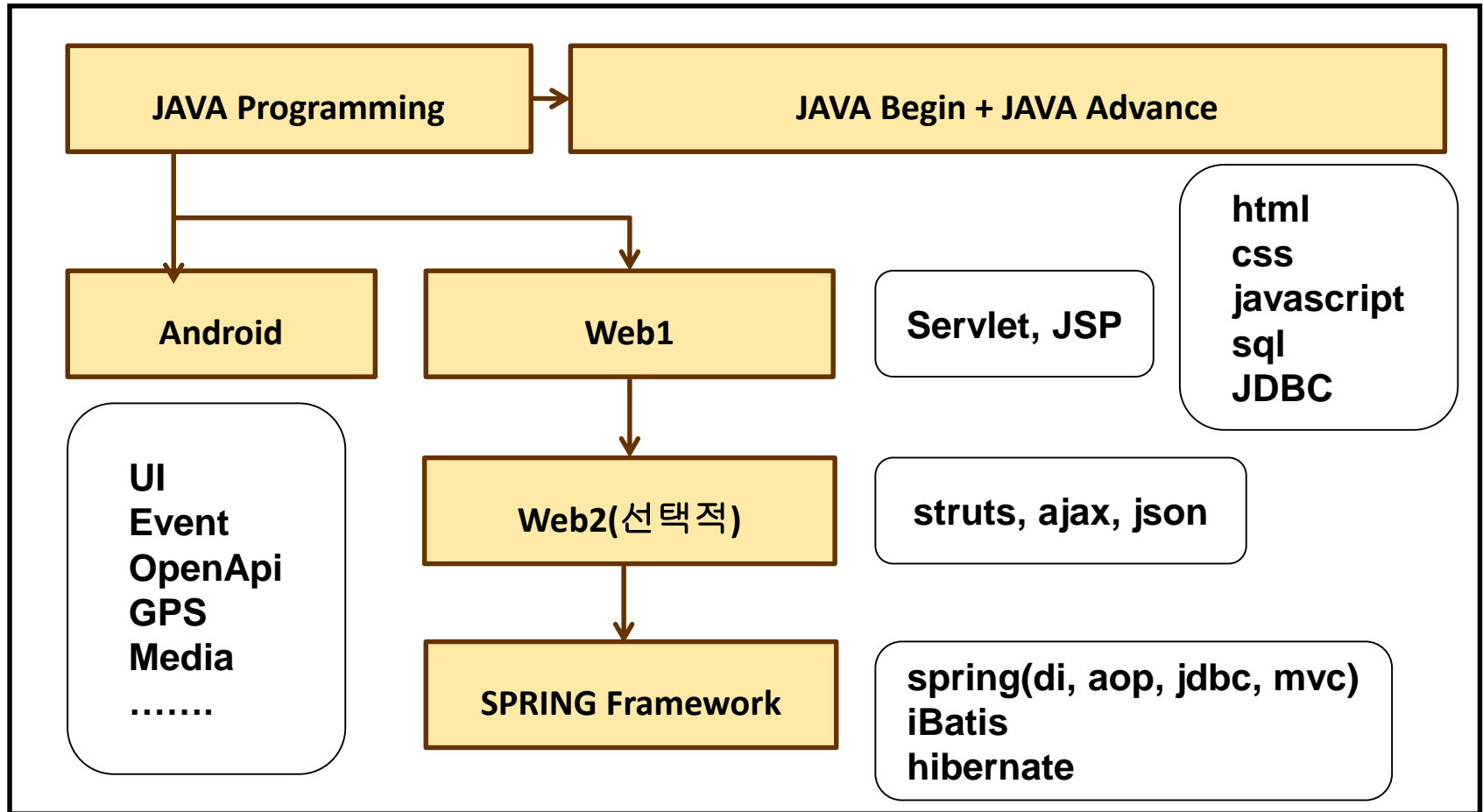


# 자바 기초

# 과정 로드맵





# 수업을 하기 위한 준비

0 절. 디렉토리 만들기

1 절. 필요파일 다운로드

- JDK : 1.8버전

- 편집기 : 이클립스 4.8 Photon

### - 수업 디렉토리 만들기

**d:\ java-begin** : 수업 동안 사용될 루트 폴더

- **bin** : 설치 경로 (**Eclipse**)
- **workspace** : 작업 소스 폴더
- **setup** : 수업에 필요한 설치파일

## - 다운로드 링크

<http://www.oracle.com>

## - 자바 다운로드 선택

The screenshot shows the Oracle website's 'Downloads' section. The 'Oracle' logo is at the top left. Navigation links include 'Sign In/Register for Account', 'Help', 'Select Country/Region', 'Communities', 'I am a...', and 'I want to...'. A search bar is on the right. The main navigation bar includes 'Products and Services', 'Solutions', 'Downloads', 'Store', 'Support', 'Training', 'Partners', and 'About'. The 'Downloads' section is expanded, showing various categories: 'Database', 'Enterprise Management', 'Developer Tools', and 'Applications'. Under 'Database', 'Java for Developers' is highlighted with a red box, and the Korean text '클릭' (Click) is written next to it. Other categories include 'Server and Storage Systems', 'Prebuilt Developer VMs', and 'Applications'.

## JDK 다운로드 링크 선택

**Java SE Development Kit 7u5**

You must accept the [Oracle Binary Code License Agreement for Java SE](#) to download this software.

**2** ☐ **Accept License Agreement** **클릭**

Product / File Description	File Size	Download
Here are the Java SE downloads in detail:		
<b>Java Platform, Standard Edition</b>		
<b>Java SE 7u5</b> This release includes security enhancements and bug fixes. <a href="#">Learn more</a>	<b>JDK</b> <b>DOWNLOAD</b>	<b>JRE</b> <b>DOWNLOAD</b>
"What Java Do I Need?" You must have a copy of	JDK 7 Docs	JRE 7 Docs
Solaris SPARC 64-bit	12.55 MB	<a href="#">jdk-7u5-linux-i586.rpm</a>
Solaris x64	14.39 MB	<a href="#">jdk-7u5-linux-i586.tar.gz</a>
Solaris x64	9.54 MB	<a href="#">jdk-7u5-linux-x64.rpm</a>
Windows x86	87.9 MB	<a href="#">jdk-7u5-linux-x64.tar.gz</a>
Windows x64	92.3 MB	<a href="#">jdk-7u5-macosx-x64.dmg</a>
		<a href="#">jdk-7u5-solaris-i586.tar.Z</a>
		<a href="#">jdk-7u5-solaris-i586.tar.gz</a>
		<a href="#">jdk-7u5-solaris-sparc.tar.Z</a>
		<a href="#">jdk-7u5-solaris-sparc.tar.gz</a>
		<a href="#">jdk-7u5-solaris-sparcv9.tar.Z</a>
		<a href="#">jdk-7u5-solaris-sparcv9.tar.gz</a>
		<a href="#">jdk-7u5-solaris-x64.tar.Z</a>
		<a href="#">jdk-7u5-solaris-x64.tar.gz</a>
		<a href="#">jdk-7u5-windows-i586.exe</a>
		<a href="#">jdk-7u5-windows-x64.exe</a>

**1** **JDK** **클릭**

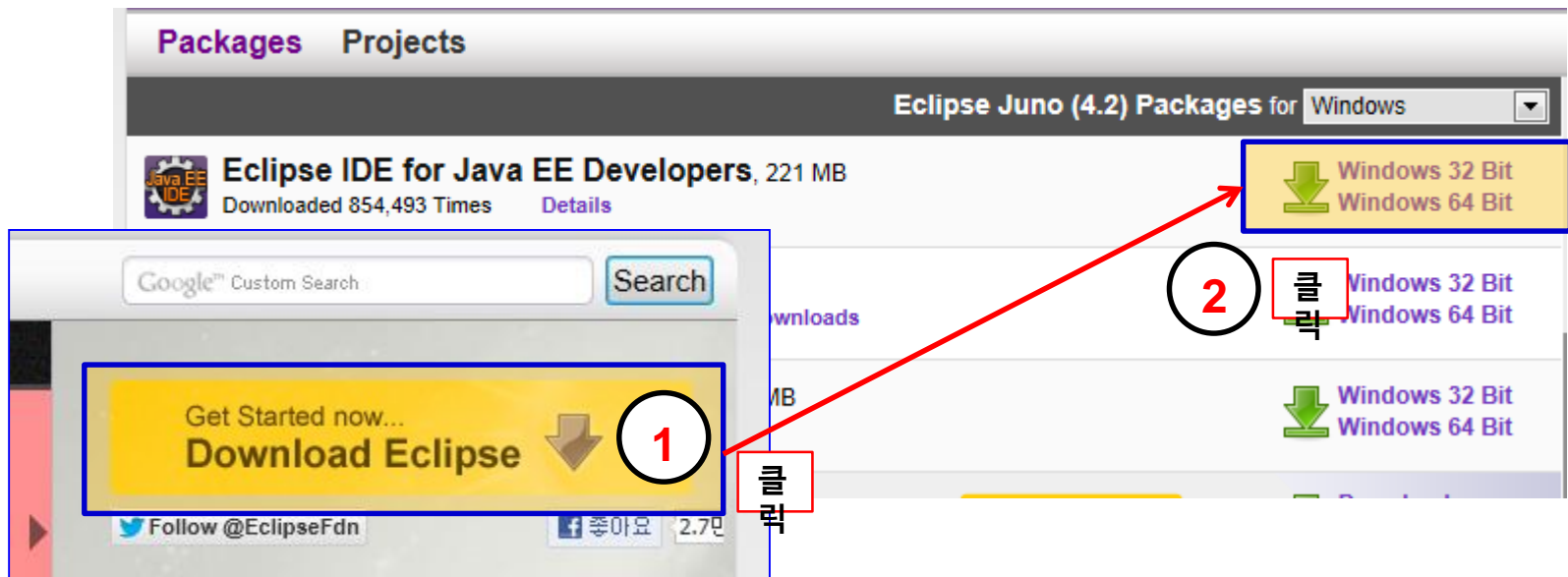
**3** **클릭**

- Windows x86 → 32bit, Windows x64 → 64bit 일 경우 선택

- 다운로드 링크

<http://eclipse.org>

- Download Eclipse 버튼 클릭



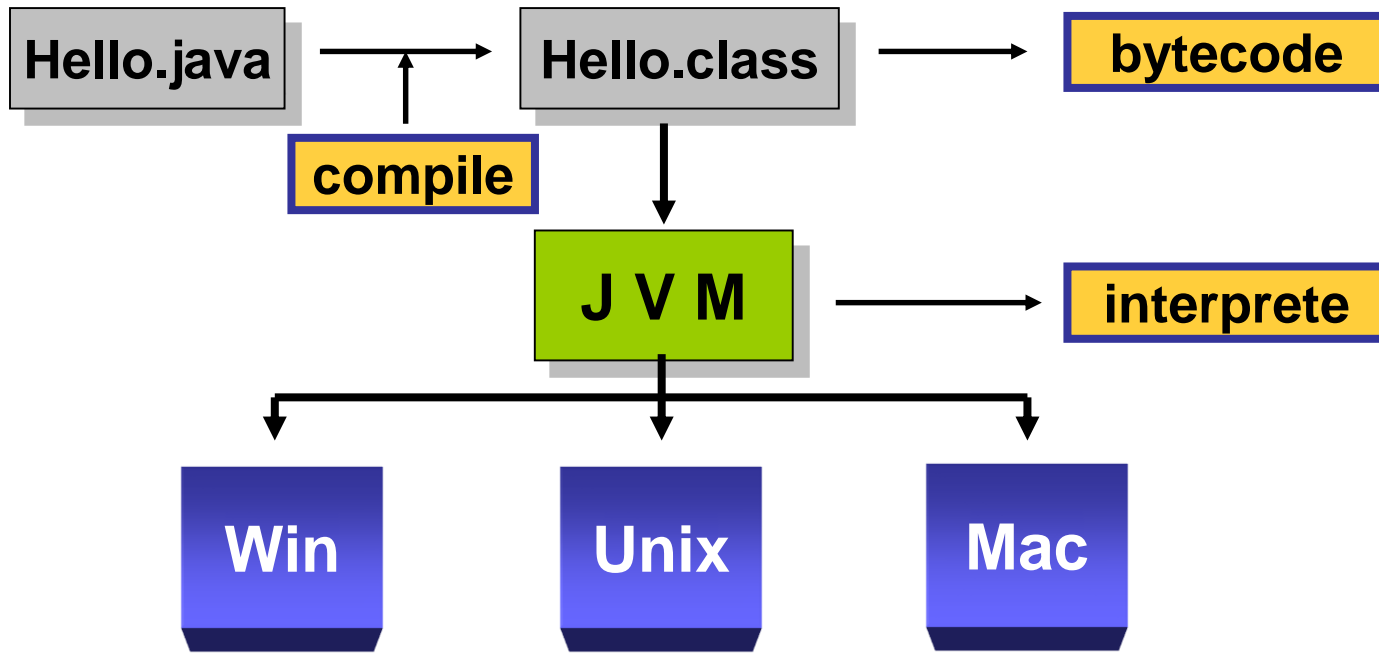


# 설치 및 실행



# 1. 자바언어의 특징

- 단순(Simple) : 메모리 관리(Garbage Collection)
- 객체지향적(Object-Oriented) : 재사용
- 컴파일 + 인터프리터 : OS에 독립적, WORA





## 2. 자바 환경 설정 및 설치

- Java 분야
  - J2SE
  - J2ME
  - J2EE
- Java 다운로드 : <http://www.oracle.com>
- JDK 설치 확인 : `java -version`
- 환경설정
  - JAVA\_HOME 설정
  - PATH 설정

## 2. 자바 환경 설정 및 설치

예제작성

- 편집기 이클립스 설치
  - 다운로드 : <http://eclipse.org/>

### 3. 메인 메소드

- 실행 명령인 java 를 실행 시 가장 먼저 호출 되는 부분
- 만약, Application 에서 main() 메소드가 없다면 절대로 실행 될 수 없음
- Application의 시작 = 특정 클래스의 main( ) 실행
- 형태 (고정된 형태)

```
public static void main(String [ ] args) { }
```

## 4. 출력문

예제작성

- print

- println

- printf

- %d : 정수

- %f : 실수

- %c : 문자

- %s : 문자열

```
System.out.println( "%s");
```

??

```
System.out.printf( "오늘은 날씨가 %s");
```

??

```
System.out.printf( "오늘은 날씨가 %s", "화창하다");
```

??

```
System.out.printf( "오늘 점심은 %d시에 %s" , 12, "강남식당");
```

??

```
System.out.printf( "오늘의 원달러 환율은 %f 원" , 1019.8);
```

??

```
System.out.printf( "오늘의 원달러 환율은 %7.1f 원" , 1019.8);
```

??

