Java

API 도큐먼트

- 자바 표준 모듈에서 제공하는 방대한 라이브러리들을 쉽게 찾아서 사용할 수 있도록 API 도큐먼트가 있다.

- 클래스와 인터페이스의 집합을 라이브러리라고 한다면,
  이러한 라이브러리들의 사용 방법을 기술한 것을 API 도큐먼트라 한다.

https://docs.oracle.com/en/java/javase/index.html

API 도큐먼트

- String 도큐먼트를 확인해보자 [java.base > java.lang > String]

Module java.base
Package java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

- 선언부를 통해 String 클래스가 어떻게 정의되었는지 볼 수 있고, 상속 계층도를 통해 전체 상속 관계를 한 눈에 파악할 수 있다.

API 도큐먼트

- 상단의 SUMMARY를 이용하면, 해당 클래스가 가지고 있는 멤버를 볼 수 있다.

  SUMMARY: NESTED | FIELD | CONSTR | METHOD

  - NESTED : 중첩 클래스/중첩 인터페이스 목록으로 이동
  - FIELD : 필드 목록으로 이동
  - CONSTR : 생성자 목록으로 이동
  - METHOD : 메소드 목록으로 이동

- 만약 링크가 없는 경우에는 공개된(public, protected) 멤버가 없다는 뜻이다.

java.base 모듈

- java.base는 모든 모듈이 의존하는 기본 모듈로, 모듈 중 유일하게 requires하지 않아도 사용 가능하다.
- java.base 모듈에 포함된 주요 패키지와 용도는 아래와 같다.

패키지	용도
java.lang	자바 언어의 기본 클래스를 제공
java.util	자료 구조와 관련된 컬렉션 클래스를 제공
java.text	날짜 및 숫자를 원하는 형태의 문자열로 만들어 주는 포맷 클래스 제공
java.time	날짜 및 시간을 조작하거나 연산하는 클래스를 제공
java.io	입출력 스트림 클래스를 제공
java.net	네트워크 통신과 관련된 클래스를 제공
java.nio	데이터 저장을 위한 Buffer 및 새로운 입출력 클래스 제공

- String, System, Integer, Double, Exception, RuntimeException 등의 클래스는 모두 java.lang 패키지에 있고, Scanner는 java.util 패키지에 있다.
- java.lang 패키지는 import 없이 사용 가능하다.

java.lang 패키지

클래스	용도
Object	• 자바 클래스의 최상위 클래스
System	• 키보드를 통한 데이터 입력 • 콘솔창에 데이터 출력 • 프로세스 종료 • 진행시간 읽기 • 시스템 속성 읽기
String	• 문자열 저장 및 조작
StringBuilder	• 효율적인 문자열 조작
Byte, Short, Character, Integer, Float, Double, Boolean	• 기본 타입의 값을 포장 • 문자열을 기본 타입으로 변환
Math	• 수학 계산
Class	• 클래스의 메타 정보(이름, 구성 멤버) 등 확인

Object 클래스

- 클래스를 선언할 때 extends 키워드로 다른 클래스를 상속하지 않으면 암시적으로 Object 클래스를 상속하게 된다.

- 즉, 자바의 모든 클래스는 Object의 자식이거나 자손 클래스가 된다.
- 때문에 Object가 가진 메소드는 모든 객체에서 사용 가능하다.

  - boolean equals(Object obj) : 객체의 번지를 비교하고 결과를 반환
  - int hashCode() : 객체의 해시코드를 반환
  - String toString() : 객체의 문자 정보를 반환

Object 클래스 – equals

- equals() 메소드는 일반적으로 재정의해서 동등 비교용으로 사용한다.

```java
package com.object;

public class Member {
    public String id;
    public Member(String id) {
        this.id = id;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Member) {
            Member target = (Member) obj;
            if(id.equals(target.id)) {
                return true;
            }
        }
        return false;
    }
}
```

```java
package com.object;

public class MemberExample {
    public static void main(String[] args) {
        Member obj1 = new Member("blue");
        Member obj2 = new Member("blue");
        Member obj3 = new Member("red");
        if(obj1.equals(obj2)) {
            System.out.println("obj1과 obj2는 같다.");
        } else {
            System.out.println("obj1과 obj2는 다르다.");
        }
        if(obj1.equals(obj3)) {
            System.out.println("obj1과 obj3는 같다.");
        } else {
            System.out.println("obj1과 obj3는 다르다.");
        }
    }
}
```

Object 클래스 – hashcode

- 객체 해시코드란 객체를 식별하는 정수를 말한다.

```
public class Student {
	private int no;
	private String name;

	public Student(int no, String name) {
		this.no = no;
		this.name = name;
	}

	public int getNo() { return no; }
	public String getName() { return name; }

	@Override
	public int hashCode() {
		int hashCode = no + name.hashCode();
		return hashCode;
	}

	@Override
	public boolean equals(Object obj) {
		if (obj instanceof Student) {
			Student target = (Student) obj;
			if (no == target.getNo() && name.equals(target.getName())) return true;
		}
		return false;
	}
}
```

Object 클래스 – hashcode

- 객체의 메모리 주소를 이용해서 해시코드를 생성하기 때문에 객체마다 다른 정수값을 반환한다.

```
package com.object;

public class StudentExample {
    public static void main(String[] args) {
        Student s1 = new Student(1, "홍길동");
        Student s2 = new Student(1, "홍길동");

        System.out.println(s1.hashCode());
        System.out.println(s2.hashCode());
        System.out.println(s1.equals(s2));
    }
}
```

Object 클래스 – toString

- toString() 메소드는 객체의 문자 정보를 반환한다.
- 객체의 문자 정보란 객체를 문자열로 표현한 값을 말하며, 기본적으로 "클래스명@16진수해시코드"로 구성되어 있다.

- 객체의 문자 정보가 중요한 경우에는 toString() 메소드를 재정의해서 간결하고 유용한 정보를 반환하도록 해야 한다.
- Date 클래스는 현재 날짜와 시간을 반환하고, String 클래스는 저장된 문자열을 반환하도록 오버라이딩하고 있다.

Object 클래스 – toString

```
public class SmartPhone {
    private String model;
    private String os;

    public SmartPhone(String model, String os) {
        this.model = model;
        this.os = os;
    }

    @Override
    public String toString() {
        return model + ", " + os;
    }
}

public class SmartPhoneExample {
    public static void main(String[] args) {
        SmartPhone myPhone = new SmartPhone("아이폰", "ios");
        System.out.println(myPhone.toString());
    }
}
```

롬복(Lombok) 사용하기

- 롬복은 JDK에 포함된 표준 라이브러리는 아니지만 개발자들이 즐겨 쓰는 자동 코드 생성 라이브러리이다.
- DTO 클래스를 작성할 때 Getter, Setter, hashCode, equals, toString 메소드를 자동 생성하여 작성할 코드의 양을 줄여준다.

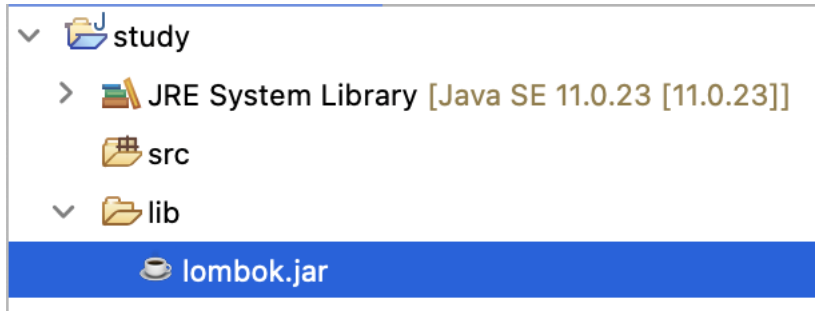
TIP

DTO (Data Transfer Object, 데이터 전송 객체) :
데이터를 전송하기 위해 사용하는 객체로
주로 클라이언트와 서버가 데이터를 주고받을 때 사용한다.

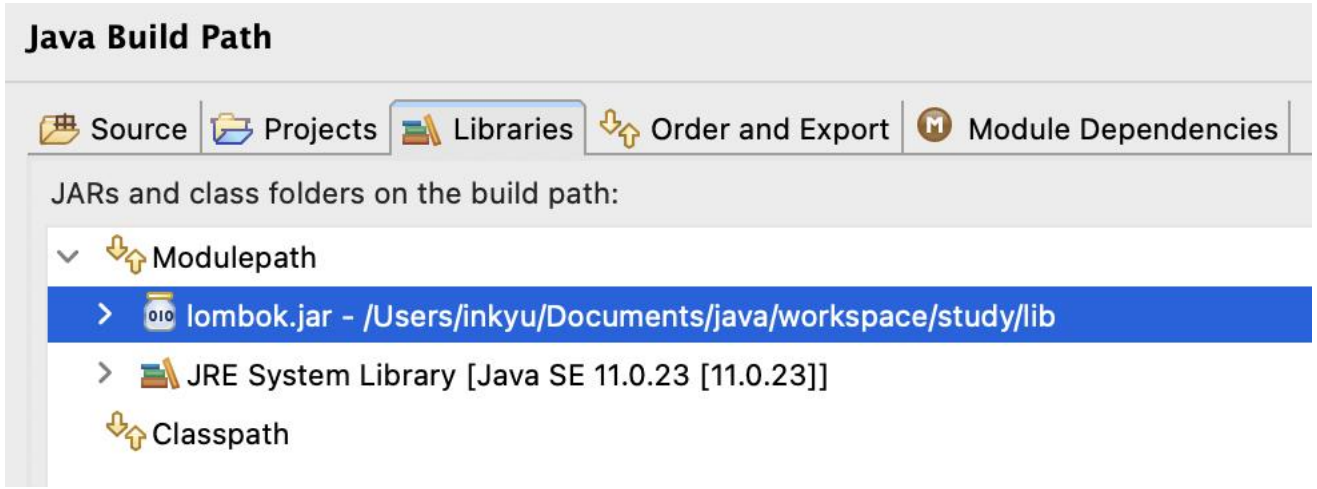
- 이클립스에서 롬복을 사용하기 위해서는 설치 과정이 필요하다. https://projectlombok.org/download
- 다운로드 받은 lombok.jar 파일이 있는 곳으로 이동해서 아래 명령어를 실행하면 설치가 진행된다.
  - java -jar lombok.jar
- 설치 완료 후 이클립스를 재시작해야 한다.

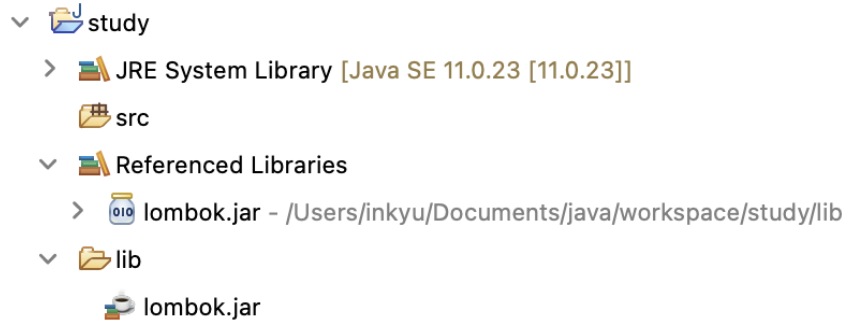
롬복(Lombok) 사용하기

- 이클립스 재시작 후, 프로젝트 안에 lib 폴더를 생성하고, 다운로드한 lombok.jar 파일을 lib 폴더 내부에 복사한다.



- 마지막으로 Build Path 에서 아래와 같이 추가해준다.





