - 문제점

```
Stack<Point> sp = new Stack<Point>(5);  
sp.push(new Point(100, 200));  
s1.compareSize(sp);           // 에러!! 다른 데이터 타입임
```

- 해결 방법 : 와일드카드 인수 사용

# 와일드카드 인수 사용

---

- 와일드카드 인수(wildcard argument)
  - 알지 못하는 타입(unknown type)이란 의미를 가짐

```
class Stack < T > {  
    // 생략...  
    public boolean compareSize(Stack<?> o) {  
        return size() > o.size() ? true : false;  
    }  
}
```

- 다른 타입을 비교하는 코드에 사용

```
boolean b = s1.compareSize(sp);        // OK!!
```

# 실습

---

- 스택 자료 구조를 표현하는 제네릭 클래스를 구현한다.
- 스택 제네릭 클래스를 사용하는 코드를 작성한다.
- 두개 숫자 중에서 큰 수를 반환하는 `max()` 제네릭 메서드를 구현한다.
- `max()` 제네릭 메서드를 사용하는 코드를 작성한다.

## 10. 예외 처리



# 10. 예외 처리

---

- 예외 처리
- 예외 처리 구문
- 예외 클래스
- 처리되지 않은 예외
- 예외 던지기
- 사용자 정의 예외 클래스
- 고급 예외 처리 기법

# 예외 처리

---

- 예외의 종류
  - 컴파일 에러(compilation error)
    - 소스 프로그램을 작성하고 컴파일러를 사용하여 컴파일 할 때 발생하는 에러
  - 실행 에러(runtime error)
    - 실행 시에 프로그램이 제대로 동작하지 않는 에러
  - 논리 에러(logic error)
    - 개발자가 원하는 대로 동작하지 않는 것

# 예외 처리 구문

---

- 예외 처리 구문

```
try {  
    // 보호 코드 블록  
    throw 예외타입인스턴스;  
}  
catch ( 예외타입 인수명 ) {  
    // 예외 처리 코드 블록  
}  
finally {  
    // 최종 실행 코드 블록  
}
```

# 예외 처리 구문

---

- 0으로 나누기 예외

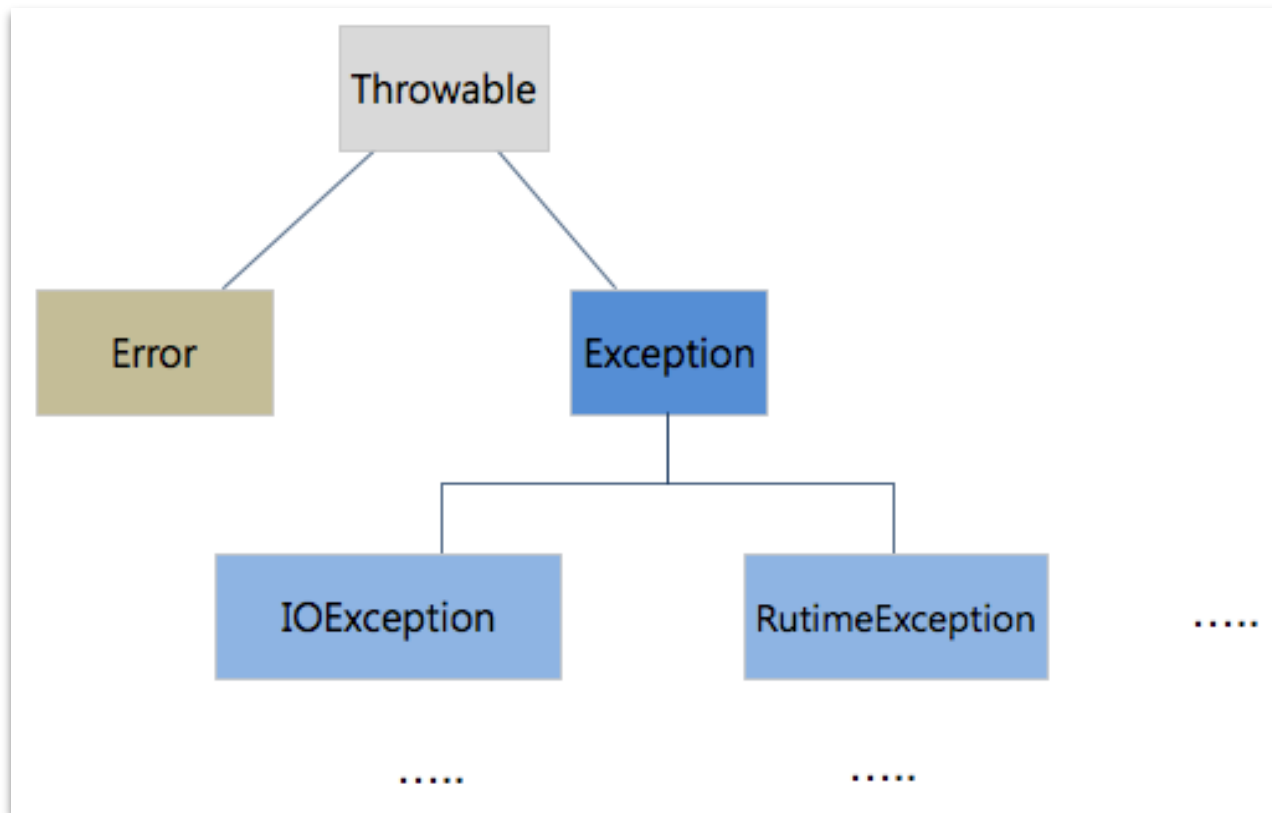
```
// 0 으로 나누기 예외
public static double divideByZero(int x, int y) {
    return x / y;
}

public static void main (String [] args) {
    int x = 50;
    int y = 0;
    double z = 0;
    try {
        z = divideByZero(x, y);
        System.out.printf("결과 : %f\n", z);
    } catch (ArithmeticException e) {
        System.out.println("0으로 나누기 예외가 발생했습니다.");
    }
    finally {
        System.out.println("처리가 완료되었습니다.");
    }
}
```

# 예외 클래스

---

- 예외 타입



# 예외 클래스

---

- Exception 서브 클래스

예외 타입	예외 발생 시점
NoSuchMethodException	메서드가 존재하지 않을 때
ClassNotFoundException	클래스가 존재하지 않을 때
CloneNotSupprotedException	객체의 복제가 지원되지 않는 상황에서 복제를 시도할 때
IllegalAccesException	클래스에 대해 잘못된 접근을 시도할 때
InstantiationException	추상 클래스나 인터페이스로부터 객체를 생성할 때
InterruptedException	스레드가 인터럽트 되었을 때
IOException	입출력 예외가 발생할 때
RuntimeException	실행 예외가 발생할 때

# 예외 클래스

---

- RuntimeException 서브 클래스

예외 타입	예외 발생 시점
ArithmeticException	0으로 나누기 등 산술적인 예외
NegativeArraysizeException	배열의 크기를 지정할 때 음수를 사용한 경우
NullPointerException	널null 객체의 메서드나 필드에 접근할 때
IndexOutOfBoundsException	배열이나 문자열의 범위를 벗어나서 접근할 때
SecurityException	보안 상 메서드를 수행할 수 없을 때

# 처리되지 않은 예외

---

- 디폴트 핸들러(default handler)
  - 프로그램에서 발생한 예외를 처리하지 않을 때
  - 예외를 표시하는 문자열을 표시하고, 예외가 발생한 위치에서부터 스택 추적(stack trace)를 출력한 다음 프로그램을 종료시킴