## 5. 클래스의 구조

예제작성

- 클래스 선언부에 올 수 있는 것
  - 주석문
  - 패키지
  - 임포트
  - 클래스 선언
- 클래스 내용부에 올 수 있는 것
  - 멤버변수
  - 메소드
  - 내부클래스



# 식별자, 자료형, 연산자

- 클래스, 메소드, 변수의 이름
- 명명규칙
  1. 클래스 : 단어의 첫 글자를 대문자로 표기. 만약, 여러 개의 단어로 이루어져 있다면 각 단어의 첫글자를 대문자로 표기.  
예> Hello, HelloWorld, BoardMng
  2. 멤버변수, 메소드 : 단어의 첫 글자를 소문자로 표기. 만약, 여러 개의 단어로 이루어져 있다면 각 단어의 첫글자를 대문자로 표기  
예> name, cnt, main(), print(), juminNo, printName()
  3. 상수 : 모든 단어를 대문자로 표기. 만약, 여러 개의 단어로 이루어져 있다면 단어와 단어 사이를 '\_'로 구분한다.  
예> MAX, MIN, MAX\_VALUE, SERVER\_NAME





## 2. 변수

- 데이터를 저장할 메모리의 위치를 나타내는 이름
- 메모리 상에 데이터를 보관할 수 있는 공간을 확보
- 적절한 메모리 공간을 확보하기 위해서 변수의 타입 등장
- '=' 를 통해서 CPU에게 연산작업을 의뢰

## 2. 변수

메모리의 단위

- 0과 1을 표현하는 bit
- 8bit = 1byte
- 2bytes = word

bit의 데이터 처리 - 2진수

- 전구의 불이 들어오고 나가는 처리

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
128	64	32	16	8	4	2	1
0	0	0	0	0	1	1	1
					4 +	2 +	1
							결과 7

## 2. 변수

- 변수

- 선언

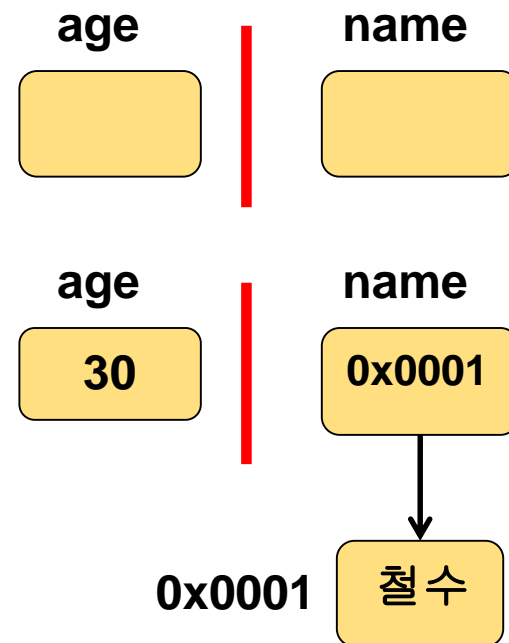
1. 자료형 변수명;
2. 예> `int age; String name; ...`

- 초기화

1. 변수명 = 저장할 값;
2. 예> `age = 30; name = "철수";`

- 선언과 초기화를 동시에

1. 자료형 변수명 = 저장할 값;
2. 예> `int age = 30;`



## 2. 자료형

- 자료형

- 기본 자료형과 참조 자료형(기본 자료형 8가지 외 모든 것)
- 기본 자료형 (맨 앞의 비트는 부호비트)

타입	세부타입	데이터형	크기	기본값	사용예
논리형		<b>boolean</b>	1bit	false	boolean b = true
문자형		<b>char</b>	2byte	null(\u0000)	char c = 'a', c1 = 65, c2 = '\uffff'
숫자형	정수형	<b>byte</b>	1byte	(byte)0	byte b = 100;
		<b>short</b>	2byte	(short)0	short s = 100;
		<b>int</b>	4byte	0	int i = 100;
		<b>long</b>	8byte	0L	long l = 100, l2 = 100L
	실수형	<b>float</b>	4byte	0.0f	float f = 3.1f, f2 = 3.1F;
		<b>double</b>	8byte	0.0d	double d = 3.1;

## 2. 형변환

예제작성

- 자료형의 크기 비교

byte < short < int < long < float < double

char < int < long < float < double

- 데이터 형변환

1) 묵시적(암묵적) : Implicit Casting

: 범위가 넓은 데이터 형에 좁은 데이터 형을 대입하는 것

: 예> byte b = 100; int i = b;

2) 명시적 : Explicit Casting

: 범위가 좁은 데이터 형에 넓은 데이터 형을 대입하는 것

: 형변환 연산자 사용 - (타입) 값;

: 예> int i = 100; byte b = i; (X), byte b = (byte) i; (O)

## 2. 자료형

예제작성

- 기본 자료형과 참조 자료형의 차이

	기본 자료형	참조 자료형
변수값	실제 사용할 값	객체 참조값
정의방식	Java 내부 이미 정의	클래스, 인터페이스, 배열, enum등..
생성방식	19, 3.14, true, 'a'	new 키워드 활용
초기화방식	default	생성자

```
int a = 100;
```

a

100

```
String s = new String("100");
```

s

0x0001

0x0001

"100"

## 2. 자료형

예제작성

### 기본 자료형 에서 꼭 기억해야 할 것들

모든 변수는 데이터를 담는 공간(상자)이다.  
기본 자료형의 타입은 무조건 소문자로 시작한다.  
상자의 타입은 담을 수 있는 데이터의 종류를 말한다.  
각 타입마다 상자의 크기가 있다.

### 변수 선언시의 주의 점

적절한 변수의 타입을 써라.  
변수의 이름은 결국 메모리상에서 찾아가는 이름 - 충돌 조심  
상자를 최초로 만들 때에만 변수의 타입이 쓰인다.  
이클립스는 미리 프로파일링을 통해서 불필요한 변수 선언을 처리

## 2. 변수, 상수, 자료형

예제작성

- 상수

- 변경될 수 없는 고정된 데이터
- 코드의 이해와 변경이 쉬움
- 분산된 상수로 인한 에러를 방지
- **final** 키워드를 이용해서 정의

예> `int age;` -> 변수, **final** `double PI = 3.14` -> 상수

- 문자열 상수(이스케이프 문자)

문자상수	내용	문자상수	내용
<code>\n</code>	줄넘김	<code>\"</code>	“ 표시
<code>\t</code>	탭만큼 띄우기	<code>\'</code>	‘ 표시
<code>\\</code>	\ 화면에 표시		



# 3. 연산자

예제작성

## - 3항 연산자

### 형식

조건식 ? 수식-1 : 수식-2;

수식-1 : 조건식의 결과가 참(true) 일 때 수행되는 식

수식-2 : 조건식의 결과가 거짓(false) 일 때 수행되는 식

```
int a = 10;
```

```
int b = 5;
```

```
int max = ( a > b ) ? a : b ;
```

max 값은 ??

### 3. 연산자

예제작성

- 산술 연산자

연산자	사용법	설 명
+, -, *		
/	op1 / op2	op1을 op2로 나눈 몫을 구한다.
%	op1 % op2	op1을 op2로 나눈 나머지를 구한다.

정수와 정수의 연산 = 정수  
정수와 실수의 연산 = 실수

### 3. 연산자

예제작성

#### - 증감 연산자

연산자	사용법	설명	사용예
++	++op (선행처리)	1 증가	int a = 5;    int b = a++; ??
	op++ (후행처리)		int a = 5;    int b = ++a; ??
--	--op (선행처리)	1 감소	int a = 5;    int b = a--; ??
	op-- (후행처리)		int a = 5;    int b = --a; ??

```
int a = 5;
System.out.println(a++);
System.out.println(++a);
System.out.println(--a);
System.out.println(a  );
System.out.println(a--);
System.out.println(a++);
```

### 3. 연산자

예제작성

- 비교 연산자 : 결과값으로 참(true), 거짓(false)이 반환

연산자	사용법	설명	사용예
>, >=, <, <=			
==	op1 == op2	서로 같은 경우	boolean b = ( (10 % 2) == 0 )
!=	op1 != op2	서로 같지 않은 경우	boolean b = ( (10 % 2) != 0 )
instanceof	op1 instanceof op2	객체의 타입을 비교	

가장 많이 사용되는 것  
== , !=

### 3. 연산자

예제작성

- 조건 연산자 : 결과값으로 참(true), 거짓(false)이 반환

연산자	사용법	설명
&&	A && B	A와 B가 참일 경우만 참 반환 (A가 거짓일 경우 B는 실행하지 않는다)
	A    B	A 또는 B 둘 중에 하나가 참일 경우 참 반환 (A가 참일 경우 B는 실행하지 않는다)
!	! A	A가 참이면 거짓, A가 거짓이면 참을 반환

### 3. 연산자

예제작성

- 배정연산자

연산자	사용법	설명
+=	op1 +=op2	op1 = op1+op2
-=	op1 -=op2	op1 = op1-op2
*=	op1 *=op2	op1 = op1*op2
/=	op1 /=op2	op1 = op1/op2



## 4. 조건문

예제작성

- if 문
- switch 문

## 4. 조건문

예제작성

- 단일 if

```
- if ( 조건식 )
    실행문장;
- if ( 조건식 )
    실행문장;
else
    실행문장;
- if ( 조건식 ) {
    .....
} else {
    .....
}
```

주의사항

- 실행문장이 복수일 때에는 블록으로 처리
- 조건식 자리에는 반드시 참과 거짓을 구분해야 한다.

```
int a = 3;
if(a = 3) => 에러 발생
if( 0 )   => 에러 발생
```

- **else** 절은 필요에 따라 기술한다.



## 4. 조건문

예제작성

- 다중 if

```
형식 - if (조건식) {  
    실행문장;  
} else if (조건식) {  
    실행문장;  
} else if (조건식) {  
    실행문장;  
} else {  
    실행문장;  
}
```

## 4. 조건문

예제작성

- 내포된 if

형식 - if (조건식) {  
    if (조건식)  
        .....  
    if (조건식)  
        .....  
    else  
        .....  
}

```
int a = 10, b = 5;  
if ( a > b ) {  
    if ( a == 10 )  
        System.out.println("a = 10");  
        System.out.println("a가 b보다 크다");  
    else  
        System.out.println("b가 a보다 크다");  
}
```

## 4. 조건문

예제작성

- switch

```
switch (수식) {  
    case 값1:  
        처리문장들;  
        break;  
    case 값2:  
        처리문장들;  
        break;  
    case 값n:  
        처리문장들;  
        break;  
    default:  
        묵시적으로 처리해야 하는 문장들;  
}
```

주의사항

1. 수식에 올 수 있는 것
  - 1.4버전 까지  
byte, short, char, int
  - 1.5버전 부터  
enum 클래스 타입
  - 1.7 버전 부터  
String 클래스 타입
2. break문 없이도  
사용이 가능하다.
3. default => else의  
역할과 동일하다.

## 4. 반복문

예제작성

### - for

- 형식     **for(1. 초기값 ; 2. 조건 ; 3. 증감) {**  
                  **4. 반복문장들**  
                  **}**

**5. 반복문 빠져나옴**

### - 실행순서

1 - 2(조건이 참일 경우)     - 4 - 3  
    - 2(조건이 참일 경우)     - 4 - 3  
    - 2(조건이 거짓일 경우) - 5

## 4. 반복문

예제작성

### - while

조건절로 지정된 조건이 참일 동안 **while** 블록을 실행

```
while(조건절) {  
    반복문장들  
}
```

예>

```
int a=10, b=20;
```

```
while(a > b)
```

```
System.out.println("이 문장은 영원히 나타나지 않는다");
```

## 4. 반복문

### - do ~ while

**do-while**문은 조건을 나중에 평가한다  
**while** 블록이 적어도 한번은 수행

```
do {  
    반복문장들  
} while(조건절);
```

ex> int a = 5, b = 10;

```
do {  
    System.out.println("무조건 실행됨");  
} while (a > b)  => 1번 실행됨.
```

**while** 와 **do-while**의 차이점  
- **do-while** 은 무조건  
한번은 실행. **While**은  
실행이 안될 수 도 있다  
다시 말해서  
같은 조건일 경우 **do-while**  
문만 실행 될 수 있다.

### - break

**break**문의 3가지 역할

**switch**문에서 **switch**문을 벗어나는데 사용

반복문 에서 반복루프를 벗어나는데 사용

중첩된 반복문을 한번에 빠져나갈때

### - continue

반복문의 특정지점에서 제어를 반복문의 처음으로 보낸다

## 4. 제어

예제작성

- break

```
int i = 1;
while(i < 100) {
    if(i == 10) break;
    System.out.println(i + "자바의 세계로 오세요! ");
    i++;
}
```

결과는 ??



## 4. 제어

### 예제 작성

#### - break

반복문이 중첩되었을 경우 가장 가까운 반복문을 빠져 나온다

```
int i, j;
for(i=1 ; i<=5 ; i++) {
    for(j=1 ; j<=i ; j++) {
        if (j > 3) break;
        System.out.print(" * ");
    }
    System.out.println();
}
```

i = 1일 때

\*

i = 2일 때

\* \*

i = 3일 때

\* \* \*

i = 4일 때

\* \* \*

i = 5일 때

\* \* \*

결과

\*

\* \*

\* \* \*

\* \* \*

\* \* \*

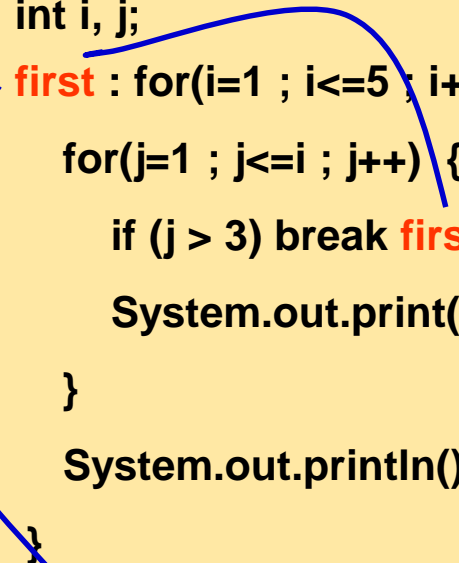
## 4. 제어

예제작성

### - break

중첩된 반복문을 한번에 빠져 나오기

```
int i, j;  
first : for(i=1 ; i<=5 ; i++) {  
    for(j=1 ; j<=i ; j++) {  
        if (j > 3) break first; // first라는 이름의 블록을 벗어난다.  
        System.out.print(" * ");  
    }  
    System.out.println();  
}
```



### - continue

반복문의 특정지점에서 제어를 **반복문의 처음으로** 보낸다

```
예> class ContinueTest {  
    public static void main(String args[]) {  
        for(int i=0; i<10; i++) {  
            if (i%2 == 0) continue;  
            System.out.println(i + " 자바의 세계로 오세요! " );  
        }  
    }  
}
```

결과는??

1 자바의 세계로 오세요!  
3 자바의 세계로 오세요!  
5 자바의 세계로 오세요!  
7 자바의 세계로 오세요!  
9 자바의 세계로 오세요!



배 열

# 배열

## - 배열이란?

나가수경연	임재범	김범수	박정현
1차 경연	40.1%	29.7%	30.2%
2차 경연	30.1%	35.7%	34.2%

**String name = “임재범”;**

**String name2 = “김범수”;**

**String name3 = “박정현”;**

**double first = 40.1;**

**double first2 = 29.7;**

**double first3 = 30.2;**

**double second = 30.1;**

**double second2 = 35.7;**

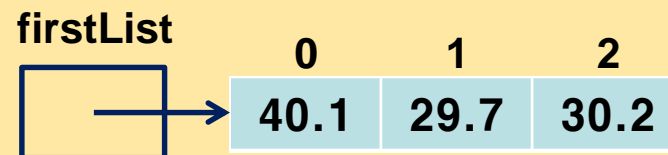
**double second3 = 34.2;**

## - 배열이란?

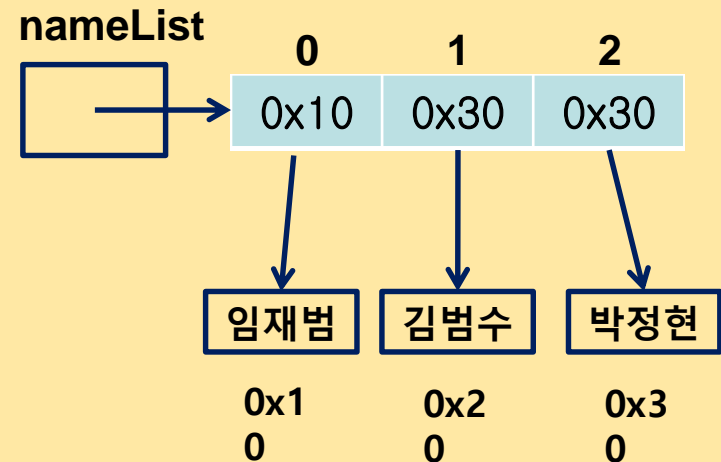
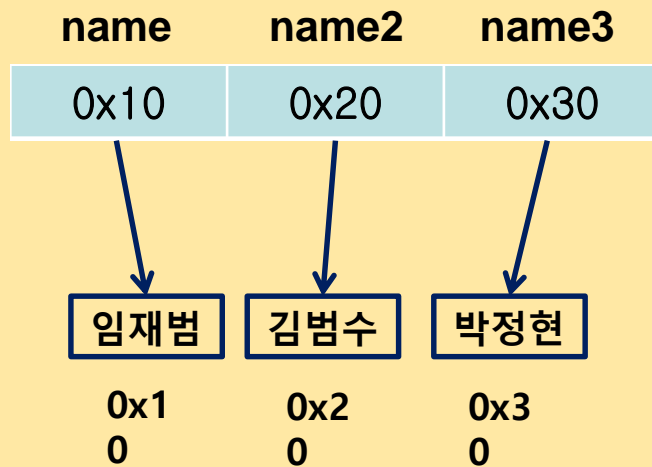
- 같은 종류의 데이터를 저장하기 위한 자료구조
- 크기가 고정되어 있다(한번 생성된 배열은 크기를 바꿀 수 없다)
- 배열을 객체로 취급
- 배열의 요소를 참조하려면 배열이름과 색인(**index**)이라고 하는
- **int** 유형의 정수값을 조합하여 사용한다.

