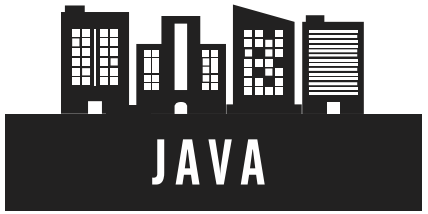
8253jang@daum.net
장희정 강사

Java Programming



01/ 자바 프로그래밍 시작

02/ 변수와 자료형

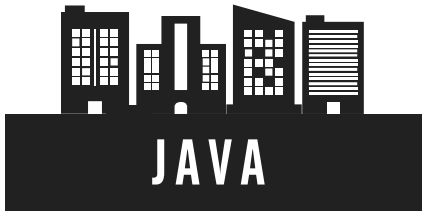
03/ 연산자

04/ 제어문

05/ 배열

06/ 객체지향 프로그래밍

07/ 제어자



08/ 상속

09/ 예외 처리

10/ 주요 API 활용

11/ 컬렉션 프레임워크

12/ 쓰레드

13/ 입출력 스트림

14/ 네트워크

01. 자바 프로그래밍 시작

01/ 자바 소개

02/ 개발 환경 구축

03/ 자바 프로그래밍

04/ 자바 가상 머신

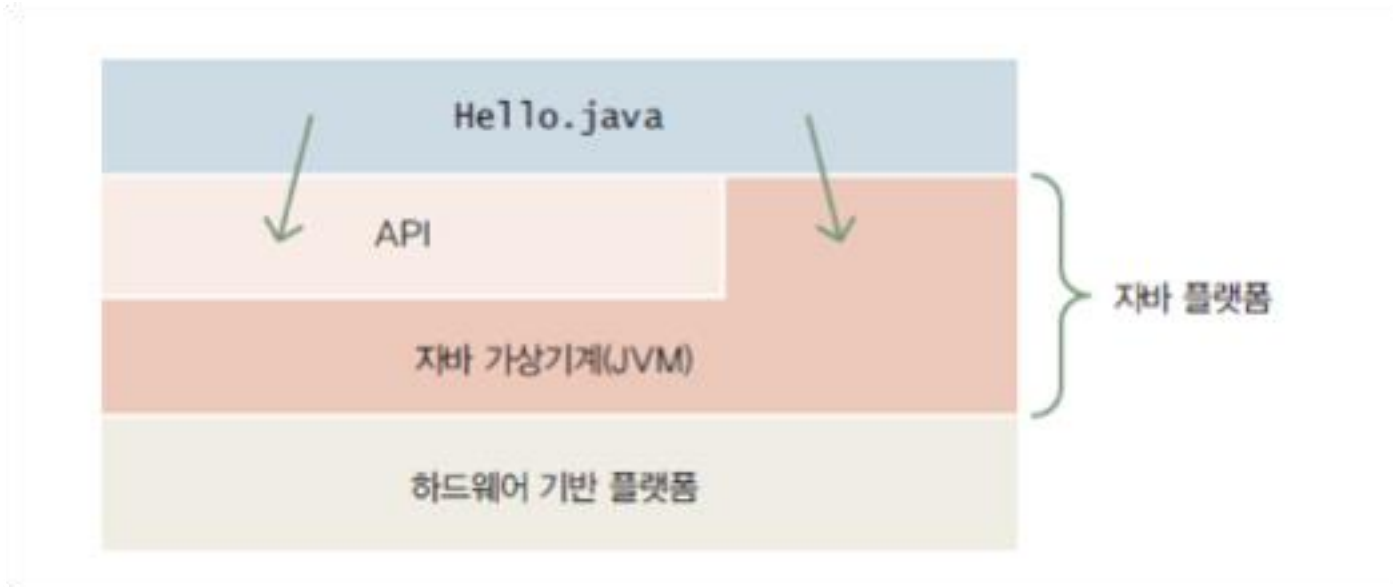
01. 자바 소개

- **자바란?**
  - 썬 마이크로 시스템스의 제임스 고슬링(James Gosling)
  - 1991년에 그린 프로젝트(Green Project)라는 이름으로
  - 가전 제품에 들어갈 소프트웨어를 위해 개발 시작
  - 1995년 자바 발표

- **자바의 특징**
  - Objected Oriented
    - 객체들 사이의 관계성에 따른 시스템의 구성 및 운영
    - 상속, 은닉, 다형성 등 객체지향 언어의 특징 모두 지원
    - S/W의 재사용성, 안전성, 유지보수의 용이성
  - 플랫폼 독립적인 언어 개발
    - 모든 플랫폼에서 호환성을 갖는 프로그래밍 언어 필요
  - 메모리 사용량이 적고 다양한 플랫폼을 가지는 가전 제품에 적용
    - 가전 제품 : 작은 량의 메모리를 가지는 제어 장치, 내장형 시스템 요구 충족

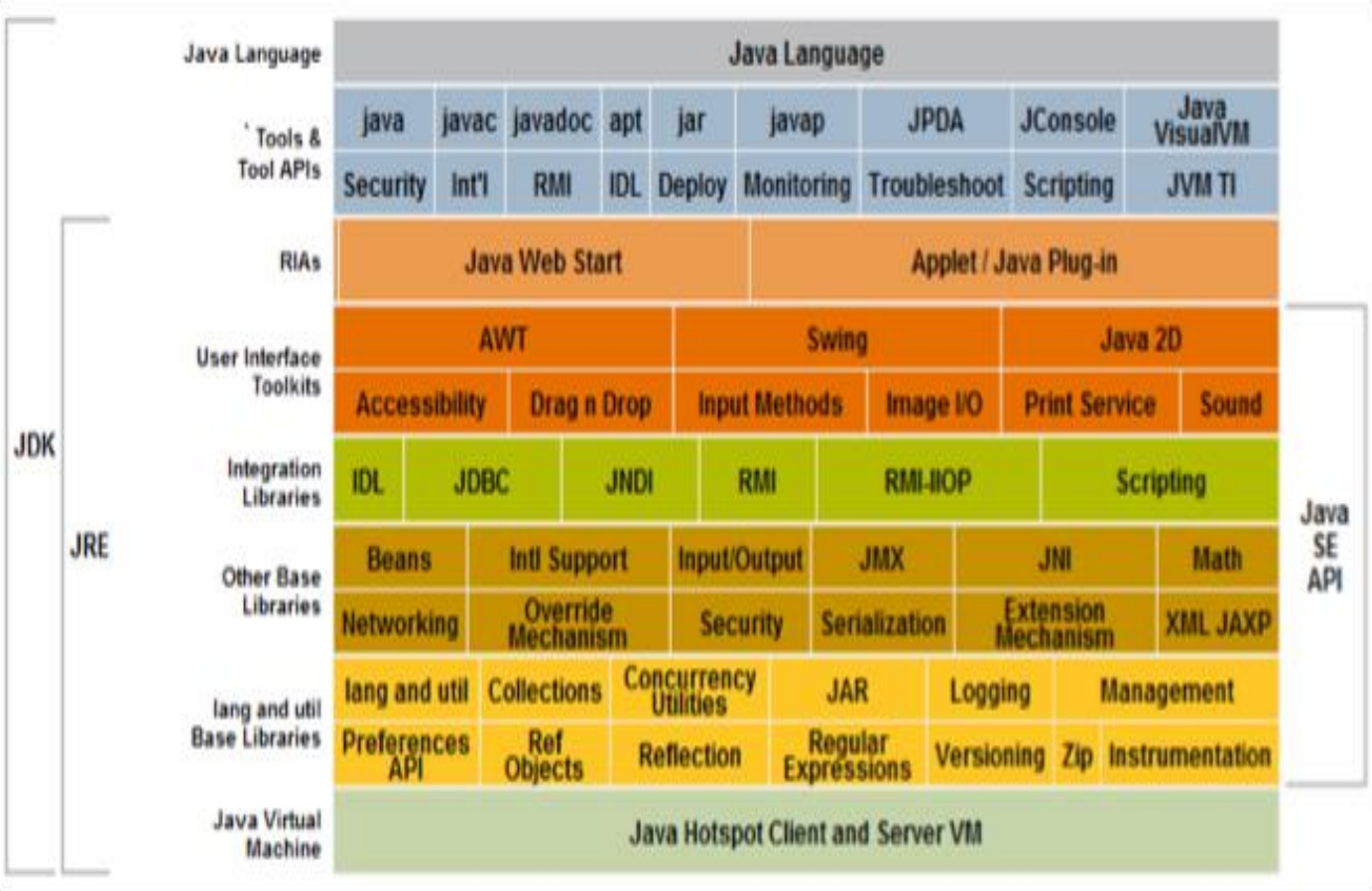
01. 자바 소개

- **자바 플랫폼**
  - Java SE (Standard Edition) : PC, 유닉스 등 범용 컴퓨팅 개발환경 지원 플랫폼
  - Java EE (Enterprise Edition) : 서버측 프로그래밍 개발환경 지원 플랫폼
  - Java ME(Micro Edition) : 핸드폰, 스마트 카드 등 소형기기 개발환경 지원 플랫폼



01. 자바 소개

- **자바 플랫폼 – Java SE**



- **JDK(Java Development Kit)**
- **JRE(Java Runtime Environment)**
- **Java SE API**

02. 개발 환경 구축

- **자바 개발도구(JDK) 설치**
  - http://www.oracle.com/technetwork/java/javase/downloads/index.html
  - 플랫폼에 맞는 프로그램 선택 다운로드

Java SE 8u151/ 8u152
Java SE 8u151 includes important bug fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release. Java SE 8u152 is a patch-set update, including all of 8u151 plus additional features (described in the release notes).
Learn more

- Installation Instructions
- Release Notes
- Oracle License
- Java SE Licensing Information User Manual
  - Includes Third Party Licenses
- Certified System Configurations
- Readme Files
  - JDK ReadMe
  - JRE ReadMe

JDK DOWNLOAD

Server JRE DOWNLOAD

JRE DOWNLOAD

Java SE Development Kit 8u152

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

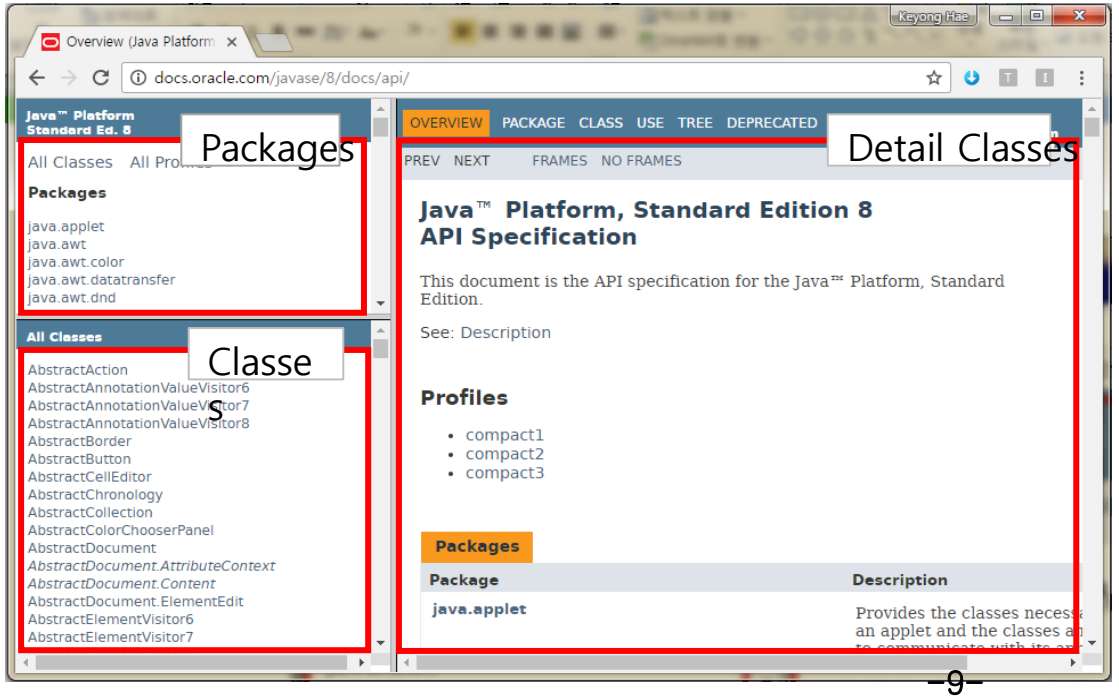
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.94 MB	jdk-8u152-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.88 MB	jdk-8u152-linux-arm64-vfp-hflt.tar.gz
Linux x86	168.99 MB	jdk-8u152-linux-i586.rpm
Linux x86	183.77 MB	jdk-8u152-linux-i586.tar.gz
Linux x64	166.12 MB	jdk-8u152-linux-x64.rpm
Linux x64	180.99 MB	jdk-8u152-linux-x64.tar.gz
macOS	247.13 MB	jdk-8u152-macosx-x64.dmg
Solaris SPARC 64-bit	140.15 MB	jdk-8u152-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.29 MB	jdk-8u152-solaris-sparcv9.tar.gz
Solaris x64	140.6 MB	jdk-8u152-solaris-x64.tar.Z
Solaris x64	97.04 MB	jdk-8u152-solaris-x64.tar.gz
Windows x86	198.46 MB	jdk-8u152-windows-i586.exe
Windows x64	206.42 MB	jdk-8u152-windows-x64.exe

02. 개발 환경 구축

- **자바 API 문서 설치**
  - 다운로드 후 Java SE 설치폴더 밑에 docs 폴더에 압축해제

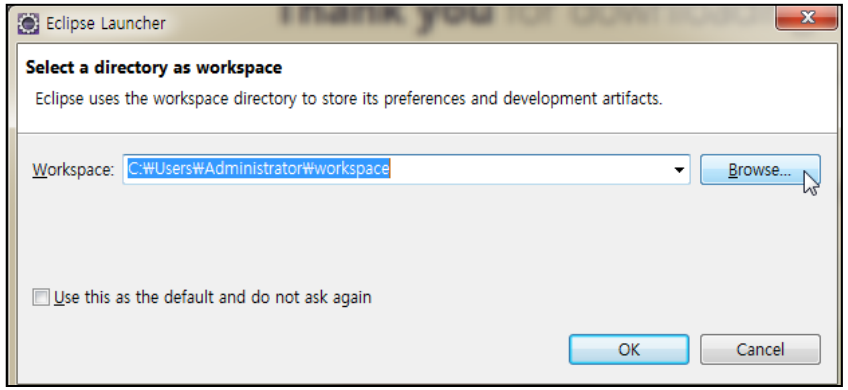
  - 온라인 : http://docs.oracle.com/javase/8/docs/api/



02. 개발 환경 구축

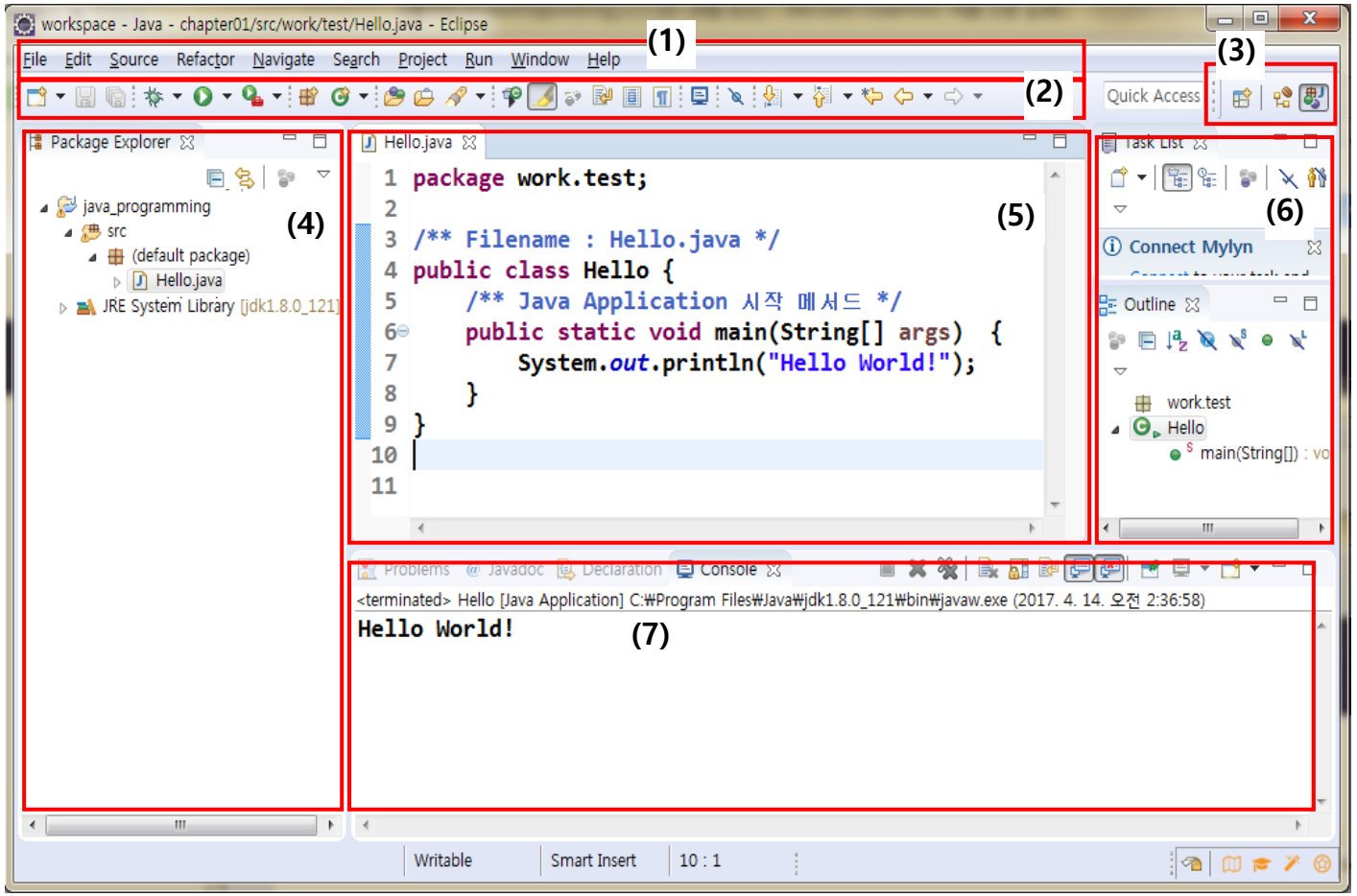
- **이클립스(Eclipse) 설치**
  - http://www.eclipse.org/downloads/

- **이클립스 워크스페이스(Workspace) 설정**



02. 개발 환경 구축

- 이클립스 화면 구성



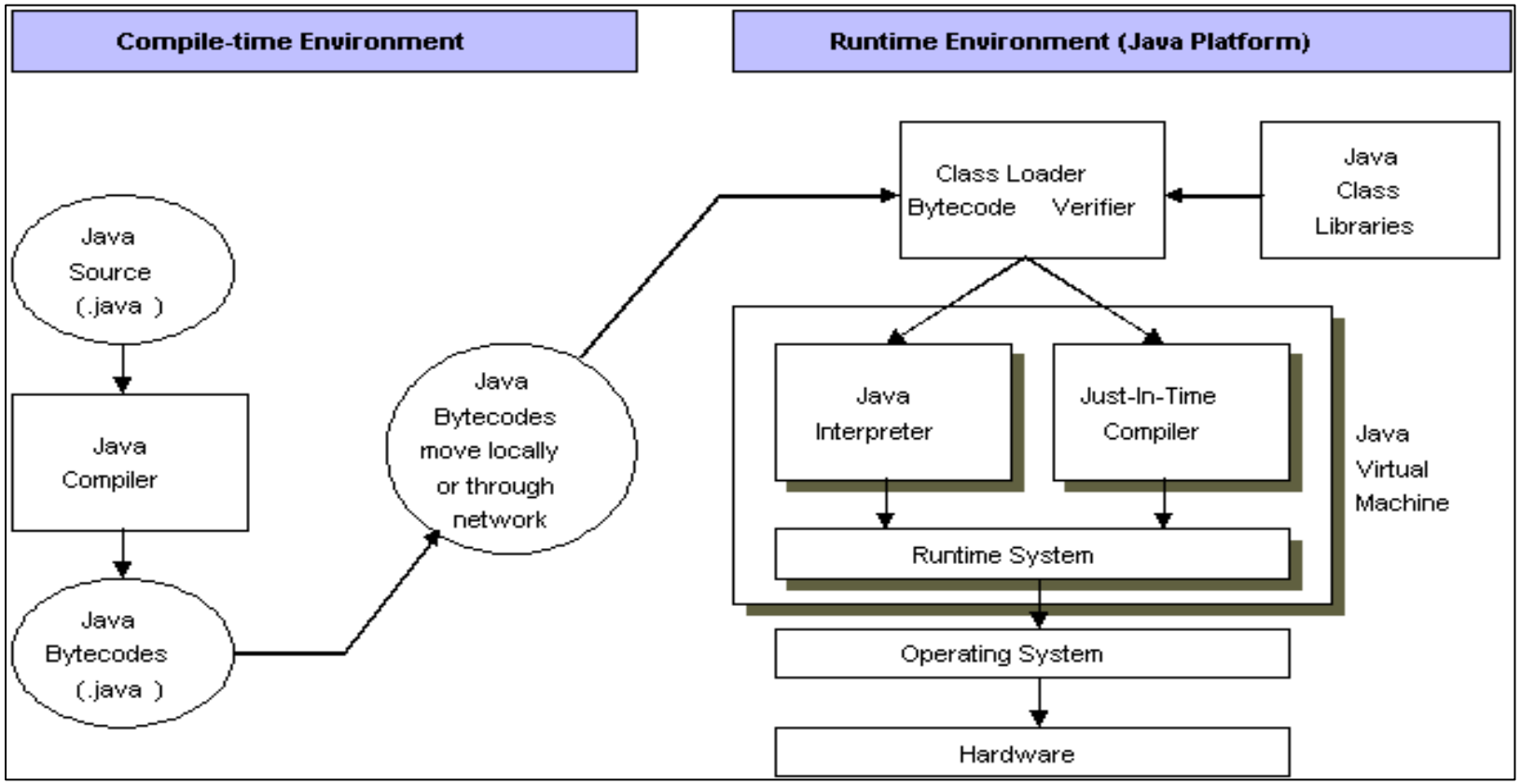
(1) Menu Bar
(2) Meun Icon
(3) Open Perspective
(4) Package Explorer
(5) Editor View
(6) Outline View
(7) Console View

03. 자바 프로그래밍

- **자바 프로그래밍 과정**
  - 자바 소스코드 파일 작성 : Hello.java
  - 자바 컴파일
    - Java 언어로 작성된 소스코드를 JVM이 인식가능한 코드인 바이트코드(ByteCode) 번역
    - Java 문법 및 구문오류 등을 체크하고, 생략된 부분의 코드 자동 추가
    - javac Hello.java
  - 자바 실행
    - ByteCode로 번역된 코드를 OS에 맞는 명령어로 번역하여 실행
    - 최적화 작업 수행 후 실행
    - java Hello

03. 자바 프로그래밍

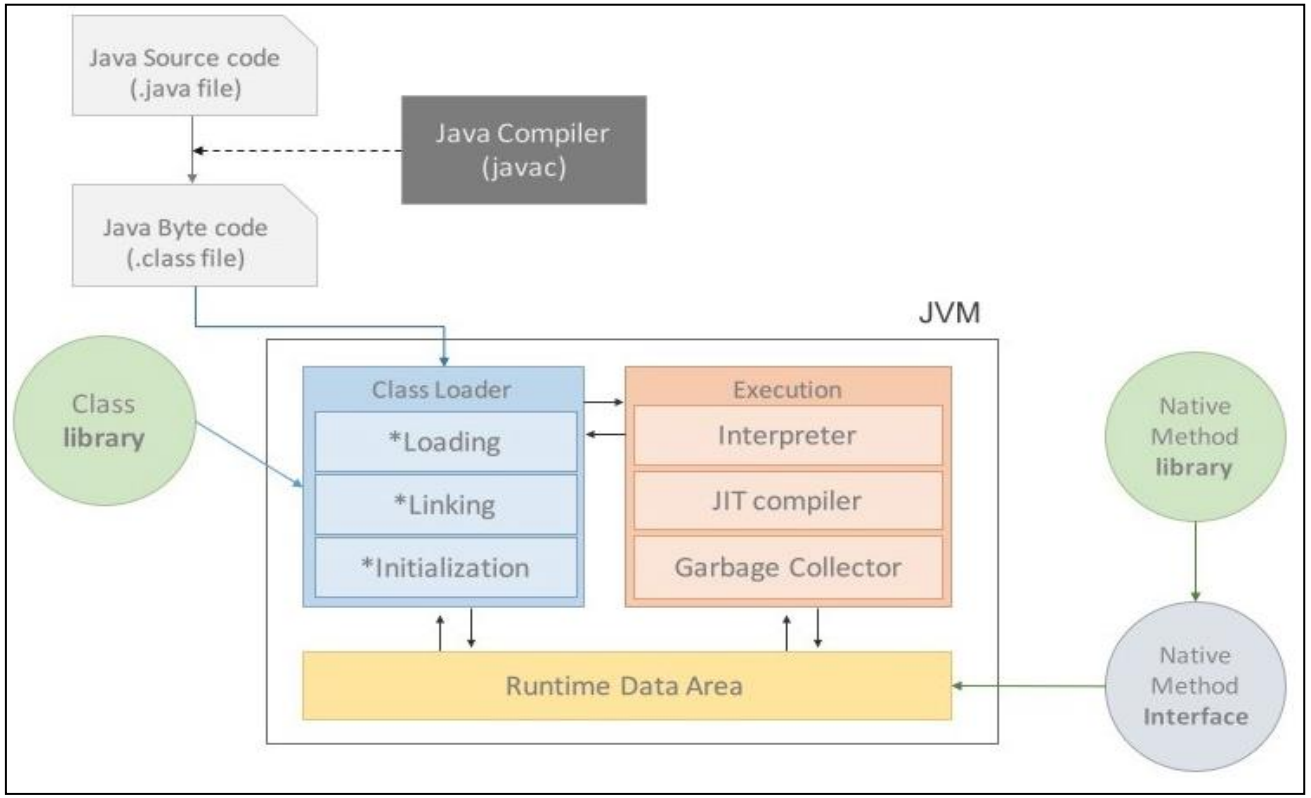
- **자바 프로그래밍 과정**



04. 자바 가상 머신(JVM : Java Virtual Machine)