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:90)
```



# 처리되지 않은 예외

---

- throws 구
  - 메서드가 예외를 발생시킬 가능성이 있다면 해당 메서드를 호출하는 호출자가 예외에 대하여 스스로를 보호할 수 있도록 어떤 예외가 발생할 가능성이 있는 지를 알려주기 위해 사용
  - 해당 메서드의 throws 구에 발생할 가능성이 있는 모든 예외 목록을 지정해야 함
  - Error와 RuntimeException 클래스 및 서브 클래스 타입의 예외는 throws 구에 지정할 필요가 없음

```
public void readFile(String filename) throws IOException {  
    FileReader file = new FileReader(filename);  
    int r;  
    while( (r = file.read()) != -1)  
        System.out.print((char)r);  
    file.close();  
}
```

# 처리되지 않은 예외

---

- throws 구
  - throws 구가 지정된 메서드를 호출하는 메서드에서 throws 구에 지정된 예외를 처리하는 try...catch 코드 블록을 작성해야 함

```
try {  
    readFile("sample.txt");  
}  
catch(IOException e) {  
    System.out.println(e);  
}
```

# 예외 던지기

---

- 예외 던지기(throw exception)
  - throw 문 사용

```
throw 예외인스턴스;
```

- 예외 인스턴스(exception instance)는 반드시 Throwable 클래스 또는 서브 클래스 타입의 인스턴스이어야 함

```
throw new IllegalAccessException("잘못된 클래스 사용입니다");
```

# 예외 던지기

---

- 예외 던지기(throw exception)
  - catch 코드 블록에서 다시 예외 던지기

```
public static void foo() throws IllegalAccessException {  
    try {  
        throw new IllegalAccessException("잘못된 클래스 사용입니다");  
    }  
    catch(IllegalAccessException e) {  
        System.out.println("예외를 지역적으로 처리했습니다.");  
        throw e;          // 외부에서 다시 예외를 처리하도록 한다  
    }  
}
```

```
public static void main (String [] args) {  
    try {  
        foo();  
    } catch (IllegalAccessException e) {  
        System.out.println("다시 예외를 처리했습니다.");  
    }  
}
```

# 사용자 정의 예외 클래스

---

- 사용자 정의 예외 클래스(user-defined exception class)
  - Exception 클래스로부터 파생되는 서브 클래스

```
// 사용자 정의 예외 클래스
class UserException extends Exception {
    public UserException(int code) {
        this.code = code;
    }
    public String getMessage() {
        return String.format("예외코드 : %d, 메시지 : %s", code, message[code]);
    }
    private int code = 0;
    private static String [] message;
    static {
        message = new String[10];
        for(int i = 0; i < 10; i++)
            message[i] = "에러 메시지" + i;
    }
}
```

# 사용자 정의 예외 클래스

---

- 사용자 정의 예외 클래스 사용
  - 예외 던지기

```
try {  
    // 예외 발생 시  
    throw new UserException(1);    // 에러 코드 1 예외 발생  
    // 생략...  
}
```

- 예외 처리

```
catch (UserException e) {  
    System.out.println(e.getMessage());  
}
```

# 고급 예외 처리 기법

---

- 다중 예외 처리
  - 수퍼 타입 예외 보다는 서브 타입 예외에 대응하는 catch 코드 블록을 먼저 둔다

```
try {  
    // 파일을 찾을 수 없는 경우 FileNotFoundException 예외 발생  
    // 파일을 읽기를 실패한 경우 IOException 예외 발생  
    // 생략...  
}  
catch(FileNotFoundException e) {  
    System.out.println(e);  
}  
catch(IOException e) {  
    System.out.println(e);  
}
```

# 고급 예외 처리 기법

---

- 다중 예외 처리
  - Exception 타입의 catch 코드 블록은 가장 마지막에 둔다

```
try {  
    // 파일을 찾을 수 없는 경우 FileNotFoundException 예외 발생  
    // 파일을 읽기를 실패한 경우 IOException 예외 발생  
    // 생략...  
}  
catch(FileNotFoundException e) {  
    System.out.println(e);  
}  
catch(IOException e) {  
    System.out.println(e);  
}  
catch(Exception e) {  
    System.out.println(e);  
}
```



# 고급 예외 처리 기법

---

- 다중 캐치(multi catch)
  - 같은 catch 코드 블록이 2개 이상의 예외 타입을 잡아 처리할 수 있도록 하는 기능
  - | 연산자 사용

```
try {  
    // 파일을 찾을 수 없는 경우 FileNotFoundException 예외 발생  
    // 0으로 나눈 경우 ArithmeticException 예외 발생  
    // 생략...  
}  
catch(FileNotFoundException | ArithmeticException e) {  
    System.out.println(e);  
}
```

# 고급 예외 처리 기법

---

- try-with-resource
  - try 보호 코드 블록 안에서 자동적으로 사용한 리소스를 닫아주는 기능

```
// 자동 리소스 닫기
public void readFile(String filename) {
    try ( FileReader file = new FileReader(filename)) {
        int r;
        while( (r = file.read()) != -1)
            System.out.print((char)r);
    }
    catch(IOException e) {
        System.out.println(e);
    }
}
```

# 고급 예외 처리 기법

---

- 예외 처리 시 유의 사항
  - 프로그램 로직을 구현하는 수단으로 사용하지 말아야 한다

# 11. 내부 클래스와 람다 표현식

# 11. 내부 클래스와 람다 표현식

---

- 내부 클래스
- 람다 표현식

# 내부 클래스

---

- 내부 클래스(inner class)
  - 다른 클래스 안에 정의된 클래스
  - 이점
    - 내부 클래스 메서드는 내부 클래스가 정의된 범위의 어떤 데이터에도 접근할 수 있음
    - 내부 클래스는 같은 패키지의 다른 클래스에 감출 수 있음
    - 익명 내부 클래스(anonymous inner class)는 많은 코드를 작성하지 않고도 쉽게 콜백(callback)을 정의할 수 있게 함

# 내부 클래스

---

- 객체 상태 접근 사용
  - 클로저(closure)

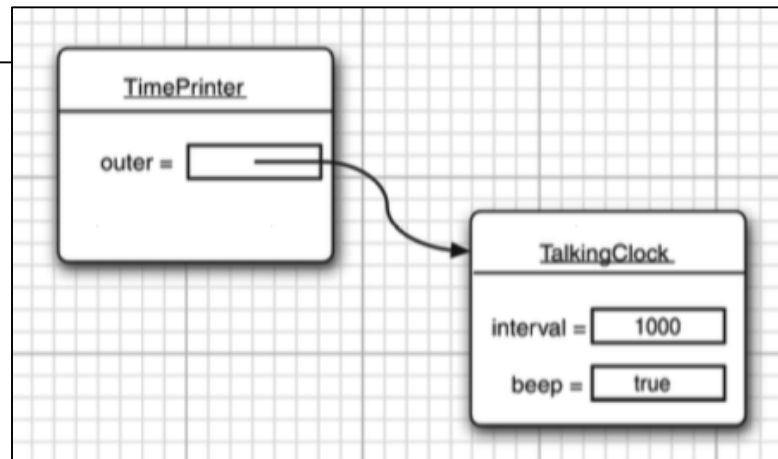
```
class TalkingClock {
    private int interval;
    private boolean beep;
    public TalkingClock(int interval, boolean beep) { .... }
    public void start() { ... }

    public class TimePrinter implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("At the tone, the time is " + new Date());
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }
}
```

# 내부 클래스

- 컴파일러 생성 코드