## - 배열

first	first2	first3
40.1	29.7	30.2



## - 배열





## - 배열의 선언

- [ ] 는 자바에서 배열을 나타낸다.
- [ ] 의 개수가 배열의 차원수를 나타낸다.  
예를 들어, [ ] 일 경우 1차원, [ ][ ] 일 경우 2차원이 된다.

### - 1차원 배열 선언

배열유형 배열이름 [ ] 또는 배열유형[] 배열이름

ex) int prime [ ], int [ ] prime

### - 다차원 배열 선언

배열유형 배열이름 [][] 또는 배열유형[][] 배열이름

ex) int prime [ ][ ], int [ ][ ] prime

\* 배열의 유형은 모든 것이 가능하다(기본형, 참조형)

# 배열

## - 배열의 선언

타입	배열이름	선언
int	iArr	<b>int [] iArr;</b>
char	cArr	<b>char [] cArr;</b>
boolean	bArr	<b>boolean [] bArr;</b>
String	strArr	<b>String [] strArr;</b>
Date	dateArr	<b>Date [] dateArr;</b>

## - 배열의 선언

`int [ ] dongList;`

**dongList**



하나의 값을 저장할 수  
있는 메모리 생성

`int [ ][ ] aptInfoList;`

**aptInfoList**



하나의 값을 저장할 수  
있는 메모리 생성

## - 배열의 생성

### - 1차원 배열

배열의 이름 = **new** 배열유형 [배열크기];

ex) **prime = new int [10]**

### - 2차원 배열

배열의 이름 = **new** 배열유형[1차원배열개수][1차원배열의크기];

배열의 이름 = **new** 배열유형[1차원배열개수] **[ ]**;

ex) **prime = new int [3][2];**

**prime = new int [3][ ];**

## - 자동 초기화

- 배열이 생성되면 자동적으로 배열요소는 기본값으로 초기화된다.

ex) int : 0

boolean : false

char : '\u0000'

참조형 : null....

- 멤버변수와 로컬변수 모두 배열이 생성이 되면 자동 초기화된다.

## - 배열의 생성

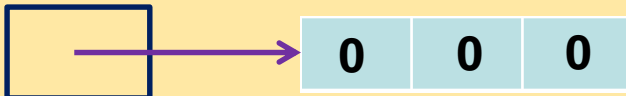
`int [ ] dongList;`

`dongList`



`dongList = new int[3];`

`dongList`



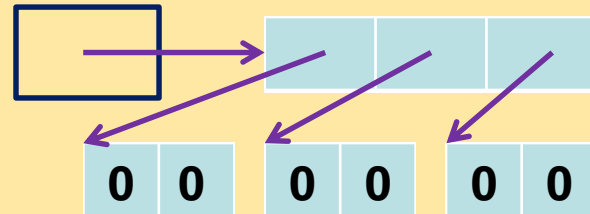
`int [ ][ ] aptInfoList;`

`aptInfoList`



`aptInfoList = new int[3][2];`

`aptInfoList`



## - 배열의 생성

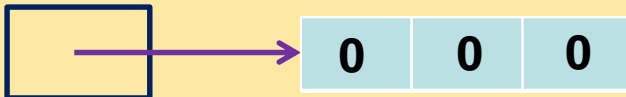
`int [ ] dongList;`

`dongList`



`dongList = new int[3];`

`dongList`



`int [ ][ ] aptInfoList;`

`aptInfoList`



`aptInfoList = new int[3][ ];`

`aptInfoList`



## - 초기화

### - 1차원 배열

배열이름[인덱스] = 값;

ex) `prime[0] = 100;`

### - 2차원 배열

배열이름[인덱스][인덱스] = 값;

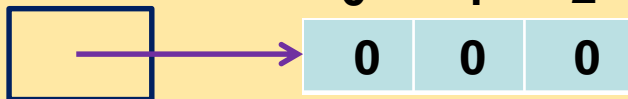
ex) `twoArr[0][1] = 100;`



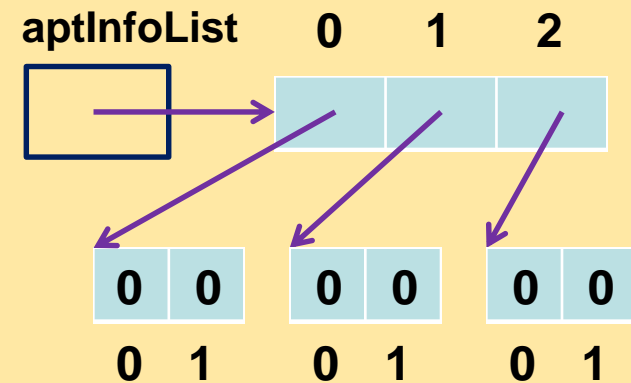
## - 배열의 초기화

```
int [ ] dongList = new int[3];
```

dongList



```
int [ ][ ] aptInfoList =  
    new int[3][2];
```



## - 배열의 초기화

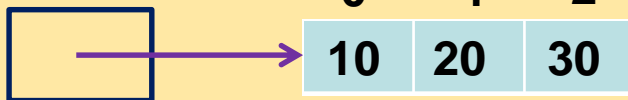
```
int [ ] dongList = new int[3];
```

```
dongList[0] = 10;
```

```
dongList[1] = 20;
```

```
dongList[2] = 30;
```

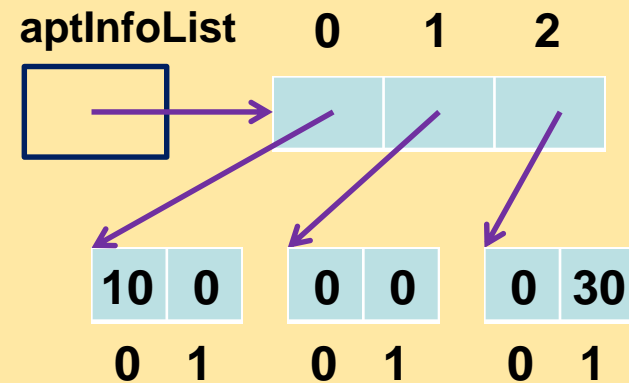
dongList



```
int [ ][ ] aptInfoList =  
    new int[3][2];
```

```
aptInfoList[0][0] = 10;
```

```
aptInfoList[2][1] = 30;
```



## - 배열의 초기화

```
int [ ] dongList = new int[3];
```

- 배열의 인덱스는 0부터 시작
- 배열의 크기 : 배열이름.length
- 마지막 요소 인덱스 : 배열크기 - 1

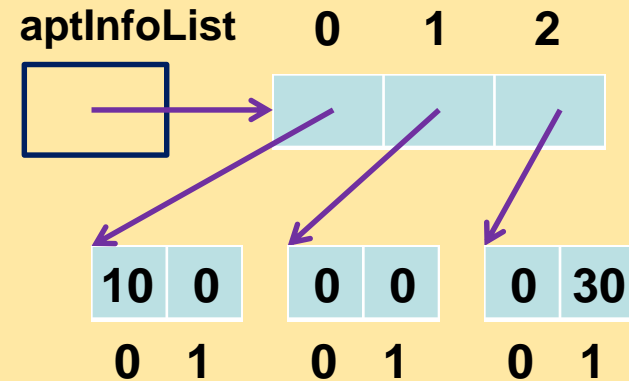
aptInfoList.length ??

aptInfoList[0].length ??

```
int [ ][ ] aptInfoList =  
    new int[3][2];
```

```
aptInfoList[0][0] = 10;
```

```
aptInfoList[2][1] = 30;
```



## - 초기화

- { } 를 활용하는 방식 : 배열 선언 시에만 설정 가능

1차원 배열 : 배열유형 [ ] 배열명 = { 값, .. 값 };

ex) int [ ] prime = { 1, 2, 3 };

2차원 배열 : 배열유형 [ ][ ] 배열명 = { { 값1, 값2 }, { 값3, 값4 } };

ex) int [ ][ ] twoArr = { { 1, 2 }, { 3, 4 }, { 5, 6 } };

- new 배열타입[ ] { 값, ....}

ex) int [ ] prime = new int[ ] { 1, 2 };

## - 배열관련 제공 API

### - **`System.arraycopy(src, srcPos, dest, destPos, length)`**

**src** : 원본배열      **srcPos** : 원본배열의 복사 시작 위치( 0부터 시작 )

**dest** : 복사할 배열    **destPos** : 복사 받을 시작 위치

**length** : 복사할 크기

예) `String [ ] oriArr = {“봄”, “여름”, “가을”};`

`String [ ] destArr = new String[oriArr.length + 1];`

`System.arraycopy( oriArr, 0, destArr, 0, oriArr.length );`

`destArr[ 3 ] = “겨울”;`

`for( int i = 0; i < destArr.length; i++)`

`System.out.println(destArr[ i ]);`

## - 배열관련 제공 API

### - **Arrays.toString(배열객체)**

: 배열안의 요소를 [ 요소, 요소, ..] 의 형태로 출력

예)

```
for( int i = 0; i < destArr.length; i++)
```

```
    System.out.println(destArr[ i ]);
```

```
=====
```

단순 배열값 확인일 경우

```
System.out.println( Arrays.toString(destArr) );
```



클 래 스



# 클래스

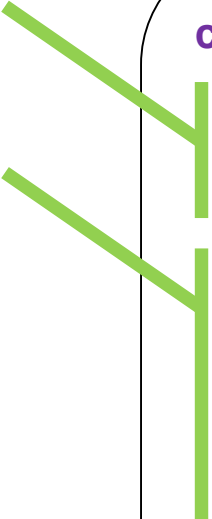
- ❖ 모든 객체들의 생산처
- ❖ 클래스 = 객체를 생성하는 틀
- ❖ 프로그래밍이 쓰이는 목적을 생각하여 어떤 객체를 만들어야 하는지 결정한다.
- ❖ 각 객체들이 어떤 특징(속성과 동작)을 가지고 있을지 결정한다.
- ❖ 객체들 사이에서 메시지를 주고 받도록 만들어 준다.



# 객체의 구성

- ❖ 속성(Attribute) - 멤버변수
- ❖ 동작(Behavior) - 메소드

```
class TV {  
    int channel;  
    int volumn;  
    public void channelUp() {  
    }  
    public void channelDown() {  
    }  
}
```



# 추상화와 클래스

❖ 필요한 객체를 설계해서 프로그램이 인식하게 하는 방법

- 클래스를 설계한다.
- 클래스로부터 객체를 생성한다.
- 생성된 객체는 클래스에 정의한 속성과 동작을 가지고 동작한다.

```
class TV {  
    int channel;  
    int volumn;  
    public void channelUp() {  
    }  
    public void channelDown() {  
    }  
}  
TV tv = new TV();  
tv.channelDown();
```

# 클래스의 선언

public / default

final / abstract

[접근제한자] [활용제한자] **class** 클래스명 {

속성 정의 (멤버변수)

기능 정의 (메소드)

}

❖ 객체(object / instance) 와 인스턴스 변수 ??



메 소 드

# 메소드

- 메소드(Method)

객체가 할 수 있는 행동을 정의

메소드의 이름은 소문자로 시작하는 것이 관례

`public / protected / default() / private`



`static / final / abstract / synchronized`



`[접근제한자] [활용제한자] 반환값 메소드이름([매개변수들]) {`

`행위 기술.....`

`}`

`public static void main(String [] a) { }`

# 메소드

- 메소드 선언

- 선언시 { } 안에 메소드가 해야 할 일을 정의

```
class Test {  
    public void call(int val) {  
        .....  
    }  
}
```

- 메소드 호출

- 호출한 메소드가 선언되어 있는 클래스를 접근한다.
  - 클래스객체.메소드 이름으로 호출

```
Test t = new Test( );
```

```
t.call( 100 );
```

- **static** 이 메소드에 선언되어 있을 때는 클래스이름.메소드 이름으로 호출

# 메소드

- 메소드 호출

- `class Test {`

- `public static void call( ) {`

- `}`

- `}`

- `static` 이 메소드에 선언되어 있을 때는 클래스이름.메소드 이름으로 호출

- `Test t = new Test( );`

- `t.call( );`

- `-----`  
`Test.call( );`

# 메소드

- 매개변수
  - 메소드에서 사용하는 것
- 인자
  - 호출하는 쪽에서 전달하는 것
- 메소드에서 받은 매개변수는 그 메소드에서 선언한 지역 변수와 똑같이 간주됩니다.
- 메소드에 매개변수가 있으면 반드시 해당 유형의 값을 전달해야만 합니다.

```
public void soundUp( int val ) {  
    // 이 위치에서 선언되는 것  
    String msg = “지역변수”;  
  
}  
=====  
soundUp(10);
```



# 메소드

- 메소드로부터 값을 받을 수도 있습니다.
- 리턴 유형은 메소드를 선언할 때 지정하며, 리턴 유형이 정해져 있으면 반드시 그 유형의 값을 리턴해야만 합니다.

```
public void volumnUp(int val) {  
    // 이 위치에서 선언되는 것  
    String msg = “지역변수”;  
}
```

```
=====  
volumnUp(10);
```

```
public int getVolumn( ) {  
    return 10;  
}
```

```
=====  
int volumn = getVolumn( );
```

# 메소드

- 메소드에 여러 개의 인자를 전달할 수 있습니다.
- 메소드에는 값이 전달되는 방식을 사용합니다.

```
public void init(int channel, int volumn) {  
}
```

```
=====
```

```
int channel = 7;    int volumn = 15;
```

channel

volumn

7

15

```
init( 7, 15);
```

```
=====
```

```
init( channel, volumn );
```



The diagram illustrates the process of passing arguments to a method. On the left, two variables, 'channel' and 'volumn', are shown with their respective values, 7 and 15, in green boxes. Blue curved lines connect these boxes to the arguments in the method call 'init( channel, volumn );' on the right. A vertical green line separates the method definition from the method call.

# 메소드

- 만약, 여러 개의 값을 리턴하고 싶다면??

배열을 이용

**Collection** 객체를 이용

- 만약, 리턴값의 유형이 double로 선언했다면, double이 받을 수 있는 값들은 모두 리턴이 가능합니다.

```
public double getVolumn( ) {  
    return 10;  
}
```

```
public double getVolumn( ) {  
    return 10;  
}  
-----  
getVolumn();
```

- 리턴값을 무조건 사용해야 하는 것은 아닙니다.



# Java 문자열과 API 사용법



# 1. 문자열 정의

- ➔ 자바에서는 문자열을 객체로 취급
- ➔ `java.lang` 패키지에 포함
- ➔ `java.lang.String`, `StringBuffer`, `StringBuilder` 클래스 제공
- ➔ `String` 클래스 : 한번 생성된 다음 변하지 않는 문자열에 사용
- ➔ `StringBuffer` 클래스 : 계속하여 변할 수 있는 문자열에 사용, 동기화 적용
- ➔ `StringBuilder` 클래스 : 계속하여 변할 수 있는 문자열에 사용, 비동기화



## **construct**

String()

String(char chars[])

String(char chars[], int startindex, int numChars)

String(String strObj)

String(byte asciiChars[])

String(byte asciiChars[], int startIndex, int numChars)



## 문자열 길이


`int length()` : 문자열의 길이를 반환

## 문자열 추출

`char charAt(int i)` : 문자열중에서 `i`번째 문자를 반환

`void getChars(int sourceStart, int sourceEnd, char target[],  
int targetStart)`

➔ 문자열의 일부를 문자 배열로(`target[]`) 제공



## 문자 비교

`boolean equals(Object str)` : `str`로 지정된 문자열과 현재의 문자열 같은지 비교

`boolean equalsIgnoreCase(String str)` : 문자열 비교시 대소문자 무시

`boolean startsWith(String str)` : 문자열이 `str`로 시작하면 `true`, 아니면 `false`

`boolean endsWith(String str)` : 문자열이 `str`로 끝나면 `true`, 아니면 `false`

`int compareTo(String str)` : 현재의 문자열과 `str`로 지정된 문자열을 비교하여 현재의 문자열이 `str`로 지정된 문자열보다 크면 양수, 같으면 0, 작으면 음수값을 반환.