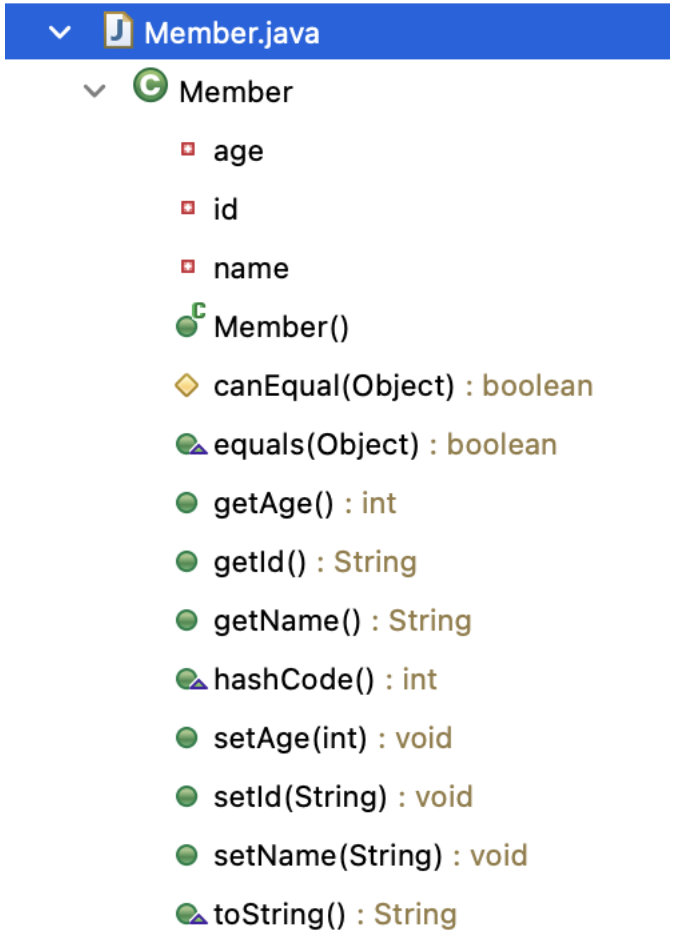
롬복(Lombok) 사용하기

- @Data가 붙게 되면 컴파일 과정에서 기본 생성자와 함께 Getter, Setter, hashCode(), equals(), toString() 메소드가 자동 생성된다.

```java
package com.lombok;

import lombok.Data;

@Data
public class Member {
    private String id, name;
    private int age;
}
```



롬복(Lombok) 사용하기

- @Data 외에도 아래와 같은 어노테이션을 사용할 수 있다.

어노테이션	설명
@NoArgsConstructor	매개변수가 없는 기본 생성자 포함
@AllArgsConstructor	모든 필드를 초기화시키는 생성자 포함
@RequiredArgsConstructor	매개변수가 없는 기본 생성자를 포함 (만약 final 또는 @NonNull이 붙은 필드가 있다면 해당 필드만 초기화시키는 생성자 포함)
@Getter	Getter 메소드 포함
@Setter	Setter 메소드 포함
@EqualsAndHashCode	equals와 hashCode 메소드 포함
@ToString	toString 메소드 포함

TIP

@Data는 @RequiredArgsConstructor, @Getter, @Setter, @EqualsAndHashCode, @ToString가 합쳐진 것과 동일한 효과를 갖는다.

System 클래스

- Java는 운영체제 상에서 바로 실행되지 않고, JVM이라는 가상 머신 상에서 실행되기 때문에 운영체제의 모든 기능을 Java에서 직접 접근하는 것은 어렵다.

- 하지만 java.lang 패키지에 속하는 System 클래스를 이용하면 운영체제의 일부 기능을 이용할 수 있게 된다.
- System 클래스의 정적 필드와 메소드를 이용하면 프로그램 종료, 키보드 입력, 콘솔 출력, 현재 시간 읽기, 시스템 프로퍼티 읽기가 가능하다.

정적 멤버		용도
필드	out	콘솔(모니터)에 문자 출력
	err	콘솔(모니터)에 에러 내용 출력
	in	키보드 입력
메소드	exit(int status)	프로세스 종료
	currentTimeMillis()	현재 시간을 밀리초 단위의 long 값으로 반환
	nanoTime()	현재 시간을 나노초 단위의 long 값으로 반환
	getProperty()	운영체제와 사용자 정보 제공
	getenv()	운영체제의 환경 변수 정보 제공

System 클래스 – out, in

```
public class InExample {
    public static void main(String[] args) throws Exception {
        int speed = 0;
        int keyCode = 0;
        while(true) {
            if (keyCode != 13 && keyCode != 10) {
                if (keyCode == 49) {
                    speed++;
                } else if (keyCode == 50) {
                    speed--;
                } else if (keyCode == 51) {
                    break;
                }
                System.out.println("------------------------");
                System.out.println("1. 증속 | 2. 감속 | 3. 중지");
                System.out.println("------------------------");
                System.out.println("현재 속도 = " + speed);
                System.out.print("선택 : ");
            }
            keyCode = System.in.read();
        }
        System.out.println("프로그램 종료");
    }
}
```

System 클래스 – exit( )

- 운영체제는 실행 중인 프로그램을 프로세스로 관리한다.
- Java 프로그램을 시작하면 JVM 프로세스가 생성되고, 이 프로세스에서 main() 메소드를 호출한다.
- 프로세스를 강제 종료하고 싶다면 System.exit() 메소드를 사용한다.

- exit() 메소드는 int 매개값이 필요한데, 이 값을 종료 상태값이라 한다.
- 종료 상태값에 어떤 값을 주더라도 프로세스는 종료되는데, 정상 종료는 0, 비정상 종료는 1 또는 -1을 주는 것이 관례이다.

System 클래스 – currentTimeMillis( ), nanoTime( )

- currentTimeMillis()와nanoTime()메소드는 1970년 1월 1일 0시부터 시작해서 현재까지 진행된 시간을 반환한다.

- 이 두 메소드는 프로그램 처리 시작을 측정하는데 주로 사용된다.

```
package com.system;

public class MeasureRunTImeExample {
    public static void main(String[] args) {
        long time1 = System.nanoTime();
        int sum = 0;
        for(int i=0; i<=1000000;i++) {
            sum += i;
        }
        long time2 = System.nanoTime();
        System.out.println("합계 : " + sum);
        System.out.println(time2-time1 +"나노초가 소요되었습니다.");
    }
}
```

System 클래스 – getProperty( ), getProperties( )

```java
package com.system;

import java.util.Properties;
import java.util.Set;

public class GetPropertyExample {
    public static void main(String[] args) {
        String osName = System.getProperty("os.name");
        String userName = System.getProperty("user.name");
        String userHome = System.getProperty("user.home");
        System.out.println(osName + ", " + userName + ", " + userHome);

        Properties props = System.getProperties();
        Set keys = props.keySet();
        for(Object key : keys) {
            System.out.println(key + ": " + System.getProperty((String) key));
        }
    }
}
```

문자열 클래스 – String

- 문자열 리터럴인 경우에 자동으로 String 객체로 생성된다.
- String 클래스의 다양한 생성자를 통해 직접 객체를 생성할 수도 있다.
- 예를 들어, 네트워크 통신으로 얻은 byte 배열을 원래 문자열로 변환하는 경우에 주로 쓰인다.

```
package com.string;

import java.util.Arrays;

public class BytesToStringExample {
    public static void main(String[] args) throws Exception {
        String data = "최인규";
        byte[] arr1 = data.getBytes();
        System.out.println(Arrays.toString(arr1));
        String str1 = new String(arr1);
        System.out.println(str1);

        byte[] arr2 = data.getBytes("EUC-KR");
        System.out.println(Arrays.toString(arr2));
        String str2 = new String(arr2, "EUC-KR");
        System.out.println(str2);
    }
}
```

문자열 클래스 – StringBuilder

- String은 내부 문자열을 수정할 수 없다.

```
String data = "JAVA";
data += "SCRIPT";
```

- 문자열을 변경이 가능한 것처럼 보이지만, 실제로는 새로운 String 객체를 생성하는 것이다.
- 새로운 String 객체가 생성되고 이전 객체는 계속 버려지기 때문에 효율성이 좋지 않다.
- StringBuilder는 내부 버퍼(데이터를 저장하는 메모리)에 문자열을 저장해두고, 그 안에서 추가, 수정, 삭제 작업이 가능하도록 설계되어 있다.
- 때문에 잦은 문자열 변경이 필요하다면 String보다는 StringBuilder를 사용하는 것이 좋다.

메소드	설명
append(기본값|문자열)	문자열을 끝에 추가
insert(위치, 기본값|문자열)	문자열을 지정 위치에 추가
delete(시작위치, 끝위치)	문자열 일부를 삭제
replace(시작위치, 끝위치, 문자열)	문자열 일부를 대체
toString()	완성된 문자열을 리턴

문자열 클래스 – StringBuilder

- String은 내부 문자열을 수정할 수 없다.

```
String data = "JAVA";
data += "SCRIPT";
```

- 문자열을 변경이 가능한 것처럼 보이지만, 실제로는 새로운 String 객체를 생성하는 것이다.
- 새로운 String 객체가 생성되고 이전 객체는 계속 버려지기 때문에 효율성이 좋지 않다.
- StringBuilder는 내부 버퍼(데이터를 저장하는 메모리)에 문자열을 저장해두고, 그 안에서 추가, 수정, 삭제 작업이 가능하도록 설계되어 있다.
- 때문에 잦은 문자열 변경이 필요하다면 String보다는 StringBuilder를 사용하는 것이 좋다.

```
public class StringBuilderExample {
    public static void main(String[] args) {
        String data = new StringBuilder()
            .append("cDef")
            .insert(0, "ABC")
            .delete(3, 4)
            .replace(4, 6, "EF")
            .toString();
        System.out.println(data);
    }
}
```

포장(Wrapper) 클래스

- 기본 타입의 값을 갖는 객체를 생성할 수 있다. 이러한 객체를 포장 객체라 한다.
- 포장 객체는 포장하고 있는 기본 타입의 값을 변경할 수는 없고, 오직 객체로 생성하는데 목적이 있다.

- 포장 객체가 필요한 이유는 컬렉션 객체 때문이다.
- 나중에 알아볼 컬렉션 객체는 기본 타입의 값은 저장할 수 없고, 객체만 저장 가능하다는 특징이 있다.

- 기본 타입의 값을 포장 객체로 만드는 과정을 박싱(boxing)이라 하고,
- 반대로 포장 객체에서 기본 타입의 값을 얻어내는 과정을 언박싱(unboxing)이라 한다.

```
Integer obj = 100;
int value = obj;
```

- 언박싱은 연산 과정에서도 발생 한다.

```
Integer obj = 100;
int value = obj + 50;
```

포장(Wrapper) 클래스

- 포장 클래스는 문자열을 기본 타입 값으로 변환할 때도 사용된다.
- 대부분의 포장 클래스에는 'parse + 기본타입' 으로 되어있는 정적 메소드가 있어서 문자열을 해당 기본타입 값으로 변환한다.

- 포장 객체의 내부 값을 비교하기 위해서는 == 이나 !=과 같은 비교연산자를 사용할 수 없다.
- 엄밀히 말하면 범위의 값을 갖는 포장 객체(boolean 또는 char 또는 -128~127 범위의 byte, short, int)는 비교 연산자 사용이 가능하다.
- 하지만 포장 객체 내 어떤 값이 저장될 지 모르는 상황이라면, equals() 메소드로 내부 값을 비교해야 한다.

수학(Math) 클래스

- Math 클래스는 수학 계산에 사용할 수 있는 메소드를 정적 메소드로 제공한다.