- **자바 가상 머신 개념**
  - 자바 바이트코드(ByteCode)를 특정 플랫폼에 맞도록 해석해서 실행
  - 자바로 작성된 프로그램은 다양한 환경의 하드웨어 및 운영체제에서 동일하게 실행
  - Java 애플리케이션을 Class Loader를 통해 읽어들여 Java API와 함께 실행.
  - 메모리 관리, Garbage Collection 수행
  - 스택기반의 가상머신

- **자바 가상 머신 구조**
  - Class Loader Subsystem
  - Runtime Data Areas
  - Execution Engine



04. 자바 가상 머신(JVM : Java Virtual Machine)

- **프로그램 실행 과정**
  - 프로그램이 실행되면 JVM은 OS로부터 프로그램이 필요로 하는 메모리를 할당
  - JVM은 이 메모리를 용도에 따라 여러 영역으로 나누어 관리

  - Java 컴파일러(javac)가 Java 소스코드(.java)를 읽어 Java 바이트코드(.class)로 변환
  - Class Loader를 통해 .class파일들을 JVM으로 로딩
  - 로딩된 class파일들은 Execution Engine을 통해 해석
  - 클래스 로딩이 끝나면 JVM은 main메소드를 찾아 지역변수, 객체변수, 참조변수를 스택에 저장
  - 해석된 바이트코드는 Runtime Data Areas에 배치되어 실질적인 명령어 수행 (CPU).
  - 위와 같은 과정 속에서 JVM은 필요에 따라 Thread Synchronization과 GC같은 관리작업을 수행

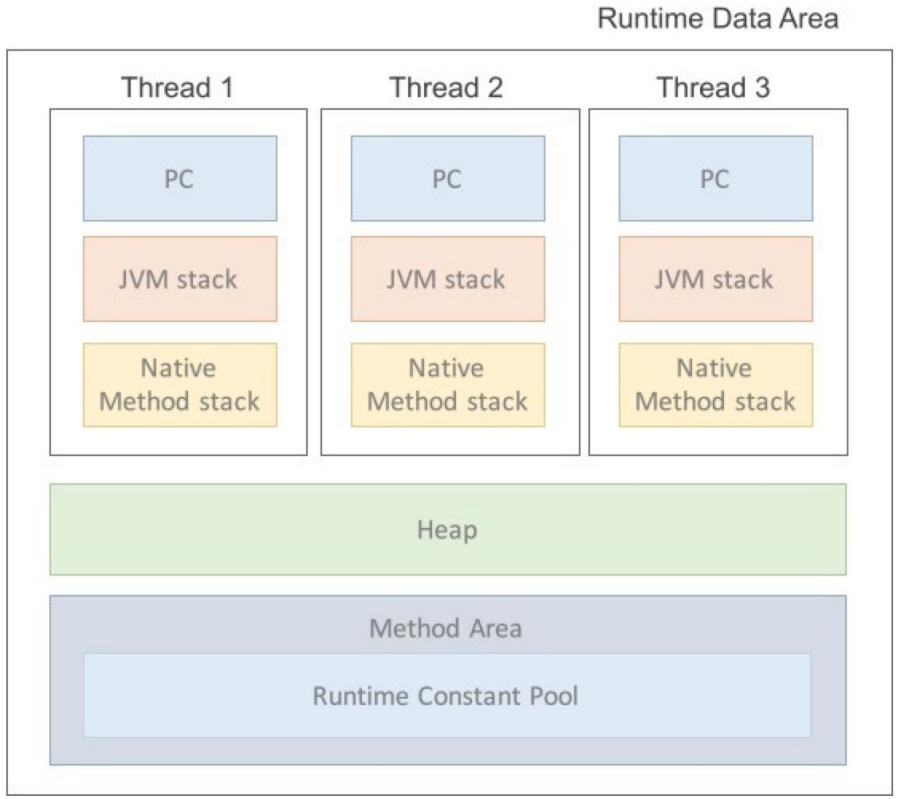
04. 자바 가상 머신(JVM : Java Virtual Machine)

- **클래스 로더(Class Loader)**
  - JVM내로 클래스들을 Load하고, Link를 통해 배치하는 작업을 수행하는 모듈
  - 런타임시 동적으로 클래스를 로드

- **Execution Engine**
  - ClassLoader를 통해 배치된 바이트코드를 명령어 단위로 실행
  - 인터프리터(Interpreter)
    - 자바 바이트 코드를 명령어 단위로 읽어서 실행
  - JIT(Just-In-Time)
    - 인터프리터 방식의 단점을 보완
    - 인터프리터 방식으로 실행하다 적절한 시점에 바이트코드 전체를 컴파일하여 네이티브 코드로 변경하고 그 이후에는 인터프리팅 하지 않고 네이티브 코드로 직접 실행하는 방식
- **Garbage Collector**
  - 메모리 관리 기능을 자동으로 수행
  - 애플리케이션이 생성한 객체의 생존 여부를 판단하여 더 이상 사용되지 않는 객체를 해제

04. 자바 가상 머신(JVM : Java Virtual Machine)

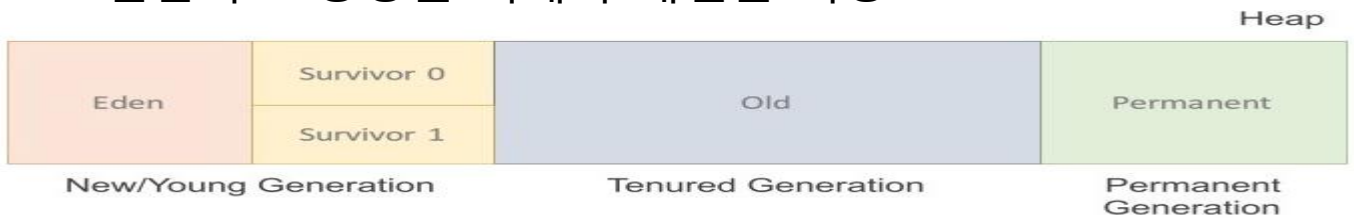
- **Runtime Data Areas**
  - JVM이 운영체제 위에서 실행되면서 할당받는 메모리 영역
  - ClassLoader에서 준비한 데이터들을 보관하는 저장소



- **PC 레지스터(PC Register)**
  - 쓰레드마다 현재 수행중인 JVM 명령의 주소를 저장
- **스택 영역(Stack Area)**
  - 메소드가 호출될 때마다 각각의 스택 프레임이 생성
  - 지역변수, 호출된 메소드의 매개변수, 반환값 등을 저장
- **Native Method Stack**
  - 바이트 코드가 아닌 실제 실행할 수 있는 기계어로 작성된 프로그램을 실행시키는 영역
  - Java가 아닌 다른 언어로 작성된 코드를 위한 공간
- **메소드 영역(Method Area)**
  - 클래스 정보를 처음 메모리 공간에 로드할 때 초기화 대상(바이트 코드)을 저장
  - Runtime Constant Pool(상수 자료형을 저장)
  - 필드/메소드 정보, 클래스/인터페이스 등 타입 저장

04. 자바 가상 머신(JVM : Java Virtual Machine)

- **힙 영역(Heap)**
  - 객체를 저장하는 가상 메모리 공간, 자동 초기화
  - new 연산자로 생성된 객체와 배열을 저장



  - Permanent Generation
    - 생성된 객체들의 주소값과 클래스 로더에 의해 로드된 클래스/메소드 등의 메타 정보가 저장되는 영역
    - Reflection을 사용하여 동적으로 클래스가 로딩되는 경우 사용
  - New/Young 영역
    - Eden : 객체들의 최초로 생성되는 공간
    - Survivor 0 / 1 : Eden에서 참조되는 객체들이 저장되는 공간
  - Old 영역
    - New 영역에서 일정시간 참조되어 살아남은 객체들이 저장되는 공간
    - Eden 영역에 객체가 가득차면 첫번째 GC(Minor GC) 수행, Eden 영역에 있는 값들은 Survivor 1 영역에 복사하고 이 영역을 제외한 나머지 영역의 객체를 삭제

02. 변수와 자료형

01/ 변수의 개념

02/ 변수의 종류

03/ 자료형

01. 변수(Variable)의 개념

- **변수 개념**
  - 하나의 값을 저장할 수 있는 메모리 공간
  - 변수의 데이터 타입에 맞는 메모리 공간이 할당
  - 변수를 사용하기 전에 반드시 변수 선언을 해야 함

- **변수 선언**

형식

```
[접근지정 제어자] [사용 제어자] 자료형  변수명 [ = 초기값 ];
```

- **식별자(identifier) 규칙**
  - 식별자는 클래스, 변수, 메서드, 상수, 패키지의 이름을 의미
  - 대소문자 구분, Camel 표기법
  - 숫자 시작 불가, 예약어(keyword) 사용 불가
  - _(밑줄), $(달러) 만 사용 가능

02. 변수의 종류

- **멤버변수 – 필드(Field)**
  - 클래스의 멤버로 선언한 변수, 클래스 멤버인 메서드, 생성자 내부에서 사용 가능한 변수
  - 각 데이터 타입의 기본값으로 자동 초기화
    - 논리형 : false
    - 문자형 : '\u0000'
    - 정수형 : 0
    - 실수형 : 0.0
    - 참조형 : null
  - 인스턴스 필드
    - 객체 생성시 해당 객체의 메모리 영역에 할당되는 멤버변수
    - 선언 : [접근지정 제어자][사용제어자] 타입 변수명 [= 명시적초기값];
    - 사용 : 참조변수명.인스턴스멤버변수명
  - 클래스 멤버변수
    - 클래스 메모리 로딩시에 할당되어 해당 인스턴스간 사이에 공유변수
    - 선언 : [접근지정 제어자] **static** 타입 변수명 [= 명시적초기값];
    - 사용 : 클래스이름.클래스멤버변수명

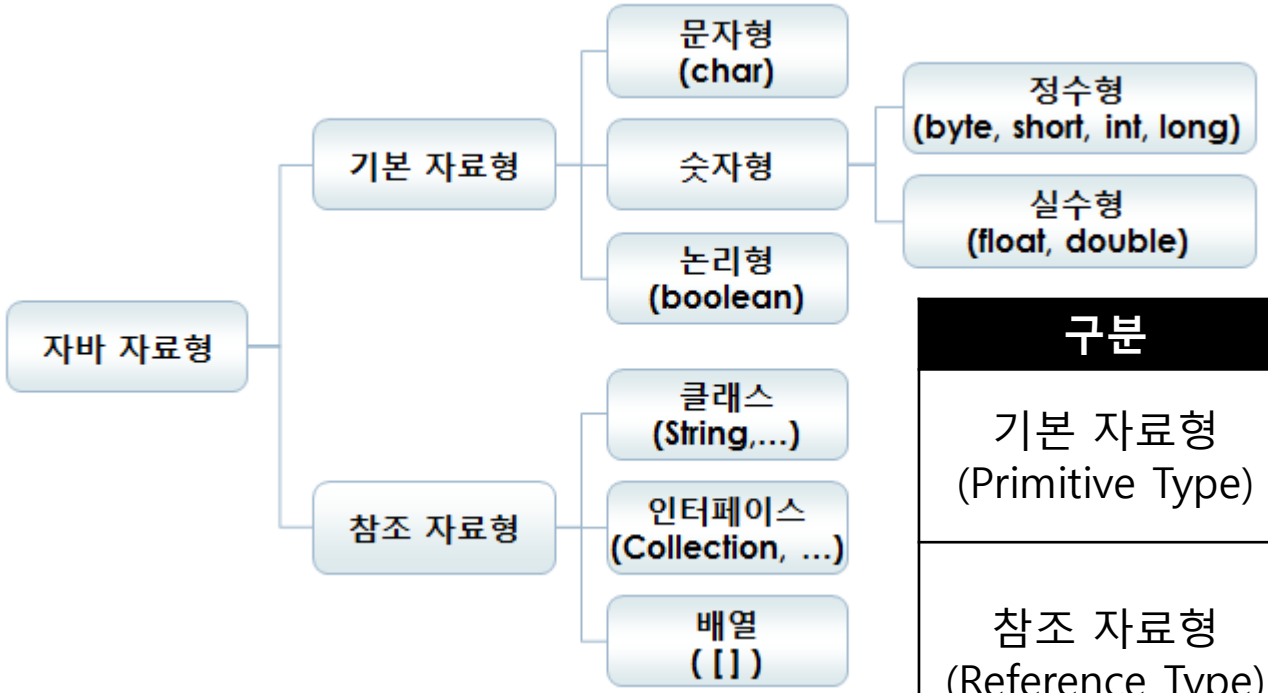
02. 변수의 종류

- **지역변수**
  - 메서드 내부, 생성자 내부, 제어문 블록 내부에 선언한 변수
  - 메서드, 생성자의 매개변수
  - 해당 블록 내부에서만 사용 가능
  - 자동 초기화 되지 않으므로 사용전 명시적 초기화 필요

03. 자료형(Data Type)

- **자료형의 개념**
  - 자료형의 개념

- **자료형의 종류**



구분	설명
기본 자료형 (Primitive Type)	• 자바 컴파일러에 의해서 해석되는 자료형 • 실제 **값**을 값는 자료형 • 자바에서 여러 형태의 타입을 미리 정의하여 제공
참조 자료형 (Reference Type)	• 자바 API에서 제공되거나 프로그래머에 의해서 정의된 클래스를 자료형으로 선언하는 경우 • 값이 아닌 객체의 **주소**를 가지고 있는 자료형 • 배열, 클래스 인터페이스

03. 자료형(Data Type)

- **기본형(Primitive Type)**
  - 기본적인 원시값을 기억하는 자료형
  - 자바에서 기본 자료형은 반드시 사용하기 전에 선언
  - OS에 따라 자료형의 길이가 변하지 않음

구분	키워드	크기 및 범위		설명
논리형	boolean	8 bit	true, false	true, false
문자형	char	16 bit	0 ~ 65,535	Unicode, ' 이용
정수형	byte	8 bit	-128 ~ 127	
	short	16 bit	-32,768 ~ 32,767	
	int	32 bit	-2,147,483,648 ~ 2,147,483,647	
	long	64 bit	-9223372036854775808L ~ 9223372036854775807L	
실수형	float	32 bit	-3.4E38 ~ 3.4E38	IEEE 754 – 1985 표준
	double	64 bit	-1.7E308 ~ 1.7E308	IEEE 754 – 1985 표준

03. 자료형(Data Type)

- **참조형(Reference Type)**
  - 객체의 참조값(reference value)을 갖는 자료형
  - 기본적으로 java.lang.Object를 상속
  - 클래스형(Class Type), 인터페이스형(Interface Type), 배열형(Array)

03. 연산자

01/ 연산자의 개념

02/ 연산자의 종류

03/ 형변환 연산자

01. 연산자의 개념

- **연산자(Operator)**
  - 어떠한 기능을 수행하는 기호
  - 단항 연산자, 이항 연산자, 삼항 연산자, 산술 연산자, 비교 연산자, 대입 연산자

- **피연산자(Operand)**
  - 연산자의 작업에 대상이 되는 것
  - 변수, 상수, 리터럴, 수식

- **연산자의 종류**
  - 단항 연산자 : 피연산자가 하나인 연산자
    - +, -, (자료형), ++, --, ~, !
  - 이항 연산자 : 피연산자가 두개인 연산자
    - +, -, *, /, %, <<, >>, >>>, >, <, >=, <=, ==, !=, &, |, &&, ||, ^
  - 삼항 연산자 : 피연산자가 세개인 연산자
    - (조건식) ? expression1 : expression2

02. 연산자의 종류

- **연산자의 종류**

종류	연산자
산술 연산자	+ - * / %
대입 연산자	=
단축 대입연산자	+= -= *= /= %=
비교 연산자	< > <= >= == !=
논리 연산자	& | && || !
비트 연산자	& | ^ ~ << >> >>>
증감 연산자	++ --
부호 연산자	+ -
조건 연산자	(조건식) ? expression1 : expression2
형변환 연산자	(데이터타입)

02. 연산자의 종류

- **연산자의 우선순위**

종류	연산방향	연산자	우선순위
단항 연산자	←	++ -- + - ~ ! (데이터타입)	높음
산술 연산자	→	* / %	
		+ -	
		<< >> >>>	
비교 연산자	→	< > <= >= instanceof	
		== !=	
논리 연산자	→	&	
		^	
		|	
		&&	
		||	
삼항 연산자	→	? :	
대입 연산자	←	= *= /= %= += -= <<= >>= >>>= &= ^= !=	낮음

03. 형변환(Type Casting)

- **형변환 개념**
  - 선언시의 자료형이 아닌 다른 자료형으로 변환하는 것
  - boolean을 제외한 7개의 기본형간 형변환 가능

- **묵시적 형변환(자동 형변환)**
  - 작은 타입의 값은 큰 타입으로 자동 형변환

- **명시적 형변환(강제 형변환)**
  - 큰 자료형의 값은 작은 자료형으로 형변환되지 않으므로 명시적 형변환
  - 자료형 변수명 = **(자료형)** 피연산자;
  - 데이터 손실 발생 가능

04. 제어문

01/ 제어문의 개요

02/ 조건문

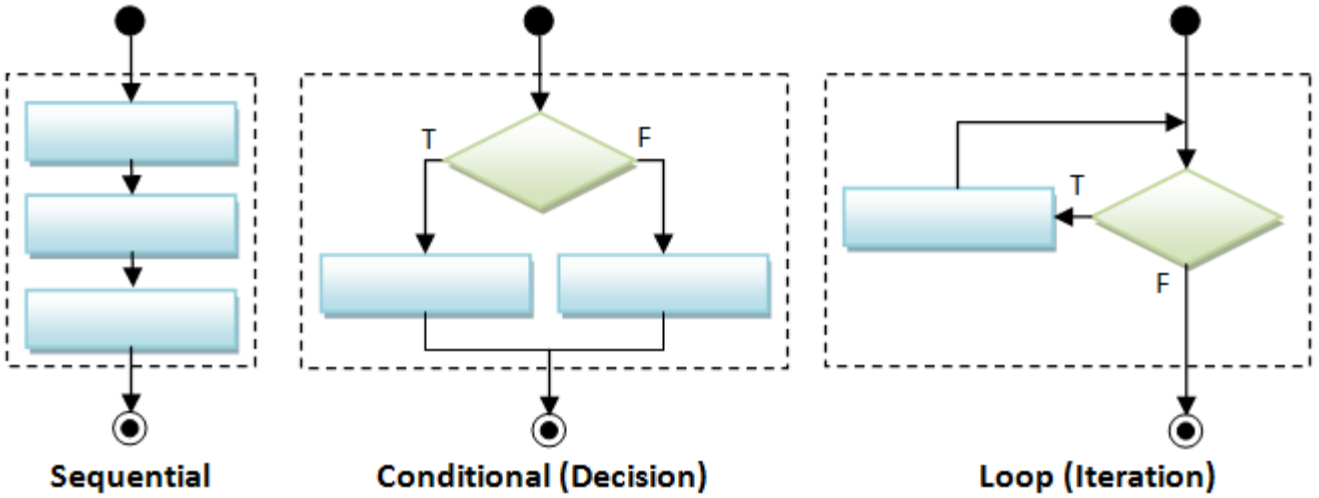
03/ 반복문

04/ 분기문

01. 제어문의 개요

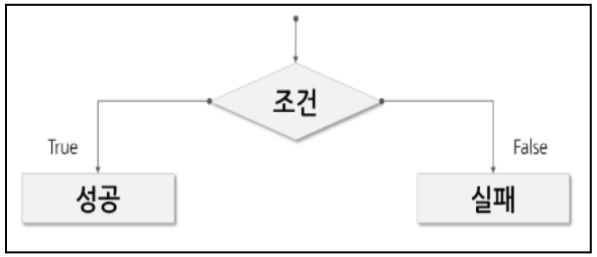
- **제어문의 개념**
  - 조건의 결과에 따라서 프로그램의 수행 순서를 제어하거나 문자들의 수행횟수를 조정하는 문장
  - 조건문, 반복문, 분기문

- **제어문의 종류**

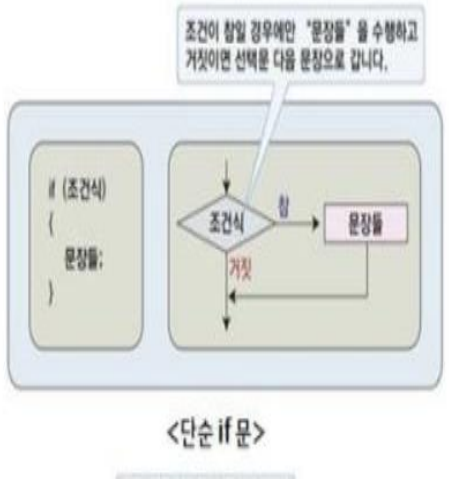


02. 조건문

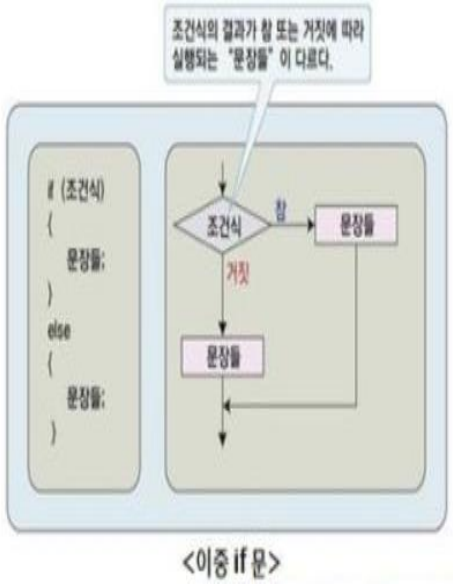
- **조건문의 개념**
  - 조건 결과에 따라 프로그램의 수행 순서를 제어



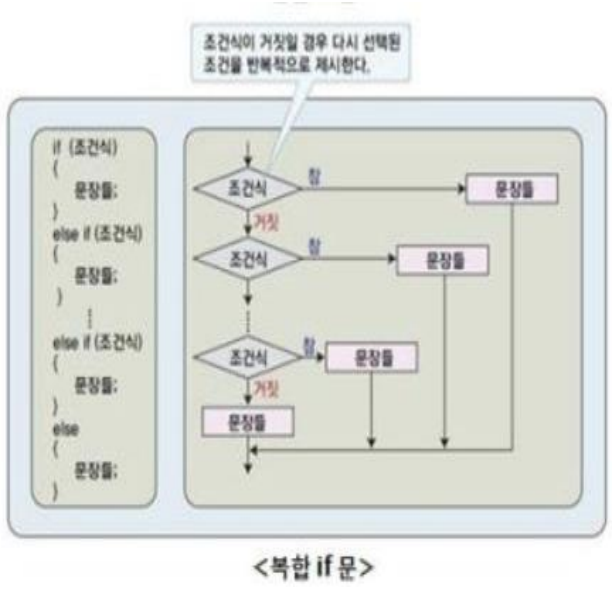
- **조건문의 종류**



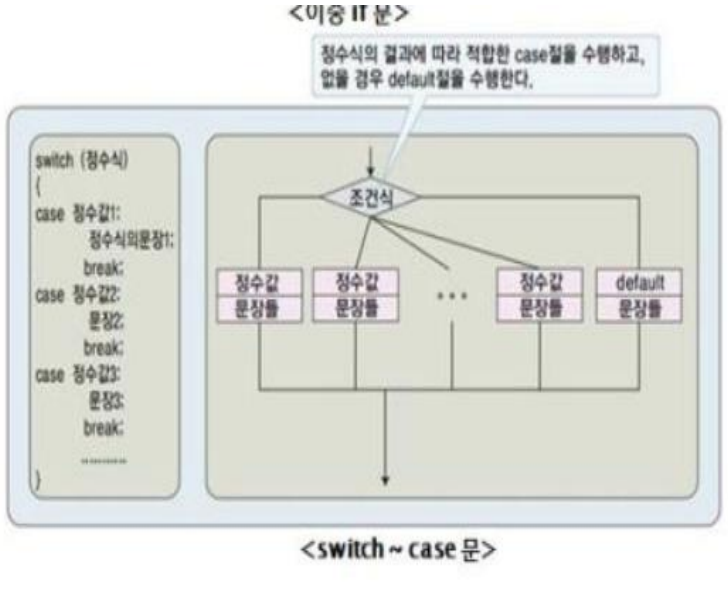
<단순 if 문>



<이중 if 문>



<복합 if 문>



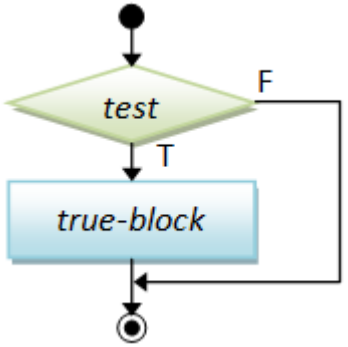
<switch ~ case 문>

02. 조건문

- **if 문**
  - 조건식의 결과는 boolean 타입이어야 함

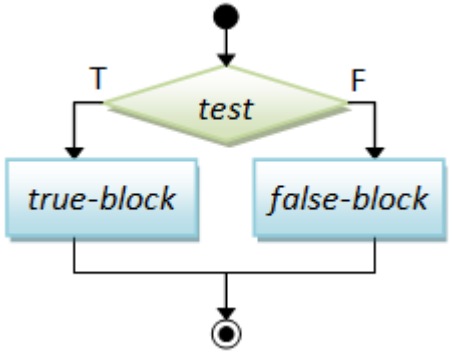
형식 #1

```
if(조건식) {
        조건식 결과가 true인 경우에 실행하는 문장;
}
```