작다는 의미는 순서(알파벳)에 따라 앞에 온다는 의미



## 문자열 탐색

`int indexOf(String str)`

`int indexOf(String str, int startIndex)`

`int lastIndexOf(String str)`

`int lastIndexOf(String str, int startIndex)`

## 문자열 변환

`String substring(int startIndex, int endIndex)`      부분 문자열을 반환

`String concat(String constr)`      결합된 문자열 반환


`String replace(char original, char replacement)`      치환된 문자열 반환

`String trim()`      문자열의 시작과 끝부분에 있는 공백이 제거된 문자열을 반환

`String toLowerCase( )`      소문자로 반환

`String toUpperCase( )`      대문자로 반환

`String [ ] split(String regexp)`      regexp를 기준으로 문자열을 나눈다.



## 형의 변환

`static String valueOf(double num)`

`static String valueOf(long num)`

`static String valueOf(Object obj)`

`static String valueOf(char chars[])`

`static String valueOf(char chars[], int startIndex, int numChars)`



# 생 성 자



# 생성자 특징

1. 클래스 명과 이름이 동일
2. 반환타입이 없다.

```
public class Dog {  
    Dog ( ) {  
        System.out.println("나는 생성자 입니다.");  
        System.out.println("클래스와 이름이 동일하고 반환타입이 없어요");  
    }  
}
```

# 생성자 특징

## 3. 디폴트 생성자

- 클래스내에 생성자가 하나도 정의되어 있지 않을 경우  
**JVM**이 자동으로 제공하는 생성자
- 형태 : 매개변수가 없는 형태, 클래스명() {}

```
class Dog {
```

```
    생성자가 하나도 없는 상태임
```

```
    JVM 이 자동으로 제공함
```

```
    ??????
```

```
}
```

```
class Main {
```

```
    public static void main(String [] a) {
```

```
        // 객체 생성
```

```
        Dog d = new Dog( );
```

```
    }
```

```
}
```

# 생성자 특징

## 4. 오버로딩을 지원한다.

- 클래스 내에 메소드 이름이 같고 매개변수의 타입 또는 개수가 다른 것

```
class Dog {  
    Dog() { }  
    Dog(String name) { }  
    Dog(int age) { }  
    Dog(String name, int age) { }  
}
```

```
class Main {  
    public static void main(String [] a) {  
        Dog d = new Dog();  
        Dog d2 = new Dog("짱");  
        Dog d3 = new Dog(3);  
        Dog d4 = new Dog("메리", 4);  
    }  
}
```

# 생성자 특징

5. 객체를 생성할 때 속성의 초기화를 담당하게 한다.

```
class Dog {  
    String name;  
    int age;  
    Dog() { }  
    Dog( String n, int a) {  
        name = n;  
        age = a;  
    }  
}
```

```
class Main {  
    public static void main(String [] a) {  
        Dog d = new Dog( );  
        d.name = “짱”;  
        d.age = 3;  
        Dog d2 = new Dog(“메리”, 4);  
    }  
}
```

# 생성자 특징

6. this의 활용 : static 영역에서는 사용이 불가능하다.

- **this.멤버변수**

- **this ( [ 인자값.. ] )** : 생성자 호출

- **this** 생성자 호출시 제한사항

  - : 생성자 내에서만 호출이 가능함

  - : 생성자 내에서 첫번째 구문에 위치해야 함

```
class Dog {  
    String name;  
    int age;  
    void info ( ) {  
        System.out.print(this.name);  
        System.out.println(this.age);  
    }  
}
```



# 생성자 특징

6. this의 활용 : static 영역에서는 사용이 불가능하다.

- this.멤버변수

- this ( [ 인자값.. ] ) : 생성자 호출

- this 생성자 호출시 제한사항

  - : 생성자 내에서만 호출이 가능함

  - : 생성자 내에서 첫번째 구문에 위치해야 함

```
class Dog {  
    String name;  
    int age;  
    Dog ( ) {  
        this("짱");  
    }  
    Dog ( String name ) {  
    }  
}
```



# **static**



# static 특징

## 1. 로딩시점

- **static** : 클래스 로딩 시
- **nonStatic** : 객체 생성시

## 2. 메모리상의 차이

- **static** : 클래스당 하나의 메모리 공간만 할당
- **nonStatic** : 인스턴트 당 메모리가 별도로 할당

# static 특징

## [실행 시 메모리 영역]

타입 정보

1. Type Information
2. Constant Pool
3. Field Information
4. Method Information
5. Class Variables

Instance  
(객체 생성시)

PC Registers

Java Virtual  
Machine  
Stacks

Method  
Area  
(공유)

Heap  
(독립적)

Native  
Method  
Stacks

# static 특징

## 3. 문법적 특징

- **static** : 클래스 이름으로 접근
- **nonStatic** : 객체 생성 후 접근

```
class Employee {  
    static int empCount;  
    String name;  
}
```

```
class Main {  
    public static void main(String [] a) {  
        Employee e = new Employee( );  
        e.name = "손오공";  
        Employee.empCount ++;  
    }  
}
```

# static 특징

## 4. static 영역에서는 non-static 영역을 집적 접근이 불가능

```
class Main {  
    String name = "길동이";  
    public static void main(String [] a) {  
        System.out.println( name );  
    }  
}
```


오류발생



# static 특징

- non-static 영역에서는 static 영역에 대한 접근이 가능

```
class Main {  
    static int count = 100;  
    public void call () {  
        System.out.println( count );  
    }  
}
```



상 속



1. 확장성, 재 사용성
2. 클래스 선언 시 **extends** 키워드를 명시

```
class Employee {  
  
}  
class Manager extends Employee {  
  
}
```

3. 관계
  - 부모 (상위, **Super**) 클래스 : **Employee**
  - 자식 (하위, **Sub**) 클래스 : **Manager**

4. 자식 클래스는 부모 클래스에 선언 되어 있는 멤버변수, 메소드를 자신의 것처럼 사용할 수 있다.

단, 접근 제한자에 따라 사용 여부가 달라진다.

```
class Employee {  
    int    no;  
    String name;  
}
```

```
class Manager extends Employee {  
    public void info( ) {  
        System.out.println( no );  
        System.out.println( name );  
    }  
}
```

## 5. super 키워드

```
public class Employee {  
    String no;  
    String name;  
    String grade;  
    int salary;  
    public Employee(String no, String name, String grade, int salary) {  
        this.no = no;  
        this.name = name;  
        this.grade = grade;  
        this.salary = salary;  
    }  
}  
  
public class Manager extends Employee {  
    String no;  
    String name;  
    String grade;  
    int salary;  
    Employee [] subEmpList;  
    public Manager03(String no, String name, String grade, int salary, Employee [] subEmpList) {  
        super(no, name, grade, salary);  
        this.subEmpList = subEmpList;  
    }  
}
```

## 6. 오버라이딩 ( 재정의 )

```
public class Employee {
```

```
    String no;
```

```
    String name;
```

```
    String grade;
```

```
    int salary;
```

```
    public void
```

```
        System.out.
```

```
        System.out.
```

```
        System.out.
```

```
        System.out.
```

```
    }
```

```
}
```

```
public class Manager extends Employee {
```

```
    public void info() {
```

```
        super.info( );
```

```
        System.out.println("-----");
```

```
        System.out.println("관리하는 직원들의 정보");
```

```
        System.out.println("-----");
```

```
        for (int i = 0; i < subEmpList.length; i++) {
```

```
            subEmpList[i].info();
```

```
        }
```

```
        System.out.println("-----");
```

```
    }
```

```
}
```



# 접근제한자



# 접근제한자

- **public** : 모든 위치에서 접근이 가능
- **protected** : 같은 패키지에서 접근이 가능, 다른 패키지 접근 불가능  
단, 다른 패키지의 클래스와 상속관계가 있을 경우 접근 가능
- **default** : 같은 패키지에서만 접근이 허용  
접근제한자가 선언이 안 되었을 경우 기본 적용
- **private** : 자신 클래스에서만 접근이 허용
  
- 클래스(외부) 사용가능 : **public, default**
- 내부클래스, 멤버변수, 메소드 사용 가능 : 4 가지 모두 가능

# 접근제한자

```
public class Test {  
    public      String name;  
    protected  String pass;  
                String addr;  
    private    int money;  
}
```


kr.co.bit.pack.a

```
class A extends Test {  
    ???  
}
```

```
import kr.co.bit.pack.a.Test;  
class Main {  
    Test t = new Test ( );  
    t.??  
}
```

kr.co.bit.pack.b

```
import kr.co.bit.pack.a.Test;  
class B extends Test {  
    ???  
}
```



# 객체의 형변환



# 형변환

1. 정의 : "=" 연산자를 기준으로 좌변과 우변의 데이터 타입이 다른 경우에 발생
2. 조건 : 좌변과 우변의 객체가 상속 관계가 있어야 함

```
public class Printer { }  
public class LGPrinter { }  
Printer p = new LGPrinter (); (오류)  
=====
```

```
public class Printer { }  
public class LGPrinter extends Printer { }  
Printer p = new LGPrinter ();
```

# 묵시적 형변환

## 3. 종류

### 1) 묵시적 형변환

- 상위(부모) 클래스 타입 = 하위(자식) 클래스 타입

**Printer p = new LGPrinter ( );**

- 부모타입은 자식타입을 포함
- 기억해야 할 중요 포인트:

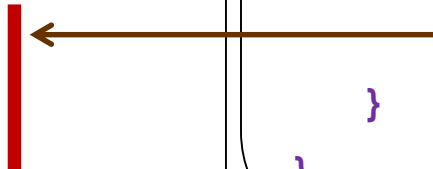
형변환된 상위 클래스 변수가 사용 할 수 있는 범위는 자신 클래스에 정의된 변수와 메소드만 사용이 가능

단, 상위클래스의 메소드를 하위클래스에서 오버라이딩 (재정의) 했을 경우 하위클래스에 선언된 메소드가 호출

# 묵시적 형변환

```
public abstract class TV {  
    boolean power;  
    public void powerOn( );  
}  
  
public class LgTV extends TV {  
    String name = "LGTV";  
    public void powerOn() {  
        power = true;  
    }  
    public void print( ) {  
        System.out.println(name);  
    }  
}
```

```
class Main {  
    public static void main(String [] a) {  
        TV tv = new LgTV( );  
        tv.power ;  
        tv.name;  
        tv.print( );  
        tv.powerOn( );  
    }  
}
```





# 명시적 형변환

## 2) 명시적 형변환

- 형변환 연산자를 이용해서 변환
- 하위클래스 타입 = (하위클래스타입) 상위 클래스 타입

**LGPrinter lg = (LGPrinter) p;**

- 기억해야 할 중요 포인트 :

상위클래스 타입 자리에 올 수 있는 객체는 실제 가리키는 메모리가 하위 클래스 타입 이어야 가능

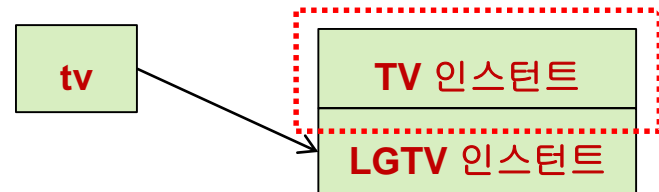
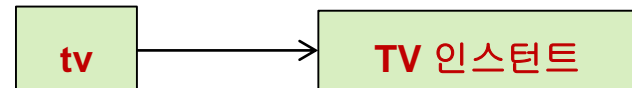
# 명시적 형변환

```
public class TV { }  
public class LgTV extends TV { }
```

```
LgTV lg = new TV(); // 컴파일 오류
```

```
TV tv = new TV( );  
LgTV lg = (LgTV) tv; // 실행시 오류
```

```
TV tv = new LgTV( );  
LgTV lg = (LgTV) tv; // 성공
```





# 추 상 클 래 스

## (abstract)

# 추상클래스

**abstract** : 클래스, 메소드, 멤버변수(X)

## 1. 추상클래스

- **abstract** 를 클래스 선언 부분에 추가함

```
abstract class Animal {  
}
```

## 2. 추상메소드

- **abstract** 를 메소드 선언 부분에 추가함
- 메소드의 선언부만 있고 바디({}) 가 없음

```
void print ( ) {  
}
```

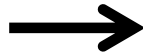


```
abstract void print ( ) ;
```

# 추상클래스

## 3. 추상 메소드를 포함하는 클래스는 반드시 추상클래스로 선언되어야 함

```
class Test {  
    abstract void print();  
}
```



```
abstract class Test {  
    abstract void print( );  
}
```

## 4. 추상클래스는 인스턴스 생성이 불가능함. (new 키워드 사용 불가능)

```
abstract class Test { }  
class Main {  
    Test t = new Test( );  
}
```



오류발생



# 추상클래스

5. 추상클래스는 일반(구현된) 메소드와 추상 메소드 모두 선언이 가능함.


```
abstract class Printer {  
    abstract void print();  
    public void call () {  
        System.out.println("구현된 메소드");  
    }  
}
```

# 추상클래스

6. 추상클래스를 상속받는 하위클래스는 상위클래스의 추상 메소드를 반드시 오버라이딩(재정의) 해야 한다.

```
abstract class Printer {  
    abstract void print();  
    public void call () {  
        System.out.println("구현");  
    }  
}
```

```
class LGPrinter extends Printer {  
    void print( ) {  
        System.out.println("재정의");  
    }  
}
```



# 추상클래스

## 7. 추상클래스의 객체변수는 하위클래스를 이용함.

```
abstract class Printer { }  
class LGPrinter extends Printer { }  
-----  
class Main {  
    public static void main(String [ ] args) {  
        Printer p = new LGPrinter ( );  
    }  
}
```



# 인 터 페 이 스 (interface)

# 정의 및 특징

- 완벽한 추상화된 객체
- 반쯤 완성된 객체
- 설계도

## 1. **interface** 키워드를 이용하여 선언

```
interface ServerConstants { }
```

## 2. 선언되는 변수는 모두 상수로 적용

```
interface ServerConstants {  
    String SERVER_IP = "127.0.0.1";  
}
```

→ **public static final** String SERVER\_IP

# 특징

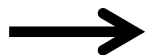
## 3. 선언되는 메소드는 모두 추상 메소드로 적용

```
interface Test {  
    void print( ) ;
```

```
    public void call ( ) {
```

```
    }
```

```
}
```



```
public abstract void print( ) ;
```



오류발생

# 특징

## 4. 객체 생성이 불가능 (추상클래스 동일한 특성)

```
interface Test { }
```

```
class Main {
```

```
    Test t = new Test( );
```

```
}
```

오류발생

## 5. 클래스가 인터페이스를 상속 할 경우에는 **extends** 키워드가 아니라 **implements** 키워드를 이용

```
interface shape {
```

```
}
```

```
class Circle implements Shape {
```

```
}
```

오류발생

# 특징

6. 인터페이스를 상속받는 하위클래스는 추상 메소드를 반드시 오버라이딩 (재정의) 해야 한다.

```
interface Printer {  
    void print( );  
}
```

```
class LGPrinter implements Printer {
```

```
    public void print( ) {  
        System.out.println("재정의");  
    }
```

```
}
```



## 7. 인터페이스 객체변수는 하위클래스를 이용함.

```
interface Printer { }  
class LGPrinter implements Printer { }  
-----  
class Main {  
    public static void main(String [ ] args) {  
        Printer p = new LGPrinter ( );  
    }  
}
```



# final

# final

## final 3가지 사용법

1. 변수 : 상수
2. 메소드 : 오버라이딩 금지
3. 클래스 : 상속

```
class Printer {  
    public final void print () {}  
}
```

```
class LGPrinter extends Printer {  
    public void print () {  
        System.out.println ("재정의");  
    }  
}
```

오류발생

# final

## final 3가지 사용법

1. 변수 : 상수
2. 메소드 : 오버라이딩 금지
3. 클래스 : 상속 금지

오류발생

```
final class String {  
    public void print () {}  
}
```

```
class MyString extends String{  
    public void print () {  
        System.out.println ("재정의");  
    }  
}
```