```
class TalkingClock {  
    public void start() {  
        ActionListener listener = new TimePrinter(this); // this 자동 추가  
    }  
    public class TimePrinter implements ActionListener {  
        public TimePrinter(TalkingClock clock) {  
            outer = clock;  
        }  
    }  
}
```





# 실습 : 내부 클래스

---

- 객체 상태 접근 예제

```
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Date;
import javax.swing.Timer;

public class TalkingClock {
    private int interval;
    private boolean beep;
    public TalkingClock(int interval, boolean beep) {
        this.interval = interval;
        this.beep = beep;
    }
    public void start() {
        ActionListener listener = new TimePrinter();
        Timer t = new Timer(interval, listener);
        t.start();
    }
    public class TimePrinter implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("At the tone, the time is " + new Date());
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }
}
```

# 실습 : 내부 클래스

---

- 객체 상태 접근 예제

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        TalkingClock clock = new TalkingClock(1000, true);
        clock.start();

        JOptionPane.showMessageDialog(null, "Quit program?");
        System.exit(0);
    }
}
```

# 내부 클래스

---

- 외부 메서드 변수 접근

```
class TalkingClock {  
    public void start(int interval, boolean beep) {  
        class TimePrinter implements ActionListener {  
            public void actionPerformed(ActionEvent event) {  
                System.out.println("At the tone, the time is " + new Date());  
                if(beep) Toolkit.getDefaultToolkit().beep();  
            }  
        }  
        ActionListener listener = new TimePrinter();  
        Time t = new Timer(interval, listener);  
        t.start();  
    }  
}
```

# 내부 클래스

---

- 익명 내부 클래스

```
new 수퍼타입(생성자 매개변수) {  
    // 내부 클래스 메서드와 데이터  
}
```

```
new 인터페이스타입() {  
    // 메서드와 데이터  
}
```

# 내부 클래스

---

- 익명 내부 클래스

```
class TalkingClock {  
    public void start(int interval, boolean beep) {  
        ActionListener listener = new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                System.out.println("At the tone, the time is " + new Date());  
                if(beep) Toolkit.getDefaultToolkit().beep();  
            }  
        }  
        Time t = new Timer(interval, listener);  
        t.start();  
    }  
}
```

# 실습 : 익명 내부 클래스

---

- 익명 내부 클래스 예제

```
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Date;
import javax.swing.Timer;

public class TalkingClock {
    public void start(int interval, boolean beep) {
        ActionListener listener = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.out.println("At the tone, the time is " + new Date());
                if (beep) Toolkit.getDefaultToolkit().beep();
            }
        };
        Timer t = new Timer(interval, listener);
        t.start();
    }
}
```

# 실습 : 익명 내부 클래스

---

- 익명 내부 클래스 예제

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        TalkingClock clock = new TalkingClock();
        clock.start(1000, true);
        JOptionPane.showMessageDialog(null, "Quit program?");
        System.exit(0);
    }
}
```

# 람다 표현식

---

- 람다(Lambda)
  - 매개변수로 전달되어 나중에 실행될 수 있는 코드 블록

```
public interface Comparator<T> {  
    int compare(T first, T second);  
}  
class LengthComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return first.length() - second.length();  
    }  
}  
Arrays.sort(strings, new LengthComparator()); // LengthComparator.compare() 호출됨
```



# 람다 표현식

- 람다 표현식(Lambda expression)

```
(String first, String second)           // 단일문으로 구현되는 경우  
-> first.length() - second.length()    // 구문으로부터 리턴 타입이 추론됨
```

```
(String first, String second) -> {      // 복합문으로 구현되는 경우  
    if(first.length() < second.length()) return -1;  
    else if(first.length() > second.length() ) return 1;  
    else return 0;  
}
```

```
() -> {                                // 매개변수가 없는 경우  
    for (int i = 100; i >= 0; i--)  
        System.out.println(i);  
}
```

```
Comparator<String> comp = (first, second)  
    // 추론할 수 있는 경우 (String first, String second)와 같음  
    -> first.length() - second.length();
```

# 람다 표현식

---

- 람다 표현식(Lambda expression)

```
public interface ActionListener {  
    void actionPerformed(ActionEvent event);  
}  
ActionListener listener = event ->          // (event) -> .. 또는 (ActionEvent event) -> 대신  
    System.out.println("The time is " + new Date());
```

# 실습 : 람다 표현식

---

- 람다 표현식 예제

```
import java.util.*;
import javax.swing.Timer;

public class Main {
    public static void main(String[] args)
    {
        String[] planets = new String[] { "Mercury", "Venus", "Earth", "Mars",
                                           "Jupiter", "Saturn", "Uranus", "Neptune" };
        System.out.println(Arrays.toString(planets));
        System.out.println("Sorted in dictionary order:");
        Arrays.sort(planets);
        System.out.println(Arrays.toString(planets));
        System.out.println("Sorted by length:");
        Arrays.sort(planets, (first, second) -> first.length() - second.length());
        System.out.println(Arrays.toString(planets));

        Timer t = new Timer(1000, event ->
            System.out.println("The time is " + new Date()));
        t.start();
        JOptionPane.showMessageDialog(null, "Quit program?");
        System.exit(0);
    }
}
```

## 12. 컬렉션 프레임워크

## 12. 컬렉션 프레임워크

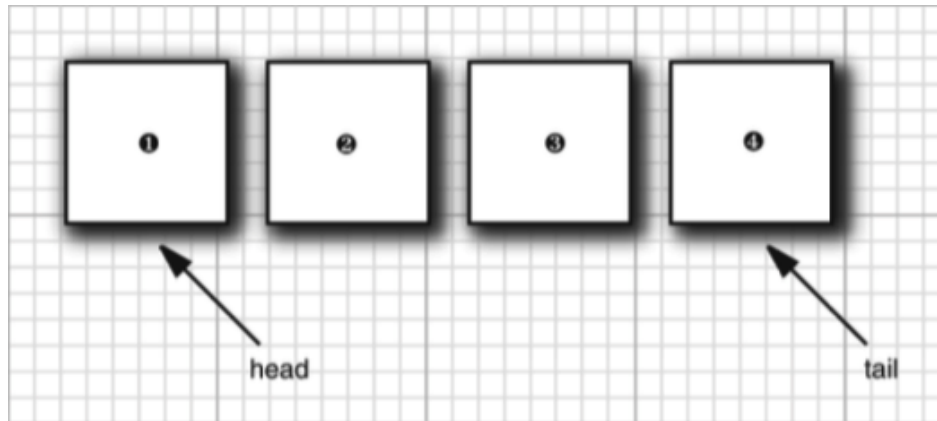
---

- 자바 컬렉션 프레임워크
- 구체적인 컬렉션
- 맵

# 자바 컬렉션 프레임워크

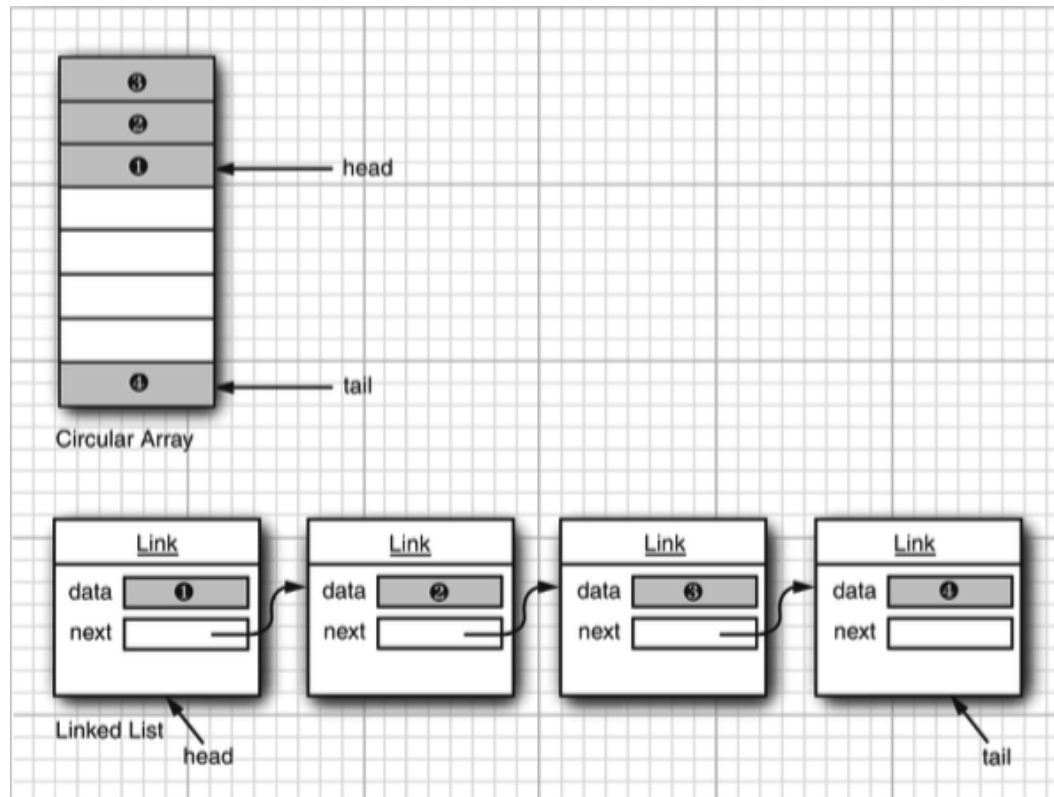
- 컬렉션 인터페이스와 구현 분리
  - 큐(queue) 인터페이스