형식 #2

```
if(조건식) {
        조건식 결과가  true인 경우에 실행하는 문장;
}
else {
        조건식 결과가 false인 경우에 실행하는 문장;
}
```

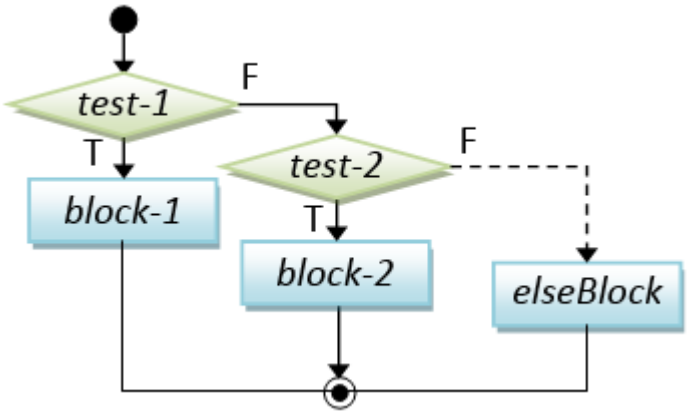


02. 조건문

- **if 문**
  - 조건식의 결과는 boolean 타입이어야 함

형식 #3

```
if(조건식1) {
        조건식1 결과가  true인 경우에 실행하는 문장;
}
else if(조건식2) {
        조건식2 결과가  true인 경우에 실행하는 문장;
}
else {
        모든 조건식 결과가 false인 경우에 실행하는 문장;
}
```

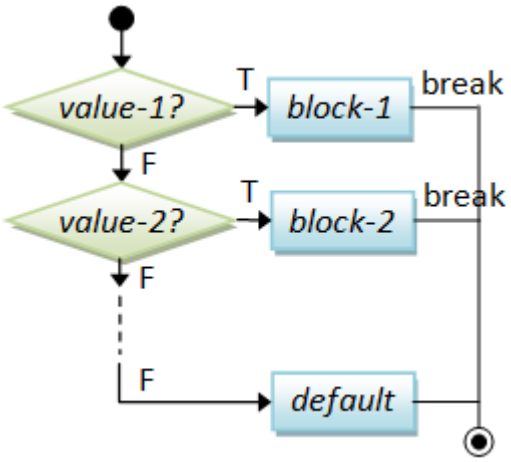


02. 조건문

- **switch ~ case 문**
  - 조건식 결과는 int, byte, short, char, String(JDK7 이상부터) 타입만 가능
  - 해당 값에 해당되는 case 구문 부터 순차적으로 수행
  - break; 구문을 만나면 switch 구문을 벗어남

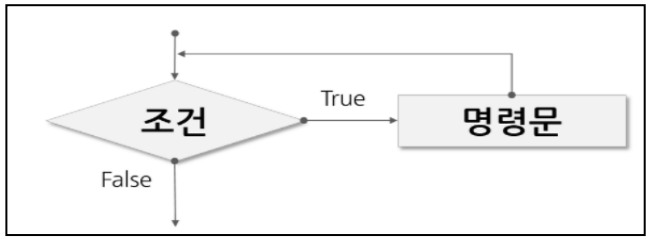
형식

```
switch(조건식) {
        case 값1:
                조건식 결과가 값1과 같은 경우 수행하는 문장;
                [break;]
        case 값2:
                조건식 결과가 값2와 같은 경우 수행하는 문장;
                [break;]

        [default :
                조건식 결과에 해당하는 case가 없는 경우 수행하
는 문장;]
}
```

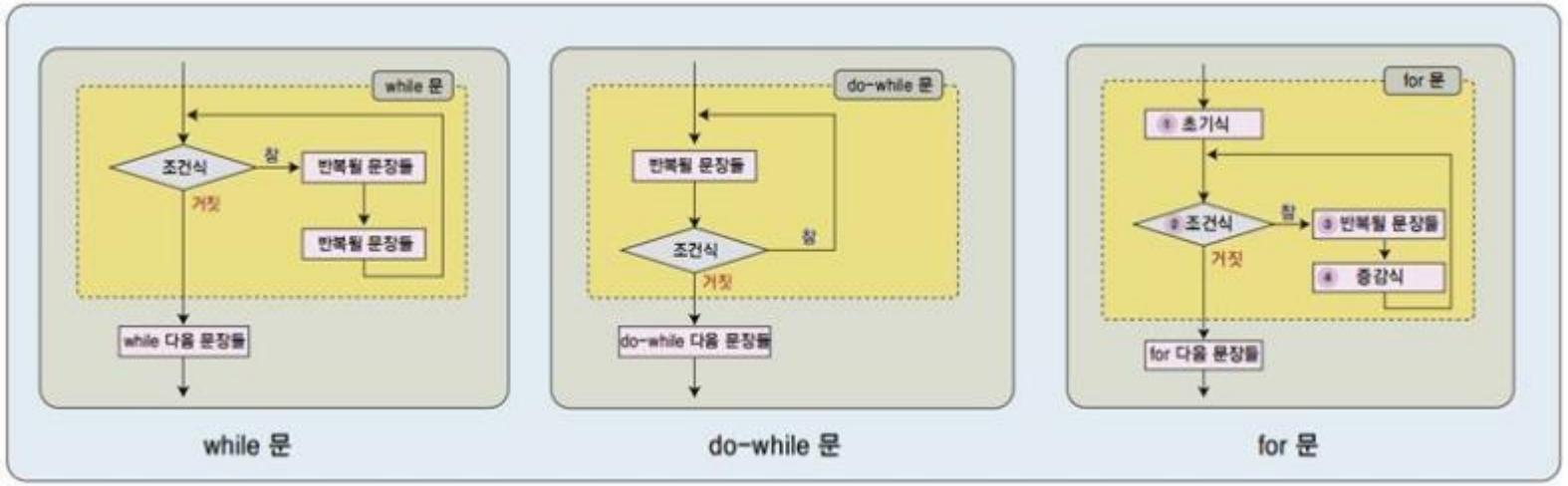


03. 반복문

- **반복문의 개념**
  - 조건 결과에 따라 하나 이상의 문장을 반복 수행하기 위한 문장
  - 조건식이 true 인 동안 반복 수행



- **반복문의 종류**

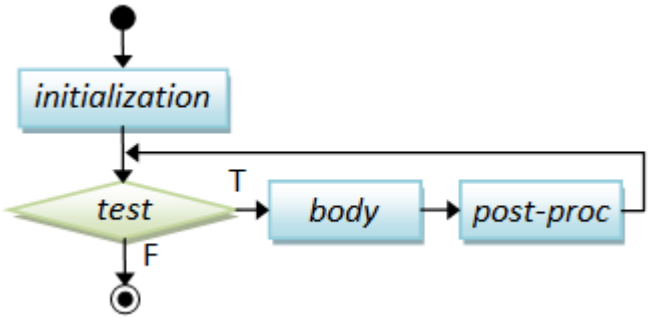


03. 반복문

- **for 문**
  - 조건식을 먼저 검증 후 반복 수행여부 결정

형식

```
for(초기식; 조건식; 증감식) {
        조건식 결과가 true일 동안 반복해서 수행할 문장;
}
```

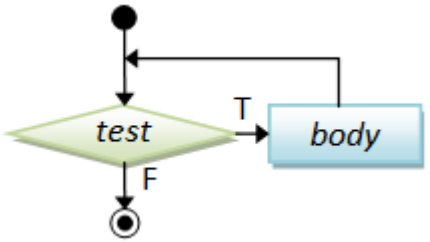


03. 반복문

- **while 문**
  - 조건식을 먼저 검증 후 반복 수행여부 결정

형식

```
while(조건식) {
        조건식 결과가 true일 동안 반복해서 수행할 문장;
}
```

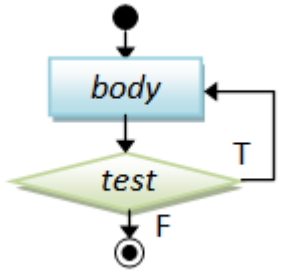


03. 반복문

- **do ~ while 문**
  - 먼저 수행 후 조건식 검증 후 반복 수행여부 결정
  - 최소 기본 1번은 수행

형식

```
do {
        조건식 결과가 true일 동안 반복해서 수행할 문장;
} while(조건식) ;
```



04. 분기문

- **break 문**
  - 이제 더 이상 반복하지 말고, 바로 for문이나 while문을 끝내때 사용하는 문장

- **continue 문**
  - for문이나 while문의 {}안에서 continue 문장을 만난 순간 continue문 아래에 있는 실행해야 하는 문장들을 건너 뛰고, 다음 반복을 시작

05. 배열

01/ 배열의 개념

02/ 1차원 배열

03/ 2차원 배열

04/ 가변 배열

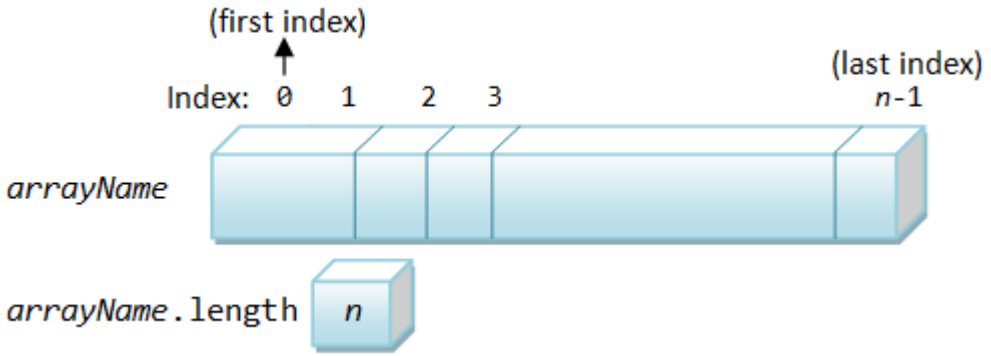
05/ 확장 for 문

01. 배열의 개념

- **배열의 개념**
  - 동일한 자료형의 데이터를 그룹으로 묶어 관리하기 위한 메모리 공간
  - 배열에 저장된 데이터인 각각의 요소는 서로 연속적으로 위치하며 인덱스로 각 값들을 구분
  - new 키워드를 이용하여 해당 자료형의 배열객체 생성하여 사용
  - 배열 객체 생성시에 배열의 크기를 지정하여 생성하며, 생성 후 배열의 크기는 불변
  - 기본형, 참조형 모두 배열 사용

- **첨자(Index)**
  - 배열의 값들을 구분해주는 정수
  - 0부터 시작하여 (배열 크기 – 1)까지임

- **배열의 크기**
  - 배열 생성 시에 결정되며, 변경 불가능
  - 배열의 length라는 필드에 저장되어 있음



02. 1차원 배열

- **1차원 배열 선언**

형식

```
[접근지정 제어자] 자료형[]  배열명;
```

- **1차원 배열 생성**

형식

```
배열명 = new  자료형[배열크기];
```

- **1차원 배열 값 초기화**

형식

```
배열명[배열요소인덱스] = 값;
```

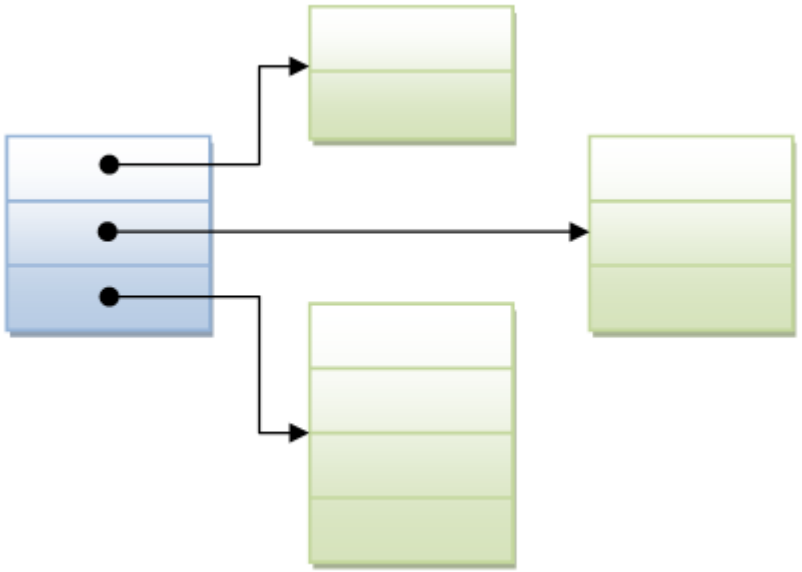
03. 2차원 배열

- **2차원 배열**
  - 행과 열로 구성되어 있는 배열
  - 행의 생성 및 열의 생성은 각각의 객체로 생성됨
  - 행마다 열의 크기를 다르게 지정하여 분리 생성 가능

- **2차원 배열 선언**

형식

```
[접근지정 제어자] 자료형[][] 배열명;
```



- **2차원 배열 생성**

형식

```
배열명 = new 자료형[행의 크기] [열의 크기];
```

04. 가변 배열

- **2차원 배열**
  - 배열의 배열을 의미
  - 배열의 요소가 배열이므로 배열 요소마다 차원과 크기를 다르게 지정하여 선언하는 배열
  - 배열의 차원을 각괄호로 구분할 때 1차원 배열의 크기만을 지정
  - 지정된 1차원을 이용해서 또 다른 1차원을 생성하여 주소를 대입하는 형식으로 선언

- **2차원 배열 선언 및 생성**

형식

```
[접근지정 제어자] 자료형[][]  배열명 = new 자료형[행의 크기] [];
```

05. 확장 for문

- **확장 for 문**
  - 조건식을 먼저 검증 후 반복 수행여부 결정

형식

```
for(자료형 변수명 : 배열명 또는 컬렉션명) {
        배열 또는 컬렉션 개수만큼 반복해서 수행할 문장;
}
```

06. 객체지향 프로그래밍

01/ 객체지향의 개요

02/ 객체지향의 특성

03/ 클래스와 객체

04/ 클래스 정의

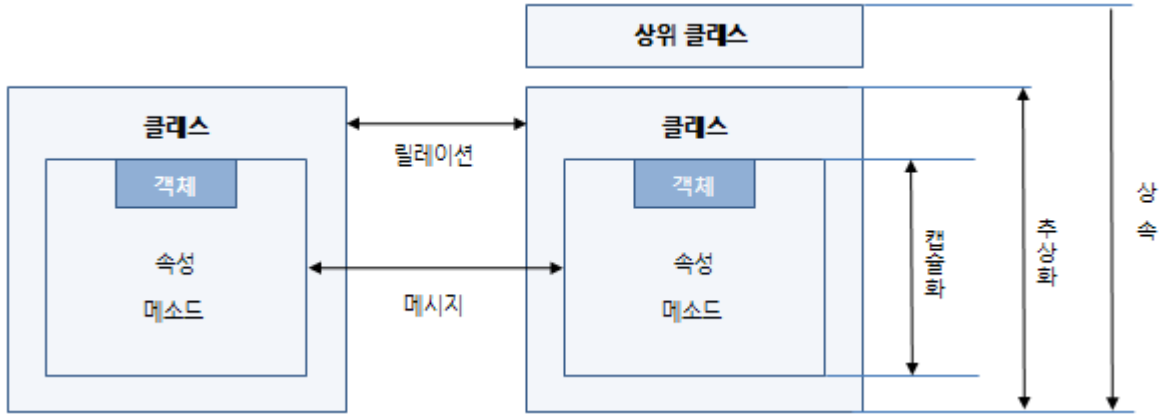
05/ 생성자

06/ 참조 변수

07/ 패키지

01. 객체지향의 개요

- **객체지향의 개념**
  - 실 세계를 구성하는 모든 것
  - 하나의 객체는 특성과 기능으로 구성
  - 실 세계의 객체들은 여러 다른 객체와 상호 작용
  - 컴퓨터, 자동차, 도서, 사람, 직원 등



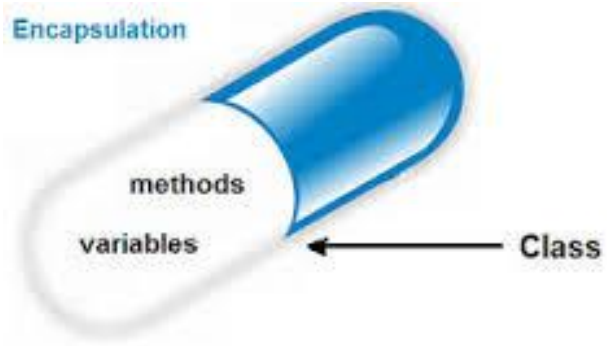
02. 객체지향의 특성

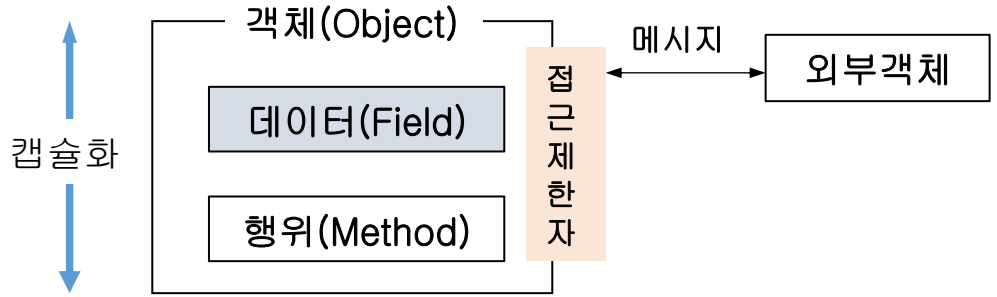
- **추상화(Abstraction)**
  - 데이터나 프로세스 등을 의미가 비슷한 개념이나 표현으로 정의해 나가는 과정
  - 각 개별 개체의 구현에 대한 상세함은 감추는 것

  - 1) 기능추상화 : 클래스 내 메소드를 정의(obj.getName())
  - 2) 자료(데이터)추상화 : 객체 클래스 자체를 데이터 타입으로 사용(String, Class)
  - 3) 제어추상화 : 제어행위에 대한 개념화, 명령 및 이벤트(if, for, while)

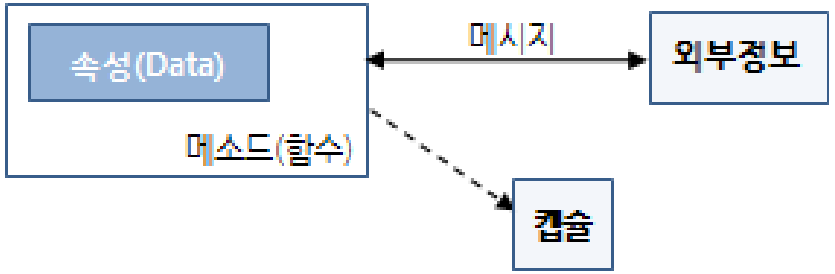
02. 객체지향의 특성

- **캡슐화(Encapsulation)**
  - 객체의 데이터(필드)과 행위(메서드)를 하나로 묶고,
  - 실제 구현 내용 일부를 외부에 감추어 은닉하는 객체지향 특징
  - 은닉의 정도를 접근제한자로 기술하여 속성이나 메서드의 접근을 제한



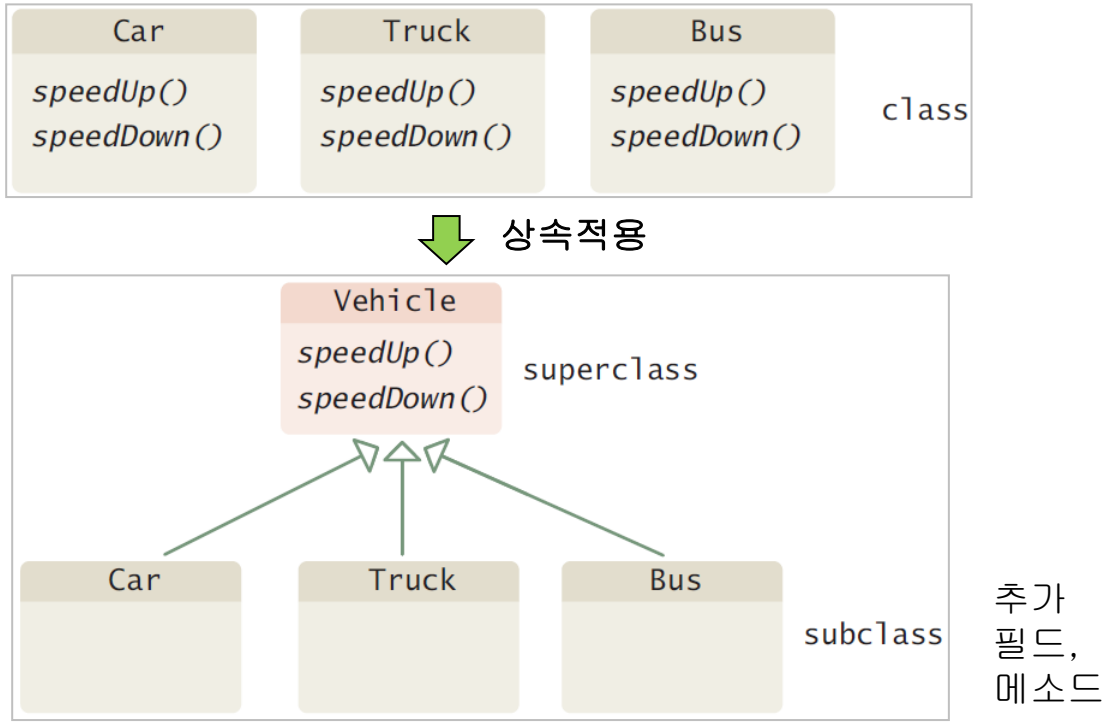


- **정보은닉**
  - 캡슐화된 항목들이 다른 객체에게 보이지 않게 하는 것
  - 메시지 전달에 의해 다른 클래스내의 메소드가 호출됨



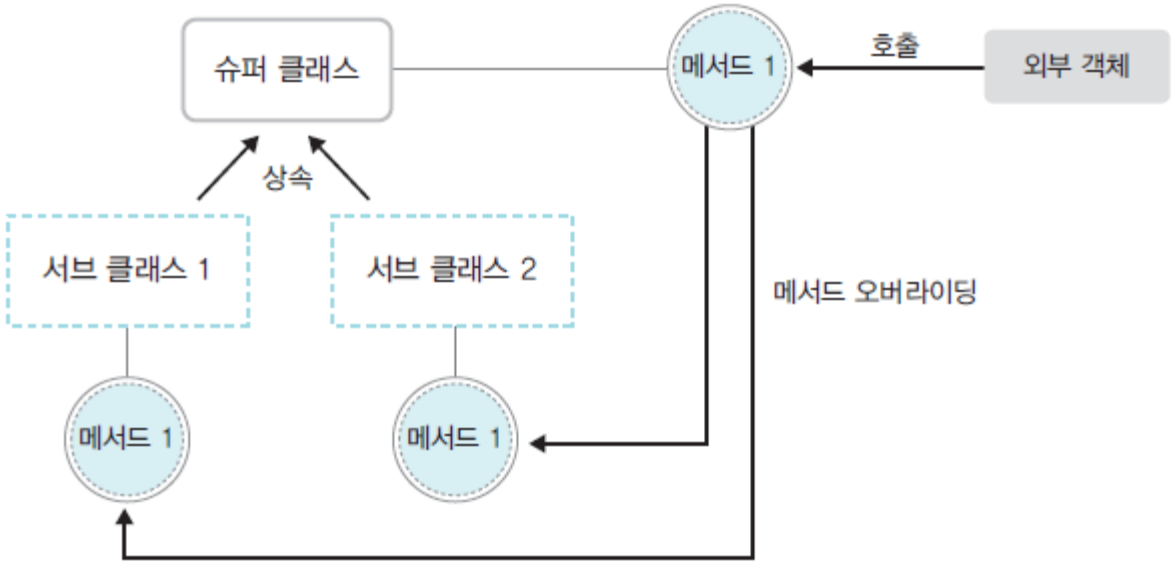
02. 객체지향의 특성

- **상속성(Inheritance)**
  - 하위 클래스에게 자신의 속성과 메소드를 사용할 수 있도록 허용하는 것



02. 객체지향의 특성

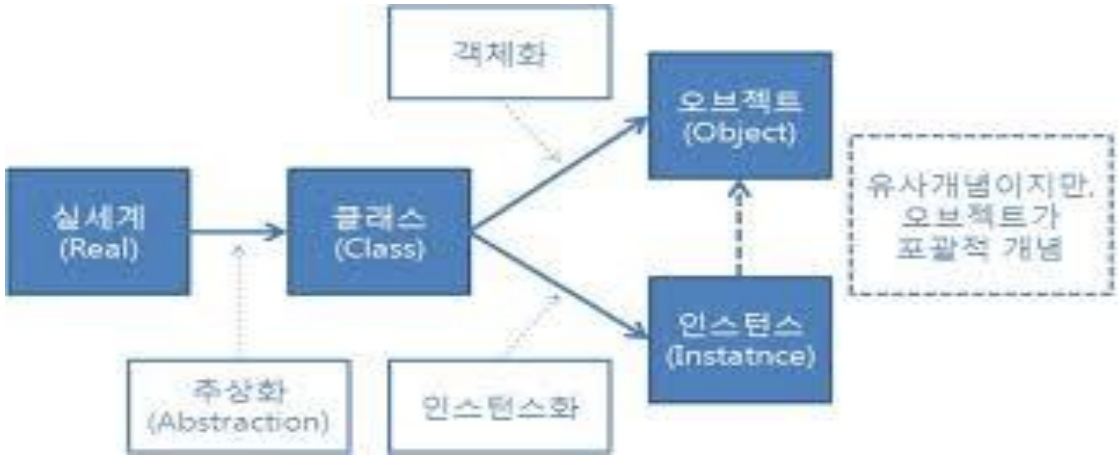
- **다형성(Polymorphism)**
  - 같은 이름을 갖고 있지만, 상황에 따라 다른 형태로 해석될 수 있다는 것을 의미
  - 다중 정의(Overload), 재정의(Override)를 제공