# 예외처리

---

10.1 예외란

10.2 예외 처리

10.3 사용자 정의 예외 생성

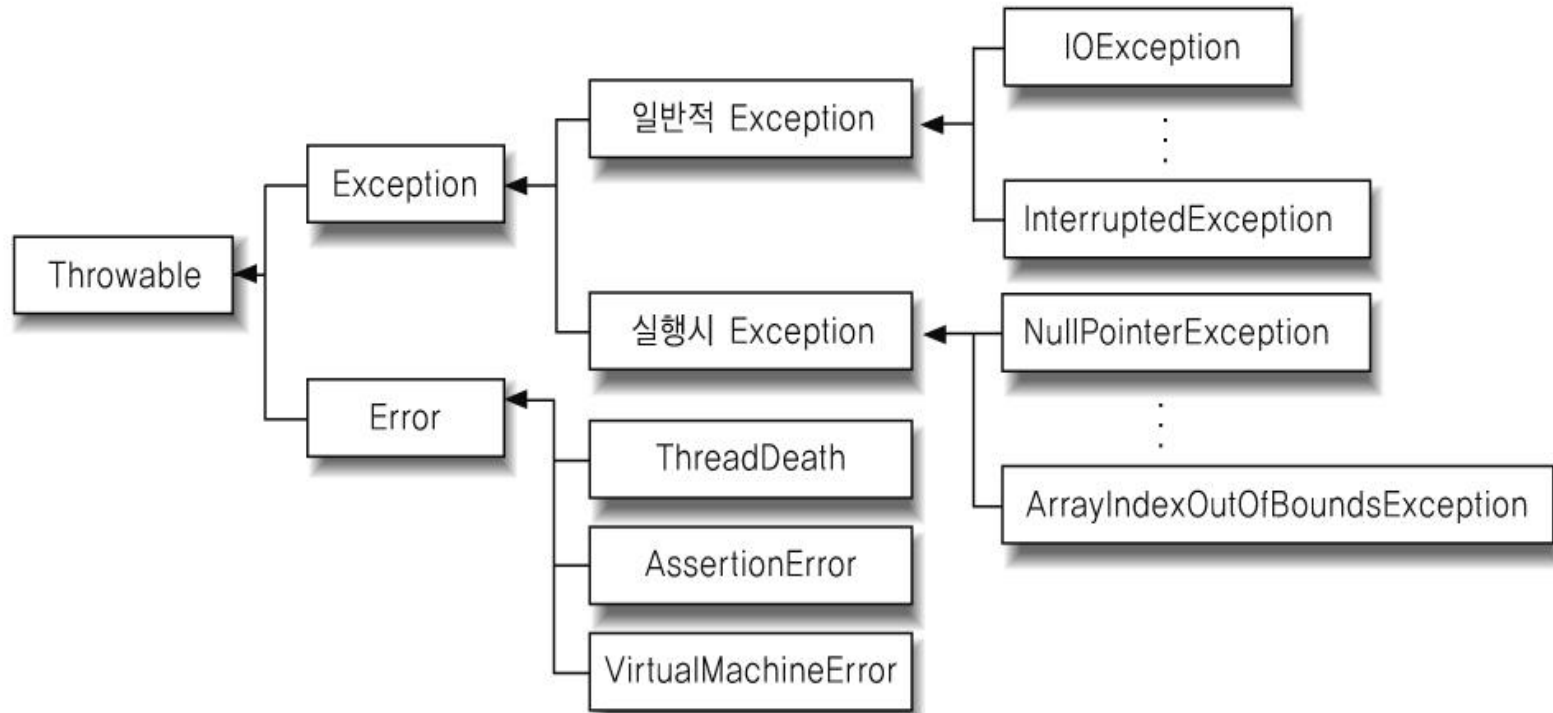


# 예외정의

---

- 예외 : 프로그램이 실행되는 동안에 발생하는 예기치 않은 에러
- 예외는 컴파일 시점과 실행 시점으로 나눌 수 있음
- 예외가 발생하는 예
  - 정수를 0으로 나누는 경우
  - 배열의 첨자가 음수 또는 범위를 벗어나는 경우
- 자바 언어는 프로그램에서 예외를 처리할 수 있는 기법을 제공
- 발생할 수 있는 예외를 클래스로 정의

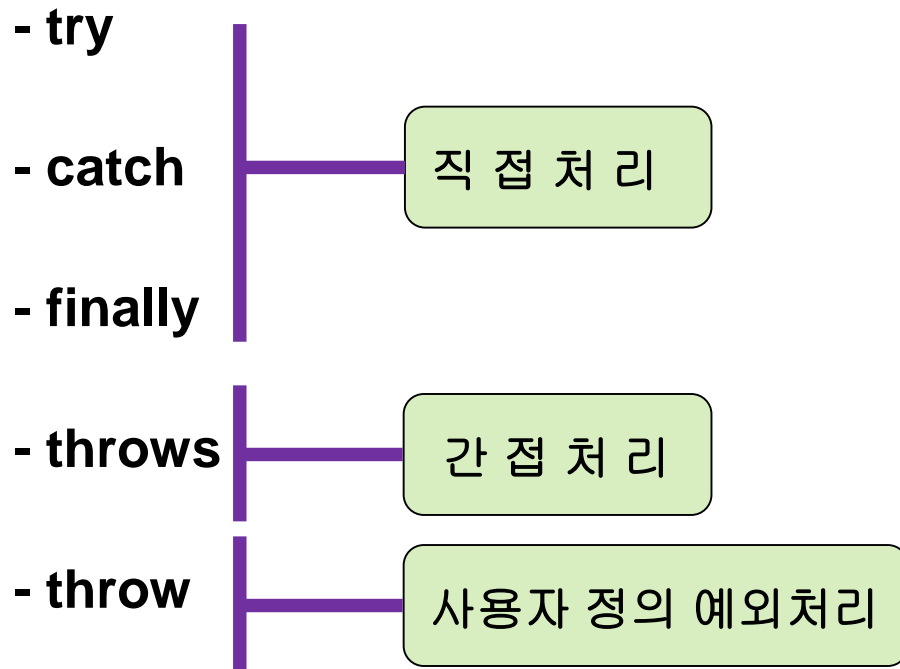
# 예외클래스



예외의 종류와 구조

# 예외관련 키워드

꼭 기억해야 할 키워드(5가지)





# 예외처리

## 1. 직접처리

- **try** : 예외가 발생할 만한 코드를 기술하는 부분
- **catch** : **try** 블록에서 예외가 발생하면 예외를 처리하는 부분
- **finally** : 예외 발생여부와 상관없이 무조건 실행하는 부분

```
try {  
    예외가 발생할 것 같은 코드 정의  
} catch(Exception e) {  
    예외처리  
} finally {          무조건 실행 }
```

### 예외처리 진행 순서

1. **try** 구문으로 진입

2 – 1. **try** 구문 안에서 예외가 발생하면

2 – 2. **catch** 구문을 순차적으로 살펴보면서 일치하는 예외가 있는지  
조사하여 해당 블록으로 간다.

2 – 3. 해당 **catch** 블록을 실행하여 에러처리를 한다.

2 – 2. **try** 구문 안에서 예외가 발생하지 않았다면 **catch** 블록을 실행하지  
않는다.

3. **finally** 블록이 있다면 예외 발생 유무에 상관없이 무조건 실행한다.

```
public static void main(String [] args) {  
    System.out.println(1);  
    try {  
        System.out.println(2);  
        예외 발생 가능 코드  
        System.out.println(3);  
    } catch(Exception e) {  
        System.out.println(4);  
    } finally {  
        System.out.println(5);  
    }  
}
```

정상 수행 시  
1 - 2 - 3 - 5

예외 발생시  
1 - 2 - 4 - 5

**throws :**

메서드 내에서 발생한 예외를 자신이 직접 처리하는 것이 아니라  
자신을 호출한 쪽으로 예외처리를 떠넘기는 역할을 하는 키워드

형식 >

```
public void print( ) throws Exception {  
    예외가 발생할 것 같은 코드 정의  
}
```

**throw :**

**JVM**이 예외를 발생시키는 것이 아니라 인위적으로 특정 시점에 예외를 발생 시킬 때 사용하는 키워드

형식>

**throw** 예외객체

**throw new Exception( );**

사용자 정의 예외 클래스 :


정의 :

- **API**에 정의된 예외상황이 아니라 프로그램 내에서 특별한 예외 상황에 맞는 예외를 정의할 경우 사용

방법 :

- **Exception** 클래스를 상속받아서 정의

```
public class UserException extends Exception {  
    }  
}
```



# 날 짜 API



# 날짜 관련 API

- 날짜관련 API 클래스
  - ✓ Date
  - ✓ Calendar
  - ✓ SimpleDateFormat





# Date

- 1.0 버전 부터 지원되는 클래스
- 1.1 버전 부터는 Calendar 클래스 사용을 권장

## ➤ 생성자

- Date()
- Date(long *msec*)

# Date

## ➤ 메 소 드

이름	설명
long getTime()	1970년 이후로 현재까지의 시간을 밀리초로 반환
int getYear()	1900년 이후부터의 년수를 반환
int getMonth()	해당되는 월을 반환. 0:1월 - 11:12월
int getDate()	1-31 사이의 날짜를 반환
int getDay()	요일을 해당되는 숫자로 반환. 0:일요일 - 6:토요일
int getHours()	0-23 까지의 시간을 반환
int getMinutes()	0-59 사이의 분을 반환
int getSeconds()	0-59 사이의 초를 반환

# Calendar

- 추상클래스
- 객체를 얻기위해 Calendar.getInstance( ) 를 활용

## ➤ 메 소 드

- 객체 얻기 : static Calendar getInstance()
- 정보 추출 : int get(int calendarField)

필드	의미	필드	의미
YEAR	년	HOUR HOUR_OF_DAY	시간
MONTH	월	MINUTE	분
DATE DAY_OF_MONTH	일	SECOND	초
DAY_OF_WEEK	요일		



# Calendar

---

- 날짜 설정 :

`void set(int year, int month, int date)`

`void set(int year, int month, int date, int hour, int minute)`

`void set(int year, int month, int date, int hour, int minute, int second)`

- Date 객체 얻기 :

`Date getTime()`      현재의 객체와 같은 날짜를 가진 Date 객체를 반환

- Date 객체 시간정보를 Calendar 로 설정하기 :

`void setTime(Date d)` Date 객체 d의 정보를 이용하여 현재의 객체를 설정

- 날짜 정보에서 해당 항목의 최대값 얻기 :

`int getActualMaximum (int calendarField)`

# SimpleDateFormat

- ◆ 날짜 객체로 부터 원하는 형태의 문자열로 변환
- ◆ 특정한 포맷 문자열을 사용하여 날짜 정보를 추출

## ➤ 주요 메소드

- SimpleDateFormat(String pattern)

pattern 에 지정된 형태로 날짜를 문자열로 변환

주요패턴 문자					
y	년	M	월	d	일
H	시간(0 - 23)	m	분	s	초
h	시간(0 - 11)				
E	요일				

- String format (Date d)

Date 객체를 매개변수로 받아서 지정된 패턴 형식으로 문자열 반환



# Collection API

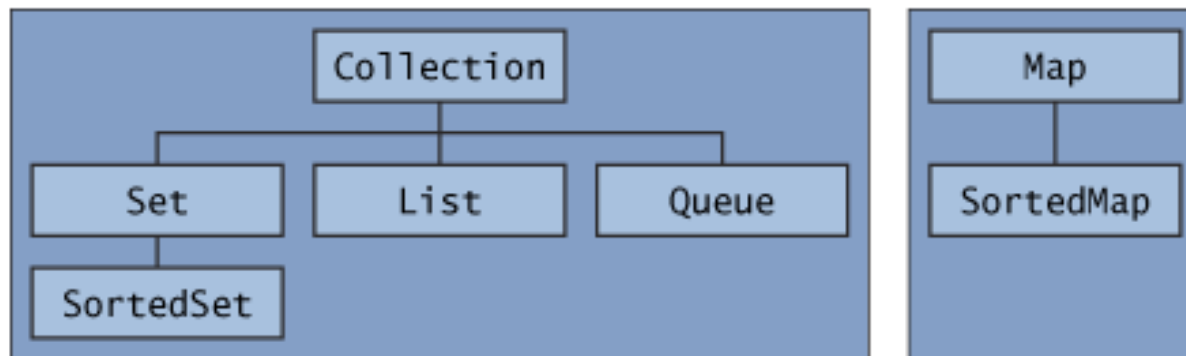


# Collection 과 자료구조

- 객체들을 한곳에 모아놓고 편리하게 사용할 수 있는 환경을 제공
- 정적 자료구조(Static structure)
  - 고정된 크기의 자료구조
  - 배열이 대표적인 정적 자료구조
  - 선언 시 크기를 명시하면 바꿀 수 없음
- 동적 자료구조(Dynamic structure)
  - 요소의 개수에 따라 자료구조의 크기가 동적으로 증가하거나 감소
  - 벡터, 리스트, 스택, 큐 등

# Collection 과 자료구조

- 자료구조들의 종류는 결국은 어떤 구조에서 얼마나 빨리 원하는 데이터를 찾는가에 따라 결정된다.
  - 순서를 유지할 것인가?
  - 중복을 허용할 것인가?
  - 다른 자료구조들에 비해서 어떤 단점과 장점을 가지고 있는가?







# Generic

- Collections Framework이 기존에는 모든 객체자료형들을 처리하기 위해서 `java.lang.Object` 타입을 사용
- JDK1.5이후에는 컴파일 시점에 자료구조에서 사용되는 Type을 체크하는 Generic 문법을 사용하는 방식으로 변화

형식 : 클래스<타입>

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



# List

- 특징: 순서가 있고, 중복을 허용 (배열과 유사)
- 장점: 가변적인 배열
- 단점: 원하는 데이터가 뒤쪽에 위치하는 경우 속도의 문제
- 구현 클래스
  - ArrayList
  - LinkedList



# ArrayList - 메소드

- 내부적으로 배열을 이용하여 데이터를 관리
- 배열과 다르게 크기가 유동적으로 변함(동적 자료구조)
- 배열을 다루는 것과 유사하게 사용 할 수 있음

# ArrayList와 Generic

- ArrayList list = new ArrayList( );
- ArrayList<String> list2 = new ArrayList<String>( );

위의 두 코드의 차이점??

list 는 모든 객체를 받을 수 있음

list2 는 String 만을 받을 수 있음

# ArrayList - 메소드

- `add(E e)` : 데이터 입력

봄	여름			
---	----	--	--	--

```
list.add ( “봄” );
```

```
list.add ( “여름” );
```

# ArrayList - 메소드

- `get(int index)` : 데이터 추출

봄	여름			
0	1	2	3	4

`String val = list.get( 0 );` → 봄이 반환

# ArrayList - 메소드

- size( ) : 크기 반환

봄	여름
0	1

`int size = list.size( );` → 2가 반환

# ArrayList - 메소드

- remove(int i) : 인덱스 위치의 데이터를 삭제

봄	여름			
0	1	2	3	4

```
list.remove( 0 );
```



# ArrayList - 메소드

- remove(Object o) : 동일한 데이터를 삭제

봄	여름			
0	1	2	3	4

```
list.remove( "봄" );
```

# ArrayList - 메소드

- `clear ( )` : 모든 데이터를 삭제

봄	여름			
0	1	2	3	4

`list.clear( );`

# ArrayList - 메소드

- contains(Object o) : 특정 데이터가 있는지 체크

봄	여름			
0	1	2	3	4

```
boolean b = list.contains("봄");
```

# ArrayList - 메소드

- isEmpty( ) : 데이터가 존재하는지 체크

봄	여름			
0	1	2	3	4

```
boolean b = list.isEmpty( );
```

# ArrayList - 메소드

- addAll(Collection c) : 기존 등록된 컬렉션 데이터 추가

**list**

봄	여름			
---	----	--	--	--

**0**

**1**

**2**

**3**

**4**

**sub**

가을	겨울
----	----

**0**

**1**

**list.addAll( sub );**

# ArrayList - 메소드

- `add(E e)` : 데이터 입력
- `get(int index)` : 데이터 추출
- `size()` : 입력된 데이터의 크기 반환
- `remove(int i)` : 특정한 데이터를 삭제
- `remove(Object o)` : 특정한 데이터를 삭제
- `clear()` : 모든 데이터 삭제
- `contains(Object o)` : 특정 객체가 포함되어 있는지 체크
- `isEmpty()` : 비어있는지 체크(true, false)
- `addAll(Collection c)` : 기존 등록된 컬렉션 데이터 입력
- `iterator()` : Iterator 인터페이스 객체 반환