```
public interface Queue<E> {  
    void add(E element);  
    E remove();  
    int size();  
}
```



# 자바 컬렉션 프레임워크

- 컬렉션 인터페이스와 구현 분리
  - 큐 구현



# 자바 컬렉션 프레임워크

---

- 컬렉션 인터페이스와 구현 분리
  - 순환 배열(circular array)

```
public class CircularArrayQueue<E> implements Queue<E> {  
    private int head;  
    private int tail;  
    CircularArrayQueue(int capacity) { ... }  
    public void add(E element) { ... }  
    public E remove() { ... }  
    public int size() { .... }  
    private E[] elements;  
}
```

- 연결 리스트(linked list)

```
public class LinkedListQueue<E> implements Queue<E> {  
    private Link head;  
    private Link tail;  
    LinkedListQueue() { ... }  
    public void add(E element) { ... }  
    public E remove() { ... }  
    public int size() { .... }  
}
```



# 자바 컬렉션 프레임워크

---

- 컬렉션 인터페이스와 구현 분리
  - 컬렉션 객체 생성

```
Queue<Customer> expressLane = new CircularArrayQueue(100);  
또는,  
Queue<Customer> expressLane = new LinkedListQueue();
```

- 컬렉션 객체 사용

```
expressLane.add(new Customer("전병선"));
```

# 자바 컬렉션 프레임워크

---

- Collection 인터페이스

```
public interface Collection<E> {  
    boolean add(E element);  
    Iterator<E> iterator();  
    ...  
}
```

- 사용 예:

```
Collection<String> c = ...;  
for(String element : c) {  
    // 요소를 사용함  
}
```

# 자바 컬렉션 프레임워크

---

- 열거자(iterator)

```
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
    void remove();  
    default void forEachRemaining(Consume<? super E> action);  
}
```

- 사용 예:

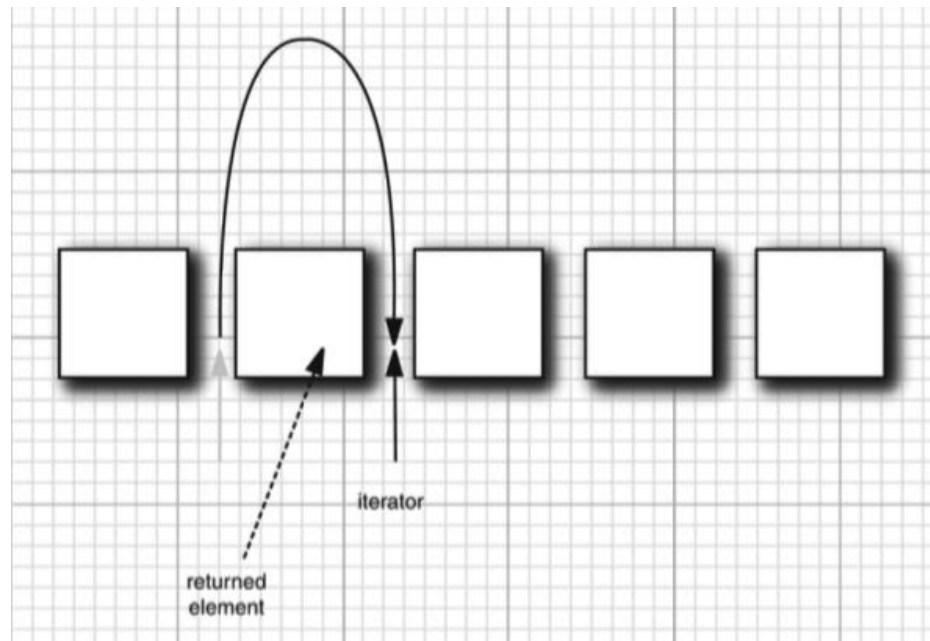
```
Collection<String> c = ...;  
Iterator<String> iter = c.iterator();  
while(iter.hasNext()) {  
    String element = iter.next();  
    // 요소를 처리함  
} // 먼저 요소가 있는 지를 체크해야 함  
// 요소가 없으면 NoSuchElementException 예외 발생
```

```
Collection<String> c = ...;  
Iterator<String> iter = c.iterator();  
iter.forEachRemaining(element -> // 요소를 처리함
```

# 자바 컬렉션 프레임워크

- 열거자(iterator)
  - 열거 항목 삭제

```
Iterator<String> iter = c.iterator();  
iter.next();      // 첫번째 항목 건너뛰고  
iter.remove();    // 삭제함
```



# 자바 컬렉션 프레임워크

---

- 열거자(iterator)
  - 열거 항목 삭제

```
Iterator<String> iter = c.iterator();  
iter.next();      // 첫번째 항목 건너뛰고  
iter.remove();    // 삭제함  
iter.remove();    // 에러!!
```