03. 클래스와 객체

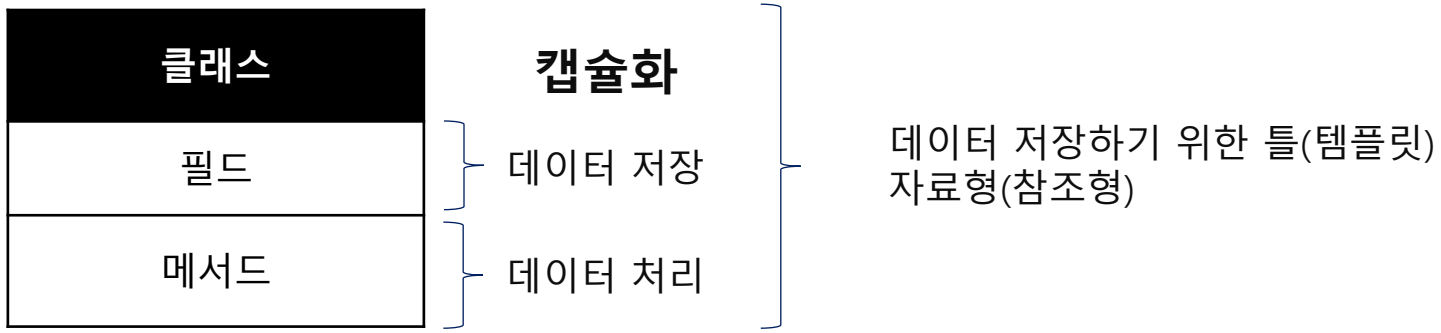
- **클래스(Class)**
  - 객체의 속성과 기능을 정의해 놓은 틀
  - 하나의 클래스를 정의하여 객체 생성시에 재사용

- **객체(Object)**
  - 클래스를 인스턴스화하여 구체화 시켜 식별가능한 것
  - 하나의 클래스를 통해서 여러 개의 객체 생성 활용
  - 클래스를 통해서 메모리에 생성된 인스턴스



04. 클래스 정의

- **클래스의 구성요소**



- **필드(Field)**
  - 객체의 특성을 표현하는 데이터를 저장하는 역할

- **메소드(Method)**
  - 객체 내부의 일을 처리하거나 객체들간의 서로 영향을 주고 받는 동적인 일을 처리하는 단위
  - 입력을 받아서 처리를 하고 결과를 반환
  - 반복되는 작업을 효율적으로 처리
  - 오버로드 가능

04. 클래스 정의

- **클래스 정의**

형식

```
[접근지정 제어자] [사용 제어자] class 클래스명 {
        // 필드 선언
        // 메소드 정의
}
```

- **필드 선언**

형식

```
[접근지정 제어자] [사용 제어자] 자료형 필드명 ;
```

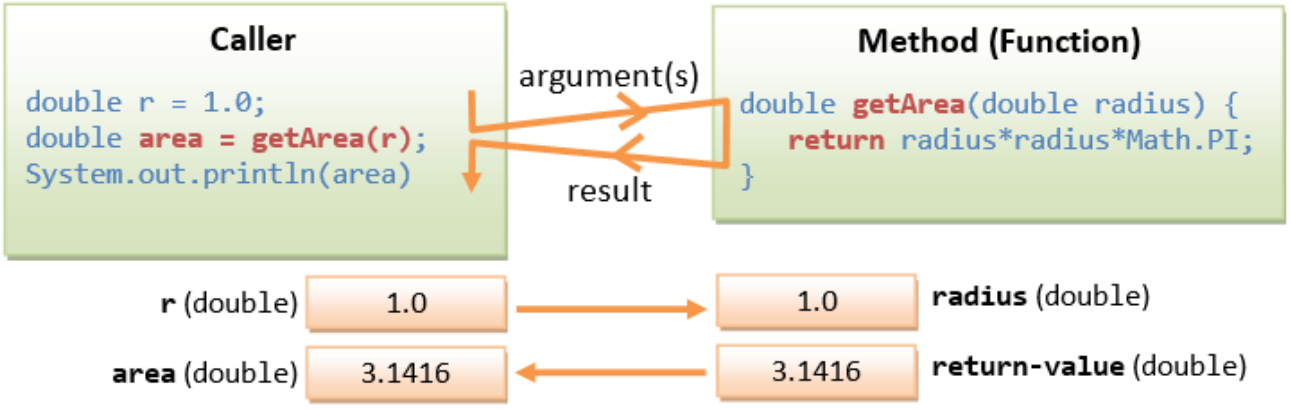
04. 클래스 정의

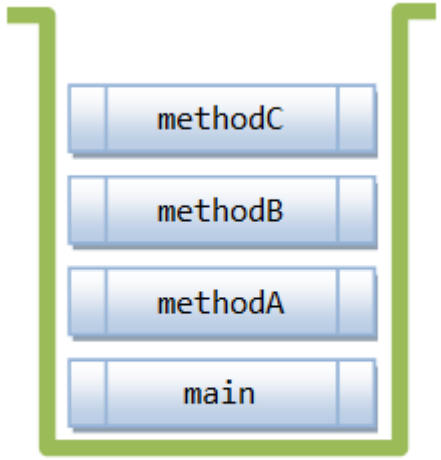
- **메소드 정의 방법**

형식

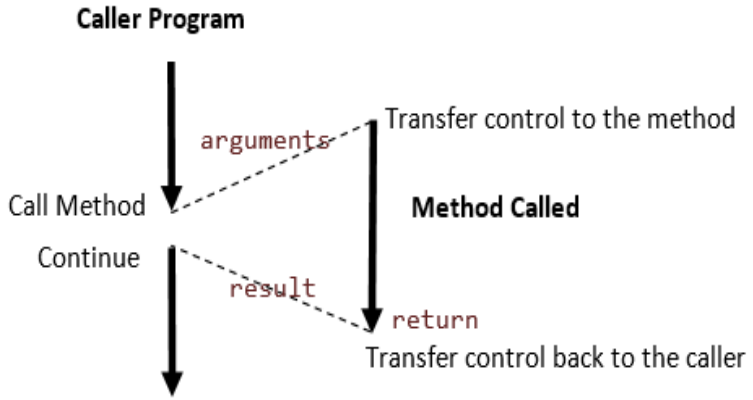
```
[접근지정 제어자] [사용 제어자] 반환자료형 메소드명(매개변수 리스트) {
        …
      [return 반환값;]
}
```

- **메소드 호출**





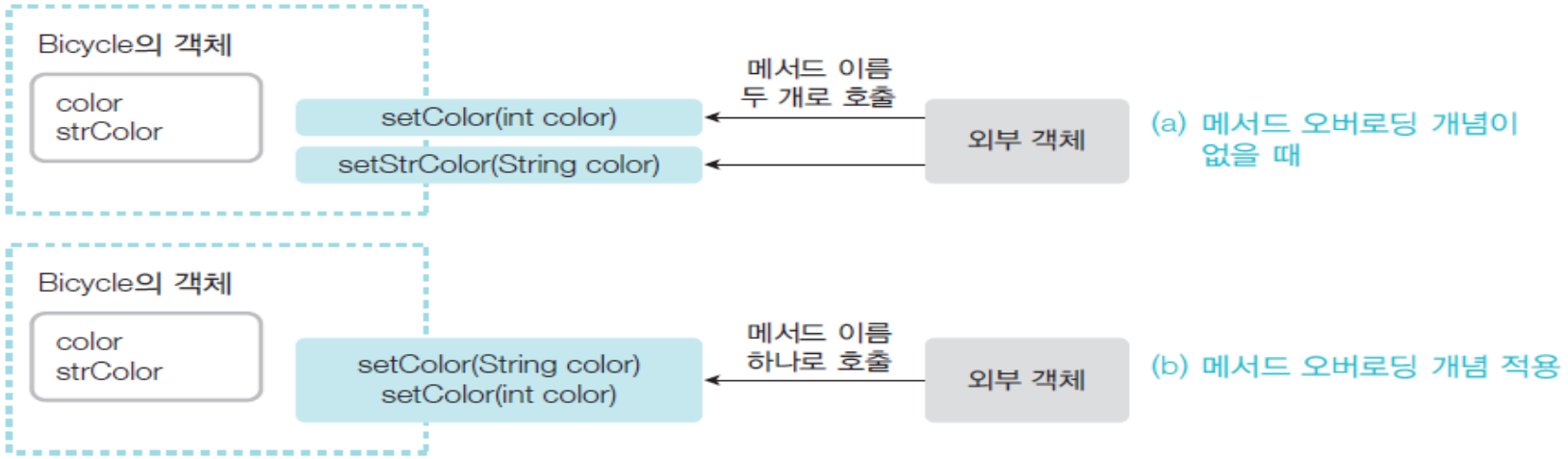
Method Call Stack
(Last-in-First-out Queue)



04. 클래스 정의

- **메소드 오버로드**
  - 같은 이름의 메소드를 여러 개 가지면서 매개변수의 유형과 개수가 다르도록 하는 기술

  - 메소드 오버로드의 조건
    - 메소드 이름이 같아야 한다.
    - 매개변수의 개수 또는 자료형이 달라야 한다.
    - 매개변수는 같고 반환 자료형이 다르면 오버로드가 성립되지 않는다



05. 생성자

- **생성자(Constructor) 개념**
  - 인스턴스 생성시 호출되는 인스턴스 초기화 메서드
  - 필드 초기화, 인스턴스 생성시 수행해야 할 작업
  - 모든 클래스는 기본 1개 이상의 생성자가 존재해야 함

- **생성자(Constructor) 정의**

형식

```
[접근지정 제어자] 클래스명(매개변수 리스트) {
        …
        인스턴스 필드 초기화 수행문;
}
```

- **기본 생성자**
  - 명시적으로 생성자를 정의하지 않는 경우에 컴파일 시점에서 자동으로 제공되는 생성자
  - 기본생성자 형식 : public 클래스이름() { super(); }

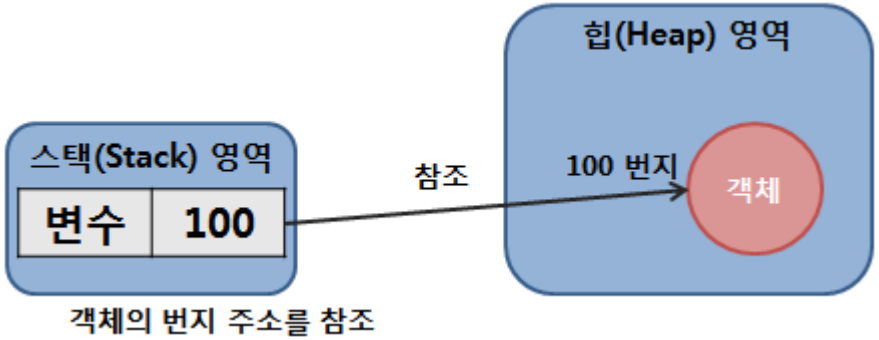
05. 생성자

- **생성자 오버로드**
  - 생성자 여러 개 가지면서 매개변수의 유형과 개수가 다르도록 하는 기술

  - 생성자 오버로드의 조건
    - 매개변수 개수 또는 자료형이 달라야 한다.

06. 참조 변수

- **참조 변수 개념**
  - **실제 값을 저장하는 것이 아니라 객체의 참조값(주소)를 저장**



- **참조 변수 선언**

형식

```
자료형   참조변수명;
```

- **객체 생성**

형식

```
참조변수명 = new 생성자();
```

06. 참조 변수

- **this**
  - 객체 생성시 자동으로 제공하는 인스턴스 자신을 가리키는 참조변수

  - 필드 접근 : this.필드명
  - 인스턴스 자신 지칭 사용 : this
  - 현재 클래스의 다른 생성자 호출 사용 : this([매개변수 값])
  - 다른 생성자 호출시에는 생성자의 첫번째 수행문으로 사용해야 함

07. 패키지(Package)

- **패키지의 개념**
  - 관련된 클래스와 인터페이스를 그룹핑하여 물리적 폴더 개념으로 분리하여 관리하는 단위
  - 클래스 간의 이름 중복으로 발생하는 충돌 방지
  - 클래스를 기능 별로 분류할 수 있어 필요한 클래스의 식별이 용이
  - 패키지는 서브 패키지로 구조화 시킬수 있으며 서브패키지 구분자는 .(dot) 사용
- **패키지 선언**
  - 선언 위치 : 소스코드 첫번째 수행문으로 선언
  - 선언 방법 : package top.sub

- **Import 문**
  - 사용한 클래스가 속한 패키지를 지정
  - java.lang 패키지, 현재 클래스와 동일한 패키지의 클래스 사용시 import 구문 생략 가능
  - 선언 위치 : package 선언문과 class 선언문 사이에 위치
  - 선언 방법 : import top.sub.클래스이름;

07. 제 어 자

01/ 제어자의 개요

02/ 접근 지정 제어자

03/ 사용 제어자

01. 제어자의 개요

- **제어자의 개념**
  - 클래스, 변수, 메소드의 선언부에 사용되어 부가적인 의미를 부여
  - 하나의 대상에 여러 개의 제어자를 조합해서 사용할 수 있으나, 접근지정 제어자는 단 하나만 사용가능

- **제어자의 종류**
  - 접근지정 제어자 (Access Modifiers)
    - 클래스, 멤버변수, 메서드, 생성자의 선언부에 사용
    - 접근지정 제어자를 통하여 접근 여부 제약 지정
    - public, protected, friendly, private
  - 사용 제어자 (Usage Modifiers)
    - 클래스, 멤버변수, 메서드, 생성자의 선언부에 사용
    - 사용 제어자를 통하여 부가적인 기능을 지정
    - static, final, abstract, transient, synchronized, native, volatile, strictfp

02. 접근지정 제어자(Access Modifier)

- **접근지정 제어자의 개념**
  - 멤버 또는 클래스에 사용, 외부에서 접근하지 못하도록 제한.
  - 클래스, 필드, 메소드, 생성자에 사용되고, 지정되어 있지 않다면 default임을 뜻함

- **접근지정 제어자의 종류**
  - public : 접근 제한이 전혀 없다.
  - protected : 같은 패키지 내에서, 다른 패키지의 자손클래스에서 접근이 가능하다.
  - default : 같은 패키지 내에서만 접근이 가능하다.
  - private : 같은 클래스 내에서만 접근이 가능하다.

- **사용가능한 접근 제어자**
  - 클래스 : public, default
  - 메소드 & 필드 : public, protected, default, private
  - 지역변수 : 없음

02. 접근지정 제어자(Access Modifier)

- **접근지정 제어자를 이용한 캡슐화(정보 은닉성)**
  - 접근지정 제어자는 클래스 내부에 선언된 데이터를 보호하기 위해서 사용
  - 유효한 값을 유지하도록, 함부로 변경하지 못하도록 접근을 제한하는 것이 필요
  - 데이터 감추기(data hiding)라고 하며, 캡슐화(encapsulation)에 해당

- **생성자의 접근 제어자**
  - 생성자에 접근 제어자를 사용함으로 인스턴스의 생성을 제한할 수 있음
  - 일반적으로 생성자의 접근 제어자는 클래스의 접근 제어자와 일치

02. 접근지정 제어자(Access Modifier)

- **접근 지정 제어자의 접근 범위**

접근지정 제어자	동일 클래스	동일 패키지	상속받은 클래스	다른 패키지
public	접근 가능	접근 가능	접근 가능	접근 가능
protected	접근 가능	접근 가능	접근 가능	X
(default)	접근 가능	접근 가능	X	X
private	접근 가능	X	X	X

- 접근 제한자 사용

접근지정 제어자	클래스	지역변수	멤버변수	상수	메서드	생성자
public	O	X	O	O	O	O
protected	X	X	O	O	O	O
(default)	O	O	O	O	O	O
private	X	X	O	O	O	O

02. 접근지정 제어자(Access Modifier)

- **getter와 setter 메소드의 역할**
  - 캡슐화된 클래스에서 정보를 은닉하기 위해 필드에 private 접근지정 제어자를 이용한 경우 외부에서 getter/setter 메소드를 이용하여 필드의 값을 변경하거나 얻어올 때 사용하느 ㄴ메소드

- **getter 메소드**

형식

```
public 필드자료형  get필드명() {
        …
      return  필드명;
}
```

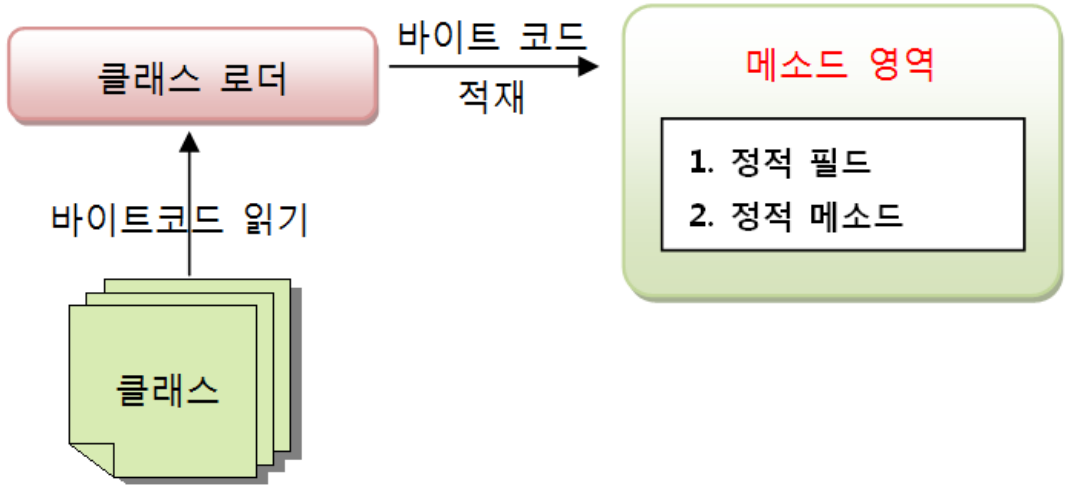
- **setter 메소드**

형식

```
public void  set필드명(필드자료형) {
        …
}
```

03. 사용 제어자

- **static**
  - '클래스의' 또는 '공통적인' 의미를 가지고 있음
  - static이 붙은 필드와 메소드, 초기화 블럭은 인스턴스를 생성하지 않고도 사용할 수 있음



03. 사용 제어자

- **static 필드**
  - 모든 인스턴스에 공통적으로 사용되는 클래스 변수가 됨
  - 클래스변수는 인스턴스를 생성하지 않고도 사용 가능
  - 클래스가 메모리에 로드될 때 생성

형식

```
[접근지정 제어자] static 자료형 필드명 ;
```

- **static 메소드**
  - 인스턴스를 생성하지 않고도 호출이 가능한 static 메소드가 됨
  - static 메소드 내에서는 인스턴스 멤버들을 직접 사용할 수 없음

형식

```
[접근지정 제어자] static 반환자료형 메소드명(매개변수 리스트) {
        …
      [return 반환값;]
}
```

03. 사용 제어자

- **final**
  - '마지막의' 또는 '변경될 수 없는' 의미를 가지고 있음

- **final 클래스**
  - 변경 될 수 없는 클래스, 확장될 수 없는 클래스가 됨
  - 다른 클래스의 조상이 될 수 없음
  - abstract과 같이 사용할 수 없음

형식

```
[접근지정 제어자] final class 클래스명 {
        // 필드 선언
        // 메소드 정의
}
```

03. 사용 제어자

- **final 메서드**
  - 변경 될 수 없는 메소드로, 오버라이딩을 통해 재정의 될 수 없음

형식

```
[접근지정 제어자] final 반환자료형 메소드명(매개변수 리스트) {
        …
      [return 반환값;]
}
```

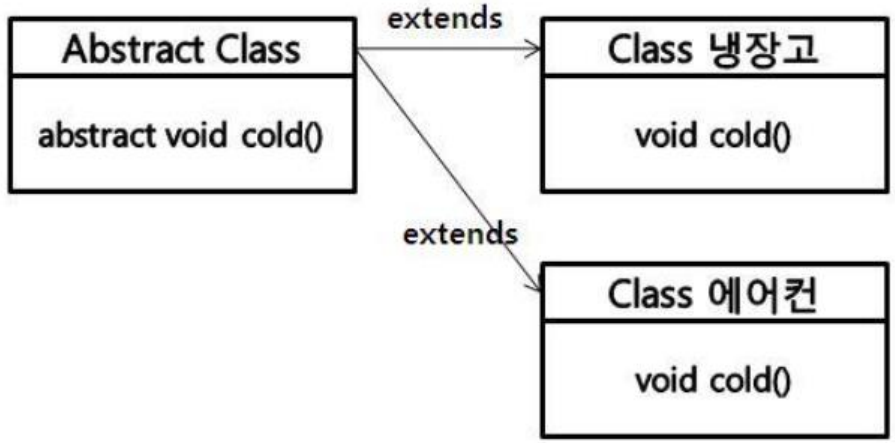
- **final 필드, final 지역 변수**
  - 변경 할 수 없는 상수가 됨
  - final이 붙은 변수는 상수이므로 보통은 선언과 초기화를 동시에 하지만, 인스턴스 변수의 경우 생성자에서 초기화 할 수 있음

형식

```
[접근지정 제어자] final 자료형 필드명 ;
```

03. 사용 제어자

- **abstract**
  - '추상의' 또는 '미완성의' 의미를 가지고 있다.
  - 메소드의 선언부만 작성하고 실제 수행 내용은 구현하지 않은 추상 메소드를 선언하는데 사용



- **abstract 클래스**
  - 클래스 내에 추상 메소드가 선언되어 있음을 의미
  - 인스턴스 생성 불가능

- **abstract 메서드**
  - 선언부만 작성하고 구현부는 작성하지 않은 추상 메소드임을 알림

08. 상속

01/ 상속 개념

02/ 메소드 오버라이드

03/ 추상 클래스

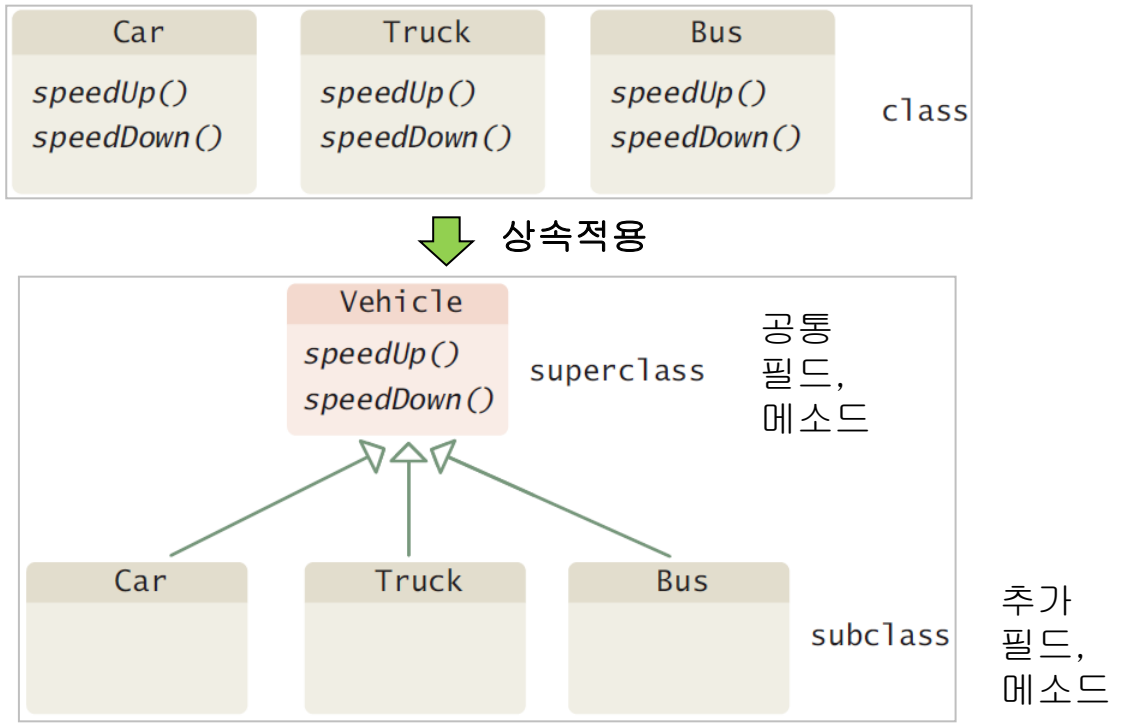
04/ 인터페이스

05/ 클래스 다이어그램

06/ 참조 자료형 형변환

01. 상속

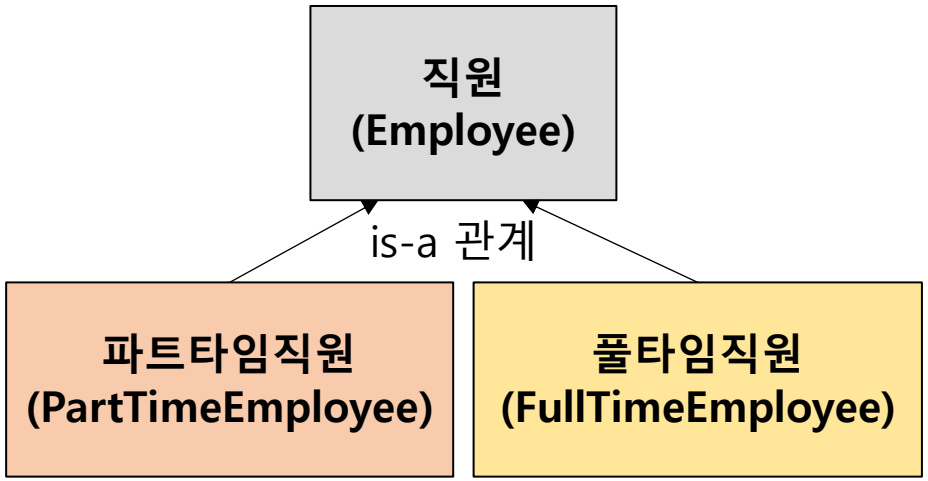
- **상속 개념**
  - 하위 클래스는 상위 클래스의 특징인 필드와 메서드를 그대로 물려받을 수 있는 특성
  - [A는 B이다]는 [이다 관계(is-a relationship)]가 성립
    B는 상위 클래스, A는 하위 클래스 관계로 규정
  - extends 키워드를 이용
  - 소스 코드의 중복 감소