# Map

- 특징 : Key(키)와 Value(값)으로 나누어 데이터 관리, 순서는 없으며, 키에 대한 중복은 없음
- 장점 : 빠른 속도
- 구현 클래스
  - HashMap
  - TreeMap

# Map - 메소드

- **V put (K key, V value) : 데이터 입력**

동일한 값이 있을 경우 새로운 값으로 대체하고 기존 값 반환

value	길동	인천			
key	name	addr			

```
map.put( "name", "길동");
```

```
map.put( "addr", "인천");
```



# Map - 메소드

- **V get (Object Key) : 데이터 추출**

Key 에 해당하는 값이 없을 경우 null 반환

value	길동	인천			
key	name	addr			

**String val = map.get("name"); → “길동”이 반환**

# Map - 메소드

- **V remove (Object Key) : 데이터 삭제**

삭제된 값을 리턴, Key 에 해당하는 값이 없을 경우 null 반환

value	길동	인천			
key	name	addr			

**String val = map.remove("addr"); → “인천”이 반환**

# Map - 메소드

- **boolean containsKey(Object Key) : 특정 키 확인**

Key 가 존재할 경우 true 반환

value	길동	인천			
key	name	addr			

**boolean flag = map.containsKey("addr"); → true 반환**

# Map - 메소드

- **void putAll(Map<K Key, V value> m) :** 컬렉션 추가  
리스트의 addAll과 같은 역할, 기존 컬렉션에 구성된 데이터를 추가할 경우

value

길동

인천

key

name

addr

value

16

key

age

map.putAll(sub);

# Map - 메소드

- `V put(K key, V value)` : 데이터 입력
- `V get(Object key)` : 데이터 추출
- `V remove(K key)` : 입력된 데이터의 크기 반환
- `boolean containsKey(Object key)` : 특정한 key 포함 여부
- `void putAll(Map<K key, V value> m)` : 기존 컬렉션 데이터 추가
- `Set<Map.Entry<K, V>> entrySet()` :  
(key 와 value) 쌍을 표현하는 Map.Entry 집합을 반환

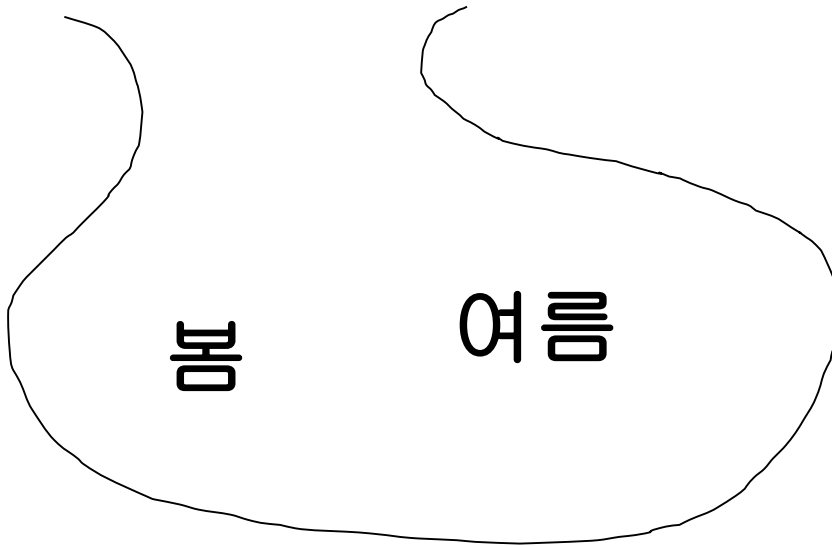


# Set

- 특징 : 순서가 없고, 중복을 허용하지 않음
- 장점 : 빠른 속도
- 단점 : 단순 집합의 개념으로 정렬하려면 별도의 처리가 필요하다.
- 구현 클래스
  - HashSet
  - TreeSet

# Set - 메소드

- `boolean add(E e)` : 데이터 입력

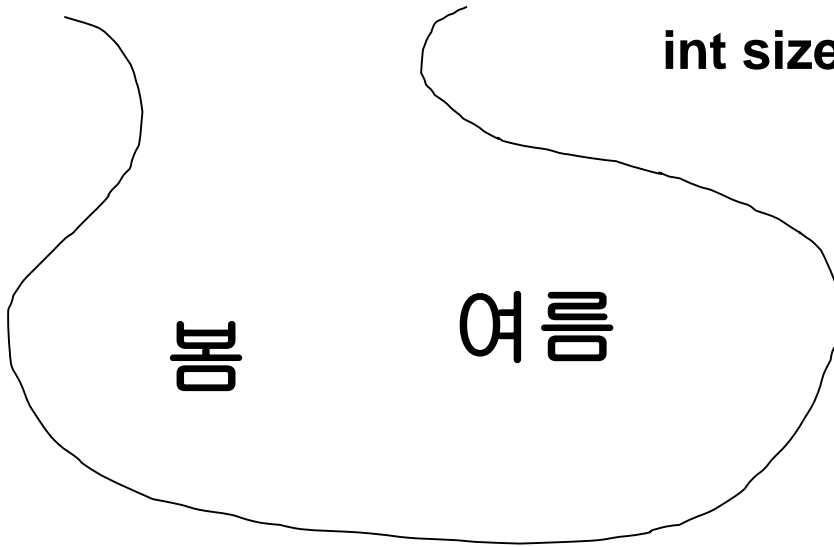


```
set.add ( “봄” );  
set.add ( “여름” );
```

# Set - 메소드

- `int size( )` : 크기 반환

`int size = set.size( );` → 2가 반환

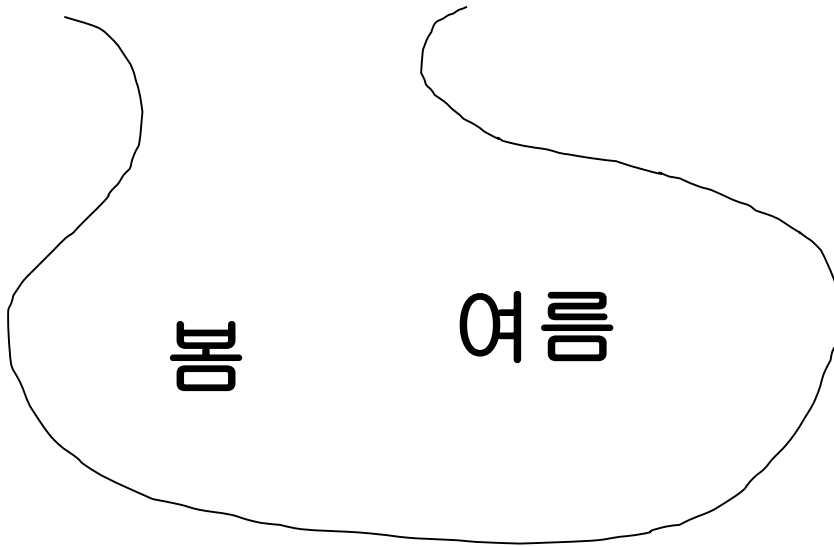




# Set - 메소드

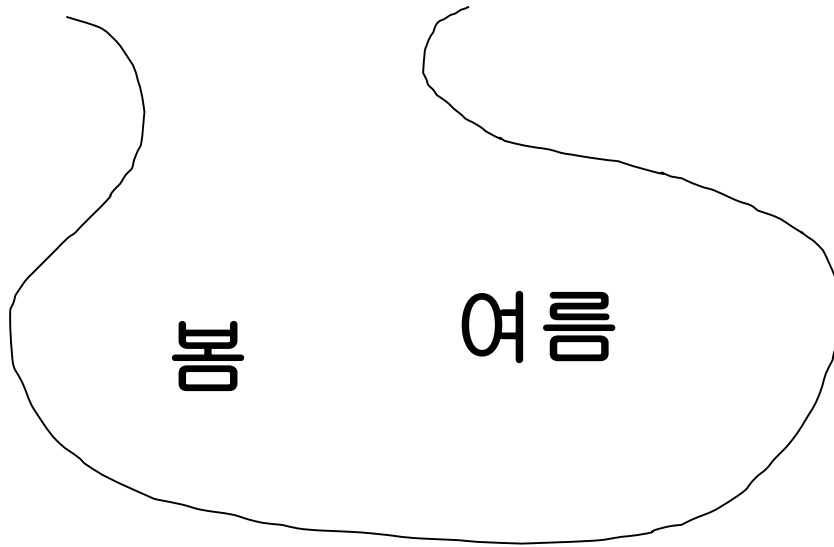
- `boolean remove(Object o)` : 동일한 데이터를 삭제

`set.remove( “봄” );`



# Set - 메소드

- void clear ( ) : 모든 데이터를 삭제



**set.clear( );**

# Set - 메소드

- contains(Object o) : 특정 데이터가 있는지 체크

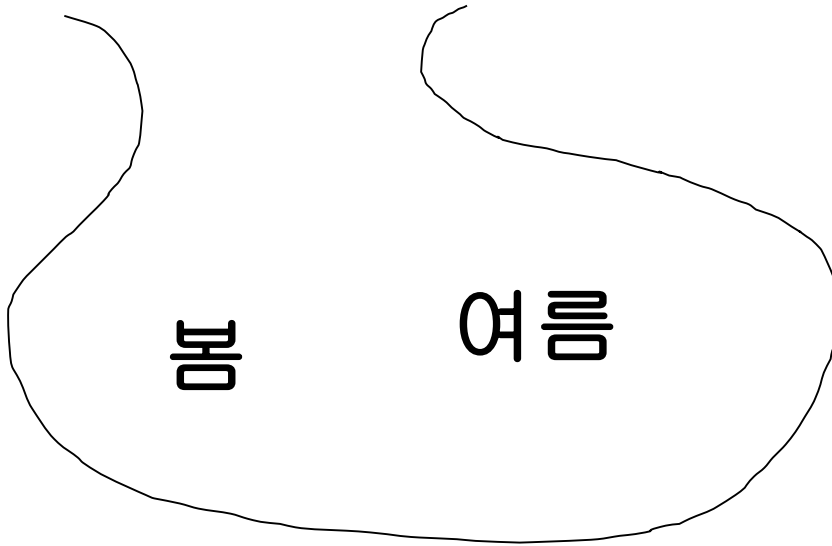
```
boolean b = set.contains("봄");
```



# ArrayList - 메소드

- isEmpty( ) : 데이터가 존재하는지 체크

```
boolean b = set.isEmpty( );
```





# Set - 메소드

---

- `add(E e)` : 데이터 입력
- `size()` : 입력된 데이터의 크기 반환
- `remove(Object o)` : 특정한 데이터를 삭제
- `clear()` : 모든 데이터 삭제
- `contains(Object o)` : 특정 객체가 포함되어 있는지 체크
- `isEmpty()` : 비어있는지 체크(true, false)
- `iterator()` : Iterator 인터페이스 객체 반환
- `toArray ()` : Set의 내용을 Object 형의 배열로 반환



# 입출력 API

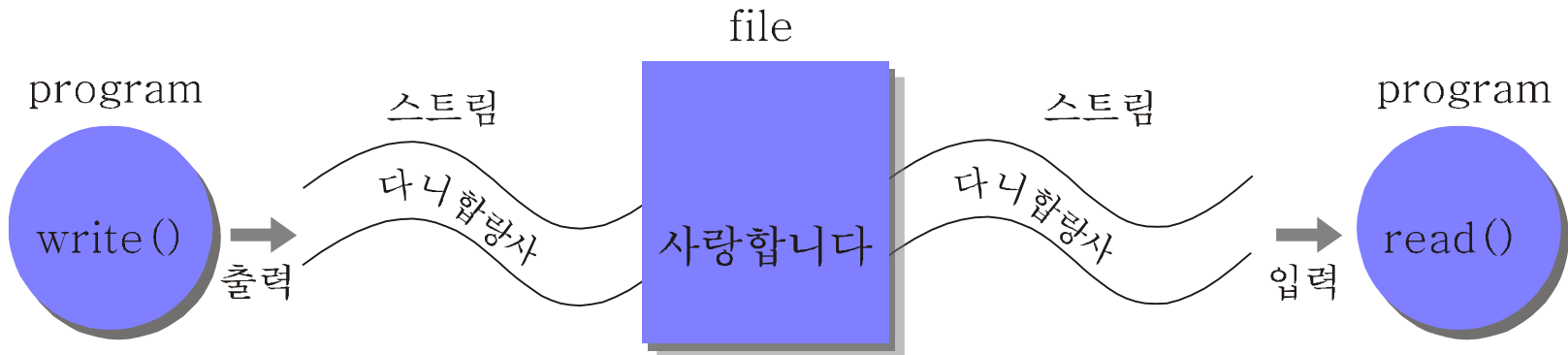


# 입출력 API

- 입출력이란
- `java.io` 패키지
- 바이트 스트림
- 문자 스트림

java.io Package를 제공함

입,출력을 위해서 스트림을 사용함(byte, character)



특 징

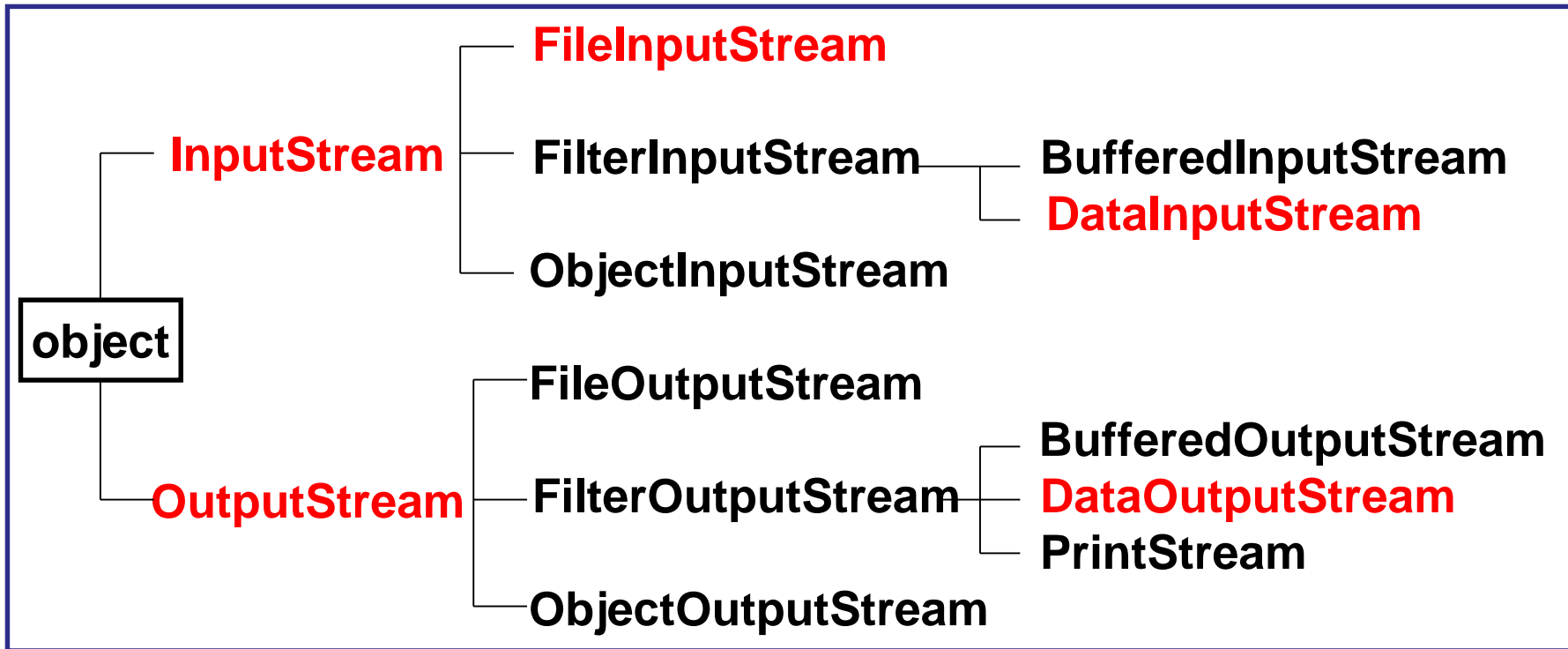
1. FIFO
2. 단방향이다.
3. 지연될 수 있다.