```
Iterator<String> iter = c.iterator();  
iter.next();      // 첫번째 항목 건너뛰고  
iter.remove();    // 삭제함  
iter.next();  
iter.remove();    // OK!!
```

# 자바 컬렉션 프레임워크

---

- 제너릭 유틸리티 메서드(generic utility method)

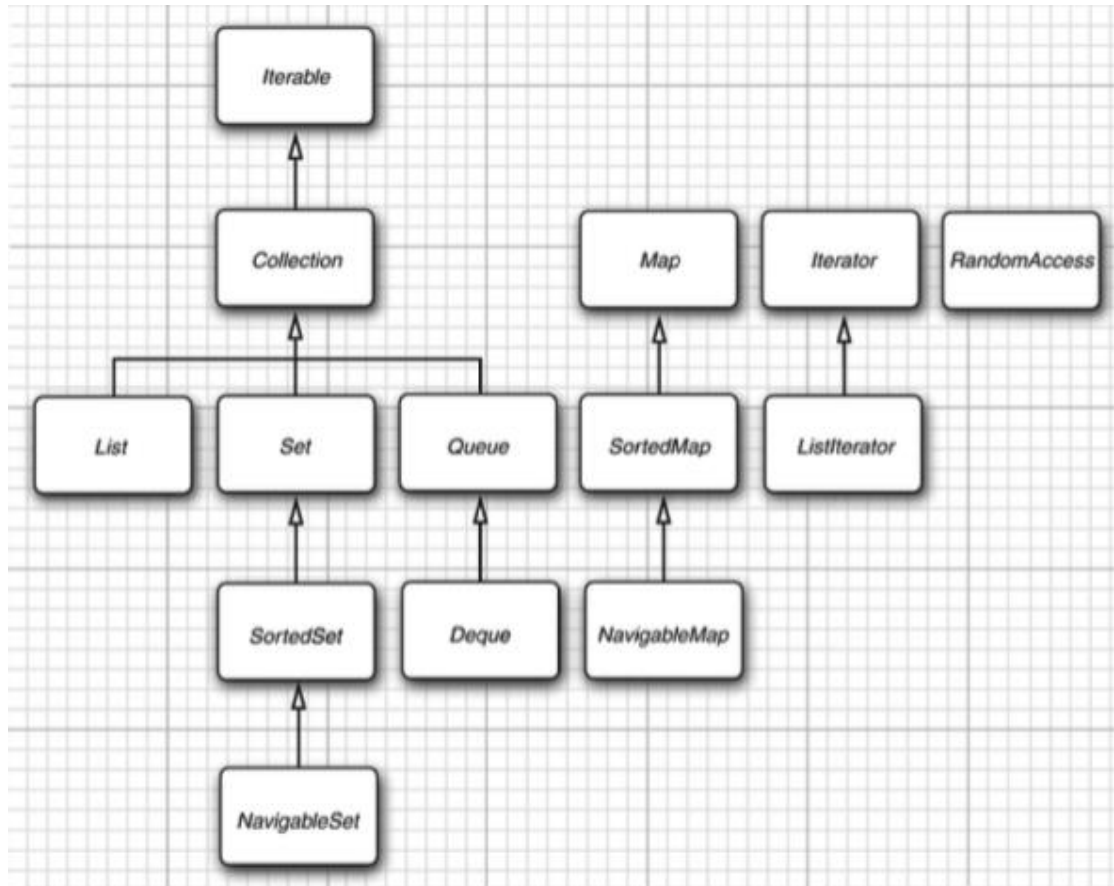
```
public static <E> boolean contains(Collection<E> c, Object obj) {  
    for(E element : c)  
        if(element.equals(obj))  
            return true;  
    return false;  
}
```

- Collection 인터페이스