구분	코드
절대값	`int v1 = Math.abs(-5);` `double v2 = Math.abs(-3.14);`
올림값	`double v3 = Math.ceil(5.3);` `double v4 = Math.ceil(-5.3);`
버림값	`double v5 = Math.floor(5.3);` `double v6 = Math.floor(-5.3);`
최대값	`int v7 = Math.max(5, 9);` `double v8 = Math.max(5.3, 2.5);`
최소값	`int v9 = Math.min(5, 9);` `double v10 = Math.min(5.3, 2.5);`
랜덤값	`double v11 = Math.random();`
반올림값	`long v12 = Math.round(5.3);` `long v13 = Math.round(5.7);`

Random 클래스

- 랜덤값을 얻기 위해서는 Math 클래스를 이용하는 방법 외에도 java.util.Random 클래스를 이용하는 방법도 있다.

```
package com.util;

import java.util.Arrays;
import java.util.Random;

public class LottoExample {
    public static void main(String[] args) {
        int[] selectNumber = new int[6];
        Random r = new Random();

        for(int i = 0; i < 6; i++) {
            selectNumber[i] = r.nextInt(45) + 1;
        }

        System.out.println(Arrays.toString(selectNumber));
    }
}
```

날짜와 시간 클래스

- java.util 패키지 내 Date와 Calendar 클래스로 날짜와 시간을 구할 수 있으며, LocalDateTime 클래스로 조작이 가능하다.

- Date 클래스에는 여러 개의 생성자가 선언되어 있으나, 대부분 Deprecated 되어 기본 생성자만 주로 사용된다.

```
package com.util;

import java.text.SimpleDateFormat;
import java.util.Date;

public class DateExample {
    public static void main(String[] args) {
        Date now = new Date();
        System.out.println(now.toString());

        SimpleDateFormat sdf = new SimpleDateFormat("yyyy년 MM월 dd일 HH시 mm분 ss초");
        System.out.println(sdf.format(now));
    }
}
```

날짜와 시간 클래스

- Calendar 클래스는 달력을 표현하는 추상 클래스이다.
- 날짜와 시간을 계산하는 방법이 지역과 문화에 따라 차이가 있어 달력은 자식 클래스에서 구현하도록 되어 있다.
- 특별한 역법을 사용하는 경우가 아니면, 직접 하위 클래스를 만들 필요 없이,
  getInstance( ) 메소드를 이용해 컴퓨터에 설정된 시간대를 기준으로 Calendar 하위 객체를 얻을 수 있다.

```
Calendar today = Calendar.getInstance();
int year = today.get(Calendar.YEAR);
int month = today.get(Calendar.MONTH) + 1;
int day = today.get(Calendar.DAY_OF_MONTH);
int week = today.get(Calendar.DAY_OF_WEEK);
int amPm = today.get(Calendar.AM_PM);
String strWeek = null;
switch(week) {
     case Calendar.MONDAY: strWeek = "월"; break;
     case Calendar.TUESDAY: strWeek = "화"; break;
     case Calendar.WEDNESDAY: strWeek = "수"; break;
     case Calendar.THURSDAY: strWeek = "목"; break;
     case Calendar.FRIDAY: strWeek = "금"; break;
     case Calendar.SATURDAY: strWeek = "토"; break;
     case Calendar.SUNDAY: strWeek = "일"; break;
}
String strAmPm = "오전";
if(amPm == Calendar.PM) strAmPm = "오후";

System.out.println(year + "년 " + month + "월 " + day + "일 " + strWeek + "요일 " + strAmPm);
```

날짜와 시간 클래스

- Date와 Calendar는 날짜와 시간 정보를 얻기에는 충분하지만, 날짜와 시간을 조작할 수는 없다.
- 조작을 위해서는 java.time 패키지의 LocalDateTime 클래스가 제공하는 메소드를 이용할 수 있다.

```
package com.time;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateTimeOperationExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy.MM.dd a HH:mm:ss");

        System.out.println(now.format(dtf));

        LocalDateTime result1 = now.plusYears(1);
        System.out.println(result1.format(dtf));
    }
}
```

날짜와 시간 클래스

```java
package com.time;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;

public class DateTimeCompareExample {
    public static void main(String[] args) {
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy.MM.dd a HH:mm:ss");
        LocalDateTime start = LocalDateTime.of(2024, 1, 1, 0, 0, 0);
        LocalDateTime end = LocalDateTime.of(2024, 12, 31, 0, 0, 0);

        if (start.isBefore(end)) {
            System.out.println("진행 중입니다.");
        } else {
            System.out.println("끝났습니다.");
        }

        long remainDay = start.until(end, ChronoUnit.DAYS);

        System.out.println("남은 날 : " + remainDay);
    }
}
```

리플렉션

- Java는 클래스와 인터페이스의 메타 정보(패키지 정보, 타입 정보, 멤버 정보 등)를 Class 객체로 관리한다.
- 메타 정보를 프로그램에서 읽고 수정하는 행위를 리플렉션(Reflection)이라 한다.

```
package com.reflection;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class Car {
    private String model, owner;
}
```

리플렉션

```
package com.reflection;

import java.lang.reflect.Field;

public class ReflectionExample {
    public static void main(String[] args) {
        Class clazz = Car.class;
        System.out.println(clazz.getPackage().getName());
        System.out.println(clazz.getSimpleName());
        Field[] fields = clazz.getDeclaredFields();
        for(Field field : fields) {
        System.out.println(field.getType().getName() + " "+ field.getName());
        }
    }
}
```

어노테이션

- 코드에서 @로 작성되는 요소를 어노테이션(Annotation)이라 한다.
- 어노테이션은 클래스 또는 인터페이스를 컴파일하거나 실행할 때, 어떻게 처리해야 할 것인지를 알려주는 설정 정보이다.

- 어노테이션의 용도는 아래와 같다.
  1. 컴파일 시 사용하는 정보 전달 - [예) @Override]
  2. 빌드 툴이 코드를 자동으로 생성할 때 사용하는 정보 전달 [예) @Data, @Getter, @Setter]
  3. 실행 시 특정 기능을 처리할 때 사용하는 정보 전달 [예) @Scheduled]

- 어노테이션은 자바 프로그램 개발 시 필수 요소가 되었다.
- 웹 개발 시 사용되는 Spring, SpringBoot 프레임워크에서는 다양한 종류의 어노테이션을 사용한다.

어노테이션

- 어노테이션도 하나의 타입이므로, 정의부터 해야 한다.
- 어노테이션의 정의는 인터페이스를 정의하는 것과 유사하다.

```
package com.annotation;

public @interface MyAnnotation { }
```

- 어노테이션은 타입과 이름으로 구성된 속성을 가질 수 있다. 속성의 기본값은 default 키워드로 지정할 수 있다.
- 기본값이 있는 속성은 생략 가능하고, 기본값이 없는 속성은 반드시 값을 기술해야 한다.

```
public @interface MyAnnotation {
    String prop1();
    int prop2() default 1;
}
```

어노테이션

- 어노테이션은 설정 정보이기 때문에 어떤 대상에 설정 정보를 적용할 것인지 명시해야 한다.
- 적용 대상을 지정할 때는 @Target 어노테이션을 사용한다.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

// 클래스, 필드, 메소드에 적용 가능한 어노테이션 (생성자에는 적용 불가)
@Target({ ElementType.TYPE, ElementType.FIELD, ElementType.METHOD })
public @interface MyAnnotation {
    String value();
    String prop1();
    int prop2() default 1;
}
```

어노테이션

- 어노테이션을 정의할 때 한 가지 더 추가해야 할 내용은 언제까지 유지할 것인지를 지정하는 것이다.
- 어노테이션의 유지 정책을 지정할 때는 @Retention 어노테이션을 사용한다.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

// 클래스, 필드, 메소드에 적용 가능한 어노테이션 (생성자에는 적용 불가)
@Target({ ElementType.TYPE, ElementType.FIELD, ElementType.METHOD })
// 실행할 때 적용하여 계속 유지되는 어노테이션
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    String value();
    String prop1();
    int prop2() default 1;
}
```

어노테이션 [실습]

```java
package com.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface PrintAnnotation {
    String value() default "-";
    int number() default 15;
}
```

어노테이션 [실습]

```
package com.annotation;

public class Service {
    @PrintAnnotation
    public void method1() {
        System.out.println("실행 내용 1");
    }

    @PrintAnnotation("*")
    public void method2() {
        System.out.println("실행 내용 2");
    }

    @PrintAnnotation(value = "#", number = 20)
    public void method3() {
        System.out.println("실행 내용 3");
    }
}
```

어노테이션 [실습]

```
package com.annotation;

import java.lang.reflect.Method;

public class PrintAnnotationExample {
    public static void main(String[] args) throws Exception {
        Method[] methods = Service.class.getDeclaredMethods();
        for(Method method: methods) {
            PrintAnnotation pa = method.getAnnotation(PrintAnnotation.class);
            printLine(pa);
            method.invoke(new Service());
            printLine(pa);
        }
    }

    public static void printLine(PrintAnnotation pa) {
        int number = pa.number();
        for (int i = 0; i < number; i++) {
            System.out.print(pa.value());
        }
        System.out.println();
    }
}
```

제네릭

- 제네릭(Generic)이란 결정되지 않은 타입을 파라미터로 처리하고, 실제 사용할 때 파라미터를 구체적인 타입으로 대체시키는 기능이다.

- 다양한 내용물을 저장하는 Box라는 클래스가 있고, Box에 넣을 내용물로 content 필드를 선언하려고 한다.
- content 필드는 다양한 내용물이 저장되기 때문에 특정 클래스 타입으로 선언할 수 없다.
  그래서 최상위 부모 클래스인 Object로 선언해야 할 것이다.

```
package com.generic;

public class Box {
    public Object content;
}
```

- 모든 객체는 부모 타입인 Object로 자동 타입 변환이 되므로, content 필드에는 어떤 객체든 대입이 가능해진다.

```
public class BoxExample {
    public static void main(String[] args) {
        Box box = new Box();
        box.content = "안녕하세요";
        box.content = 100;
    }
}
```

제네릭

- 하지만 문제는 Box 안의 내용물을 얻을 때이다.
- 내용물의 타입을 안다면 강제 타입 변환을 거쳐 얻을 수 있다.

```
Box box = new Box();

box.content = "안녕하세요";
String content1 = (String) box.content;

box.content = 100;
int content2 = (int) box.content;
```

- 하지만 어떤 내용물이 저장되어 있는지 모를 수 있다. 그렇기에 content의 타입을 Object로 하는 것은 좋은 방법이 아니다.

제네릭

- Box를 생성할 때 저장할 내용물의 타입을 명시하면 Box는 content에 무엇이 대입되고, 읽을 때 어떤 타입으로 제공할 지를 알 수 있게 된다.

```
package com.generic;

public class Box <T> {
    public T content;
}



public class BoxExample {
    public static void main(String[] args) {
        Box<String> box1 = new Box<>();
        box1.content = "안녕하세요";
        String content1 = box1.content; // 강제타입 변환없이 얻을 수 있다.

        Box<Integer> box2 = new Box<>();
        box2.content = 100;
        int content2 = box2.content; // 강제타입 변환없이 얻을 수 있다.
    }
}
```

제네릭 타입

- 제네릭 타입은 결정되지 않은 타입을 파라미터로 가지는 클래스와 인터페이스를 말한다.
- 제네릭 타입은 선언부에 '<>' 부호가 붙고 그 사이에 타입 파라미터들이 위치한다.