01. 상속

- **상속 개념**

부모, 상위, 슈퍼 클래스



자식, 하위, 서브 클래스

```
class Employee {

}
```

```
class PartTimeEmployee extends Employee {

}
```

```
class FullTimeEmployee extends Employee {

}
```

01. 상속

- **super**
  - 자식 객체 생성할 때는 부모 객체부터 생성 후 자식 객체 생성
  - 생성자나 인스턴스 메소드에서만 사용 가능
  - runtime시 해당 코드(인스턴스 메소드, 생성자)가 실행되고 있는
  - 그 객체 자기 자신의 부모 객체를 가리키는 특별한 참조 변수
  - 어떤 객체를 가리키게 될 지는 runtime시에 결정 됨

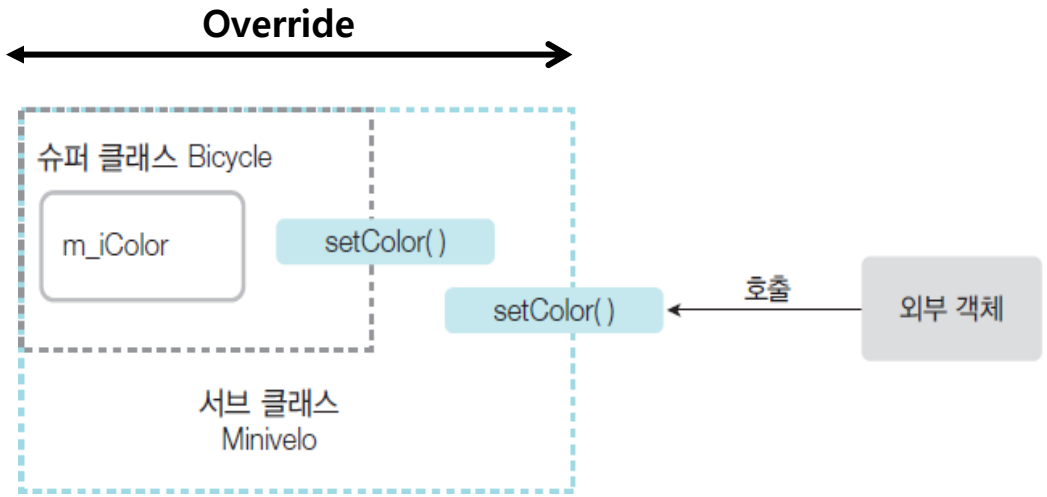
01. 상속

- **생성자 호출**
  - 묵시적 부모 클래스 생성자 호출
    - Object 클래스의 생성자까지 호출
  - 명시적 부모 클래스 생성자 호출
    - 부모 생성자 호출 완료 후 자식 생성자 호출 완료
    - 부모 객체 생성할 때, 부모 생성자 선택해 호출
    - 반드시 자식 생성자의 첫 줄에 위치, 부모 생성자 없다면 컴파일 오류 발생

```
자식클래스( 매개변수선언, ... ) {
   super( 매개값, ... );
   ...
}
```

02. 메소드 오버라이드(Method Override)

- **메소드 오버라이드 개념**
  - 부모 클래스에서 정의한 메서드를 서브 클래스에서 재정의하는 것
  - 부모 클래스에서 정의한 어떤 메서드의 내용이 서브 클래스에 적합하지 않거나 새롭게 변경된 내용이 있을 때 사용.



- **메소드 오버라이드 조건**
  - 부모 클래스의 메소드와 메소드 구성 요소 모두가 동일해야 한다.
  - 이때 메소드 이름, 매개변수, 반환자료형이 모두 같아야 한다.

02. 메소드 오버라이드

- **바인딩(Binding) 개념**
  - 속성과 개체 또는 연산과 기호를 연관(association)시키는 것
  - 변수에 변수와 관련된 속성(자료형, 값)을 연관시키는 것
  - 메소드를 호출하는 부분에서 메소드가 위치한 메모리 번지를 연결시켜주는 것

- **바인딩 유형**

구분	정적바인딩 (Static Binding)	동적 바인딩 (Dynamic Binding)
시점	컴파일 또는 링크 시간에 확정되는 바인딩	실행시간
예	식별자가 그 대상인 메모리 주소, 자료형, 값 등을 배정	실행시간에 호출되는 메소드 결정
특징	컴파일 시간에 많은 정보가 결정되므로 실행 효율 향상	런타임 시간에 결정되므로 유연성 향상

03. 추상 클래스(Abstract Class)

- **추상 클래스 개념**
  - 적어도 하나 이상의 추상 메소드를 가진 클래스는 반드시 추상이어야 함
  - 인스턴스 객체 생성 불가능, 키워드 final 이용 불가능

- **추상 클래스의 용도**
  - 실체 클래스의 공통된 필드와 메서드의 이름 통일할 목적
  - 실체 클래스 설계 규격을 만들고자 할 때

03. 추상 클래스(Abstract Class)

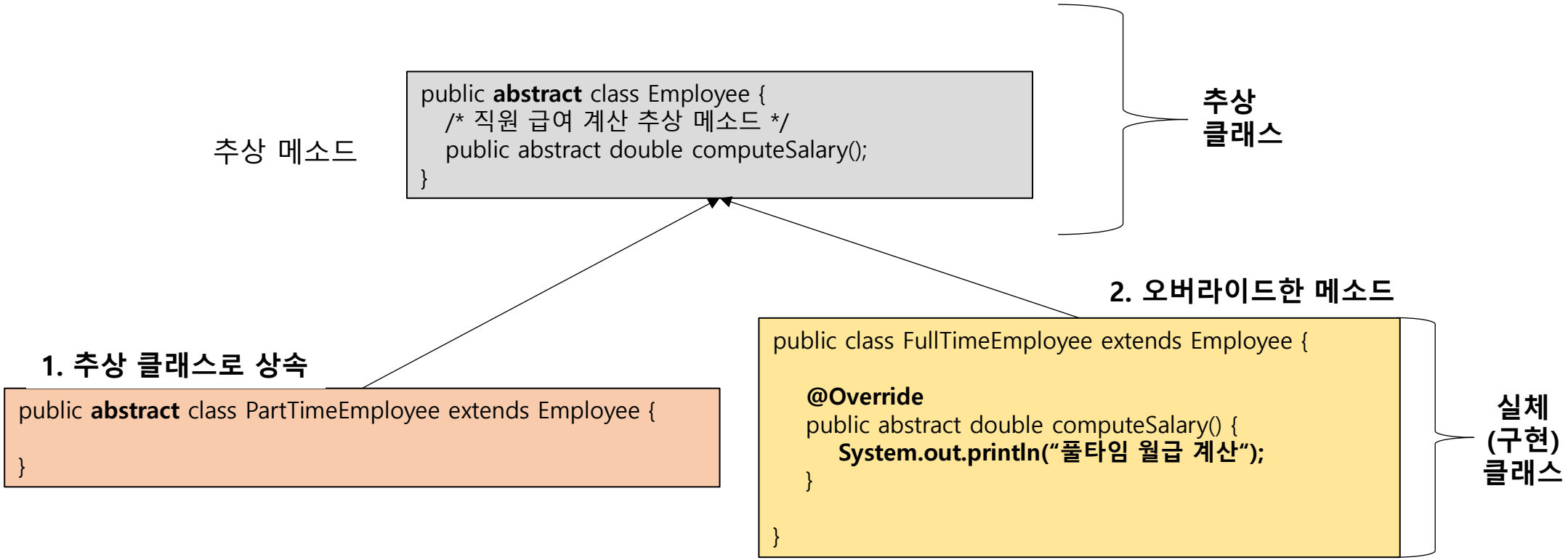
- **추상 클래스 정의**

형식

```
[접근지정 제어자] abstract class 클래스명 {
        // 필드 선언

        // 일반 메소드 정의

        // 추상 메소드 정의
}
```

03. 추상 클래스(Abstract Class)

- **추상 클래스 상속과 추상 메소드 구현**



03. 추상 클래스(Abstract Class)

- **추상 메소드 개념**
  - 몸체가 없는 메서드
  - 추상 메소드 정의 시 반환형 앞에 키워드 abstract를 기술
  - 추상 메소드는 private와 final이 사용될 수 없음

- **추상 메소드 구현 방법**
  - 추상 클래스에는 메소드의 선언부만 작성 (추상 메소드)
  - 실체 클래스에서 메소드의 실행 내용 작성(오버라이드)

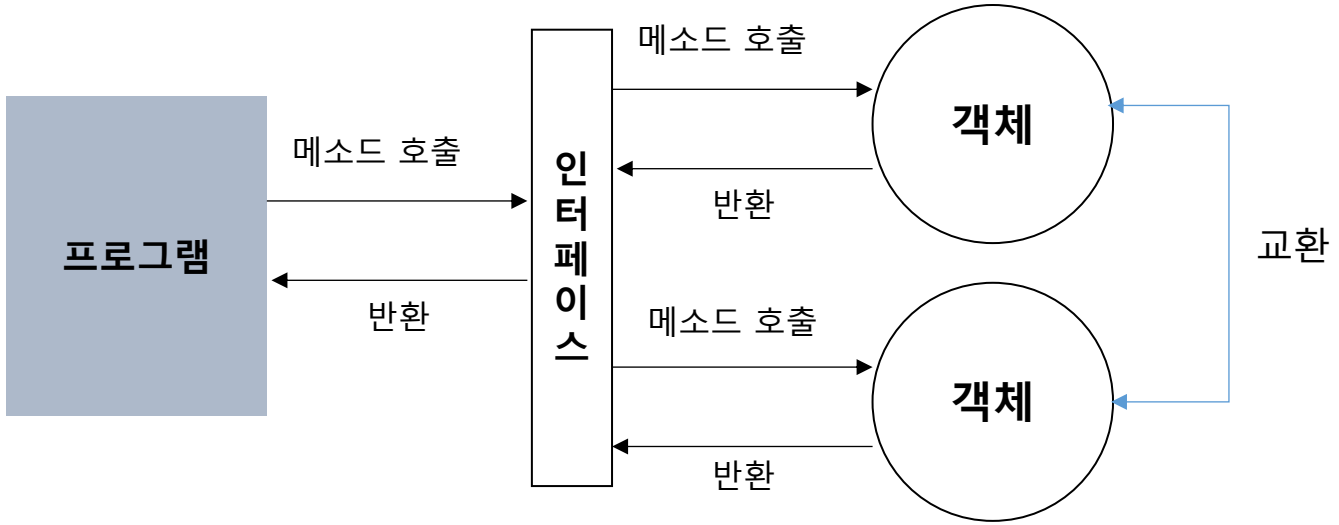
- **추상 메소드 정의**

형식

```
[접근지정 제어자] abstract 반환자료형 메소드명(매개변수 리스트);
```

04. 인터페이스(Interface)

- **인터페이스 개념**
  - 해야 할 작업의 구체적 구현 없이 기능만 선언한 클래스
  - 하위 클래스가 수행해야 하는 메소드와 필요한 상수만을 추상적으로 정의
  - 인터페이스는 다중 상속(multiple inheritance)을 지원



04. 인터페이스(Interface)

- **인터페이스 정의**

형식

```
[접근지정 제어자] [사용 제어자] interface 인터페이스명 {
        // 상수 선언
        // 추상 메소드 정의
}
```

- **인터페이스 구성 요소**
  - 상수 : public static final
  - 추상 메소드 : public abstract
  - 디폴트 메소드 : public default
  - 클래스 메소드 : public static
  - 디폴트 메소드, 클래스 메소드 JDK8부터 추가

04. 인터페이스(Interface)

- **인터페이스 구현**
  - implements 키워드 이용

형식

```
[접근지정 제어자] class 클래스명 implements 인터페이스명 {


        // 추상 메소드 오버라이드
}
```

04. 인터페이스(Interface)

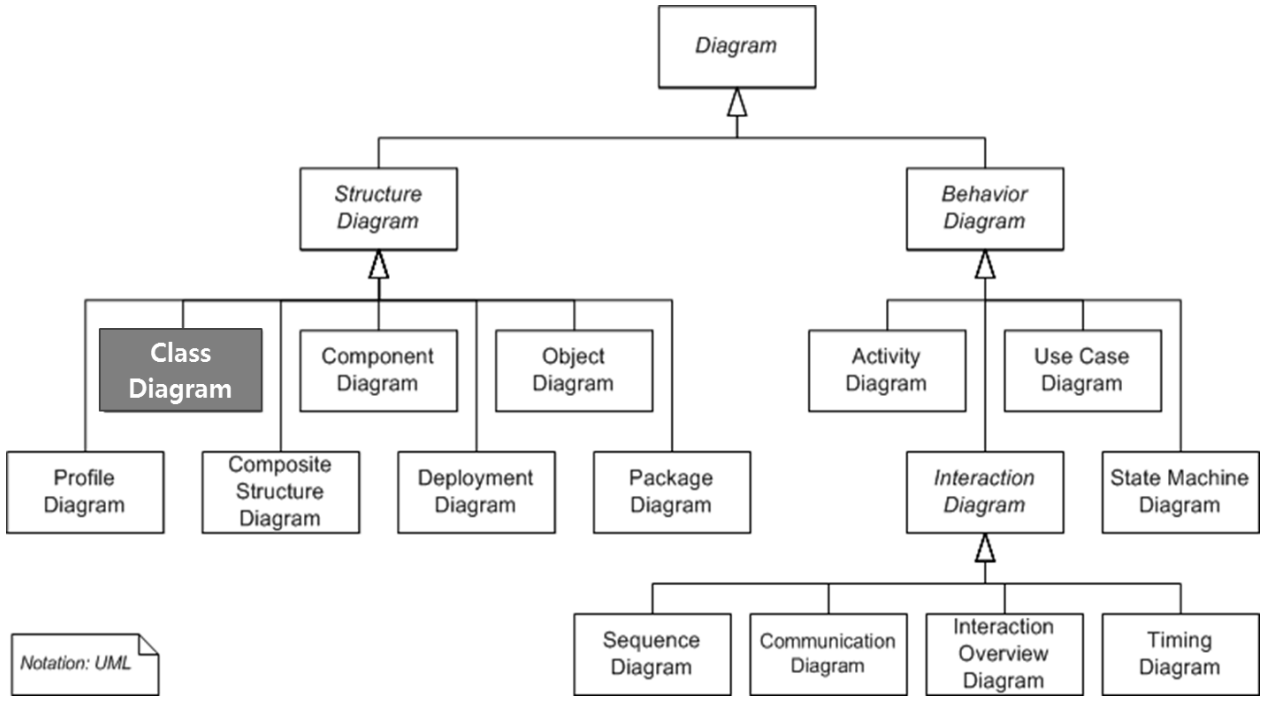
- **익명 구현 객체**
  - 명시적인 구현 클래스 작성 생략하고 바로 구현 객체를 얻는 방법
  - 이름 없는 구현 클래스 정의와 동시에 객체 생성

형식

```
인터페이스명 객체명 = new 인터페이스명() {
        // 추상 메소드 오버라이드

        // 필드
        // 메소드
}
```

05. 클래스 다이어그램

- **UML(Unified Modeling Language)**
  - 소프트웨어 공학에서 사용되는 표준화된 범용 모델링 언어
  - 객체 지향 소프트웨어 집약 시스템을 개발할 때 산출물을 명세화, 시각화, 문서화할 때 사용
  - Structure Diagram : 정적이고, 구조 표현을 위한 다이어그램
  - Behavior Diagram : 동적이고, 시퀀셜한 표현을 위한 다이어그램



05. 클래스 다이어그램

- **클래스 다이어그램 개념**
  - 시스템의 논리적인 구조(클래스)를 표현
  - 객체지향 개발에서 가장 공통적으로 많이 사용
  - 클래스 내부의 정적인 내용이나 클래스 사이의 관계를 표현

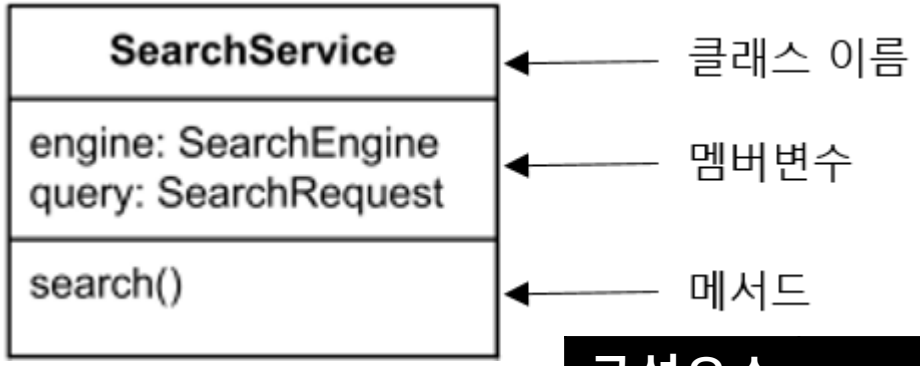
- **클래스 다이어그램의 용도**
  - 다른 사람들과의 의사소통 또는 설계 논의
  - 전체 시스템의 구조 및 클래스의 의존성 파악
  - 유지보수를 위한 설계의 back-end 문서

- **작성 시 주의사항**
  - Access : 다이어그램을 보고 직관적으로 이해할 수 있도록 의미있고 명확한 이름을 부여
  - Simplicity : 불필요한 내용을 제외하고 모델을 간결하게 그림(getter/setter/생성자는 생략 가능)
  - Cost : 개발비용보다 더 들어가는 경우도 있으므로 비용을 고려하여 작성여부 검토

05. 클래스 다이어그램

- **클래스 다이어그램 구성요소**
  - 클래스
  - 관계

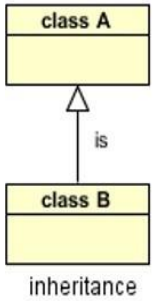
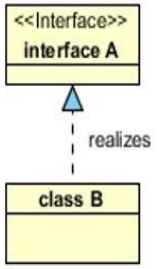
- **클래스**
  - 객체지향에서 사용되는 클래스를 표현하는 모델링 언어



구성요소	설명
필드	[visiblity] property-name : property-type
메소드	[visiblity] operation-name (property1, ...) : return-type

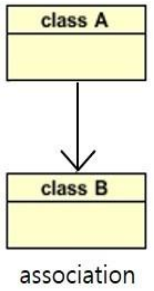
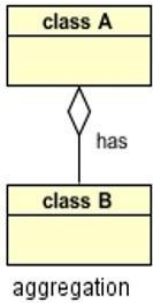
05. 클래스 다이어그램

- **관계(Relationship)**
  - 클래스의 관계를 표현하기 위한 모델링 언어

유형	표기법	설명
Generalization	 inheritance	상속관계 class A가 Super class, class B가 Sub class
Realization	 realization	인터페이스를 구현하는 관계 interface A가 있고, class B가 interface A를 구현한 경우

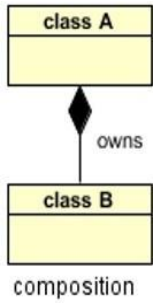
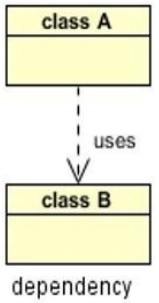
05. 클래스 다이어그램

- **관계(Relationship)**

유형	표기법	설명
Association		멤버변수로 가지고 사용하는 관계 class A가 class B를 멤버변수로 가지고 있고, 사용하는 경우
Aggregation		Association의 한 종류, 전체와 부분의 관계 멤버변수로 가지고 있지만, new를 직접 수행하지 않는 관계 class A가 class B를 멤버변수로 가지고 있는 경우. class A가 전체개념이고, class B가 부분의 개념 하지만 class A가 직접 class B를 new 하지는 않음

05. 클래스 다이어그램

- **관계(Relationship)**

유형	표기법	설명
Composition	 composition	Association의 한 종류, 전체와 부분의 관계 멤버변수로 가지고 있고, 직접 new를 하는 관계 class A가 class B를 멤버변수로 가지고 있는 경우. class A가 전체개념이고, class B가 부분의 개념 class A가 직접 class B를 new 해서 생성
Dependency	 dependency	지역변수, 매개변수, 반환값으로 사용 멤버변수로 갖지 않고 사용하는 관계 class A의 메서드가 class B의 메서드를 호출하거나, class B를 메서드의 인자로 받는 등 사용하는 경우

06. 참조 자료형 형변환

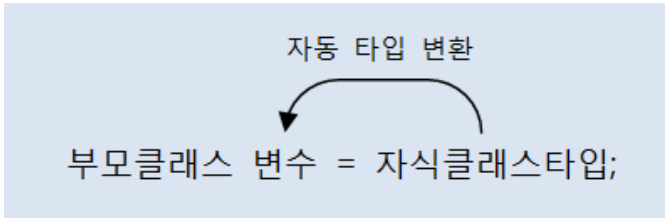
- **참조 자료형 형변환 개념**
  - 서로 상속관계에 있는 클래스사이에만 가능하기 때문에 자손타입의 참조변수를 조상타입의 참조변수로, 조상타입의 참조 변수를 자손타입의 참조변수로의 형변환만 가능

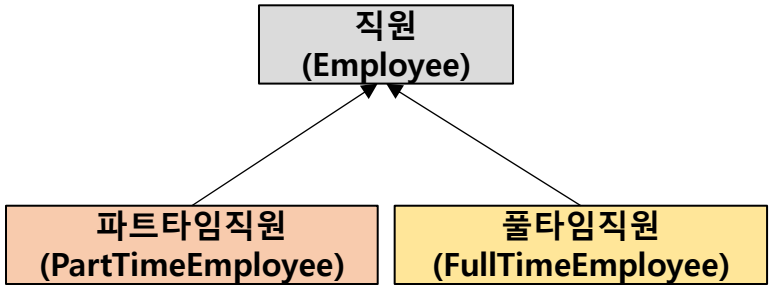
- **묵시적 형변환(Promotion, 자동 형변환)**
  - 자손타입 -> 조상타입 : 형변환 생략가능

- **명시적 형변환(Casting, 강제 형변환)**
  - 자손타입 <- 조상타입 : 형변환 생략불가

06. 참조 자료형 형변환

- **자동 형변환(Promotion)**
  - 프로그램 실행 도중에 자동 타입 변환이 일어나는 것






```
PartTimeEmployee ptEmp = new PartTimeEmployee();
Employee emp = ptEmp;
```

```
Employee emp = new PartTimeEmployee();
```

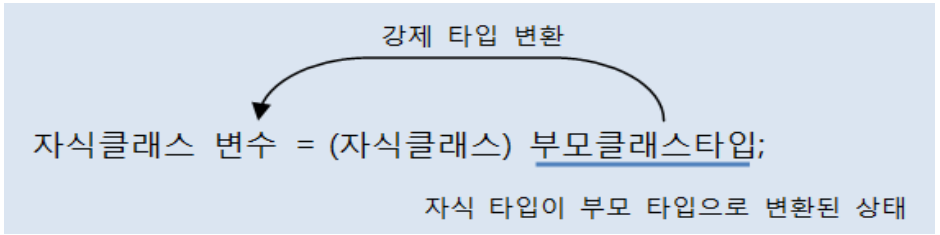
  - 매개변수의 다형성(매개변수가 클래스 타입일 경우)
    - 해당 클래스의 객체 대입이 원칙이나 자식 객체 대입하는 것도 허용, 자동 타입 변환

```
class Company {
      void hire(Employee emp) {
          System.out.println(emp.getName() + "을 고용하다");
      }
}
```

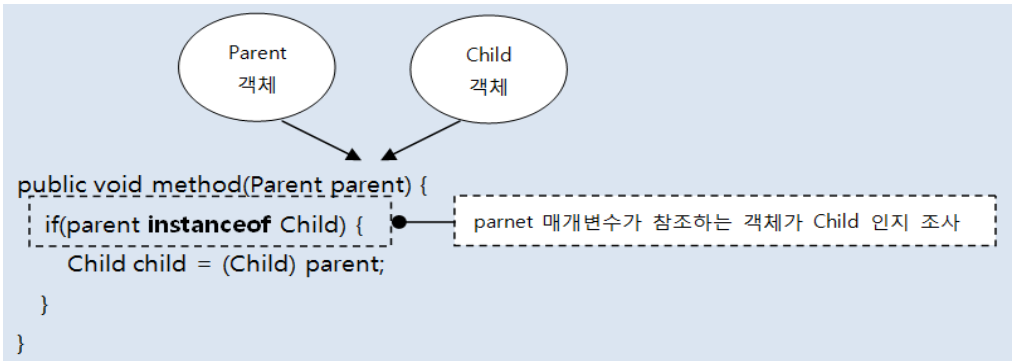
```
FullTimeEmployee fullEmp = new FullTimeEmployee ();

Company com = new Company();
com.hire(fullEmp);          // 자동 타입 변환 발생 ( Employee emp = fullEmp )
```

06. 참조 자료형 형변환

- **강제 형변환(Casting)**
  - 부모 타입을 자식 타입으로 변환하는 것
  - 자식 타입을 부모 타입으로 자동 변환 후, 다시 자식 타입으로 변환할 때만 가능



- **객체 타입 확인(instanceof 연산자)**
  - 부모 타입이면 모두 자식 타입으로 강제 타입 변환할 수 있는 것 아님
  - 먼저 자식 타입인지 확인 후 강제 타입 실행해야 함 - ClassCastException 예외 발생 가능