## IO 처리단위

	byte	Char
입 력	InputStream	Reader
출 력	OutputStream	Writer

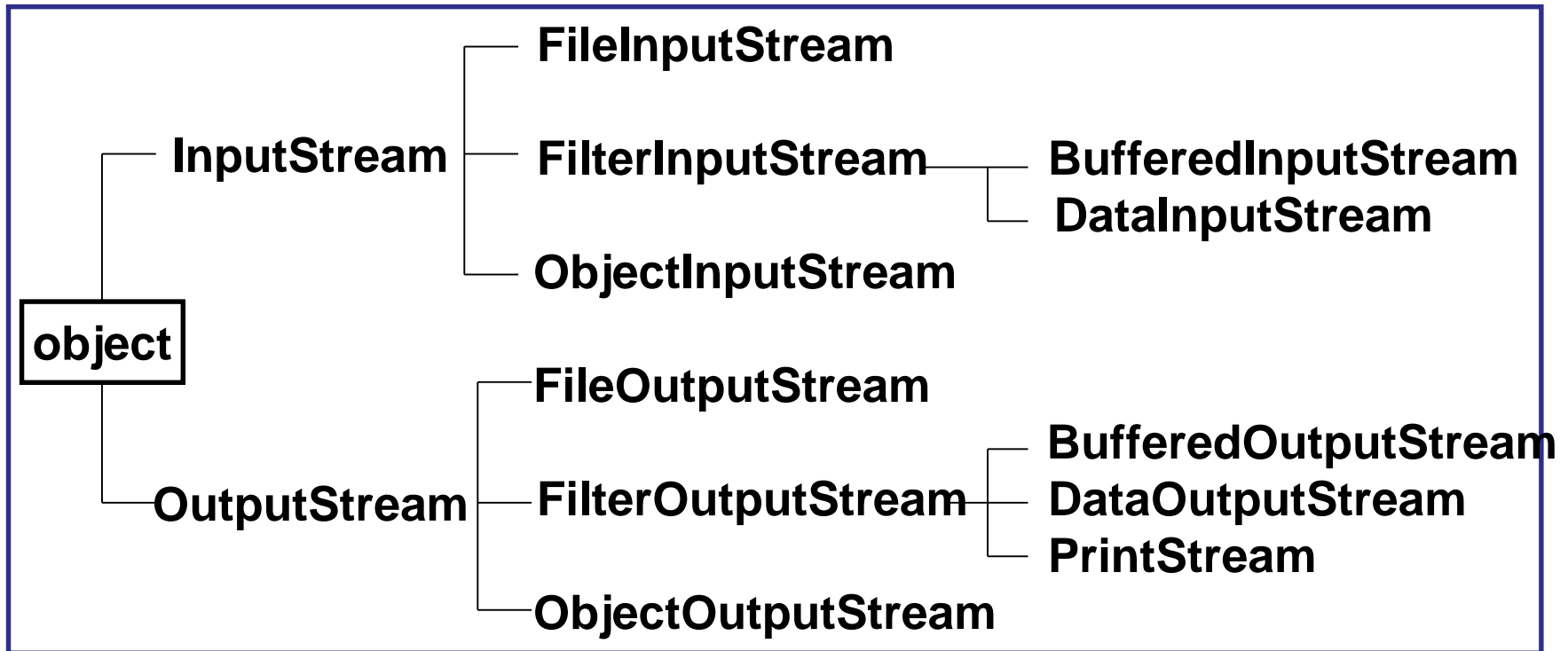
## 바이트 스트림



## 문자 스트림



## 바이트 스트림



## InputStream

`void close()` 입력 스트림을 닫는다

`int read()`

- ➔ 입력 스트림으로 부터 한 바이트를 읽어 `int` 형 값을 반환하다.  
읽은 바이트가 파일의 끝이면 `-1`을 반환

`int read(byte buffer[])`

- ➔ 입력 스트림으로부터 *buffer* 배열 크기만큼의 문자를 읽어 *buffer*에 저장

`int read(byte buffer[], int offset, int numbytes)`

- ➔ 입력 스트림으로부터 *numbytes*에 지정한 만큼의 바이트를 읽어 *buffer*의 *offset* 위치에 저장하고 읽은 바이트의 개수를 반환

`int available()` 현재 읽기 가능한 바이트의 수를 반환

`int skip(long numChars)`

- ➔ *numChars*로 지정된 바이트 수 만큼을 스킵하고 스킵 된 바이트의 수를 반환

## OutputStream

**void close()**    출력 스트림을 닫는다

**void flush()**    출력 버퍼에 저장된 모든 데이터를 출력 장치로 전송

**void write(int c)**    c의 하위 8비트를 스트림으로 출력

**void write(byte buffer[])**    buffer 배열에 있는 바이트들을 스트림으로 출력

**void write(byte buffer[], int index, int size)**

➔ buffer 배열의 index 위치부터 size 크기 만큼의 바이트들을 스트림으로  
출력

## FileInputStream

### construct

`FileInputStream (String filepath)`

`FileInputStream(File fileObj)`

## FileOutputStream

### construct

`FileOutputStream(String filepath)`

`FileOutputStream (String filepath, boolean append)`

`FileOutputStream (File fileObj)`

FileInputStream

파 일

FileInputStream

read()

FileOutputStream

FileOutputStream

write()

파 일



## BufferedInputStream

### construct

`BufferedInputStream(InputStream InputStream)`

`BufferedInputStream(InputStream InputStream, int bufSize)`

## BufferedOutputStream

### construct

`BufferedOutputStream(OutputStream outputStream)`

`BufferedOutputStream(OutputStream outputStream, int bufSize)`

## DataInputStream

### construct

`DataStream(InputStream InputStream)`

## DataOutputStream

### construct

`DataOutputStream(OutputStream outputStream)`

DataInput, DataOutput 인터페이스를 사용한 클래스  
기본 자료형 데이터를 바이트 스트림으로 입,출력

## DataInput

### method

**boolean readBoolean(boolean *b*)**

스트림으로부터 읽은 **boolean**을 반환

**byte readByte() throws IOException**

스트림으로부터 읽은 **byte**를 반환

**char readChar() throws IOException**

스트림으로부터 읽은 **char**를 반환

**double readDouble() throws IOException**

스트림으로부터 읽은 **double**을 반환

**float readFloat() throws IOException**

스트림으로부터 읽은 **float**를 반환

**long readLong() throws IOException**

스트림으로부터 읽은 **long**을 반환

**short readShort() throws IOException**

스트림으로부터 읽은 **short**를 반환

**int readInt() throws IOException**

스트림으로부터 읽은 **int**를 반환

## DataOutput

### method

<b>void writeBoolean(boolean b)</b>	<b>b</b> 를 스트림으로 출력
<b>void writeByte(int i)</b>	<b>i</b> 의 하위 8비트를 스트림으로 출력
<b>void writeBytes(String s)</b>	문자열 <b>s</b> 를 스트림으로 출력
<b>void writeChar(int i)</b>	<b>i</b> 의 하위 16비트를 스트림으로 출력
<b>void writeChars(String s)</b>	문자열 <b>s</b> 를 스트림으로 출력
<b>void writeDouble(double d)</b>	<b>d</b> 를 스트림으로 출력
<b>void writeFloat(float f)</b>	<b>f</b> 를 스트림으로 출력
<b>void writeInt(int i)</b>	<b>i</b> 를 스트림으로 출력
<b>void writeLong(long l)</b>	<b>l</b> 을 스트림으로 출력
<b>void writeShort(short s)</b>	<b>s</b> 를 스트림으로 출력

동 작 방 식

writeBoolean()  
writeByte()  
writeDouble()

DataOutputStream

파 일

DataInputStream

readBoolean()  
readByte()  
readDouble()

## Reader

`void close()` 입력 스트림을 닫는다

`int read()`

- ➔ 다음 문자를 읽어 반환한다. 입력 스트림에 읽을 문자가 없으면 대기한다.  
읽은 문자가 파일의 끝이면 -1을 반환

`int read(char buffer[])`

- ➔ 입력 스트림으로부터 *buffer* 배열 크기만큼의 문자를 읽어 *buffer*에 저장

`int read(char buffer[], int offset, int numChars)`

- ➔ 입력 스트림으로부터 *numChars*에 지정한 만큼의 문자를 읽어 *buffer*의 *offset* 위치에 저장하고 읽은 문자의 개수를 반환

`void mark(int numChars)`          입력 스트림의 현재의 위치에 mark 한다.

`boolean markSupported()`

- ➔ 현재의 입력 스트림이 `mark()`와 `reset()`을 지원하면 `true`를 반환

## Writer

**void close()**

출력 스트림을 닫는다

**void flush()**

출력 버퍼에 저장된 모든 데이터를 출력 장치로 전송

**void write(int c)**

c의 하위 16비트를 스트림으로 출력

**void write(char buffer[])** buffer 배열에 있는 문자들을 스트림으로 출력

**void write(char buffer[], int index, int size)**

➔ buffer 배열의 index 위치부터 size 크기만큼의 문자들을 스트림으로 출력

**void write(String s)**

문자열 s를 스트림으로 출력

**void write(String s, int index, int size)**

➔ 문자열의 index 위치부터 size 크기만큼의 문자들을 스트림으로 출력

## FileReader

### construct

`FileReader (String filepath)`

`FileReader(File fileObj)`

## FileWriter

### construct

`FileWriter(String filepath)`

`FileWriter (String filepath, boolean append)`

`FileWriter (File fileObj)`



## BufferedReader

**BufferedReader(Reader *inputStream*)**

**BufferedReader(Reader *inputStream*, int *bufSize*)**

**String readLine() throws IOException**

라인 단위로 읽어 온다.

## BufferedWriter

**BufferedWriter(Writer *outputStream*)**

**BufferedWriter(Writer *outputStream*, int *bufSize*)**

**void newLine() throws IOException**

새로운 라인에 출력



# Thread API

# Thread 생성자

## - 생성자

생성자	설명
Thread( )	스레드를 생성
Thread( String name )	Name 이라는 이름을 가진 스레드 생성
Thread( Runnable r )	Runnable 인터페이스를 구현한 클래스 객체로 스레드를 생성

# Thread 메서드

## - 메서드

메서드	설명
<code>static void sleep(long msec)</code>	msec 시간만큼 스레드를 대기시킨다.(1/1000 초)
<code>String getName( )</code>	스레드의 이름을 반환한다.
<code>void setName(String name)</code>	스레드의 이름을 설정한다.
<code>void start( )</code>	스레드를 시작한다.( <code>run( )</code> 메소드를 호출 )
<code>void run( )</code>	스레드가 해야할 일을 정의한다.
<code>void setPriority(int p)</code>	스레드의 우선순위를 설정한다.
<code>int getPriority( )</code>	설정된 우선순위를 반환한다.

# Thread 생성방법

## - Thread 를 생성하는 2가지 방법

Thread 클래스로부터 직접 상속 받아 스레드를 생성  
Runnable 인터페이스를 사용하는 방법

```
class ThreadTest extends
    Thread {

    ThreadTest tt = new
        ThreadTest();
    tt.start();

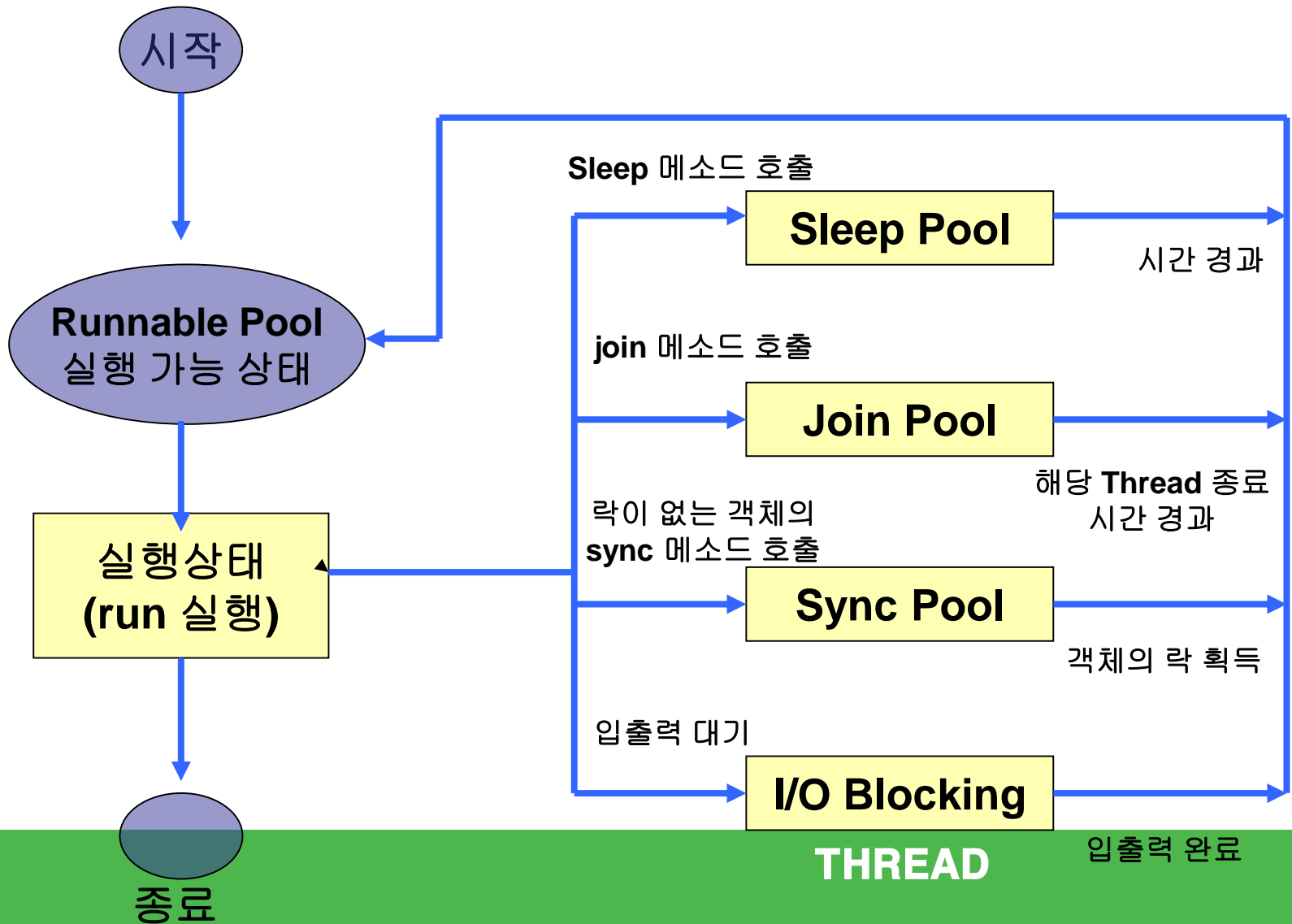
    public void run() {
    }
}
```

```
class ThreadTest implements
    Runnable {

    ThreadTest tt = new
        ThreadTest();
    Thread t = new Thread(tt);
    t.start();

    public void run() {
    }
}
```