- 타입 파라미터는 일반적으로 대문자 알파벳 한 글자로 표현한다.
- 외부에서 제네릭 타입을 사용하기 위해서는 타입 파라미터에 구체적인 타입을 지정해야 한다.
- 만약 지정하지 않으면 Object 타입이 암묵적으로 사용된다.

```java
package com.generic;

public class Product <K, M> {
    private K kind;
    private M model;

    public K getKind() { return this.kind; }
    public M getModel() { return this.model; }
    public void setKind(K kind) { this.kind = kind; }
    public void setModel(M model) { this.model = model; }
}
```

```java
package com.generic;

public class Tv { }
```

```java
package com.generic;

public class Car { }
```

제네릭 타입

```
package com.generic;

public class GenericExample {
    public static void main(String[] args) {
        Product<Tv, String> product1 = new Product<>();
        product1.setKind(new Tv());
        product1.setModel("스마트 TV");
        Tv tv = product1.getKind();
        String tvModel = product1.getModel();

        Product<Car, String> product2 = new Product<>();
        product2.setKind(new Car());
        product2.setModel("SUV 자동차");
        Car car = product2.getKind();
        String carModel = product2.getModel();
    }
}
```

제네릭 타입

- 이번에는 인터페이스를 제네릭 타입으로 선언해보자.

```
package com.generic;

public interface Rentable<P> {
    P rent();
}

public class Appliance {
    public void turnOn() {
        System.out.println("전원을 켭니다.");
    }
}

public class Tool {
    public void use() {
        System.out.println("도구를 사용합니다.");
    }
}
```

제네릭 타입

```
package com.generic;

public class ToolAgency implements Rentable<Tool>{
    @Override
    public Tool rent() {
        return new Tool();
    }
}


public class ApplianceAgency implements Rentable<Appliance>{
    @Override
    public Appliance rent() {
        return new Appliance();
    }
}
```

제네릭 타입

```
package com.generic;


public class GenericExample2 {
    public static void main(String[] args) {
        ApplianceAgency aa = new ApplianceAgency();
        Appliance appliance = aa.rent();
        appliance.turnOn();

        ToolAgency ta = new ToolAgency();
        Tool tool = ta.rent();
        tool.use();
    }
}
```

제네릭 - 제한된 타입 파라미터

- 경우에 따라서는 타입 파라미터를 대체하는 구체적인 타입을 제한할 필요가 있다.
- 예를 들어 숫자를 연산하는 제네릭 메소드는 Number 또는 자식 클래스(Byte, Short, Integer, Long, Double)로 제한할 필요가 있다.

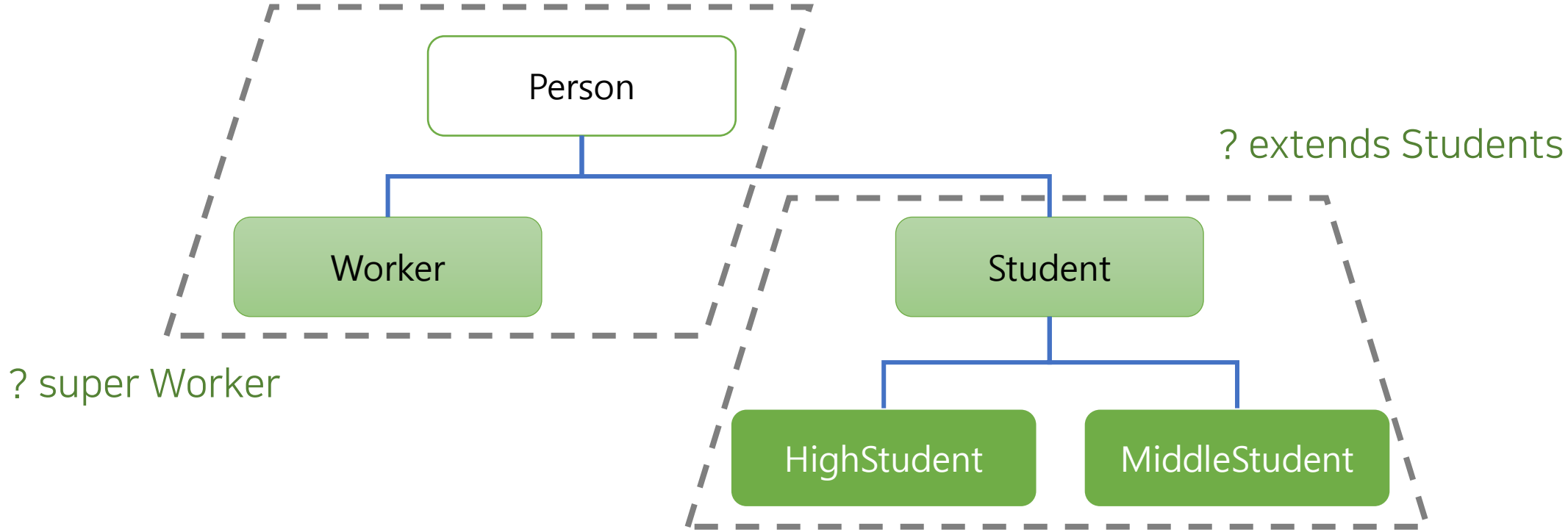
- 이처럼 모든 타입으로 대체할 수는 없고, 특정 타입과 자식 또는 구현 관계에 있는 타입만 대체할 수 있도록 하는 타입 파라미터를 제한된 타입 파라미터(Bounded Type Parameter)라고 한다.

```
public class GenericExample3 {
    public static <T extends Number> boolean compare(T t1, T t2) {
        // Number에 정의되어 있는 메소드 사용
        double v1 = t1.doubleValue();
        double v2 = t2.doubleValue();
        return (v1 == v2);
    }
    public static void main(String[] args) {
        boolean result1 = compare(10, 20);
        System.out.println(result1);
        boolean result2 = compare(20L, 20D);
        System.out.println(result2);
        boolean result3 = compare(4.5, 4.5);
        System.out.println(result3);
    }
}
```

제네릭 – 와일드카드 타입 파라미터

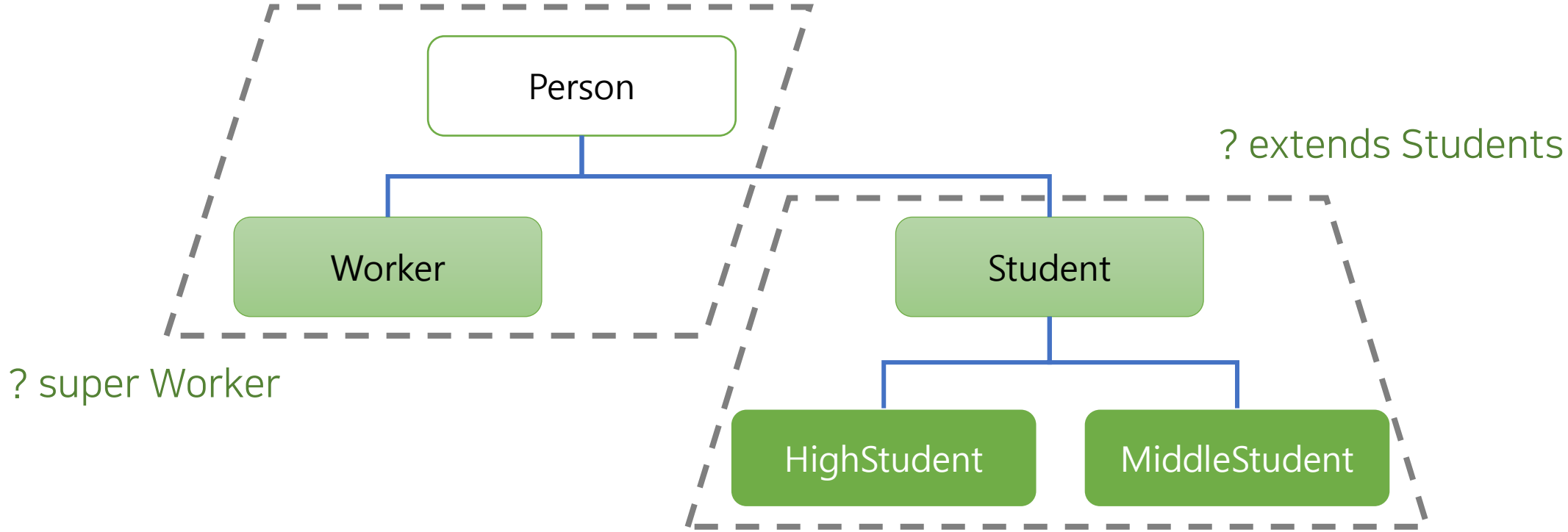
- 제네릭 타입을 매개값이나 반환 타입으로 사용할 때, 타입 파라미터로 와일드카드(?)를 사용할 수 있다.
- 와일드카드(?)는 범위에 있는 모든 타입으로 대체할 수 있다는 표시이다.

- 예를 들어 아래와 같은 상속 관계가 있다고 가정해보자.



제네릭 – 와일드카드 타입 파라미터

- 타입 파라미터의 대체 타입으로 Student의 자식 클래스만 가능하도록 선언할 수 있으며,
- 반대로 Worker의 부모 클래스만 가능하도록 선언할 수도 있다.



제네릭 – 와일드카드 타입 파라미터

- Course 클래스
  - 메소드 registerCourse1() : 모든 사람이 들을 수 있는 과정을 등록
  - 메소드 registerCourse2() : 학생만 들을 수 있는 과정을 등록
  - 메소드 registerCourse3() : 직장인과 일반인만 들을 수 있는 과정을 등록

```java
package com.generic;

public class Person { }
class Worker extends Person {}
class Student extends Person {}
class HighStudent extends Student {}
class MiddleStudent extends Student {}
```

```java
package com.generic;


public class Applicant<T> {
    public T kind;
    public Applicant(T kind) {
        this.kind = kind;
    }
}
```

제네릭 – 와일드카드 타입 파라미터

- Course 클래스
  - 메소드 registerCourse1() : 모든 사람이 들을 수 있는 과정을 등록
  - 메소드 registerCourse2() : 학생만 들을 수 있는 과정을 등록
  - 메소드 registerCourse3() : 직장인과 일반인만 들을 수 있는 과정을 등록

```
public class Course {
    public static void registerCourse1(Applicant<?> applicant) {
        System.out.println(applicant.kind.getClass().getSimpleName() + ": Course1 등록");
    }
    public static void registerCourse2(Applicant<? extends Student> applicant) {
        System.out.println(applicant.kind.getClass().getSimpleName() + ": Course2 등록");
    }
    public static void registerCourse3(Applicant<? super Worker> applicant) {
        System.out.println(applicant.kind.getClass().getSimpleName() + ": Course3 등록");
    }
}
```

제네릭 – 와일드카드 타입 파라미터

```
public class GenericExample4 {
    public static void main(String[] args) {
        Applicant<Person> p = new Applicant<>(new Person());
        Applicant<Worker> w = new Applicant<>(new Worker());
        Applicant<Student> s = new Applicant<>(new Student());
        Applicant<HighStudent> hs = new Applicant<>(new HighStudent());
        Applicant<MiddleStudent> ms = new Applicant<>(new MiddleStudent());

        Course.registerCourse1(p);
        Course.registerCourse1(w);
        Course.registerCourse1(s);
        Course.registerCourse1(hs);
        Course.registerCourse1(ms);

        Course.registerCourse2(s);
        Course.registerCourse2(hs);
        Course.registerCourse2(ms);

        Course.registerCourse3(p);
        Course.registerCourse3(w);
    }
}
```

멀티 스레드

- 운영체제는 실행 중인 프로그램을 프로세스(Process)로 관리한다.
- 멀티 태스킹은 두 가지 이상의 작업을 동시에 처리하는 것을 말하는데, 이 때 운영체제는 멀티 프로세스를 생성해서 처리한다.
- 그러나 멀티 태스킹이 꼭 멀티 프로세스를 의미하는 것은 아니다.

- 하나의 프로세스 내에서 멀티 태스킹을 할 수 있도록 만들어진 프로그램도 있다. [카카오톡 : 채팅을 하면서 파일 전송 작업도 동시에 수행]

- 하나의 프로세스가 두 가지 이상의 작업을 처리할 수 있는 이유는 멀티 스레드(Multi Thread)가 있기 때문이다.
- 스레드(Thread)는 코드의 실행 흐름을 말하는데, 프로세스 내에 스레드가 두 개라면 두 개의 코드 실행 흐름이 생긴다는 의미이다.

TIP

멀티 프로세스 : 프로그램 단위의 멀티 태스킹
멀티 스레드 : 프로세스 단위의 멀티 태스킹 (프로그램 내부)

- 멀티 프로세스는 서로 독립적이므로 하나의 프로세스에서 오류가 발생해도 다른 프로세스에 영향을 미치지 않지만,
  멀티 스레드는 하나의 스레드가 예외를 발생시키면, 프로세스가 종료되므로 다른 스레드에게 영향을 미친다.