09. 예외 처리

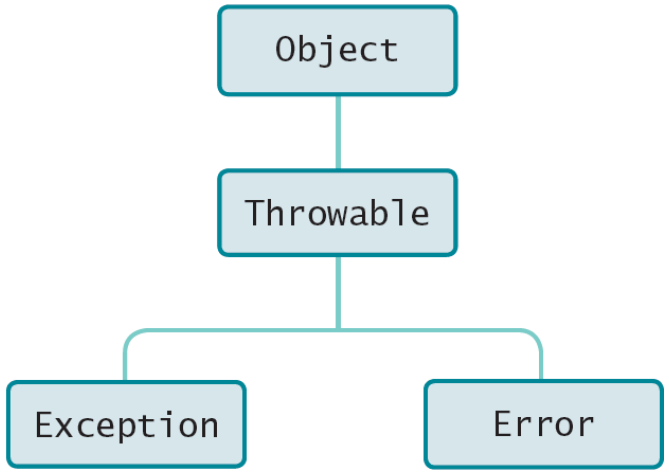
01/ 예외와 예외 클래스

02/ 예외처리 방법

03/ 사용자 정의 예외

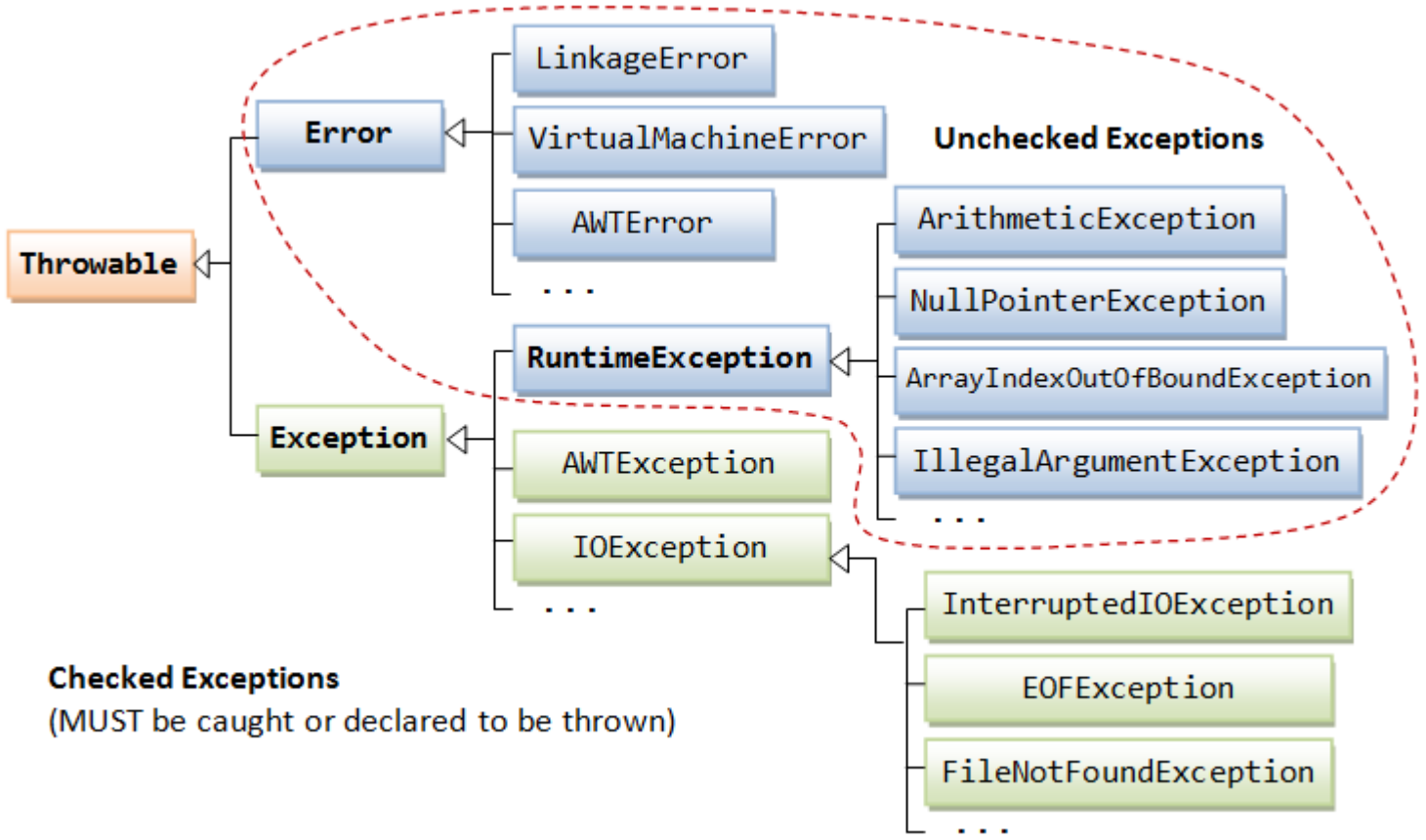
01. 예외와 예외 클래스

- **에러와 예외**
  - 오류(Error)
    - 시스템에 비정상적인 상황이 생겼을 때 발생
    - 시스템 레벨에서 발생하기 때문에 심각한 수준의 오류
    - 개발자가 미리 예측하여 처리할 수 없으므로 오류에 대한 처리를 하지 않아도 됨
  - 예외(Exception)
    - 개발자가 구현한 로직에서 발생
    - 발생할 상황을 미리 예측하여 처리할 수 있음



01. 예외와 예외 클래스

- 예외 클래스



01. 예외와 예외 클래스

- **예외 종류**

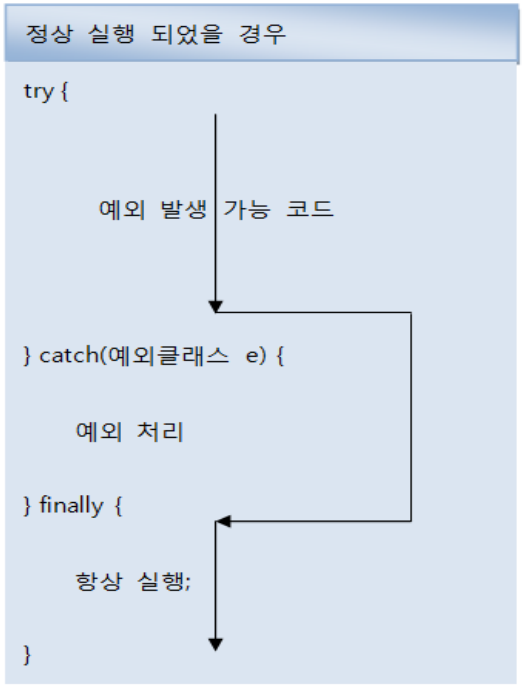
종류	Compile Time Exception (Checked Exception)	RunTime Exception (UnChecked Exception)
확인시점	컴파일 단계	실행 단계
처리 여부	반드시 예외를 처리해야 함	반드시 예외 처리를 할 필요는 없음
예외발생시 트랜잭션 처리	Rollback 하지 않음	Rollback 함
대표 예외	Exception 하위 클래스 중 RuntimeException을 제외한 모든 예외 IOException, SQLException	RuntimeException 하위 클래스 ArithmeticException, NullpointerException

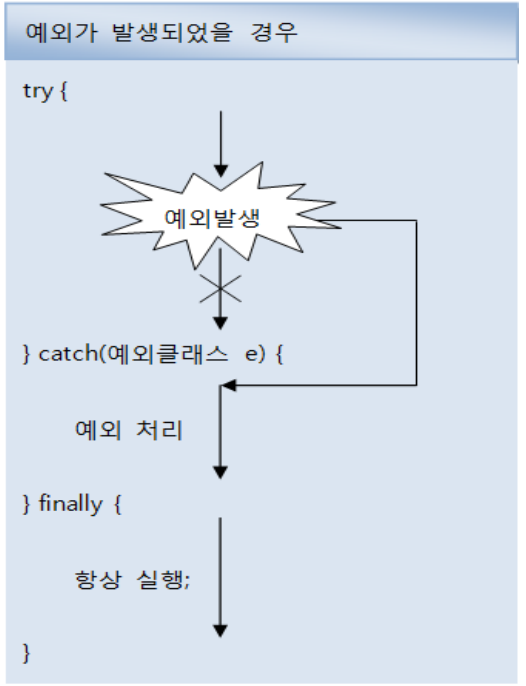
02. 예외 처리 방법

- **try~catch~finally**
  - 예외(Exception)이 발생한 메소드 내에서 직접 처리

형식

```
try {
      // 예외 발생 가능한 문장;

} catch(예외 객체) {
      // 예외 발생시 수행할 문장;

} catch(예외 객체) {
      // 예외 발생시 수행할 문장;

} finally {
      // 예외 발생 여부와 상관없이 수행할 문장;
      // 자원 반납
}
```



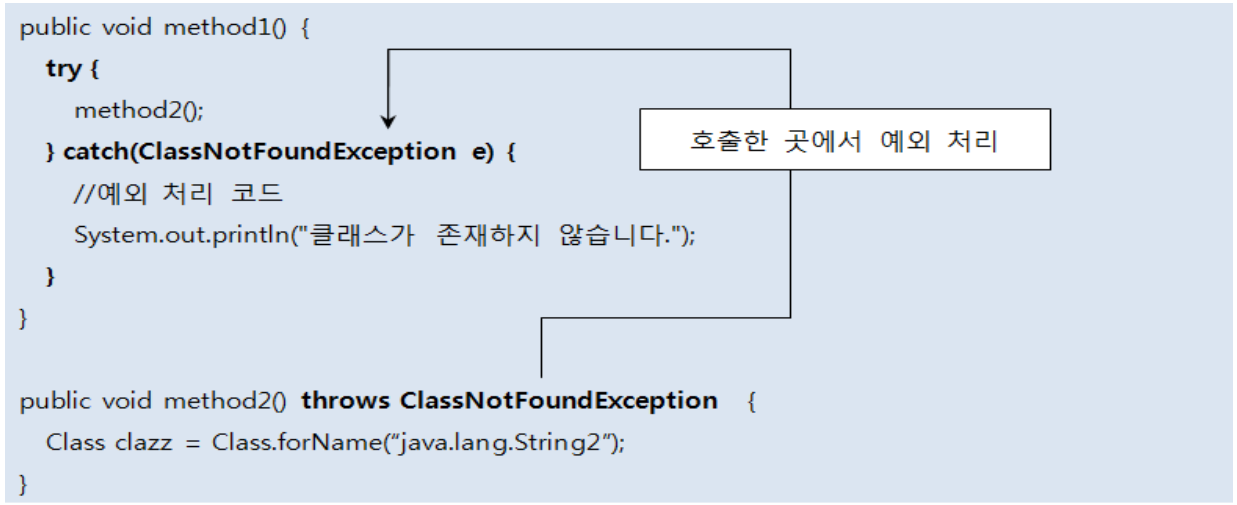


02. 예외 처리 방법

- **throws**
  - 상위 메서드로 예외처리를 미루는 방법

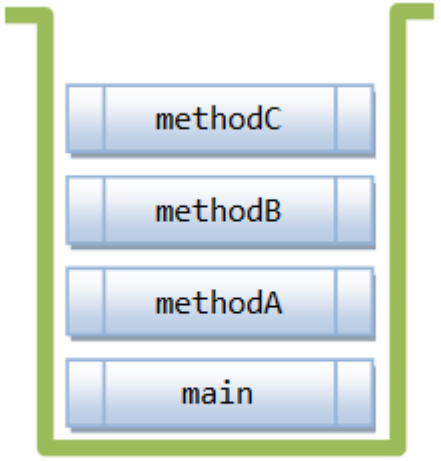
형식

```
[접근지정 제어자] [사용 제어자] 반환자료형 메소드명(매개변수 리스트) throws Exception클래스명 {
        …
}
```

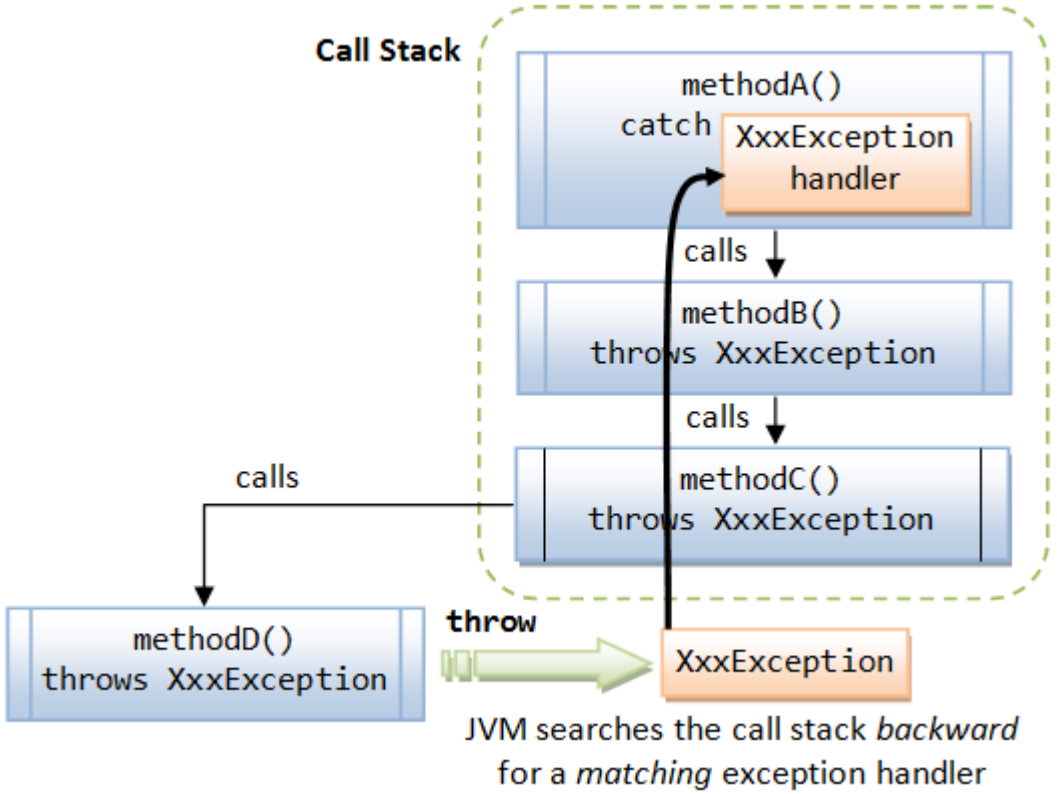


02. 예외 처리 방법

- 예외와 콜스택(Call Stack)



Method Call Stack
(Last-in-First-out Queue)



JVM searches the call stack *backward*
for a *matching* exception handler

03. 사용자 정의 예외

- **사용자 정의 예외 개념**
  - 자바 표준 API에서 제공하지 않는 예외

- **사용자 정의 예외 클래스 정의**

형식

```
[접근지정 제어자] [사용 제어자] class 클래스명 extends 예외클래스명 {

}
```

- **사용자 정의 예외 발생**

형식

```
throw 예외객체명;
```

10. 주요 API 활용

01/ 주요 API 소개

02/ Object 클래스

03/ String 클래스

04/ Wrapper 클래스

05/ Class 클래스

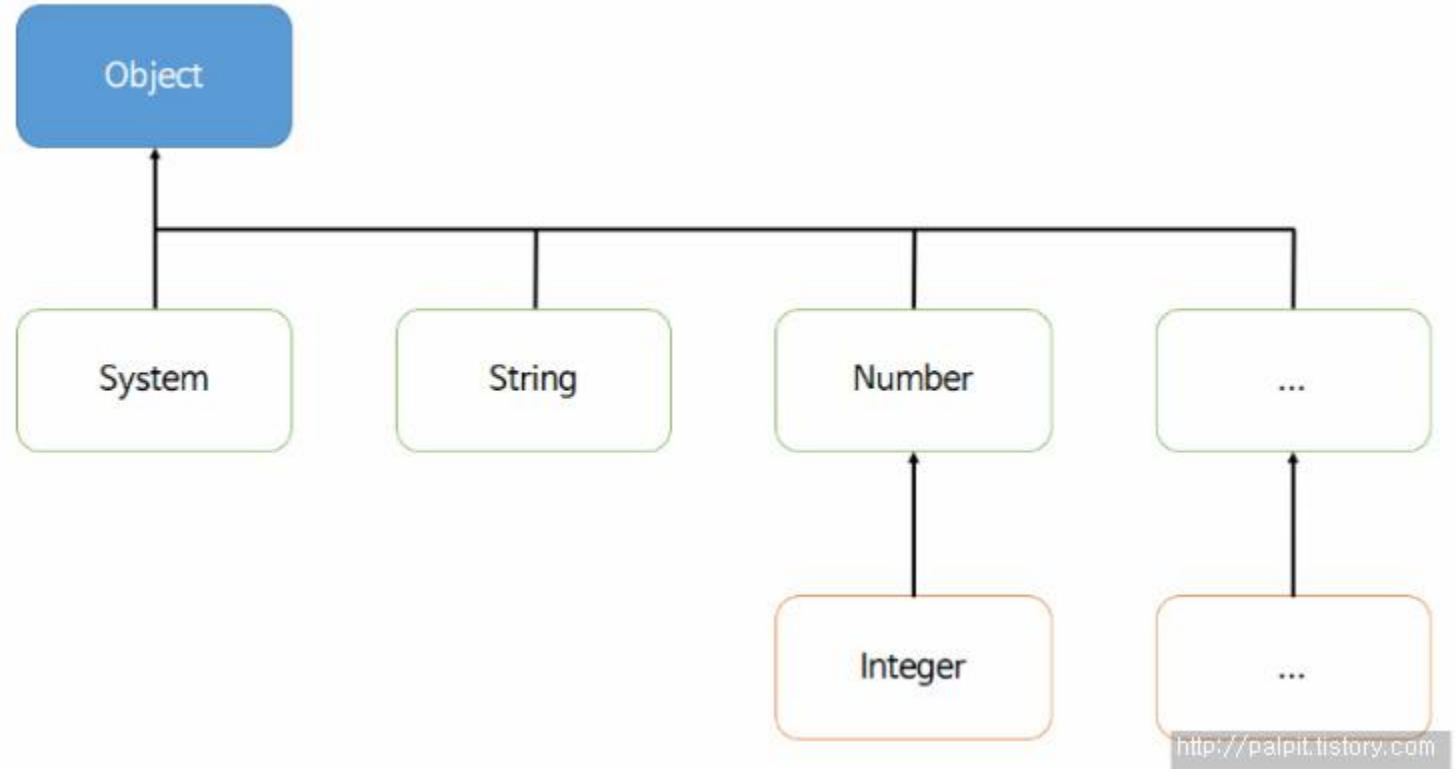
01. 주요 API 소개

- **자바 주요 API 패키지**

패키지	설명
java.applet	애플릿을 생성하는 데 필요한 클래스, 애플릿과 상호작용하는 애플릿 클래스와 인터페이스를 담고 있다.
java.awt	사용자 인터페이스를 생성하고 그래픽과 이미지를 사용하는 클래스를 담고 있다.
java.awt.event	이벤트를 처리하는 인터페이스와 클래스를 담고 있다.
java.awt.image	이미지를 생성하고 수정하는 클래스를 담고 있다.
java.io	데이터 입출력과 파일 시스템을 담고 있다.
java.lang	자바 프로그램 언어를 사용하는 기본 클래스를 담고 있다.
java.net	응용 프로그램끼리 통신할 수 있게 하는 클래스를 담고 있다.
java.rmi	원격 객체의 메서드를 호출하고, 분산 자바 프로그램을 제작하는 패키지와 인터페이스를 담고 있다.
java.security	프로그램 보안(데이터 암호화와 복호화, 접근 권한)에 필요한 클래스와 인터페이스를 담고 있다.
java.util	시간 조작, 난수 발생, 문자열 토큰화 등 관련 유틸리티 클래스를 담고 있다.
java.util.zip	표준 zip, gzip 파일 형식을 읽고 쓸 수 있는 클래스를 담고 있다.

02. Object 클래스

- **Object 클래스**
  - 자바에서 모든 클래스의 최상위 클래스
  - 새로운 클래스를 정의하는 경우 자동으로 클래스 Object가 상위 클래스



02. Object 클래스

- **Object 클래스 주요 메소드**

메소드	설명
boolean equals(Object obj)	두 개의 객체가 같은지 비교하여 같으면 true를, 같지 않으면 false를 반환
String toString()	현재 객체의 문자열을 반환한다.
protected Object clone()	객체를 복사한다
protected void finalize()	가비지 컬렉션 직전에 객체의 리소스를 정리할 때 호출
Class getClass()	객체의 클래스형을 반환
int hashCode()	객체의 코드값을 반환
void notify()	wait된 스레드 실행을 재개할 때 호출
void notifyAll()	wait된 모든 스레드 실행을 재개할 때 호출
void wait()	스레드를 일시적으로 중지할 때 호출
void wait(long timeout)	주어진 시간만큼 스레드를 일시적으로 중지할 때 호출
void wait(long timeout, int nanos)	주어진 시간만큼 스레드를 일시적으로 중지할 때 호출

02. Object 클래스

- **equals() 메서드**
  - 기본적으로 == 연산자와 동일한 결과 리턴 (번지 비교)

```
public boolean equals(Object obj) { ... }
```

```
Object obj1 = new Object();
Object obj2 = new Object();

boolean result = obj1.equals(obj2);
                 기준 객체    비교 객체

boolean result = (obj1 == obj2)
```

결과가 동일

Object 클래스

- **toString() 메서드**
  - 클래스의 요약 정보 출력 용도로 사용

```
클래스타입@해시코드
```

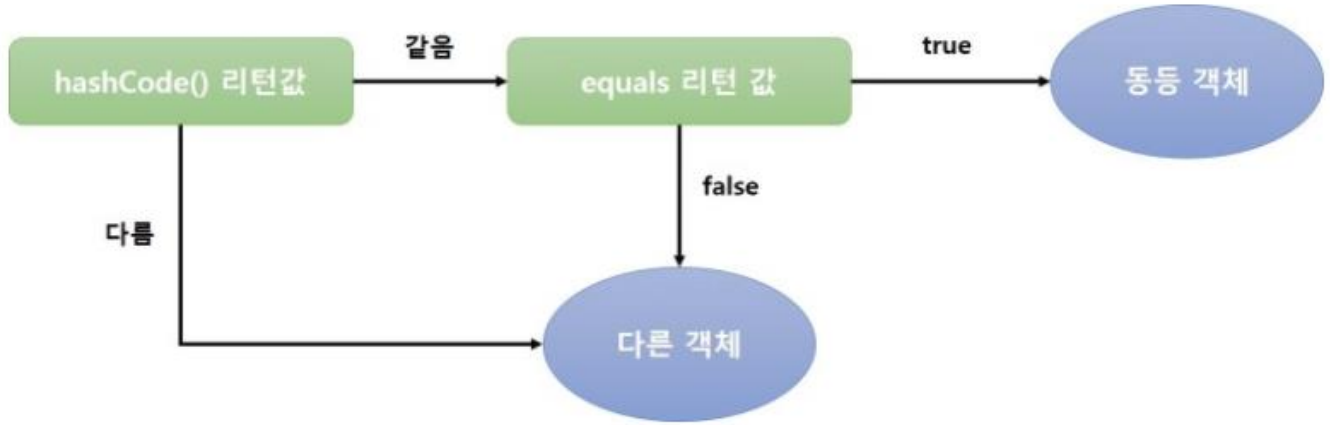
  - 주로 오버라이드해서 사용함
  - 객체가 가지고 있는 값을 문자열로 만들어서 반환하는 메소드

```
class Employee {
    private int employeeNo;
    private String name;

    @Override
     public String toString() {
         return this.employeeNo + name;
     }
}
```

02. Object 클래스

- **hashCode() 메서드**
  - 객체의 해시코드 : 객체 식별할 정수값, 객체의 메모리 번지 이용해 해시코드 생성
  - 객체 저장 데이터 동일여부 확인시 오버라이드 필요

  - 논리적 동등 비교
    - hashCode()를 오버라이드
    - HashSet, HashMap, Hashtable 등에서 사용

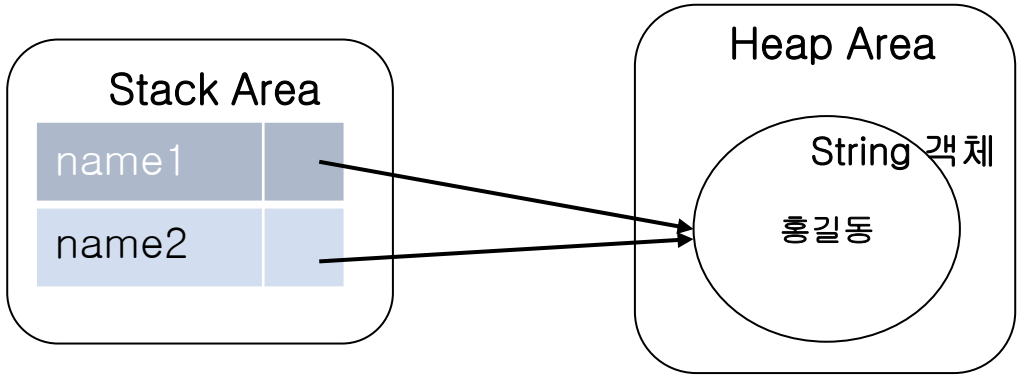


03. String 클래스

- **String 클래스**
  - 문자열을 저장하는 클래스

- **문자 리터럴 이용**

```
String name1 = “홍길동”
String name2 = “홍길동”
```



String 클래스

- **StringBuffer/StringBuilder 클래스**
  - 버퍼에 문자열 저장, 버퍼 내부에서 추가, 수정, 삭제 작업 가능
  - 멀티 스레드환경 : StringBuffer 사용
  - 단일 스레드환경 : StringBuilder 사용

```
StringBuilder sb = new StringBuilder();
StringBuilder sb = new StringBuilder(16);
StringBuilder sb = new StringBuilder("Java");
```