```
int size()  
boolean isEmpty()  
boolean contains(Object obj)  
boolean containsAll(Collection<?> c)  
boolean equals(Object other)  
boolean addAll(Collection<? extends E> from)  
boolean remove(Object obj)  
boolean removeAll(Collection<?> c)  
void clear()  
boolean retainAll(Collection<?> c)  
Object[] toArray()  
<T> T[] toArray(T[] arrayToFill)
```

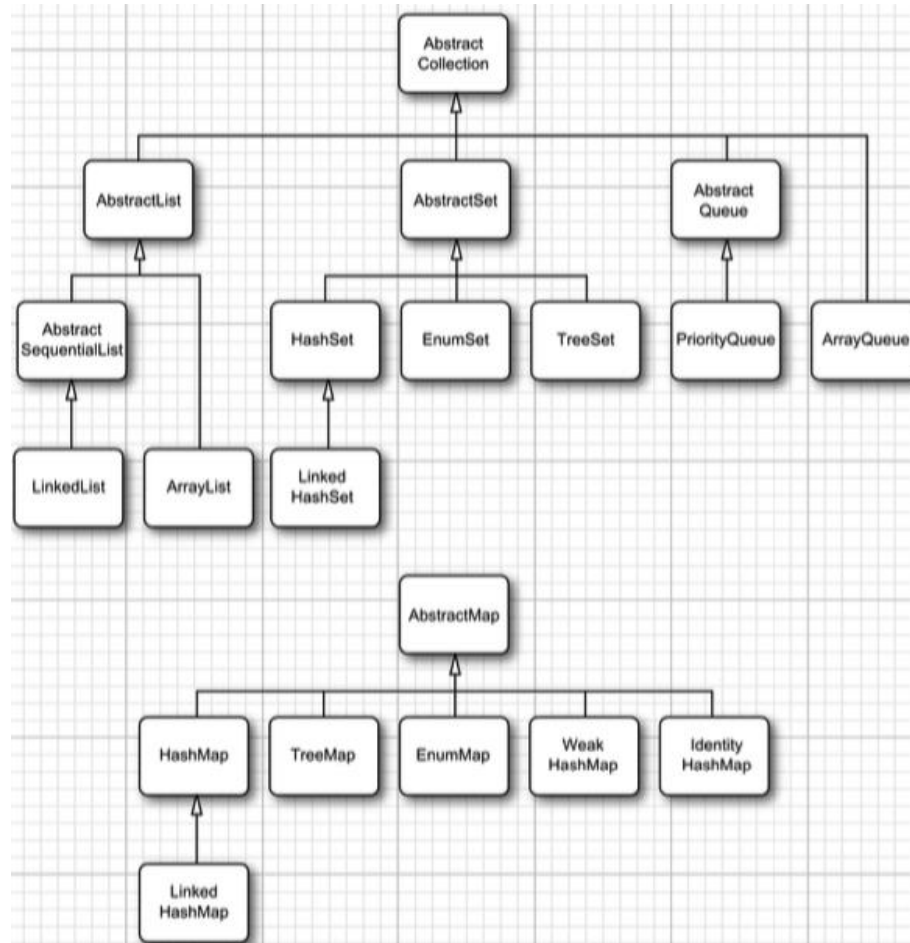
# 자바 컬렉션 프레임워크

- 컬렉션 프레임워크 인터페이스



# 구체적 컬렉션

- 구체적 컬렉션(concrete collection)





# 구체적 컬렉션

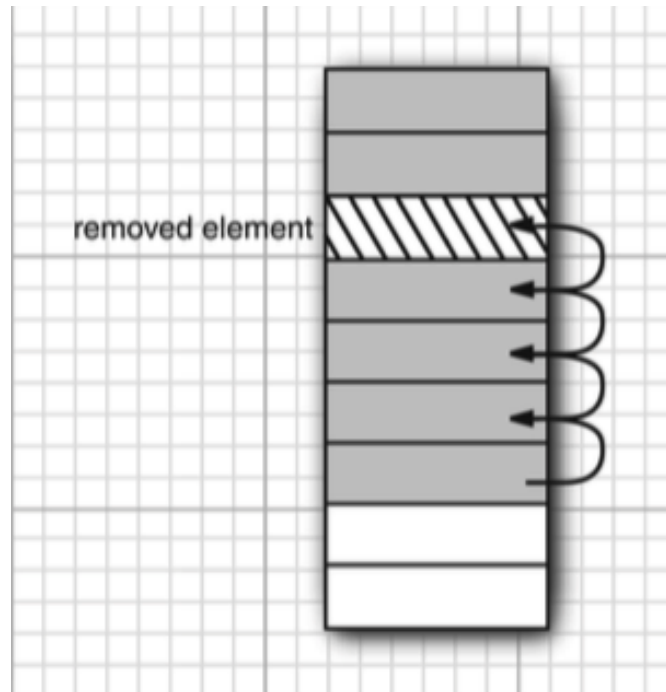
- 구체적 컬렉션(concrete collection)

컬렉션 타입	설명
ArrayList	동적으로 늘어나고 줄어드는 인덱스를 갖는 시퀀스
LinkedList	어느 위치에든 효율적으로 삽입하고 삭제할 수 있는 순서화된 시퀀스
ArrayDeque	순환 배열로서 구현되는 양끝을 갖는 큐
HashSet	중복되지 않는 비순서화된 컬렉션
TreeSet	정렬된 세트
EnumSet	열거되는 타입 값의 세트
LinkedHashSet	요소가 삽입된 순서를 기억하는 세트
PriorityQueue	가장 작은 요소를 효과적으로 제거할 수 있는 컬렉션
HashMap	키/값 연관을 저장하는 데이터 구조
TreeMap	키가 정렬되는 맵
EnumMap	키가 열거되는 타입에 속하는 맵
LinkedHashMap	추가되는 항목들의 순서를 기억하는 맵
WeakHashMap	다른 곳에서 사용되지 않으면 가비지 컬렉터에 의해 수집되는 값을 갖는 맵
IdentityHashMap	equals가 아니라 == 로 비교되는 키를 갖는 맵

# 구체적 컬렉션

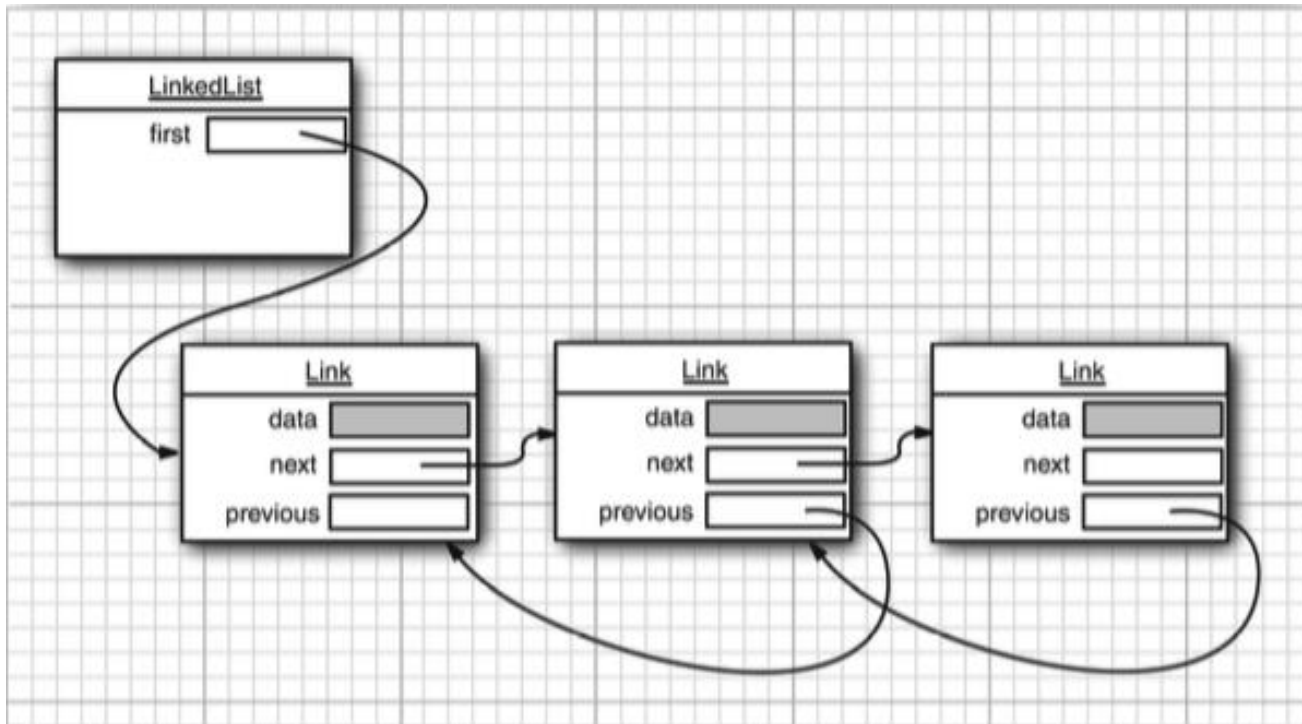
---

- 연결 리스트(Linked List)
  - 배열과 ArrayList의 문제점 : 배열 가운데 요소 삭제



# 구체적 컬렉션

- 연결 리스트(Linked List)
  - 이중 연결 리스트(doubly linked list)



# 구체적 컬렉션

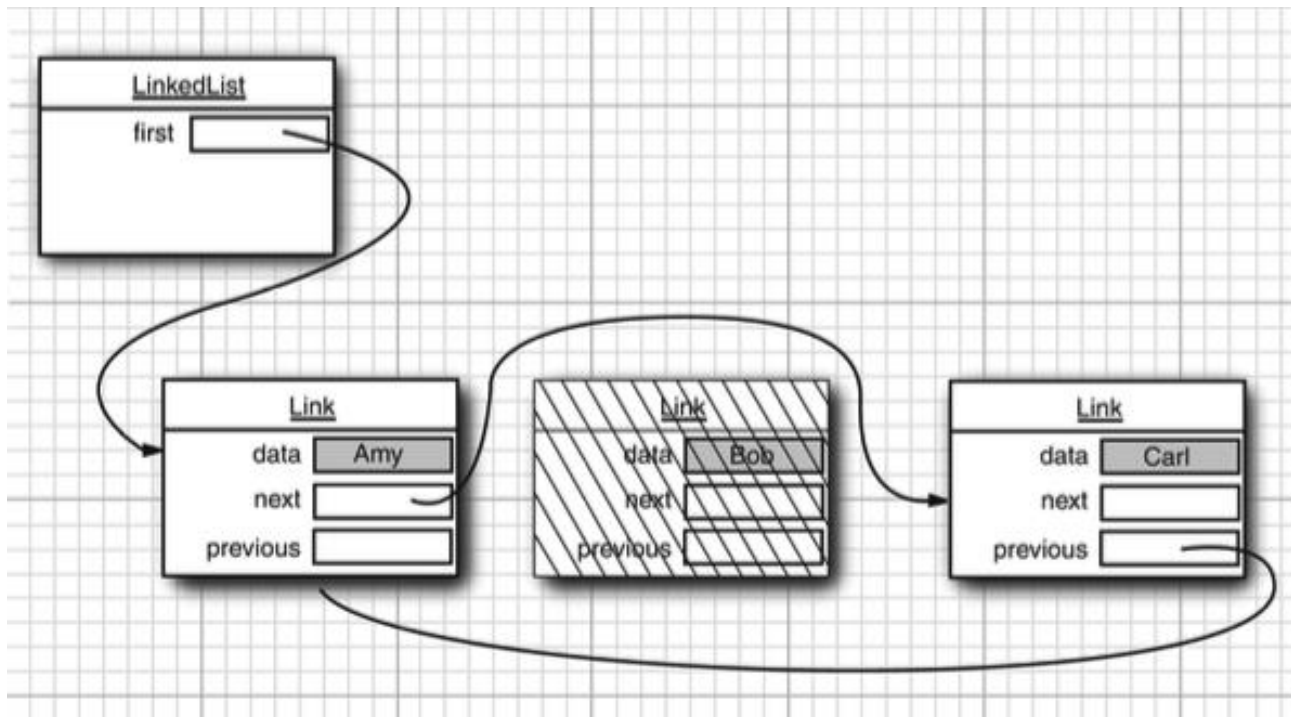
---

- 연결 리스트(Linked List)
  - 요소 삭제 : `Iterator.remove()`