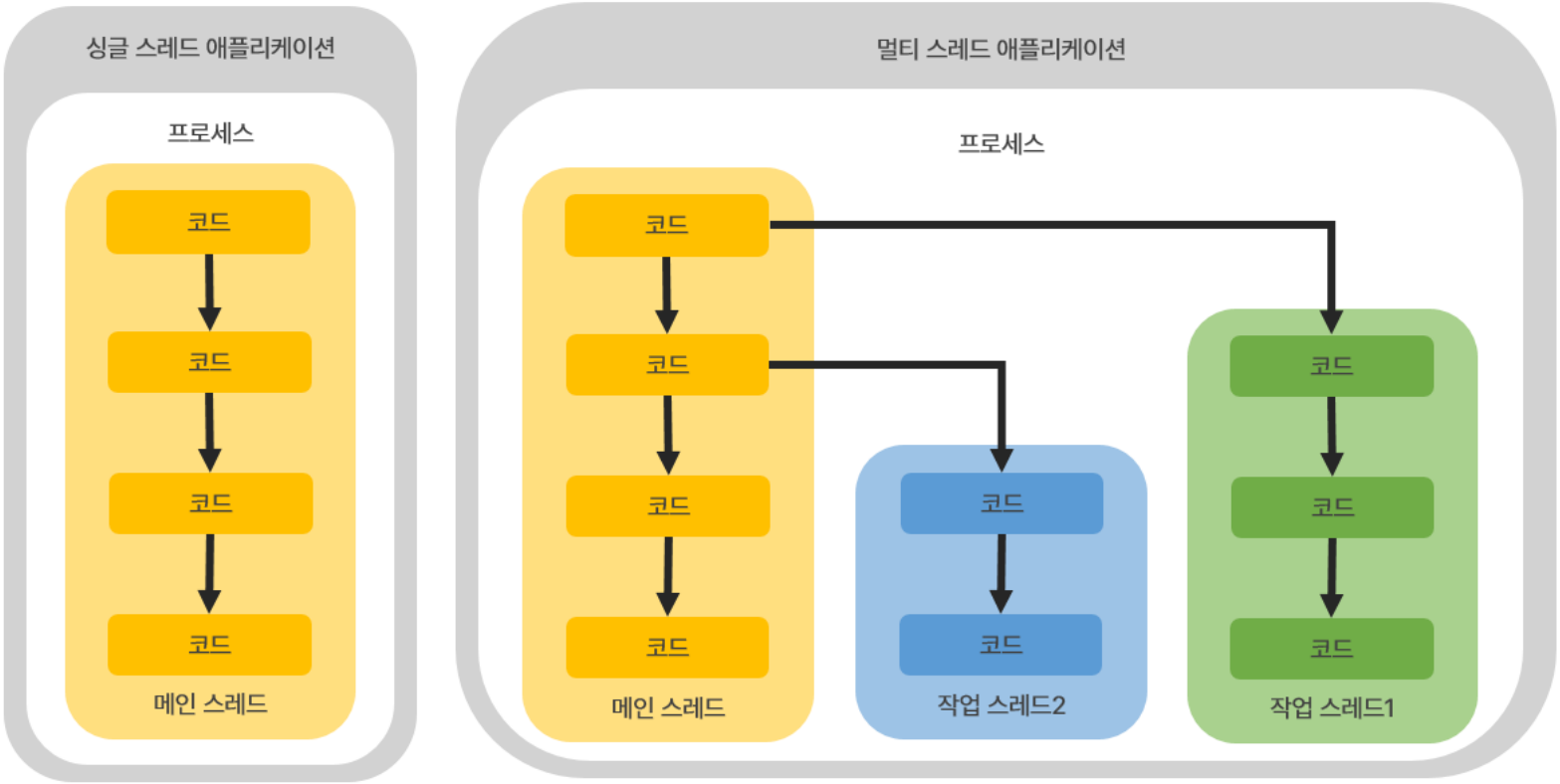
멀티 스레드 – 메인 스레드

- 모든 Java 프로그램은 메인 스레드(Main Thread)가 main() 메소드를 실행하면서 시작된다.
- 메인 스레드는 main() 메소드의 첫 코드부터 순차적으로 실행하고,
  main() 메소드의 마지막 코드를 실행하거나 return 문을 만나면 실행을 종료한다.

- 메인 스레드는 필요에 따라 추가 작업 스레드들을 만들어서 실행시킬 수 있다.

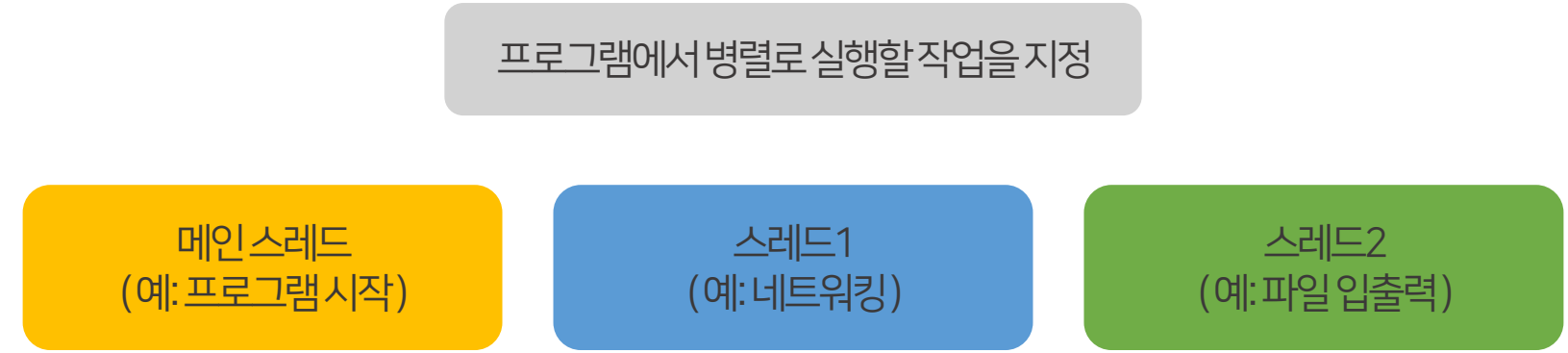
멀티 스레드 – 메인 스레드

- 싱글 스레드에서는 메인 스레드가 종료되면 프로세스도 종료된다.
- 하지만 멀티 스레드 환경에서는 실행 중인 스레드가 하나라도 있다면 프로세스가 종료되지 않는다.



멀티 스레드 – 작업 스레드 생성과 실행

- 멀티 스레드로 실행되는 프로그램을 개발하려면 먼저 몇 개의 작업을 병렬로 실행할지 결정하고, 각각의 작업 별로 스레드를 생성해야 한다.
- 메인 스레드는 반드시 존재하기 때문에 추가적인 작업 수만큼 스레드를 생성한다.



- 작업 스레드도 객체로 관리하기 때문에 클래스가 필요하다. 여기에는 두 가지 방법이 있는데,
  - Thread 클래스로 직접 객체를 생성하는 방법
  - 하위 클래스를 만들어 생성하는 방법

멀티 스레드 – Thread 클래스로 스레드 직접 생성하기

- java.lang 패키지에는 Thread 클래스로부터 작업 스레드 객체를 직접 생성하기 위해서는 Runnable 구현 객체를 매개값으로 갖는 생성자를 호출하면 된다.
- Runnable은 스레드가 작업을 실행할 때 사용하는 인터페이스로, 그 안에는 run() 메소드가 정의되어 있다.
- 구현 클래스는 run() 메소드에 스레드가 실행할 코드를 재정의(오버라이딩)해주면 된다.

```
package com.thread;

public class ThreadExample1 {
    public static void main(String[] args) {
        Thread thread = new Thread(new Task());
    }
}

class Task implements Runnable {
    @Override
    public void run() {
        // 스레드가 실행할 코드
    }
}
```

멀티 스레드 – Thread 클래스로 스레드 직접 생성하기

- 일반적으로는 명시적인 Runnable 구현 클래스를 작성하지 않고, 익명 구현 객체를 통해 작성하는 방식이 더 많이 사용된다.

```
package com.thread;

public class ThreadExample1 {
    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                // 스레드가 실행할 코드
            }
        });
    }
}
```

멀티 스레드 – Thread 클래스로 스레드 직접 생성하기

- 작업 스레드 객체가 생성되었다고 해서, 작업 스레드가 바로 실행되는 것은 아니다.
- 작업 스레드를 실행하기 위해서는 스레드 객체의 start() 메소드를 호출해야 한다.

```
package com.thread;

public class ThreadExample1 {
    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                // 스레드가 실행할 코드
            }
        });

        thread.start();
    }
}
```

```java
public class ThreadExample1 {
    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i=5; i>0; i--) {
                    System.out.println("스레드"+ i);
                    try {
                        Thread.sleep(500);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        });
        thread.start();

        for (int i=0; i<5; i++) {
            System.out.println("메인"+ i);
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

멀티 스레드 – Thread 자식 클래스로 스레드 생성하기

- 작업 스레드 객체를 생성하는 또 다른 방법은 Thread의 자식 객체로 만드는 것이다.
- Thread 클래스를 상속한 다음 run() 메소드를 재정의해서 스레드가 실행할 코드를 작성하고 객체를 생성하면 된다.

```
package com.thread;

public class ThreadExample2 {
    public static void main(String[] args) {
        Thread thread = new WorkerThread();
        thread.start();
    }
}

class WorkerThread extends Thread {
    @Override
    public void run() {
        // 스레드가 실행할 코드
    }
}
```

멀티 스레드 – Thread 자식 클래스로 스레드 생성하기

- 일반적으로는 명시적인 자식 클래스를 생성하지 않고, 익명 자식 객체를 통해 작성하는 방식이 더 많이 사용된다.

```
package com.thread;

public class ThreadExample2 {
    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                // 스레드가 실행할 코드
            }
        };
        thread.start();
    }
}
```

```java
public class ThreadExample2 {
    public static void main(String[] args) {
        Thread thread = new Thread() {
            @Override
            public void run() {
                for (int i = 5; i > 0; i--) {
                    System.out.println("스레드" + i);
                    try {
                        Thread.sleep(500);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        };
        thread.start();


        for (int i = 0; i < 5; i++) {
            System.out.println("메인" + i);
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

멀티 스레드 – 스레드 이름

- 스레드는 각각 자신의 이름이 있다.
- 메인 스레드는 'main'이라는 이름을 가지고 있고, 작업 스레드는 자동적으로 'Thread-0, Thread-1, Thread-2, …'와 같은 이름을 갖는다.
- 만약 'Thread-n' 대신 다른 이름을 설정하고 싶다면, Thread 클래스의 setName() 메소드를 사용하면 된다.
- 현재 코드를 어떤 스레드에서 실행하고 있는지 확인하기 위해서는
  Thread 클래스의 정적 메소드인 currentThread()로 스레드 객체의 참조를 얻은 다음, getName() 메소드를 사용하면 된다.

```
Thread thread = new Thread() {
    @Override
    public void run() {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName());
    }
};
thread.setName("MyThread");
thread.start();
```

멀티 스레드 – 스레드 상태

- 스레드는 객체가 생성, 실행, 종료가 될 때까지 다양한 상태를 가진다.
- 각 스레드의 상태는 Thread.State 타입으로 정의되어 있고, 6가지의 상수 집합이다.
  - (NEW, RUNNABLE, TERMINATED, TIMED_WAITING, BLOCKED, WAITING)

- 처음 객체가 생성되면 NEW 상태를 가지며, 이후 start() 메소드로 실행하면 RUNNABLE 상태가 된다.
  RUNNABLE 상태에서는 실행과 실행 대기를 반복하면서 CPU를 다른 스레드들과 나눠 사용한다.
  이후 run() 메소드가 종료되면 TERMINATED 상태가 된다.
- RUNNABLE 상태에서는 상황에 따라 TIMED_WAITING, BLOCKED, WAITING 상태로 전환될 수 있다.

```java
public class ThreadExample3 {
    public static void main(String[] args) {
        Thread.State state;
        Thread thread = new Thread() {
            @Override
            public void run() {
                for(int i=0;i<1000000000;i++) {}
            };
        };

        state = thread.getState();
        System.out.println("스레드 상태1: " + state);
        thread.start();
        state = thread.getState();
        System.out.println("스레드 상태2: " + state);

        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        state = thread.getState();
        System.out.println("스레드 상태3: " + state);
    }
}
```

멀티 스레드 – 스레드 일시정지

- 실행 중인 스레드를 일정 시간 멈추게 하고 싶다면 Thread 클래스의 sleep() 메소드를 이용하면 된다.
- 매개값에는 얼마 동안 일시 정지 상태로 있을 것인지를 밀리세컨드 단위로 부여하면 된다.