- **StringTokenizer 클래스**
  - 문자열이 특정 구분자로 연결되어 있는 경우, 구분자를 기준으로 분리하기 위해 사용
  - String.split() : 정규표현식으로 구분
  - StringTokenizer 클래스 : 문자로 구분

04. Wrapper 클래스

- **Wrapper 클래스**
  - 기본 자료형을 참조 자료형으로 포장한 포장 클래스

구분	정수형				실수형		문자형	논리형
기본형	byte	short	int	long	float	double	char	boolean
참조형	Byte	Short	Integer	Long	Float	Double	Character	Boolean

  - 자동 박싱(boxing) : 기본 자료형에서 래퍼 객체로 자동 변환
    - Double radius = 2.59; // boxing
  - 자동 언박싱(unboxing) : 래퍼 객체에서 기본 자료형으로 자동 변환
    - double r = radius; // unboxing

Class 클래스

- **Class 클래스**
  - 클래스와 인터페이스의 메타 데이터를 관리하는 클래스
  - 클래스명, 생성자/필드/메소드 정보 등을 관리
  - 리플렉션(Reflection)을 위해 사용됨

- **getClass(), forName() 메소드**
  - Object 클래스내에 있는 getClass() 이용하여 클래스 정보 반환

```
Class clazz = obj.getClass();
```

- **리플렉션(Reflection)**
  - Class 객체를 이용하여 클래스 정보를 얻어올 수 있음

```
Constructor[] constructors = clazz.getDeclaredConstructors();
Field[] fields= clazz.getDeclaredFields();
Method[] methods = clazz.getDeclaredMethods()_;
```

Class 클래스

- **newInstance() 메소드**
  - new 연산자를 사용하지 않고 동적 객체 생성
  - 런타임 시에 클래스 이름이 결정되는 경우에 사용

```
Class clazz = Class.forName("클래스명");
Object obj = new clzss.netInstance();
```

11. 컬렉션 프레임워크

01/ 컬렉션 프레임워크 기

02/ List 컬렉션

03/ Set 컬렉션

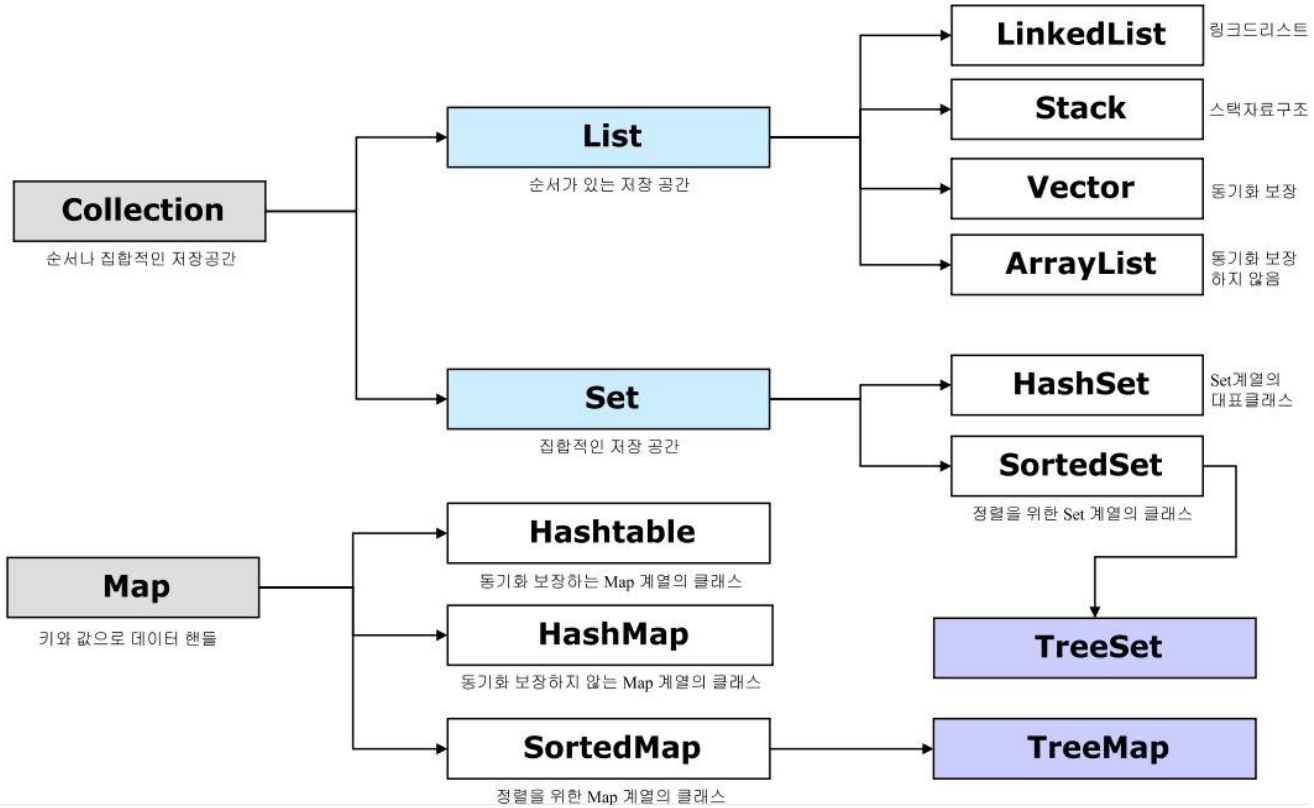
04/ Map 컬렉션

05/ Iterator 인터페이스

06/ 제네릭

01. 컬렉션 프레임워크 개요

- **컬렉션 프레임워크란?**
  - 여러 객체 원소의 삽입과 삭제가 편리한 자료 구조를 지원하는 인터페이스
  - 인터페이스를 구현한 여러 클래스로 컬렉션 프레임워크를 구성



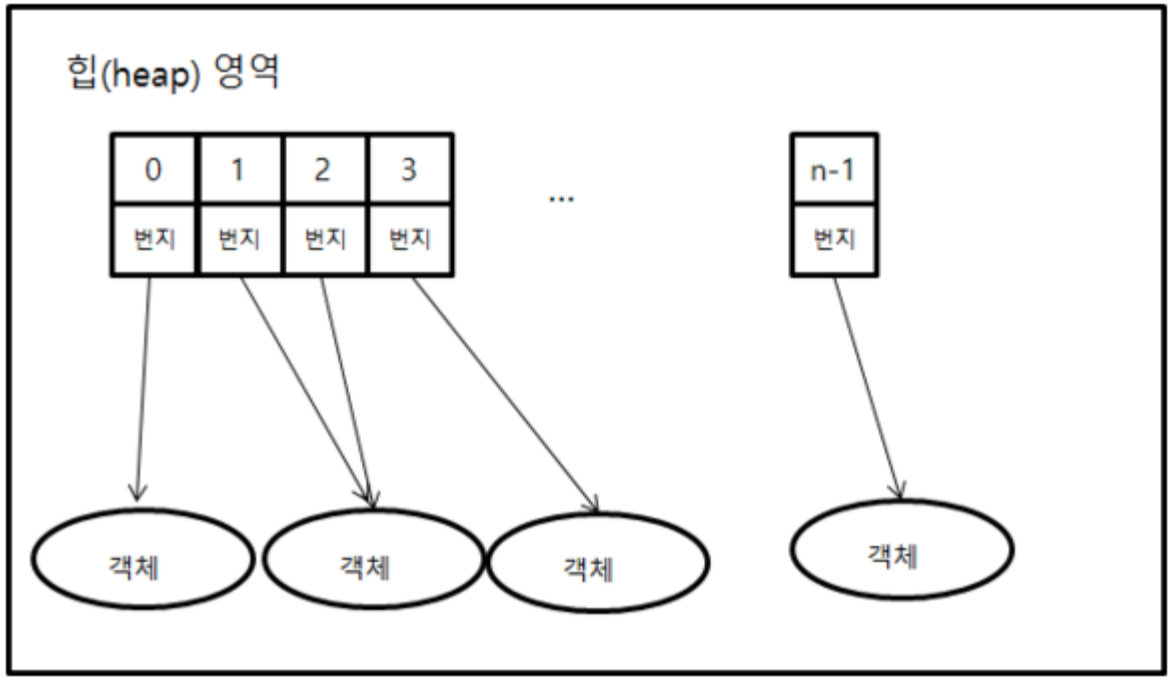
01. 컬렉션 프레임워크 개요

- **컬렉션 프레임워크 특징**
  - 인터페이스 Collection
    - 인터페이스 Set과 List 그리고 Queue로 분류, 객체 하나를 컬렉션의 원소로 처리
    - 인터페이스 Set은 수학의 집합을 표현한 인터페이스, 원소의 중복을 허용하지 않음
    - 인터페이스 List와 Queue는 원소의 중복을 허용하고 순서도 의미가 있음
  - 인터페이스 Map
    - 키(key)와 값(value)의 쌍으로 구성된 자료 구조를 컬렉션의 원소로 처리

인터페이스	설명	구현 클래스
List	순서가 있는 데이터의 집합 데이터의 중복 가능	ArrayList, LinkedList, Stack, Vector
Set	순서를 유지하지 않는 데이터의 집합 데이터의 중복 불가능	HashSet, TreeSet
Map	키와 값의 쌍으로 이루어진 데이터의 집합 순서는 유지되지 않음 키는 중복 불가능, 값은 중복 가능	HashMap, TreeMap, Hashtable, Properties

02. List 컬렉션

- **List 인터페이스**
  - 순차적으로 나열된 원소를 처리하는 구조
  - 인덱스로 관리, 중복해서 객체 저장 가능
  - 구현 클래스 : ArrayList, Vector, LinkedList



02. List 컬렉션

- **List 인터페이스 주요 메소드**

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 맨 끝에 추가
	void add(int index, E e)	주어진 인덱스에 객체 추가
객체 검색	boolean contain(Object o)	주어진 객체가 저장되어있는 여부 반환
	E get(int index)	주어진 인덱스에 저장된 객체 반환
	boolean isEmpty()	컬렉션이 비어있는지 여부 반환
	int size()	저장되어 있는 객체수 반환
객체 삭제	void clear()	저장된 모든 객체 삭제
	E remove(int index)	주어진 인덱스에 저장된 객체 삭제
	boolean remove(Object o)	주어진 객체 삭제

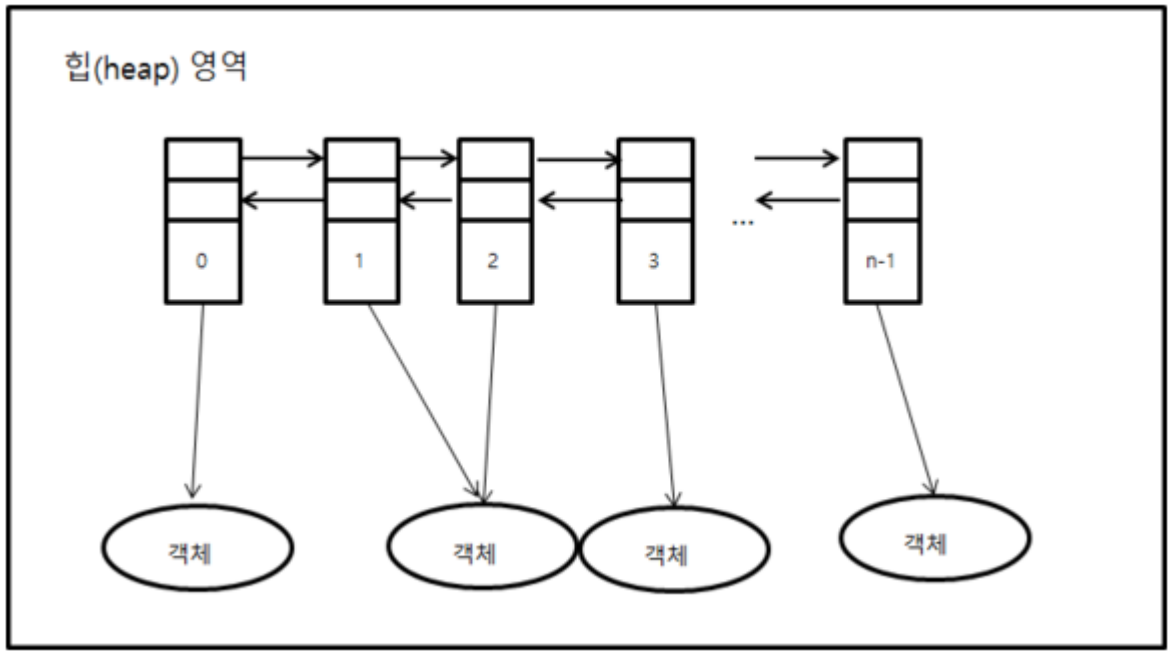
02. List 컬렉션

- **ArrayList 클래스**
  - List 인터페이스의 구현 클래스
  - 객체 추가하면 객체가 인덱스로 관리
  - 배열은 생성시 설정한 크기가 불변이나 ArrayList는 자동으로 크기 증가

- **Vector 클래스**
  - ArrayList와 동일한 내부 구조
  - ArrayList와 차이점은 동기화된 메소드로 구성되어 있어 멀티쓰레드에서 동시에 실행할 수 없음

02. List 컬렉션

- **LinkedList 클래스**
  - 인접 참조를 연결해서 체인처럼 관리



03. Set 컬렉션

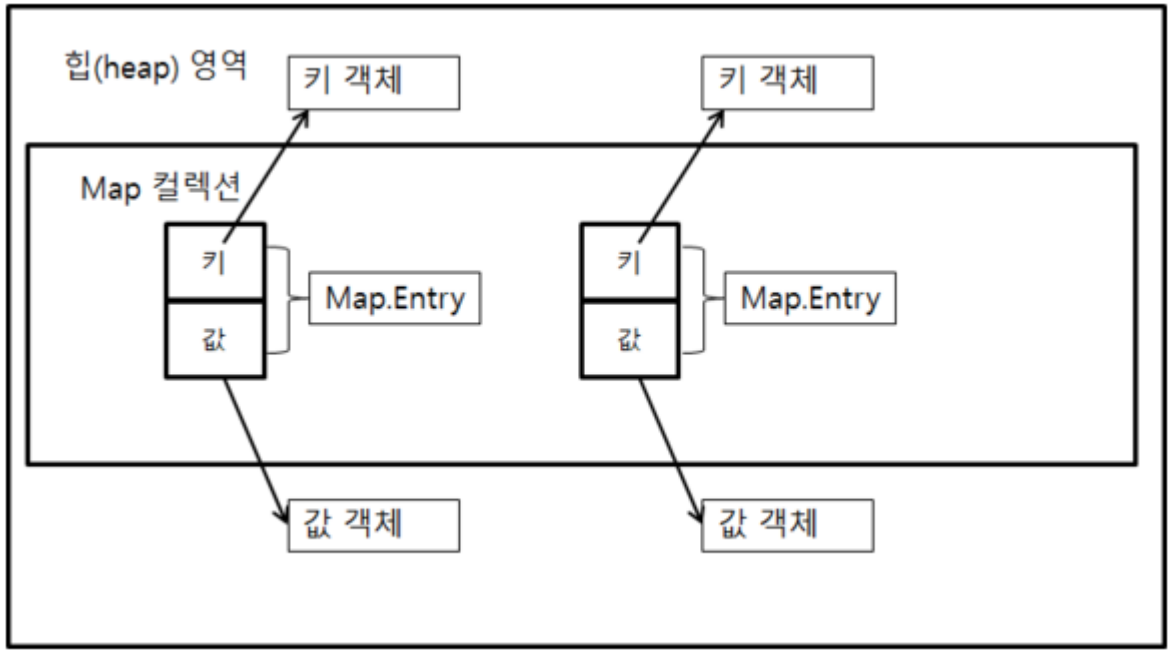
- **Set 인터페이스**
  - 중복을 허용하지 않는 컬렉션
  - 저장 순서가 유지되지 않음, 하나의 null만 저장 가능
  - 구현 클래스 : HashSet, LinkedHashSet, TreeSet

- **Set 인터페이스 주요 메소드**

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 맨 끝에 추가
객체 검색	boolean contain(Object o)	주어진 객체가 저장되어있는 여부 반환
	boolean isEmpty()	컬렉션이 비어있는지 여부 반환
	int size()	저장되어 있는 객체수 반환
객체 삭제	void clear()	저장된 모든 객체 삭제
	boolean remove(Object o)	주어진 객체 삭제

04. Map 컬렉션

- **Map 인터페이스**
  - 키(key)와 값(value)으로 구성된 Map.Entry 객체를 저장하는 구조
  - 메서드 put(key, value)으로 저장, 메서드 get(key)로 조회
  - 키는 중복될 수 없지만 값은 중복 저장 가능
  - 구현 클래스 : HashMap, Hashtable, LinkedHashMap, Properties, TreeMap



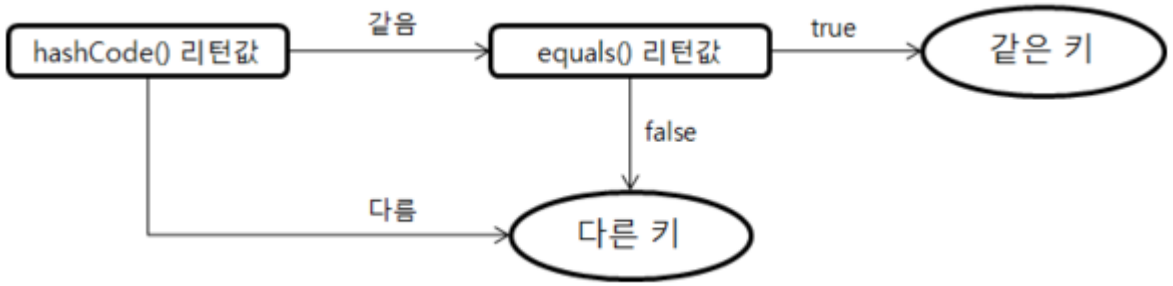
04. Map 컬렉션

- **Map 인터페이스 주요 메소드**

기능	메소드	설명
객체 추가	V put(K key, V value)	주어진 키와 값을 추가
객체 검색	boolean containKey(Object key)	주어진 키가 있는지 여부 반환
	boolean containValue(Object value)	주어진 값이 있는지 여부 반환
	Set<Map.Entry<K, V>> entrySet()	모든 Map.Entry 객체를 Set을 반환
	Set<K> keySet()	모든 키의 값을 Set으로 반환
	V get(Object key)	주어진 키에 해당하는 값 반환
	boolean isEmpty()	컬렉션이 비어있는지 여부 반환
	int size()	저장되어 있는 객체수 반환
	Collection<V> values()	저장된 모든 값을 Collection으로 반환
객체 삭제	void clear()	저장된 모든 Map.Entry(키와 값)을 삭제
	boolean remove(Object key)	주어진 키와 일치하는 Map.Entry 삭제

04. Map 컬렉션

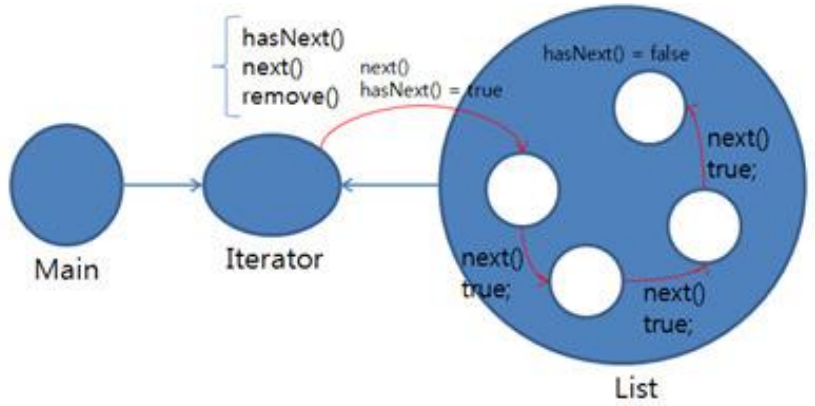
- **HashMap 클래스**
  - Map 인터페이스를 구현한 클래스
  - 키로 사용할 객체는 hashCode()와 equals() 메소드를 재정의하여 동등 객체가 될 조건을 정해야 함



- **Hashtable 클래스**
  - HashMap과 동일한 내부구조
  - HashMap과의 차이점은 동기화된 메소드로 구성되어 있어 멀티쓰레드에서 동시에 실행할 수 없음

05. Iterator 인터페이스

- **Iterator 인터페이스**
  - 모든 컬렉션으로부터 정보를 얻을 수 있는 인터페이스
  - Set과 List 안의 데이터를 조회



- **Iterator 인터페이스의 주요 메소드**

기능	메소드	설명
객체 검색	boolean hasNext()	반복할 객체가 있으면 true, 없으면 false
	E next()	컬렉션에서 하나의 객체반환
객체 삭제	void remove()	객체 삭제

06. 제네릭(Generic)

- **제네릭 개념**
  - 일반적인 코드를 작성, 이 코드를 다양한 객체에 대하여 재사용하는 프로그래밍 기법
  - 다양한 타입의 객체들을 다루는 메소드나 컬렉션에 컴파일시 자료형 체크를 해주는 기능
  - 객체의 타입 안정성을 높이고 형변환의 번거로움이 줄어듬
  - 클래스 내부에서 사용할 자료형을 나중에 인스턴스를 생성할 때 확정
  - 제네릭 타입이 인스턴스화 될 때 컴파일러는 타입 파라미터 정보를 제거하여 호환성 유지

```
List list = new ArrayList();
list.add("hello");
String str = (String) list.get(0);
```

```
List<String> list = new ArrayList<String>();
list.add("hello");
String str = list.get(0);
```

- **타입 매개변수**
  - E – Element
  - K – Key
  - N – Number
  - T – Type
  - V - Value

06. 제네릭(Generic)

- **클래스 제네릭 타입**

```
class ClassGenericType <T> {
	private T t;

	public void set(T t) {
		this.t = t;
	}
	public T get() {
		return t;
	}
}
```

- **인터페이스 제네릭 타입**

```
interface InterfaceGenericType <T1, T2> {
	T1 doSomething(T2 t);
	T2 doSomething2(T1 t);
}
```

06. 제네릭(Generic)

- **메소드 제네릭 타입**
  - 필요한 메소드에 제네릭 타입을 선언
  - 메소드 매개변수에 타입 매개변수 T가 선언되어 있으면 메소드의 반환 자료형 앞에 제네릭 타입을 선언

```
class MethodGenericType {
	public static <T> int methodGeneric(T[] list, T item) {
		int count = 0;
		for (T t : list) {
			if (item == t) {
				count++;
			}
		}
		return count;
	}
}
```

06. 제네릭(Generic)

- **와일드 제네릭 타입**
  - ?는 알 수 없는 타입
  - <?>
    - 모든 객체 자료형, 내부적으로는 Object로 인식
  - <? super 객체자료형>
    - 명시된 객체 자료형의 상위 객체, 내부적으로는 Object로 인식
  - <? extends 객체자료형>
    - 명시된 객체 자료형을 상속한 하위객체, 내부적으로는 명시된 객체 자료형으로 인식

06. 제네릭(Generic)

- **와일드 제네릭 타입**

```
class WildcardGenericType {

	public List<? extends Number> method1() {
		return new ArrayList<Long>();
	}

	// Bounded wildcard parameterized type
	public <T> List<? extends String> method2(T param) {
		return new ArrayList<String>();
	}

	// Unbounded wildcard parameterized type
	public List<?> method3() {
		return new ArrayList<>();
	}
}
```

12. 쓰레드

01/ 쓰레드 개념

02/ 쓰레드 생성

03/ 쓰레드 상태

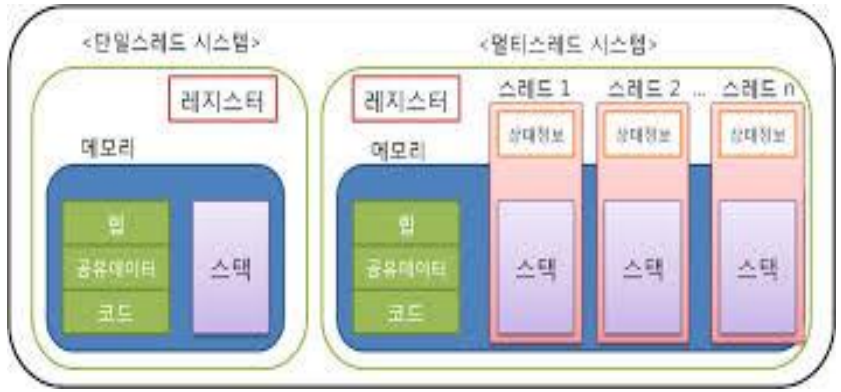
04/ 동기화

01. 쓰레드(Thread) 개념

- **쓰레드 개념**
  - 프로세스 내에서 실행되는 흐름의 단위를 말한다. 일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행할 수 있다. 이러한 실행 방식을 멀티스레드(multithread)라고 한다.