```
List<String> staff = new LinkedList<>();  
staff.add("Amy");  
staff.add("Bob");  
staff.add("Carl");  
Iterator iter = staff.iterator();  
iter.next();      // 첫번째 요소로 이동  
iter.next();      // 두번째 요소로 이동  
iter.remove();    // 마지막으로 이동한 요소 삭제
```

# 구체적 컬렉션

- 연결 리스트(Linked List)
  - 요소 삭제



# 구체적 컬렉션

---

- 연결 리스트(Linked List)
  - ListIterator 열거자

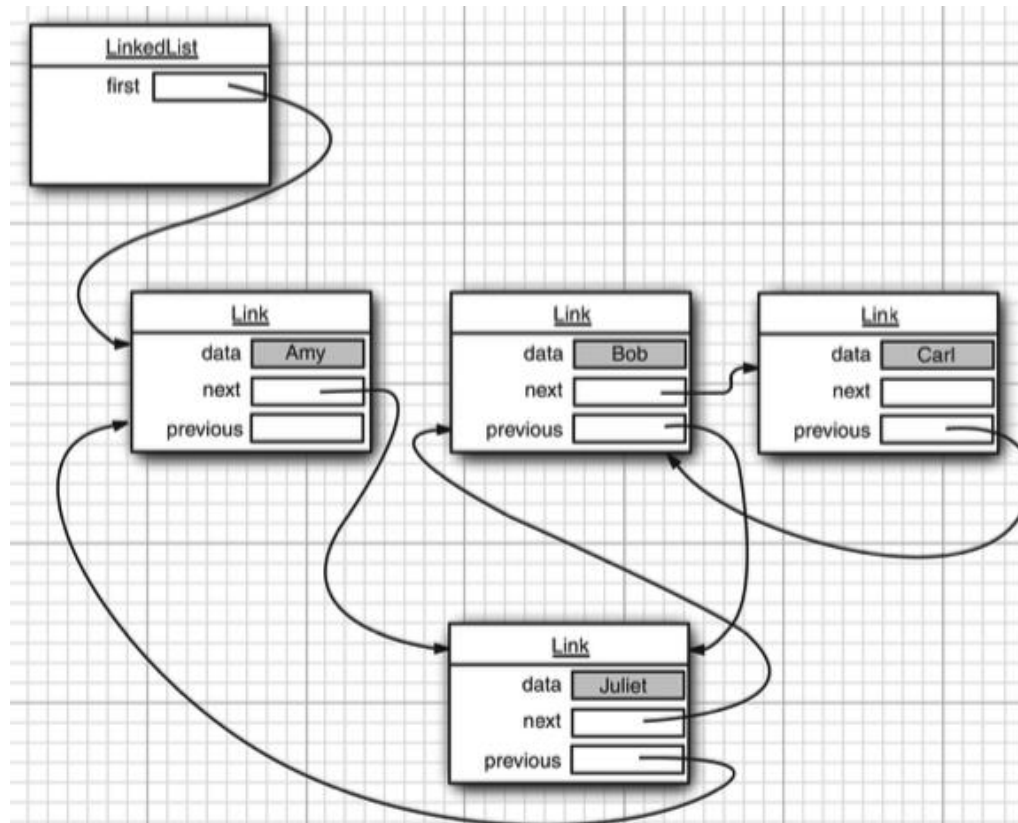
```
interface ListIterator<E> extends Iterator<E> {  
    void add(E element);  
    E previous();  
    boolean hasPrevious();  
    void set(E element);  
}
```

- 요소 추가 : ListIterator.add()

```
List<String> staff = new LinkedList<>();  
staff.add("Amy");  
staff.add("Bob");  
staff.add("Carl");  
ListIterator<String> iter = staff.listIterator();  
iter.next();          // 첫번째 요소로 이동  
iter.add("Juliet");
```

# 구체적 컬렉션

- 연결 리스트(Linked List)
  - 요소 추가



# 구체적 컬렉션

---

- 연결 리스트(Linked List)
  - 요소 대체 : ListIterator.set()

```
List<String> staff = new LinkedList<>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
String oldValue = iter.next();           // 첫번째 요소로 반환
iter.set(newValue);                     // 첫번째 요소에 newValue 저장
```



# 구체적 컬렉션

---

- 연결 리스트(Linked List)
  - 여러 열거자 사용

```
List<String> list = ...;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next();           // ConcurrentModificationException 예외 발생!
```

- 여러 열거자는 읽기 전용으로만 사용함
- 변경이 필요할 때는 하나의 열거자만 사용함

# 구체적 컬렉션

---

- 연결 리스트(Linked List)
  - 랜덤 액세스 : `LinkedList.get()`

```
LinkedList<String> list = ...;  
String obj = list.get(n);
```

- 효율적이지 않음

```
for(int i = 0; i < list.size(); i++)    // 극도로 비효율적임!!!!  
    // 항목에 대해 작업을 수행함
```

# 실습 : LinkedList 사용

---

- 연결 리스트(Linked List) 사용

```
public class Main {  
    public static void main(String[] args) {  
        List<String> a = new LinkedList<>();  
        a.add("Amy");  
        a.add("Carl");  
        a.add("Eriaca");  
        List<String> b = new LinkedList<>();  
        b.add("Bob");  
        b.add("Doug");  
        b.add("Frances");  
        b.add("Gloria");  
  
        // b를 a로 병합  
        ListIterator<String> aiter = a.listIterator();  
        Iterator<String> biter = b.iterator();  
        while(biter.hasNext()) {  
            if(aiter.hasNext()) aiter.next();  
            aiter.add(biter.next());  
        }  
        System.out.println(a);  
    }  
}
```

# 실습 : LinkedList 사용

---

- 연결 리스트(Linked List) 사용

```
// b에서 모든 두번째 단어 삭제
biter = b.iterator();
while(biter.hasNext()) {
    biter.next();
    if(biter.hasNext()) {
        biter.next();
        biter.remove();
    }
}
System.out.println(b);

// a에서 b에 있는 모든 단어 삭제
a.removeAll(b);
System.out.println(a);
}
```

# 구체적 컬렉션

---

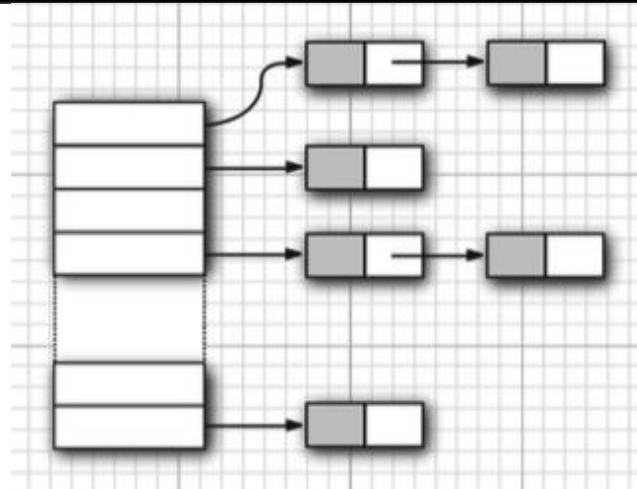
- 배열 리스트(Array List)
  - 동적으로 객체 배열을 재할당하는 기능을 구현함