- 스레드는 다른 스레드와 독립적으로 실행하지만, 가끔은 다른 스레드가 종료될 때까지 기다렸다가 실행을 해야 하는 경우도 있다.
  - 예) 스레드 B의 작업이 완료되고, 그 결과값을 받아 처리하는 스레드 A가 있는 경우

- 이를 위해서는 join() 메소드를 이용할 수 있다.
- join() 메서드는 sleep() 메서드와 다르게, 해당 스레드에게 CPU의 사용권도 넘겨준다는 특징이 있다.

멀티 스레드 – 스레드 일시정지

```
public class ThreadA extends Thread {
    @Override
    public void run() {
        System.out.println("스레드A 시작");
        ThreadB threadB = new ThreadB();
        threadB.start();
        try {
            threadB.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(threadB.sum * 2);
        System.out.println("스레드A 끝");
    }
}
```

```
public class ThreadB extends Thread {
    public int sum = 0;
    @Override
    public void run() {
        System.out.println("스레드B 시작");
        for(int i = 0; i<100; i++) {
            sum += i;
        }
        System.out.println("스레드B 끝");
    }
}
```

```
public class ThreadExample4 {
    public static void main(String[] args) {
        ThreadA threadA = new ThreadA();
        threadA.start();
    }
}
```

멀티 스레드 – 스레드 양보

- 스레드가 처리하는 작업은 반복처리가 많은 편인데, 가끔은 반복이 무의미한 반복으로 처리되는 경우가 있다.
- 이 때는 다른 스레드에게 실행을 잠시 양보하고, 자신은 실행 대기 상태로 가는 것이 프로그램 성능에 도움을 준다.
- Thread는 yield() 메소드를 제공하여, yield()를 호출한 스레드는 실행 대시 상태로 돌아가고, 다른 스레드가 실행 상태가 된다.

```
public class ThreadC extends Thread {
    public boolean work = true;
    @Override
    public void run() {
        while(true) {
            if (work) {
                System.out.println(Thread.currentThread().getName() + "작업처리");
            } else {
                Thread.yield();
            }
        }
    }
}
```

멀티 스레드 – 스레드 양보

```
public class ThreadExample5 {
    public static void main(String[] args) {
        ThreadC thread0 = new ThreadC();
        ThreadC thread1 = new ThreadC();
        thread0.start();
        thread1.start();
        try {
            Thread.sleep(5000);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
        thread0.work = false;

        try {
            Thread.sleep(5000);
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
        thread0.work = true;
    }
}
```

멀티 스레드

- 스레드 동기화
  - 멀티 스레드는 하나의 객체를 공유해서 작업도 할 수 있다.
    하지만 다른 스레드에서 객체를 조작한다면, 의도했던 결과와는 다른 결과를 초래할 수 있다.
    스레드가 사용 중인 객체를 다른 스레드가 변경할 수 없도록 객체에 잠금을 걸 수 있다.
  - 동기화 메소드와 동기화 블록은 동시 접근이 불가능하다.

- wait(), notify()
  - 두 개의 스레드를 가지고 정확히 교대로 번갈아가며 작업하도록 처리할 수 있다.
  - 한 스레드가 작업을 완료하면 notify() 메소드를 호출해, 일시정지에 있는 다른 스레드를 실행대기 상태로 만들고,
    자신은 두 번 작업을 하지 않도록 wait() 메소드를 호출하여 일지정지 상태로 만든다.
  - 주의할 점은 wait(), notify() 메소드 모두 동기화 메소드 또는 동기화 블록 내에서만 사용 가능하다는 것이다.

- 스레드 안전 종료
  - 조건 분기 처리 또는 interrupt() 메소드 사용으로 후, 리소스 정리 작업을 추가하는 것으로 스레드를 안전하게 종료할 수 있다.

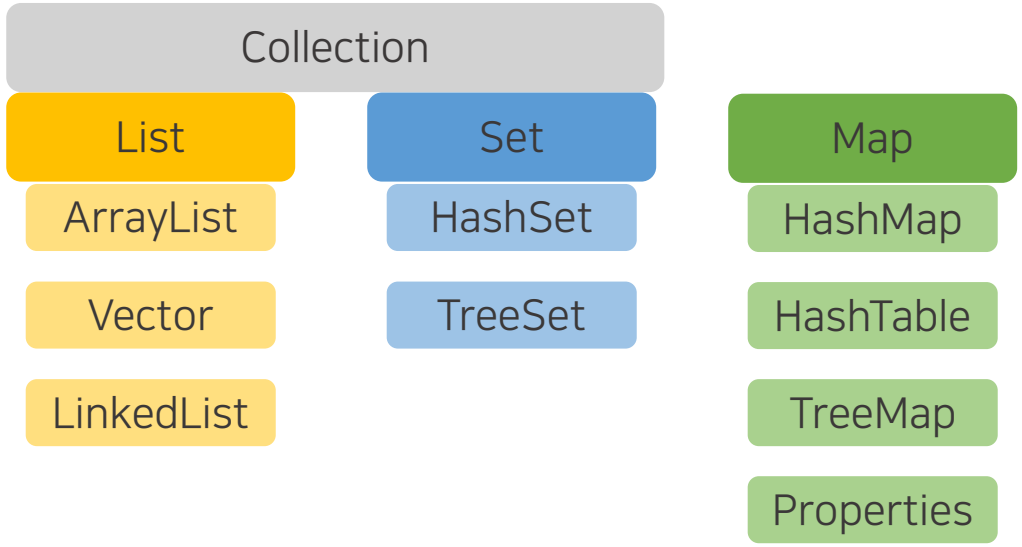
- 데몬 스레드
  - 데몬 스레드는 주 스레드의 작업을 돕는 보조적인 스레드로, 주 스레드가 종료되면 자동으로 종료된다.
  - 데몬 스레드로 만들기 위해서는 start() 호출 전에 setDaemon(true)를 호출하면 된다.

컬렉션 자료구조

- Java는 널리 알려져 있는 자료구조(Data Structure)를 바탕으로 객체들을 효율적으로 이용할 수 있도록 관련된 인터페이스와 클래스들을 java.util 패키지에 포함시켜 놓았다.

- 이들을 총칭해서 컬렉션 프레임워크(Collection Framework)라고 부른다.
- 컬렉션 프레임워크는 몇 가지 인터페이스를 통해서 다양한 컬렉션 클래스를 이용할 수 있도록 설계되어 있다.

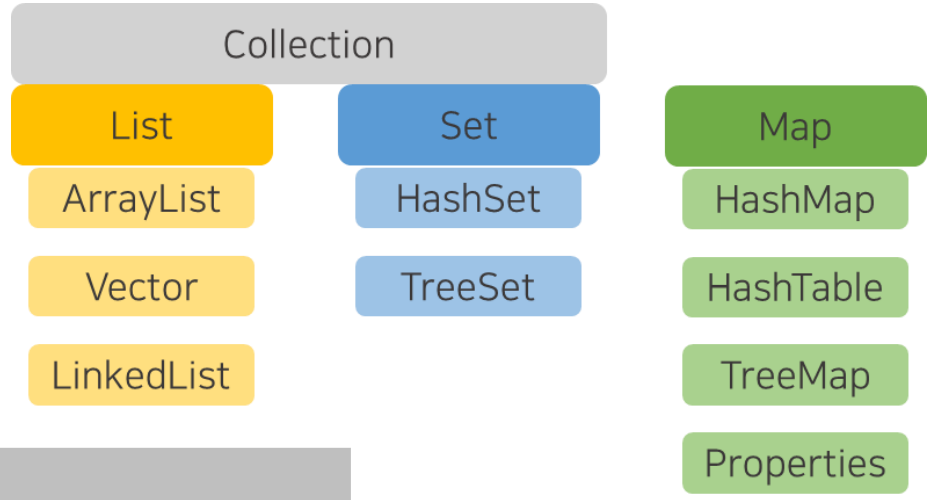
- 주요 인터페이스로는 List, Set, Map 등이 있다.



컬렉션 자료구조

- List와 Set은 객체를 추가, 삭제, 검색하는 방법에 있어서 공통점이 있기 때문에
  공통된 메소드만 따로 모아 Collection 인터페이스로 정의해두고 이것을 상속하고 있다.

- Map은 키와 값을 하나의 쌍으로 묶어서 관리하는 구조로 되어있어, List와 Set과는 사용법이 다르다.



인터페이스 분류		특징
Collection	List	• 순서를 유지함, 중복 저장 가능함
	Set	• 순서를 유지하지 않음, 중복 저장 불가능함
Map		• 키와 값으로 구성된 엔트리, 키는 중복 저장 불가능함

컬렉션 자료구조 - List 컬렉션

- List 컬렉션은 객체를 인덱스로 관리하게 때문에 객체를 저장하면 인덱스가 부여되고, 인덱스로 객체를 검색, 삭제할 수 있는 기능을 제공한다.
- List 컬렉션에는 ArrayList, Vector, LinkedList 등이 있는데,
  List 컬렉션에서 공통적으로 사용 가능한 List 인터페이스 메소드는 아래와 같다.

기능	메소드	설명
객체 추가	`boolean add(E e)`	주어진 객체를 맨 끝에 추가
	`void add(int index,E element)`	주어진 인덱스에 객체를 추가
	`set(int index, E element)`	주어진 인덱스의 객체를 새로운 객체로 바꿈
객체 검색	`boolean contains(Object o)`	주어진 객체가 저장되어 있는 지 여부 반환
	`E get(int index)`	주어진 인덱스에 저장된 객체 반환
	`isEmpty()`	컬렉션이 비어 있는지 유무 반환
	`int size()`	저장되어 있는 전체 객체 수를 반환
객체 삭제	`void clear()`	저장된 모든 객체 삭제
	`E remove(int index)`	주어진 인덱스에 저장된 객체를 삭제
	`boolean remove(Object o)`	주어진 객체를 삭제

컬렉션 자료구조 - List 컬렉션 (ArrayList)

- ArrayList는 List 컬렉션에서 가장 많이 사용되는 컬렉션이다.
- ArrayList에 객체를 추가하면 내부 배열에 객체가 생성된다.
- 일반 배열은 고정된 길이라면 ArrayList는 제한 없이 객체를 추가할 수 있다는 차이가 있다.

```
List<E> list = new ArrayList<>(); // E에 지정된 타입의 객체만 저장
List<E> list = new ArrayList<E>(); // E에 지정된 타입의 객체만 저장
List list = new ArrayList(); // 모든 타입의 객체를 저장
```

컬렉션 자료구조 - List 컬렉션 (Vector)

- Vector는 ArrayList와 동일한 내부 구조를 가지고 있으며,
  동기화된 메소드로 구성되어 있어서 멀티 스레드가 동시에 Vector 메소드를 실행할 수 없다는 점에서 차이가 있다.
- 즉, 멀티 스레드 환경에서도 안전하게 객체를 추가, 삭제할 수 있다.

```
List<E> list = new Vector<E>(); // E에 지정된 타입의 객체만 저장
List<E> list = new Vector<E>(); // E에 지정된 타입의 객체만 저장
List list = new Vector(); // 모든 타입의 객체를 저장
```

컬렉션 자료구조 - List 컬렉션 (LinkedList)

- LinkedList는 ArrayList와 사용 방법은 동일하지만 내부 구조는 완전히 다르다.
- ArrayList는 내부 배열에 객체를 저장하지만, LinkedList는 인접 객체를 사슬 형태로 연결해서 관리한다.