# Thread 상태

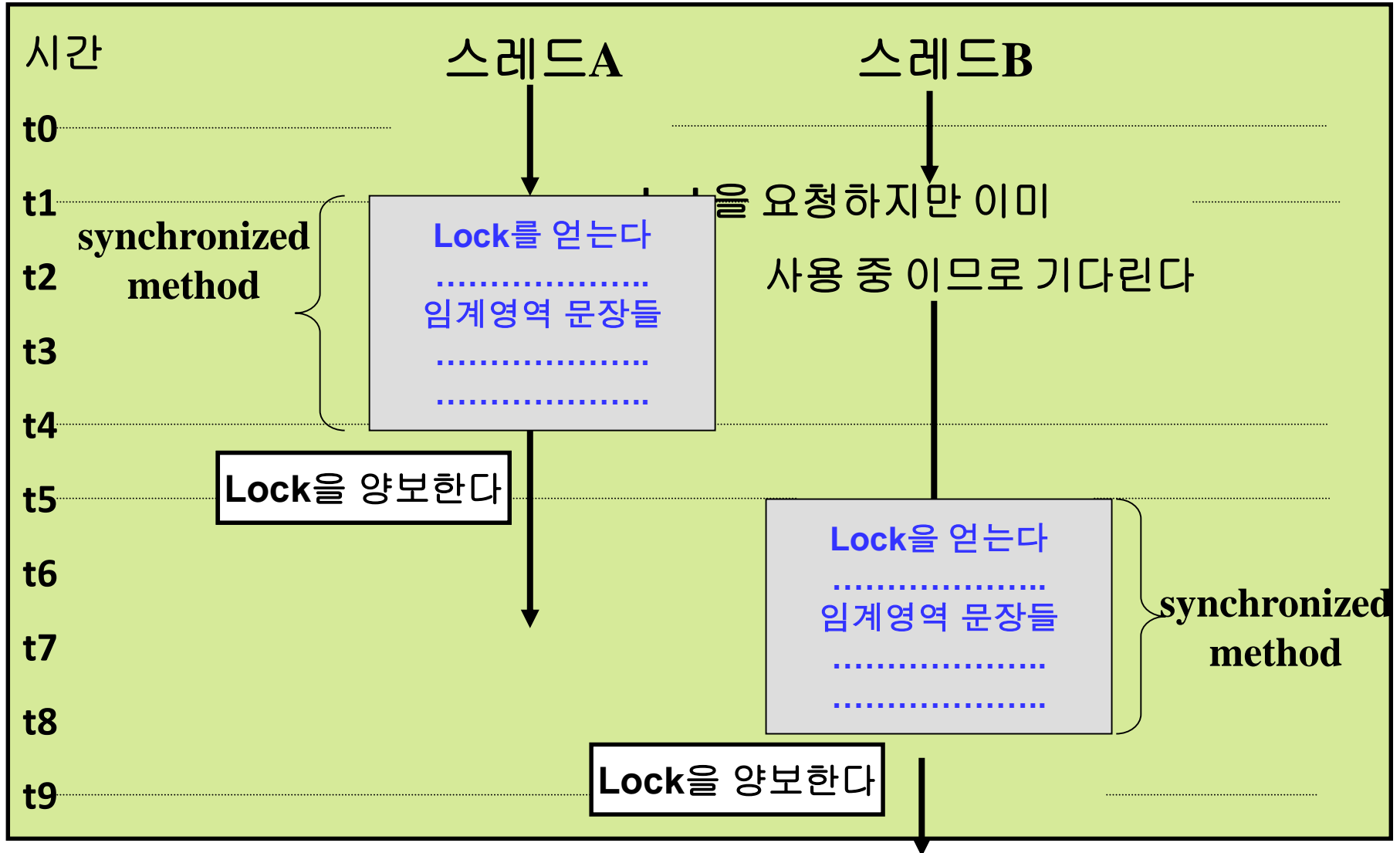




# Thread 우선순위

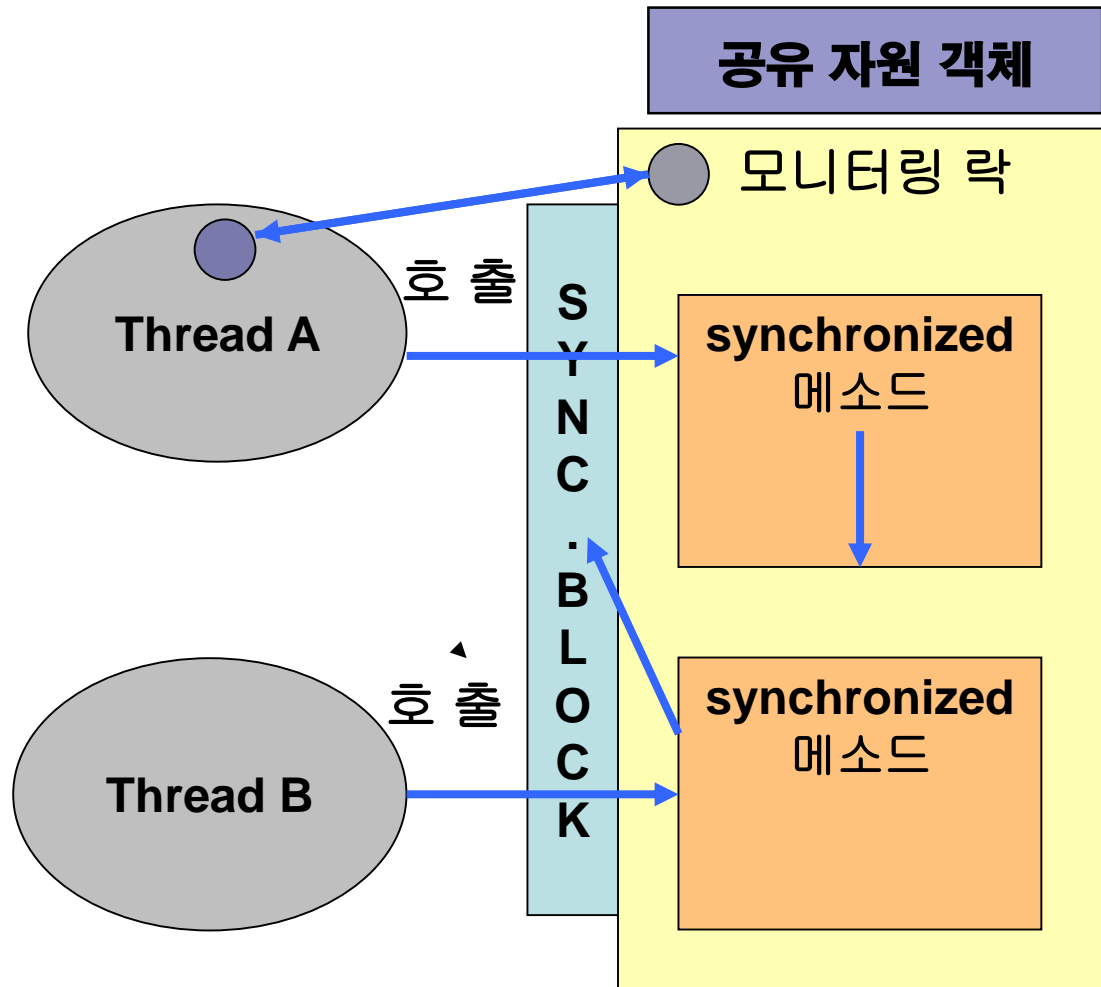
- 스레드에 우선 순위를 부여하여 우선 순위가 높은 스레드에게 실행의 우선권을 부여할 수 있다(**JVM**마다 다를 수 있다)
- **setPriority(int priority)** 메소드를 이용하여 우선 순위 부여
- **getPriority()** 메소드를 이용하여 설정된 우선 순위를 가져온다.
- 우선 순위를 지정하기 위한 상수 제공
  - static final int MAX\_PRIORITY**                      우선순위 10
  - static final int MIN\_PRIORITY**                      우선순위 1
  - static final int NORM\_PRIORITY**                      우선순위 5

# Thread 동기화





# Thread 동기화 동작원리





# Network API



# 목차

1. 관련 용어
2. **java.net** 패키지
3. 인터넷 주소와 **URL**
4. **TCP** 소켓 프로그래밍

# 관련용어 – 소켓(SOCKET)

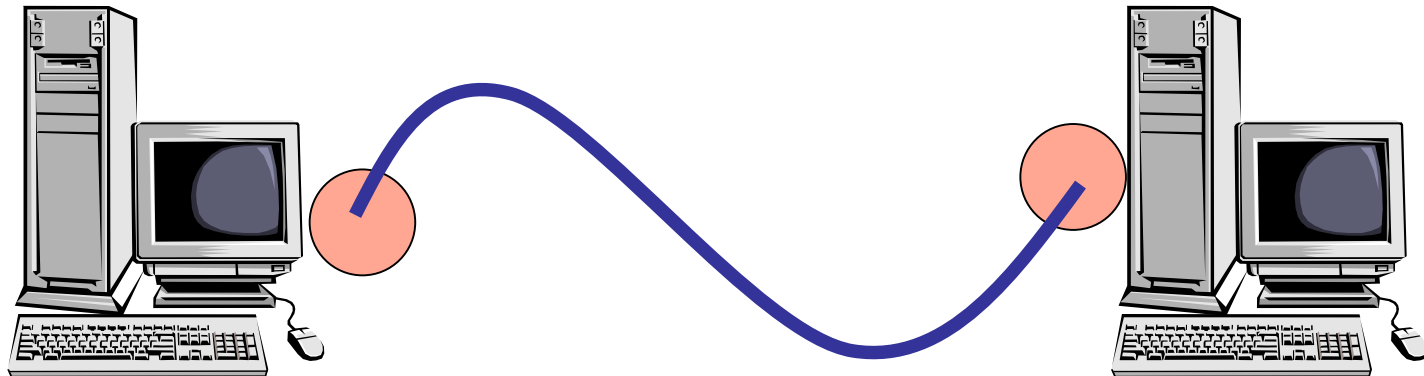
소켓 : 컴퓨터가 연결된 통신의 끝점.

소켓에 쓰는 일은 상대에게 데이터를 전달

소켓에서 읽는 일은 상대가 전송한 데이터를 수신하는 것.

자바에서 사용하는 소켓은 **TCP**와 **UDP**를 이용한다.

웹은 소켓 통신을 사용한다.(TCP)



# 관련용어 – 호스트, 포트

## 호스트(Host)

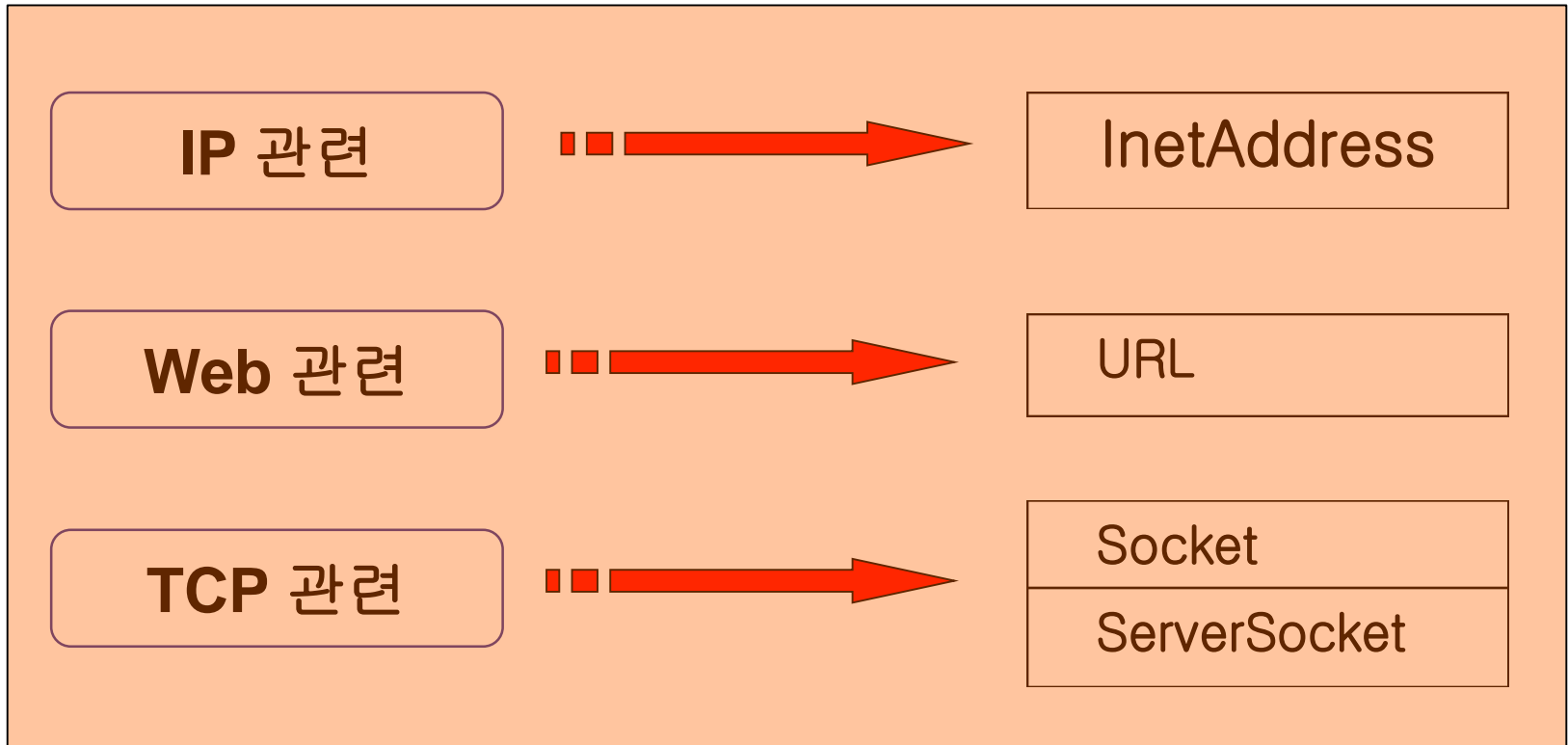
호스트 주소 : 하나의 컴퓨터에 할당된 고유 이름  
인터넷 상에서 **IP** 주소나 도메인명으로 나타난다.

## 포트(Port)

포트번호 : 한 컴퓨터에서 여러 서비스의 제공을 가능하게 함.  
한 호스트에 있는 여러 개의 서비스를 구분하기 위해서 사용

하나의 호스트는 여러 개의 포트를 가질 수 있다.  
서버 어플리케이션은 클라이언트의 요청을 위해 대기할 때 미리 정해진 포트를 감시한다.  
호스트는 전화번호에 포트는 내선번호에 비교할 수 있다.

# java.net API



# API - InetAddress

## method

**String getAddress()** 주소 정보를 나타내는 문자열을 반환

**String getHostName()** 컴퓨터 이름을 나타내는 문자열을 반환

**InetAddress getLocalHost()** 현재 컴퓨터를 나타내는 **InetAddress** 객체를 반환

**InetAddress getByName(String *hostName*)**

➔ *hostName*으로 지정된 컴퓨터를 나타 내는 **InetAddress** 객체를 반환

**InetAddress[] getAllByName(String *hostName*)**

➔ *hostName*으로 지정된 모든 컴퓨터(하나의 도메인 이름으로 여러 대의 컴퓨터를 사용하는 경우)를 나타내는 **InetAddress** 객체들의 배열을 반환

# API - URL [protocol://host:port/filename(경로포함) ]

## construct

URL(String protocol, String host, int port, String file)

URL(String protocol, String host, String file)

URL(String urlString)

## method

String getFile() → URL의 파일 이름을 반환

String getHost() → URL의 호스트 이름을 반환

String getPort() → URL의 포트 번호를 반환. 묵시적인 포트인 경우 -1 반환

String getProtocol() → URL의 프로토콜 이름을 반환

String toExternalForm() → 전체 URL의 문자열 객체를 반환

InputStream openStream() → 지정된 URL로부터 정보를 읽어들이기 위한 객체를 반환



# API - ServerSocket

## Server

### construct

`ServerSocket(int port)`

### method

`Socket accept()`

➔ 클라이언트의 요청을 받아들이는 다음 클라이언트와 연결된 소켓 클래스 객체를 반환함.

`void close()` 서버 소켓을 닫는다.

# API - Socket

## Client

### construct

`Socket(String hostName, int port)`

### method

`InputStream getInputStream()`

➔ 현재의 소켓과 관련된 `InputStream` 객체를 반환

`OutputStream getOutputStream()`

➔ 현재의 소켓과 관련된 `OutputStream` 객체를 반환

`void close()`    소켓을 닫는다

# API - Socket

## method

**InetAddress getInetAddress()**

➔ 현재 소켓에 연결된 컴퓨터의 주소를 반환

**InetAddress getLocalAddress()**

➔ 현재 소켓을 사용하고 있는 컴퓨터의 주소를 반환

**int getPort()**

➔ 현재 소켓에 연결된 컴퓨터의 포트 번호를 반환

**int getLocalPort()**

➔ 현재 소켓이 사용하고 있는 포트 번호를 반환

# 동작방식