```
List<String> list = new ArrayList<>();
```

# 구체적 컬렉션

- 해시 세트(Hash Sets)
  - 해시 테이블(hash table)
    - 객체를 빨리 찾을 수 있도록 하는 데이터 구조
    - 해시 코드(hash code)

String	Hash Code
"Lee"	76268
"lee"	107020
"ee1"	100300



# 실습 : HashSet 사용

- 해시 세트(Hash Sets) 사용

```
public class Main {
    public static void main(String[] args) {
        Set<String> words = new HashSet<>();
        long totalTime = 0;

        try (Scanner in = new Scanner(System.in)) {
            while(in.hasNext()) {
                String word = in.next();
                long callTime = System.currentTimeMillis();
                words.add(word);
                callTime = System.currentTimeMillis() - callTime;
                totalTime += callTime;
            }
        }
        Iterator<String> iter = words.iterator();
        for(int i = 1; i <= 20 && iter.hasNext(); i++)
            System.out.println(iter.next());
        System.out.println("...");
        System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
    }
}
```

# 맵

---

- 맵(map)
  - 키/값 쌍으로 저장하고 키로 값을 찾음
  - Map 인터페이스

```
public interface Map<K, V> {  
    V get(Object key);  
    default V getOrDefault(Object key, V defaultValue);  
    V put(K key, V value);  
    void putAll(Map<? extends K, ? extends V> entries);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    default void forEach(BiConsumer<? super K, ? super V> action);  
}
```

- SortedMap 인터페이스

```
public interface SortedMap<K, V> extends Map<K, V> {  
    Comparator<? super K> comparator();  
    K firstKey();  
    K lastKey();  
}
```



# 맵

---

- 맵 구현 클래스
  - HashMap 클래스

```
public class HashMap<K, V> implements Map<K, V> {  
    public HashMap() { ... }  
    public HashMap(int initialCapacity) { ... }  
    public HashMap(int initialCapacity, float loadFactor) { ... }  
}
```

- TreeMap 클래스

```
public class TreeMap<K, V> implements SortedMap<K, V> {  
    public TreeMap() { ... }  
    public TreeMap(Comparator<? super K> c) { ... }  
    public TreeMap(Map<? extends K, ? extends V> entries) { ... }  
    public TreeMap(SortedMap<? extends K, ? extends V> entries) { ... }  
}
```

# 맵

---

- 맵 항목 갱신

```
counts.put(word, counts.get(word) + 1); // NullPointerException 예외 발생!!
```

- getOrDefault() 메서드 사용

```
counts.put(word, counts.getOrDefault(word) + 1);
```

- putIfAbsent() 메서드 사용

```
counts.putIfAbsent(word, 0);  
counts.put(word, counts.get(word) + 1);
```

- merge() 메서드 사용

```
counts.merge(word, 1, Integer::sum);
```

# 실습 : HashMap 사용

---

- HashMap 사용

```
public class Employee {  
    private String name;  
    private double salary;  
  
    public Employee(String name) {  
        this.name = name;  
        salary = 0;  
    }  
  
    public String toString() {  
        return "[name=" + name + ", salary=" + salary + "];"  
    }  
}
```

# 실습 : HashMap 사용

---

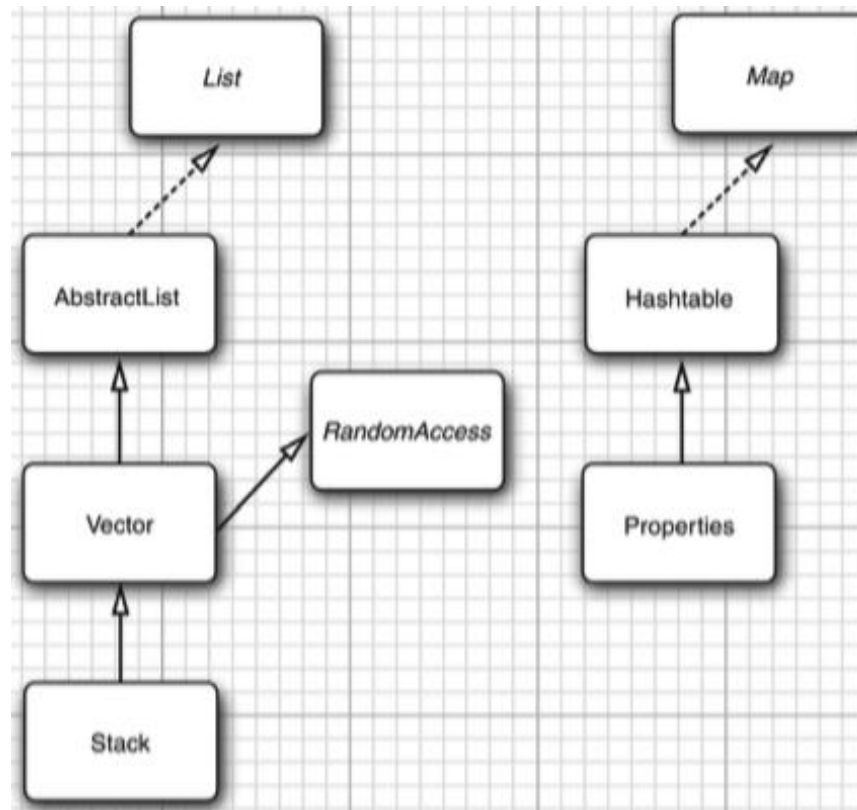
- HashMap 사용

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Map<String, Employee> staff = new HashMap<>();
        staff.put("144-25-5464", new Employee("Amy Lee"));
        staff.put("567-24-2546", new Employee("Harry Hacker"));
        staff.put("157-62-7935", new Employee("Gary Cooper"));
        staff.put("456-62-5527", new Employee("Francesca Cruz"));
        System.out.println(staff);
        staff.remove("567-24-2546");
        staff.put("456-62-5527", new Employee("Francesca Miller"));
        System.out.println(staff.get("157-62-7935"));
        staff.forEach((k, v) ->
            System.out.println("key=" + k + ", value=" + v));
    }
}
```

# 레거시 컬렉션

- 레거시 컬렉션(legacy collection)
  - 컬렉션 프레임워크 이전부터 있었던 컨테이너 클래스



# 레거시 컬렉션

---

- Hashtable 클래스
  - HashMap 클래스와 동일
- Enumeration 인터페이스
  - Iterator 인터페이스와 동일

```
Enumeration<Employee> e = staff.elements();
while(e.hasMoreElements()) {
    Employee e = e.nextElement();
    // ...
}
```

- Vector 클래스
  - ArrayList 클래스가 더 효율적임

# 레거시 컬렉션

---

- 속성 맵(property map)
  - 키와 값이 문자열임
  - 테이블이 파일에 저장되고 로드될 수 있음
  - 설정 정보 저장에 주로 사용됨

```
public class Properties {  
    public Properties() { ... }  
    public Properties(Properties defaults) { ... }  
    String getProperty(String key) { ... }  
    String getProperty(String key, String defaultValue) { .... }  
    void load(InputStream in) { ... }  
    void store(OutputStream out, String commentString) { ... }  
}
```

# The END

---

- 감사합니다!