- LinkedList는 특정 위치에서 객체를 삽입하거나 삭제하면 바로 앞뒤 링크만 변경하면 되므로
  빈번한 객체의 삭제와 삽입이 일어난다면, ArrayList를 사용하는 것보다는 LinkedList를 사용하는 것이 더 바람직하다.

```
List<E> list = new LinkedList<E>(); // E에 지정된 타입의 객체만 저장
List<E> list = new LinkedList<E>(); // E에 지정된 타입의 객체만 저장
List list = new LinkedList(); // 모든 타입의 객체를 저장
```

컬렉션 자료구조 - Set 컬렉션

- List 컬렉션은 저장 순서를 유지하지만, Set 컬렉션은 저장 순서가 유지되지 않는다.
  또한 객체를 중복해서 저장할 수 없고, 하나의 null만 저장할 수 있다.
- Set 컬렉션은 수학의 집합에 비유될 수 있다.
- Set 컬렉션은 순서가 없기 때문에 인덱스로 관리하지 않는다. 즉, 인덱스를 매개값으로 갖는 메소드가 없다.
- Set 컬렉션에는 HashSet, LinkedHashSet, TreeSet 등이 있는데,
  Set 컬렉션에서 공통적으로 사용 가능한 Set 인터페이스 메소드는 아래와 같다.

기능	메소드	설명
객체 추가	`boolean add(E e)`	주어진 객체를 성공적으로 저장하면 true, 중복 객체면 false 반환
객체 검색	`boolean contains(Object o)`	주어진 객체가 저장되어 있는지 여부 반환
	`isEmpty()`	컬렉션이 비어 있는지 유무 반환
	`Iterator<E> iterator()`	저장된 객체를 한 번씩 가져오는 반복자 반환
	`int size()`	저장되어 있는 전체 객체 수를 반환
객체 삭제	`void clear()`	저장된 모든 객체 삭제
	`boolean remove(Object o)`	주어진 객체를 삭제

컬렉션 자료구조 - Set 컬렉션 (HashSet)

- Set 컬렉션 중에서 가장 많이 사용되는 것이 HashSet이다.

```
Set<E> set = new HashSet<E>(); // E에 지정된 타입의 객체만 저장
Set<E> set = new HashSet<E>(); // E에 지정된 타입의 객체만 저장
Set set = new HashSet(); // 모든 타입의 객체를 저장
```

- HashSet은 hashCode() 메소드의 리턴값이 같고, equals() 메소드가 true를 반환하면 동일한 객체로 판단하고 중복 저장하지 않는다.

- Set 컬렉션은 인덱스로 객체를 가져올 수 없는 대신 객체를 한 개씩 반복해서 가져와야 한다. 여기에는 두 가지 방법이 있다.

```
Set<E> set = new HashSet<>();

for(E e: set) {
    // ...
}
```

```
Set<E> set = new HashSet<>();

Iterator<E> iterator = set.iterator();
while(iterator.hasNext()) {
    E e = iterator.next();
    // ...
}
```

컬렉션 자료구조 - Set 컬렉션 (TreeSet)

- TreeSet은 이진 트리(Binary Tree)를 기반으로 검색 기능을 강화한 Set 컬렉션이다.
- 이진 트리는 여러 개의 노드(node)가 트리 형태로 연결된 구조로,
  루트 노드라고 불리는 하나의 노드에서 시작해 각 노드에 최대 2개의 노드를 연결할 수 있는 구조를 가지고 있다.
- TreeSet에 저장되는 객체는 저장과 동시에 오름차순으로 정렬된다. (낮은 것은 왼쪽 자식 노드에, 높은 것은 오른쪽 자식 노드에 저장)
- 어떤 객체든지 오른차순으로 정렬될 수 있는 것은 아니고, Comparable 인터페이스를 구현하고 있는 객체만이 정렬 가능하다.
- Comparable 인터페이스와 Comparator 인터페이스는 뒤에서 알아보도록 한다.

```
TreeSet<E> treeSet = new TreeSet<E>();
TreeSet<E> treeSet = new TreeSet<>();
```

- Set 타입 변수에 대입해도 되지만, TreeSet 타입으로 대입한 이유는 검색 관련 메소드가 TreeSet에만 정의되어 있기 때문이다.

컬렉션 자료구조 - Map 컬렉션

- Map 컬렉션은 키(key)와 값(value)으로 구성된 엔트리(Entry) 객체를 저장한다.
- 키와 값은 모두 객체이고, 키는 중복 저장할 수 없지만 값은 중복 저장할 수 있다는 특징이 있다.
- 만약 동일한 키로 값을 저장하면, 기존의 값이 새로운 값으로 대치된다.
- Map 컬렉션에는 HashMap, HashTable, LinkedHashMap, Properties, TreeMap 등이 있는데, Map 컬렉션에서 공통적으로 사용 가능한 Map 인터페이스 메소드는 아래와 같다.

기능	메소드	설명
객체 추가	`V put(K key, V value)`	주어진 키와 값을 추가, 저장이 되면 값을 반환
객체 검색	`boolean containsKey(Object key)`	주어진 키가 있는지 여부 반환
	`boolean containsValue(Object value)`	주어진 값이 있는지 여부 반환
	`Set<Map.Entry<K, v>> entrySet()`	키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 Set에 담아서 반환
	`V get(Object key)`	주어진 키의 값을 반환
	`boolean isEmpty()`	컬렉션이 비어있는지 여부 반환
	`Set<K> keyset()`	모든 키를 Set 객체에 담아서 반환
	`int size()`	저장된 키의 총 수를 반환
	`Collection<V> values()`	저장된 모든 값을 Collection에 담아서 반환
객체 삭제	`void clear()`	모든 Map.Entry(키와 값)을 삭제
	`V remove(Object key)`	주어진 키와 일치하는 Map.Entry 삭제. 삭제가 되면 값을 반환

컬렉션 자료구조 - Map 컬렉션 (HashMap)

- HashMap은 키로 사용할 객체가 hashCode() 메소드의 리턴값이 같고, equals() 메소드가 true를 반환하는 경우, 동일 키로 간주하고 중복 저장을 허용하지 않는다.

```
Map<K, V> map = new HashMap<K, V>();
Map<K, V> map = new HashMap<>();
```

- K, V는 타입 파라미터로 K에는 key의 타입, V에는 value의 타입을 기술한다.
- key의 타입과 value의 타입을 작성하지 않고 생성할 수 있지만, 그렇게 생성하는 경우는 거의 없다.

컬렉션 자료구조 - Map 컬렉션 (Hashtable)

- Hashtable 은 HashMap과 동일한 내부 구조를 가지고 있으며,
  동기화된 메소드로 구성되어 있어서 멀티 스레드가 동시에 Hashtable 메소드를 실행할 수 없다는 점에서 차이가 있다.
- 즉, 멀티 스레드 환경에서도 안전하게 객체를 추가, 삭제할 수 있다.

```
Map<K, V> map = new Hashtable<K, V>();
Map<K, V> map = new Hashtable<>();
```

컬렉션 자료구조 - Map 컬렉션 (Properties)

- Properties는 Hashtable의 자식 클래스이기 때문에 Hashtable의 특징을 그대로 가지고 있다.
- Properties는 키와 값을 String 타입으로 제한한 컬렉션으로, 주로 확장자가 .properties인 프로퍼티 파일을 읽을 때 사용한다.
- 프로퍼티 파일은 키와 값이 = 기호로 연결되어 있는 텍스트 파일로,
  일반 텍스트 파일과 다르게 ISO 8859-1 문자셋으로 저장되며, 한글인 경우에는 \u+유니코드로 표현되어 저장된다.
- Properties를 사용하면 쉽게 프로퍼티의 파일 내용을 코드에서 읽어올 수 있다.
- 먼저 Properties 객체를 생성하고, load() 메소드로 프로퍼티 파일의 내용을 메모리로 로드한다.

```
Properties properties = new Properties();
properties.load(Xxx.class.getResourceAsStream("파일명.properties"));
```

- 일반적으로 프로퍼티 파일은 클래스 파일들과 함께 저장된다. 따라서 클래스 파일을 기준으로 상대 경로로 읽는 것이 편리하다.

컬렉션 자료구조 - Map 컬렉션 (TreeMap)

- TreeMap은 이진 트리(Binary Tree)를 기반으로 검색 기능을 강화한 Map 컬렉션이다.
- TreeSet과의 차이점은 키와 값이 저장된 Entry를 저장한다는 점이다.
- TreeMap에 저장되는 객체는 저장과 동시에 오름차순으로 정렬된다. (키를 기준으로 낮은 것은 왼쪽 자식 노드에, 높은 것은 오른쪽 자식 노드에 저장)
- 어떤 객체든지 오른차순으로 정렬될 수 있는 것은 아니고, Comparable 인터페이스를 구현하고 있는 객체만이 정렬 가능하다.
- Comparable 인터페이스와 Comparator 인터페이스는 뒤에서 알아보도록 한다.

```
TreeMap<K, V> treeMap = new TreeMap<K, V>();
TreeMap<K, V> treeMap = new TreeMap<>();
```

- Map 타입 변수에 대입해도 되지만, TreeMap 타입으로 대입한 이유는 검색 관련 메소드가 TreeMap에만 정의되어 있기 때문이다.

컬렉션 자료구조 - Comparable과 Comparator

- TreeSet에 저장되는 객체와 TreeMap에 저장되는 키 객체는 저장과 동시에 오름차순으로 정렬된다.
- 어떤 객체든 정렬될 수 있는 것은 아니고 객체가 Comparable 인터페이스를 구현하고 있어야 가능하다.
- Integer, Double, String 타입은 모두 Comparable을 구현하고 있기 때문에 상관 없지만,
  사용자가 직접 정의한 객체를 저장할 때에는 반드시 Comparable을 구현하고 있어야 한다.

- Comparable 인터페이스에는 compareTo() 메소드가 정의되어 있다. 따라서 사용자 정의 클래스에서 이 메소드를 재정의해서 비교 결과를 정수값으로 반환해야 한다.

반환 타입	메소드	설명
int	compareTo(T o)	주어진 객체와 같으면 0을 반환 주어진 객체보다 작으면 음수를 반환 주어진 객체보다 크면 양수를 반환

컬렉션 자료구조 - Comparable과 Comparator

- Comparable 인터페이스가 구현되어 있지 않은 객체를 TreeSet에 저장하거나 TreeMap의 키로 저장하기 위해서는 TreeSet 또는 TreeMap을 생성할 때 비교자(Comparator)를 제공해야 한다.
- 비교자는 Comparator 인터페이스를 구현한 객체를 의미한다.
- Comparator 인터페이스에는 compare() 메소드가 정의되어 있고, 이를 재정의해서 비교 결과를 정수값으로 반환하면 된다.

반환 타입	메소드	설명
int	compare(T o1, T o2)	o1, o2가 동등하면 0반환 o1을 앞에 오게하려면 음수를 반환 o1을 뒤에 오게하려면 양수를 반환

컬렉션 자료구조 - LIFO와 FIFO 컬렉션

- 후입선출(LIFO : Last In First Out)은 나중에 넣은 객체가 먼저 빠져나가고,
  선입선출(FIFO : First In First Out)은 먼저 넣은 객체가 먼저 빠져나가는 구조를 말한다.

- 컬렉션 프레임워크는 LIFO 자료구조를 제공하는 스택(Stack) 클래스와 FIFO 자료구조를 제공하는 큐(Queue) 인터페이스를 제공하고 있다.

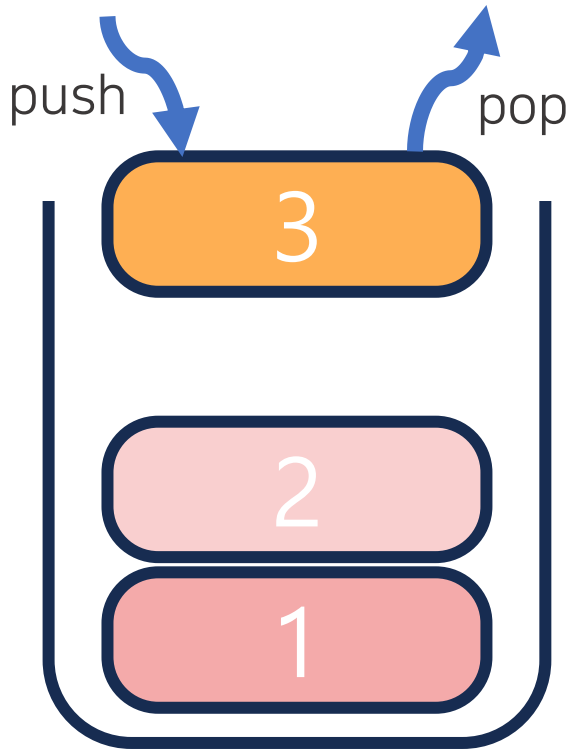
컬렉션 자료구조 - LIFO와 FIFO 컬렉션 (스택)

- Stack 클래스는 LIFO 자료구조를 구현한 클래스로 아래와 같이 생성한다.

```
Stack<E> stack = new Stack<E>();
Stack<E> stack = new Stack<>();
```

- Stack 클래스가 가진 주요 메소드는 아래와 같다.

반환 타입	메소드	설명
E	push(E item)	주어진 객체를 스택에 넣는다.
E	pop()	스택의 맨 위 객체를 빼낸다.



컬렉션 자료구조 - LIFO와 FIFO 컬렉션 (큐)

- Queue 인터페이스는 FIFO 자료구조에서사용되는 메소드를 정의하고 있다.

반환 타입	메소드	설명
boolean	offer(E e)	주어진객체를 큐에 넣는다.
E	poll()	큐에서 객체를 빼낸다.

- Queue 인터페이스를 구현한 대표적인 클래스는 LinkedList이다.
  따라서 LinkedList 객체는 Queue 인터페이스 변수에 아래와 같이 대입할 수 있다.

```
Queue<E> queue = new LinkedList<E>();
Queue<E> queue = new LinkedList<>();
```

컬렉션 자료구조 - 동기화된 컬렉션

- 컬렉션 프레임워크의 대부분의 클래스들은 싱글 스레드 환경에서 사용할 수 있도록 설계되었다.
- 그렇게 때문에 여러 스레드가 동시에 컬렉션에 접근하면 의도하지 않게 요소가 변경될 수 있는 불안전한 상태가 된다.

- Vector와 HashTable은 동기화된(Synchronized) 메소드로 구성되어 있기 때문에 멀티 스레드 환경에서 안전하게 요소를 처리할 수 있지만, 다른 것들은 동기화된 메소드로 구성되어 있지 않아 멀티 스레드 환경에서 안전하지 않다.

- 만약 멀티 스레드 환경에서 사용하고자 한다면, 비동기화된 메소드를 동기화된 메소드로 래핑하는 Collections의 synchronizedXXX() 메소드를 사용하면 된다.

리턴 타입	메소드	설명
List<T>	synchronizedList(List<T> l)	List를 동기화된 List로 반환
Map<K, V>	synchronizedMap(Map<K, V> m)	Map을 동기화된 Map으로 반환
Set<T>	synchronizedSet(Set<T> s)	Set을 동기화된 Set으로 반환

컬렉션 자료구조 - 수정할 수 없는 컬렉션

- 요소를 추가, 삭제할 수 없는 컬렉션은 수정할 수 없는 (unmodifable) 컬렉션이라 한다.
- 컬렉션 생성 시 저장된 요소를 변경하고 싶지 않을 때 사용할 수 있다.

- 수정할 수 없는 컬렉션을 만드는 방법에는 세 가지 방법이 있다.
  1. List, Set, Map 인터페이스의 정적 메소드인 of() 로 생성

```
List<E> immutableList = List.of(E element1, …);
Set<E> immutableSet = Set.of(E element2, …);
Map<E> immutableMap = Map.of(K k1, V v1, …);
```

  2. List, Set, Map 인터페이스의 정적 메소드인 copyOf()로 생성

```
List<E> immutableList = List.copyOf(Collection<E> coll);
Set<E> immutableSet = Set.of(Collection<E> coll);
Map<E> immutableMap = Map.of(Map<K, V> map);
```

  3. 배열로부터 수정할 수 없는 List 컬렉션을 생성

```
String arr = { … };
List<String> immutableList = Arrays.asList(arr);
```

람다식

- 람다식은 쉽 게 말해 메서드를 "하나의 식"으로 표현한 것이다. 하나의 식이기 때문에 훨씬 간략하게 표현이 가능하다.
- 또한 메서드의 이름과 반환값이 없어지므로 "익명함수"라고도 한다.

- Java는 함수형 프로그래밍을 위해 Java 8 부터 람다식(Lambda Expressions)을 지원한다.

TIP

함수형 프로그래밍(Functional Programming) : 함수를 정의하고, 해당 함수를 데이터 처리부로 보내 데이터를 처리하는 기법으로 데이터 처리부에서는 데이터만 가지고 있을 뿐, 데이터의 처리 방법이 정해져 있지 않기 때문에 외부에서 제공된 함수에 의존한다.

- 람다식은 데이터 처리부에 제공되는 함수 역할을 하는 매개변수를 가진 중괄호 블록으로, 익명 구현 객체로 변환하여 동작한다.
- 익명 구현 객체를 람다식으로 표현하기 위해서는 인터페이스가 단 하나의 추상 메소드만을 가지고 있어야 한다.
- 단 하나의 추상 메소드를 가지고 있는 인터페이스를 함수형 인터페이스라고 부르고, @FunctionalInterface 어노테이션을 붙인다.

- 람다식에서 매개변수가 하나인 경우에는 소괄호 생략이 가능하고, 실행문이 한 줄인 경우에는 중괄호 생략이 가능하다.

람다식 - 익명 구현 객체 생성

```
package com.lambda;

@FunctionalInterface
public interface MyInterFace {
    void action(int x);
}


public class LambdaExample {
    public static void main(String[] args) {
        MyInterFace normal = new MyInterFace() {
            @Override
            public void action(int x) {
                System.out.println(x + "을 활용한 동작 진행");
            }
        };
        normal.action(3);
    }
}
```

TIP

어노테이션은 선택사항이지만, 컴파일 과정에서 검사를 할 수 있도록 해준다.

람다식 - 익명 구현 객체 생성 [Lambda]

```java
package com.lambda;

@FunctionalInterface
public interface MyInterFace {
    void action(int x);
}


public class LambdaExample2 {
    public static void main(String[] args) {
        MyInterFace lambda = (x) -> {
            System.out.println(x + "을 활용한 동작 진행");
        };
        lambda.action(3);
    }
}
```

TIP

어노테이션은 선택사항이지만, 컴파일 과정에서 검사를 할 수 있도록 해준다.

람다식 - 익명 구현 객체 사용 (인터페이스 타입의 매개변수에 대입)

```
public class LambdaExample3 {
    public static void main(String[] args) {
        action(new MyInterFace() {
            @Override
            public void action(int x) {
                System.out.println(x + "을 활용한 동작 진행");
            }
        });
    }
    public static void action(MyInterFace mi) {
        int x = 3;
        mi.action(x);
    }
}
```

람다식 - 익명 구현 객체 사용 (인터페이스 타입의 매개변수에 대입)

```
public class LambdaExample4 {
    public static void main(String[] args) {
        action((x) -> {
            System.out.println(x + "을 활용한 동작 진행");
        });
    }

    public static void action(MyInterFace mi) {
        int x = 3;
        mi.action(x);
    }
}
```

람다식

- 함수형 인터페이스의 추상 메소드가 매개변수를 가지고 있지 않은 경우, 람다식은 아래와 같이 작성할 수 있다.
- 실행문이 두 개 이상인 경우에는 중괄호를 생략할 수 없고, 하나일 경우는 생략 가능하다.

```
( ) -> {
    실행문;
    실행문;
}
```

```
( ) -> {
    실행문;
}
```

```
( ) -> 실행문;
```

람다식

- 함수형 인터페이스의 추상 메소드에 매개변수가 있을 경우, 람다식은 아래와 같이 작성할 수 있다
- 매개변수를 선언할 때, 타입을 생략하거나, 구체적인 타입 대신에 var를 사용할 수도 있다. (타입을 생략하는 것이 일반적이다)
- 매개변수가 하나일 경우에 괄호를 생략할 수 있다. 이때는 타입 또는 var를 붙일 수 없다.

```
(타입 매개변수, …) -> {
    실행문;
    실행문;
}
```

```
(var 매개변수, …) -> {
    실행문;
    실행문;
}
```

```
(매개변수, …) -> {
    실행문;
    실행문;
}
```

```
(타입 매개변수, …) -> 실행문;
```

```
(var 매개변수, …) -> 실행문;
```

```
(매개변수, …) -> 실행문;
```

```
매개변수 -> 실행문;
```

```
매개변수 -> {
    실행문;
    실행문;
}
```

람다식

- 함수형 인터페이스의 추상 메소드에 리턴값이 있을 경우, 람다식은 아래와 같이 작성 가능하다.

```
(매개변수, …) -> {
    실행문;
    return 값;
}
```

- return문 하나만 존재하는 경우에는 중괄호와 함께 return 키워드를 생략할 수 있다.

```
(매개변수, …) -> 값;
```

람다식 - 메소드 참조

- 메소드 참조(Method Reference)는 자바 8에서 도입된 기능으로, 기존 메소드나 생성자를 참조하여 람다 표현식을 단축할 수 있다.
  이를 통해 코드의 가독성을 향상시키고, 불필요한 매개변수를 제거할 수 있다.
- 람다식은 매개변수를 받아 값을 전달하는 역할을 할 수 있지만,
  메소드 참조를 이용하면 불필요한 매개변수를 제외하고 더 간결하게 작성할 수 있다.

```
(x, y) -> 클래스.정적메소드(x, y);
```

```
클래스 :: 정적메소드
```

```
(x, y) -> 참조변수.메소드(x, y);
```

```
참조변수 :: 메소드
```

람다식 - 매개변수의 메소드 참조

- 람다식에서 제공되는 두 개의 매개변수(x, y)에서 x의 메소드를 호출해서 y를 매개값으로 사용하는 경우, 아래와 같이 사용할 수 있다.

```
(x, y) -> x.메소드(y);
```

```
x의 클래스 :: 메소드
```

- 작성 방법은 정적 메소드 참조와 동일하지만, 매개변수의 인스턴스 메소드가 사용된다는 점에서 차이가 있다.

람다식 - 생성자 참조

- 생성자를 참조한다는 것은 객체를 생성한다는 것을 의미한다.
- 람다식이 단순히 객체를 생성하고 반환하도록 구성된다면, 람다식을 생성자 참조로 대체할 수 있게 된다.

```
(x, y) -> { return new 클래스(x); }
```

```
클래스 :: new
```

- 생성자가 오버로딩되어 여러 개가 있을 경우,
  컴파일러는 함수형 인터페이스의 추상 메소드와 동일한 매개변수 타입과 개수를 가지고 있는 생성자를 찾아 실행한다.