- **프로세스(process)**
  - 실행 중인 프로그램
  - 사용자가 작성한 프로그램이 운영체제에 의해 메모리 공간을 할당 받아 실행 중

- **쓰레드(thread)**
  - 프로세스 내에서 실제로 작업을 수행하는 주체
  - 모든 프로세스에는 1개 이상의 쓰레드가 존재하여 작업 수행
  - 두 개 이상의 쓰레드를 가지는 프로세스를 멀티 쓰레드라고 함



02. 쓰레드 생성

- **Thread 클래스 상속**
  - 쓰레드를 상속받는 클래스를 정의
  - run() 메소드를 오바라이드하여 내용부를 구현

```
class  MyThread extends Thread {

        public void run() {
              // 쓰레드에서 수행할 기능
        }
}
```

```
MyThread th = new MyThread();
th.start();                    //  쓰레드 실행
```

02. 쓰레드 생성

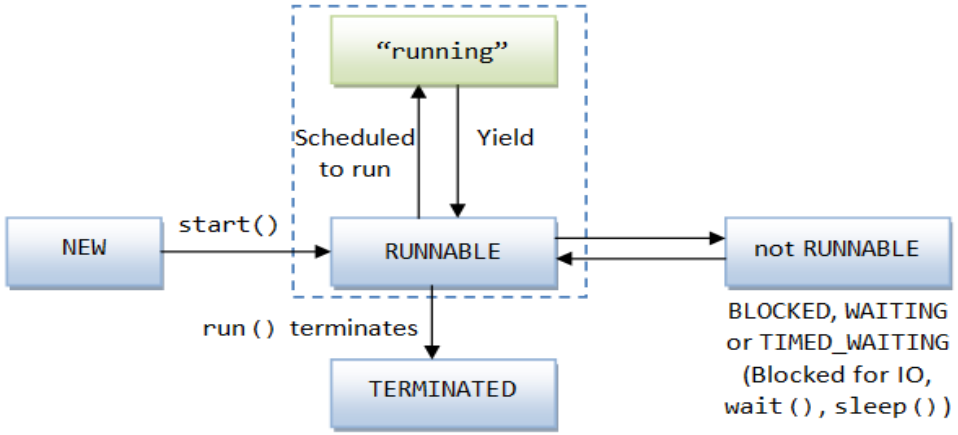
- **Runnable 인터페이스 구현**
  - Runnable을 구현하는 클래스를 정의
  - run() 메소드를 오바라이딩하여 내용부를 구현

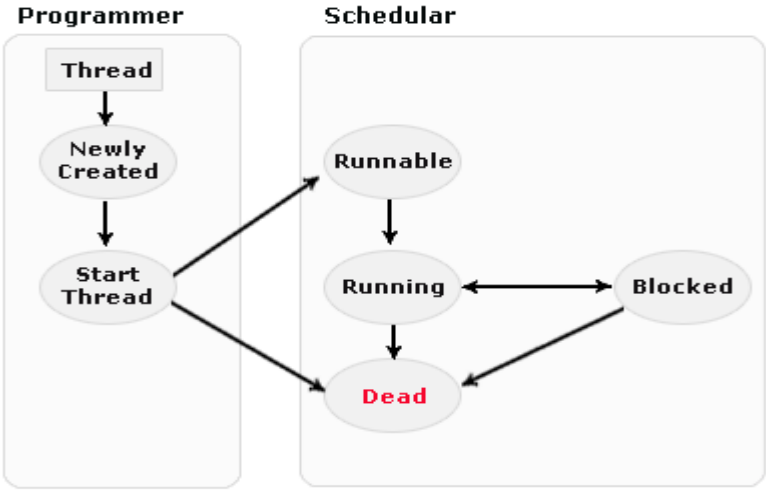
```
class  MyRunnable implements Runnable {

        public void run() {
              // 쓰레드에서 수행할 기능
        }
}
```

```
Thread th = new Thread(new MyRunnable());
th.start();                    //  쓰레드 실행
```

03. 쓰레드 상태

- **쓰레드 상태 전이**

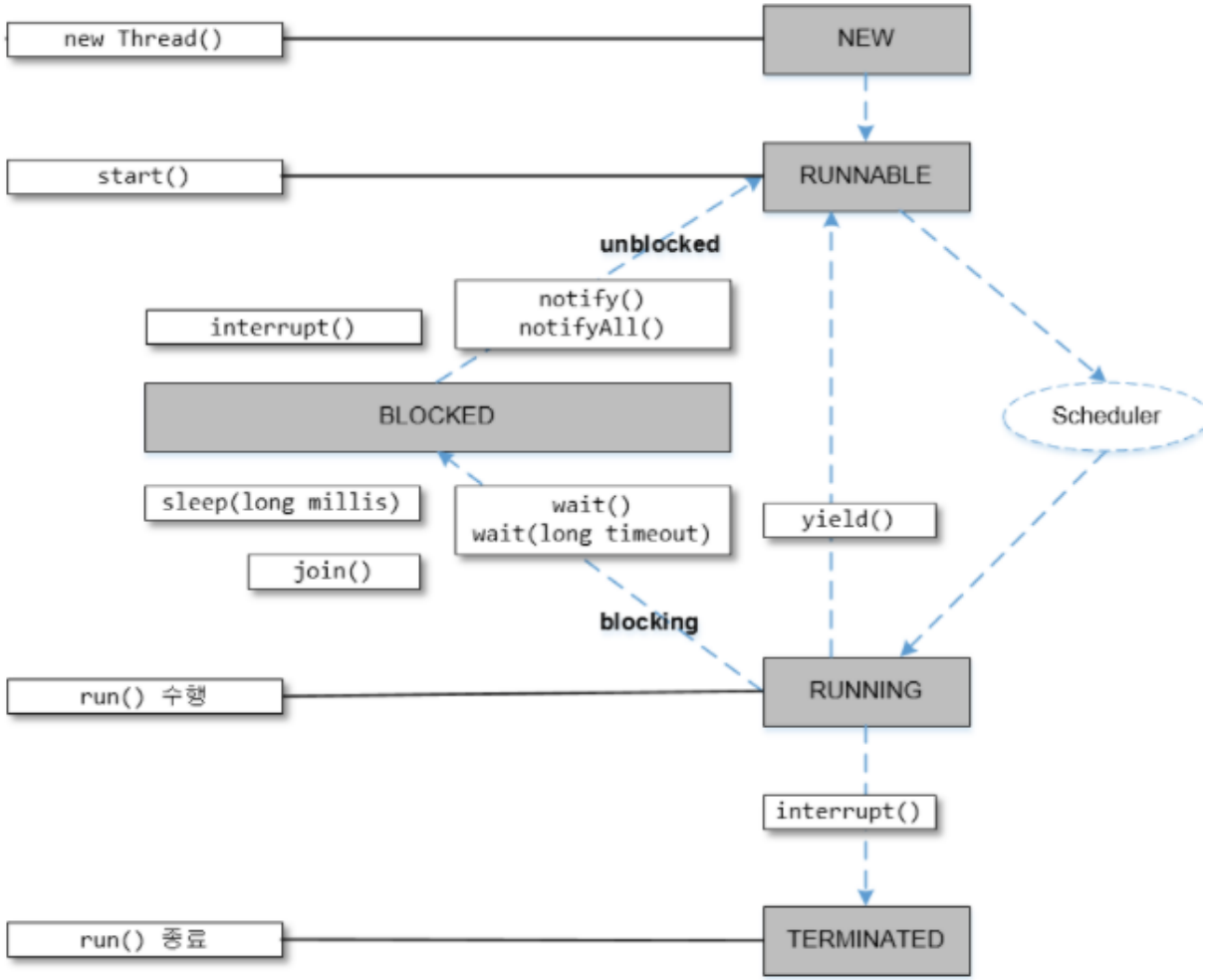




상태	열거상수	설명
객체생성	NEW	쓰레드는 객체가 생성, start() 메소드가 호출되지 않은 상태
실행대기	RUNNABLE	메소드 start()가 호출되면 이동, 실행상태로 이동 가능한 상태
일시정지	WAITING	다른 쓰레드가 통지할때까지 기다리는 상태
	TIMED_WEIGHTING	주어진 시간 동안 정지된 상태
	BLOCKED	사용하고자 하는 객체의 락이 풀릴 때까지 기다리는 상태
종료	TERMINATED	쓰레드가 완전히 종료된 상태 더 이상 NEW 또는 RUNNABLE 등의 다른 상태로 전이가 불가능한 상태

03. 쓰레드 상태

- **쓰레드 관련 주요 메소드**



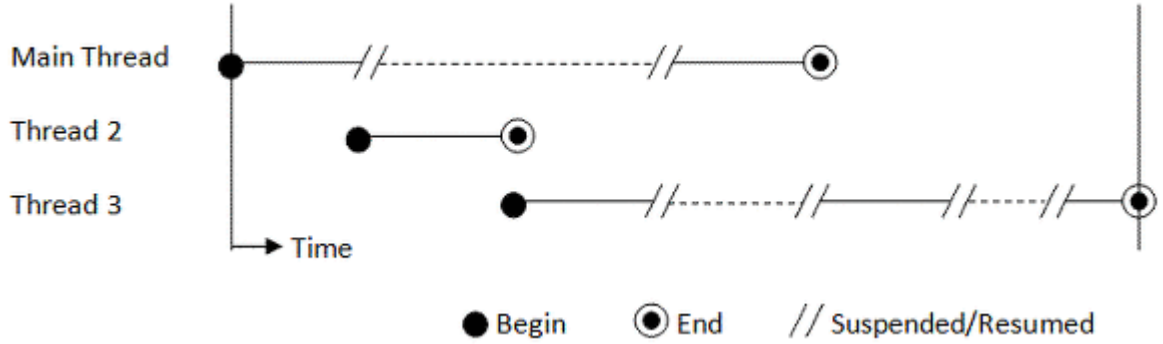
03. 쓰레드 상태

- **쓰레드 관련 주요 메소드**

메소드	설명
interrupt()	쓰레드가 일시 정지 상태가 되면, InterruptedException 예외가 발생하여 실행 대기 상태로 전이하거나 종료 상태로 전이
notify() notifyAll()	동기화 블록에서 wait() 메소드에 의해 일시 정지 상태로 된 쓰레드를 실행대기 상태로 전이
sleep(long mills) sleep(long mills), int nanos)	주어진 시간 동안 쓰레드를 일시 정지시킨 후, 시간이 지나면 다시 실행 대기 상태로 전이
join() join(long millis) join(long millis, int nanos)	join() 메소드를 호출한 쓰레드는 일시 정지 상태가 되며, 실행 대기 상태로 가려면 join() 메소드를 멤버로 가지는 쓰레드가 종료되거나 주어진 시간이 지나야 함
wait() wait(long millis) wait(long millis, int nanos)	동기화 블럭내에서 쓰레드를 일시정지 상태로 전이 후, 주어진 시간이 지나면 자동적으로 실행대기 상태로 전이
yield()	실행 중에 우선순위가 동일하거나 높은 순위의 다른 쓰레드에게 실행을 양복하고 대기 상태로 전이

04. 동기화

- **멀티 쓰레드와 동기화**
  - 여러 개의 쓰레드들이 있다면 이 쓰레드들은 서로 번갈아가면서 실행
  - 운영체제의 스케줄링 정책에 따라 쓰레드의 순서가 결정



  - 프로세스 내에서 실행되는 여러 개의 쓰레드간 공유되는 자원들이 있는 경우에 동기화 문제 발생
  - 임계영역
    - 멀티 쓰레드에 의해 공유자원이 참조할 수 있는 코드의 범위
    - 한번에 한 쓰레드만 접근이 가능한 영역

04. 동기화

- **synchronized**
  - 동기화 문제가 발생하는 최소 단위는 객체이며, 문제 발생 시점은 객체가 소유한 내부 변수
  - 동기화 문제를 해결하기 위해 모든 객체에 Lock을 포함, Lock은 모든 객체가 인스턴스화될 때 힙 영역에 객체가 저장될 때 자동으로 생성
  - Lock은 동기화가 필요한 부분에서 synchronized 키워드를 사용

```
// 1) 메소드의 동기화 방법
public synchronized void Method() {
        // 임계영역 코딩
}

// 2) 특정 블록의 동기화 방법
public void normalMethod() {
        synchronzied(/* 동기화할 객체 또는 클래스명 */) {
            // 임계영역 코딩
        }
}
```

04. 동기화

- **wait(), notify(), notifyAll() 메소드**
  - 동기화된 블록에서 쓰레드간의 제어권을 넘기기 위해 사용되는 메소드
  - synchronized 블록내에서만 사용 가능
  - synchronized 블록이 아닌 경우에 사용할 경우 java.lang.illegalMonitorStateException이 발생

  - wait() 메소드 : 어떤 객체에 대해 쓰레드 대기
  - notify() 메소드 : 대기중인 쓰레드가 있을 경우 우선순위가 높은 쓰레드 하나만을 깨움
  - notifyAll() 메소드 : 대기중인 스레드 전부를 깨움

13. 입출력 스트림

01/ 스트림

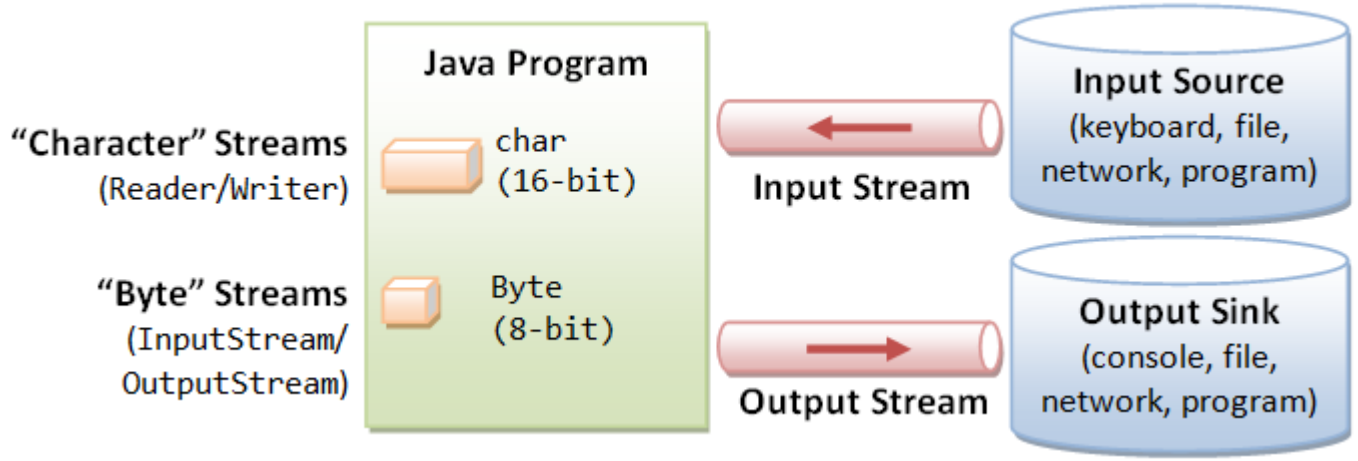
02/ 바이트 단위 스트림

03/ 문자 단위 스트림

04/ 보조 스트림

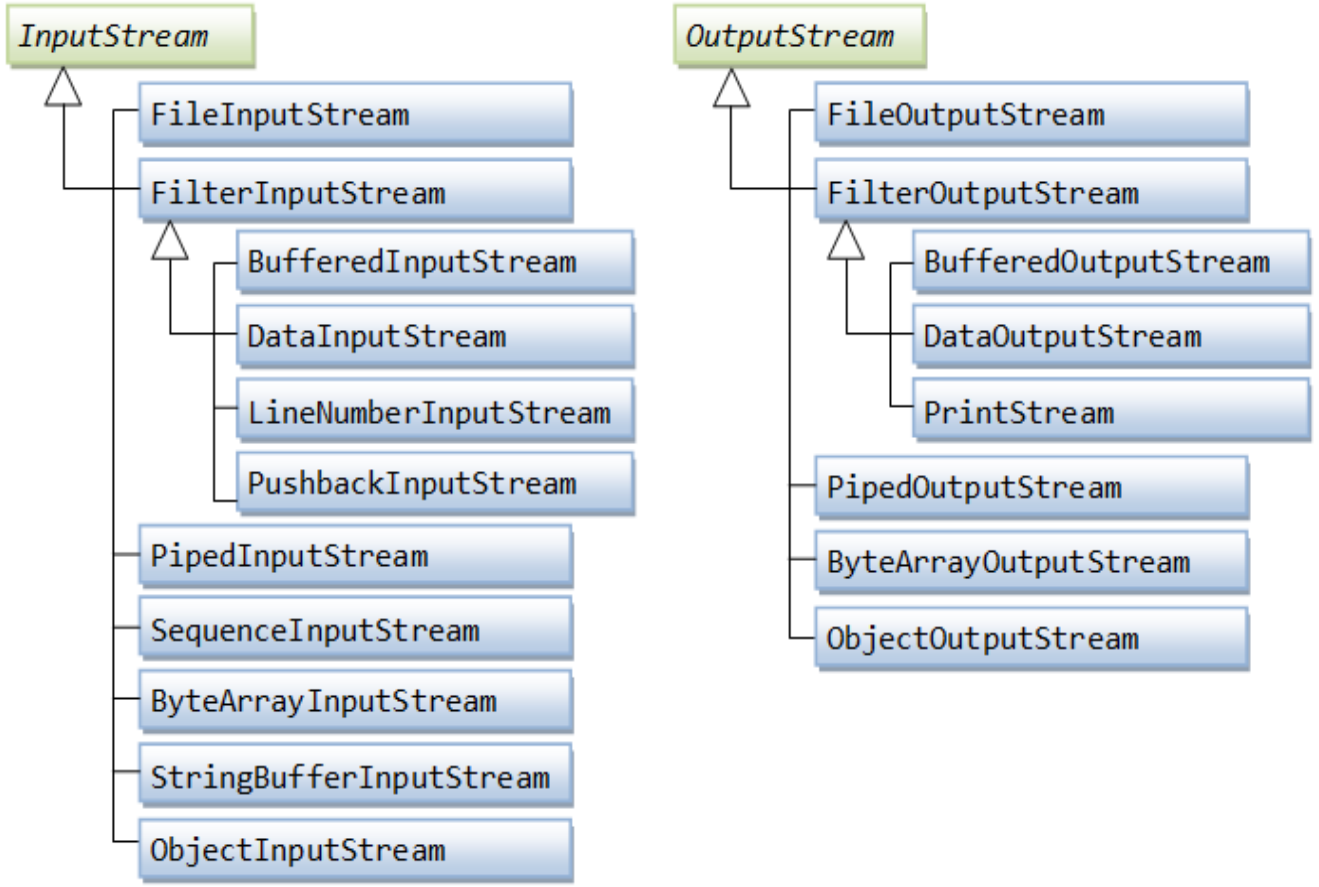
01. 스트림(Stream)

- **Stream 개념**
  - 입출력 장치와 프로그램 간 데이터 전송 통로
  - 단방향 통신을 제공하기 때문에 입력과 출력을 처리하려면 두 개의 스트림이 필요
  - 스트림은 연속된 데이터 흐름



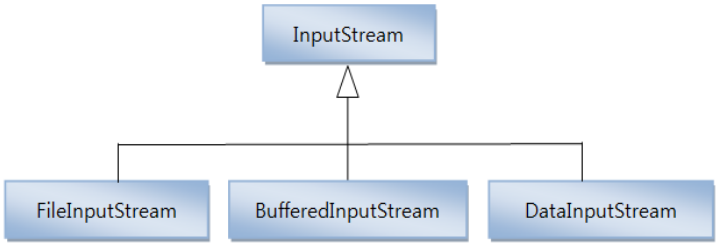
02. 바이트 단위 스트림

- **클래스 구조**
  - 추상 클래스인 InputStream과 OutputStream을 최상위 클래스



02. 바이트 단위 스트림

- **InputStream 클래스**
  - 바이트 기반 입력 스트림의 최상위 클래스로 추상 클래스

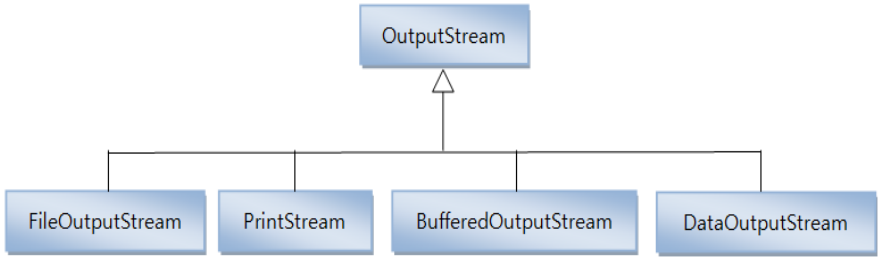


- **InputStream 주요 메서드**

메서드	설명
int read()	입력 스트림에서 1바이트씩 읽어서 0~255 사이의 값 반환, 끝에 도달하면 -1을 반환
int read(byte[] b)	b의 크기만큼 읽어 b에 저장, 읽은 바이트 수 반환
int read(byte[] b, int off, int len)	len만큼 읽어 b의 off 위치에 저장, 읽은 바이트 수 반환
int available()	읽은 바이트 수 반환
void close()	입력 스트림을 닫아 입력 스트림과 관련 자원 반환

02. 바이트 단위 스트림

- **OutputStream 클래스**
  - 바이트 기반 출력 스트림의 최상위 클래스로 추상 클래스

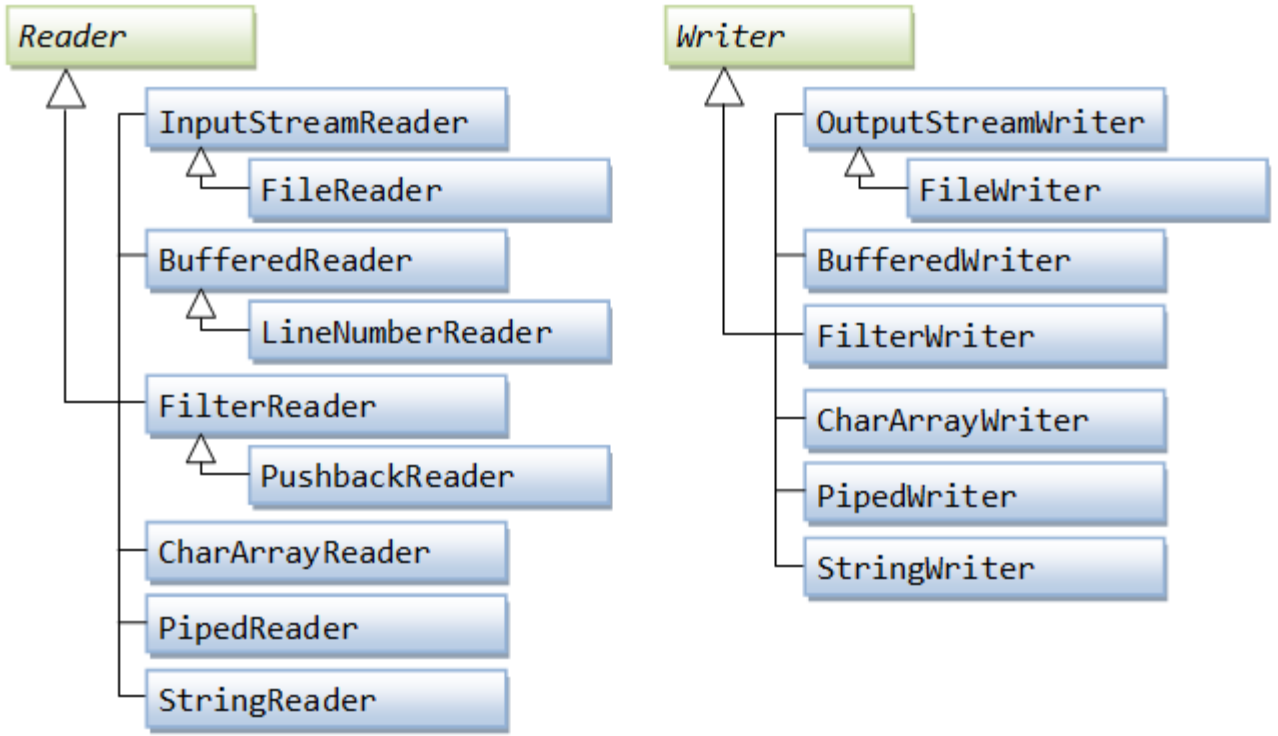


- **OutputStream 주요 메소드**

메서드	설명
void write(int b)	b의 하위 8비트를 출력
void write(byte[] b)	b의 내용을 출력
void write(byte[] b, int off, int len)	b의 off 위치로부터 len만큼 바이트 출력
void flush()	버퍼에 남은 바이트를 출력
void close()	출력스트림을 닫아 출력스트림과 관련된 자원 반환

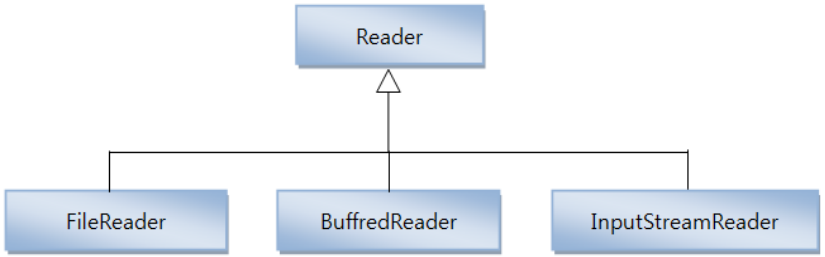
03. 문자 단위 스트림

- **클래스 구조**
  - 추상 클래스인 Reader와 Writer를 최상위 클래스



03. 문자 단위 스트림

- **Reader 클래스**
  - 문자 기반 입력 스트림의 최상위 클래스로 추상 클래스

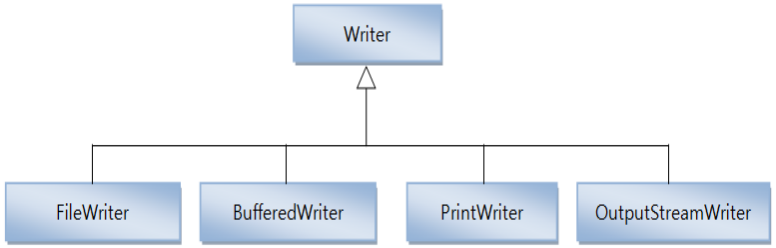


- **Reader 주요 메서드**

메서드	설명
int read()	단일 문자를 읽어 유니코드 값 반환
int read(char[] b)	문자를 b의 크기만큼 읽어 b에 저장, 읽은 문자 개수를 반환
abstract int read(char[] b, int off, int len)	len만큼 읽어 b의 off 위치에 저장, 읽은 문자 개수를 반환

03. 문자 단위 스트림

- **Writer 클래스**
  - 문자 기반의 출력 스트림의 최상위 클래스로 추상 클래스

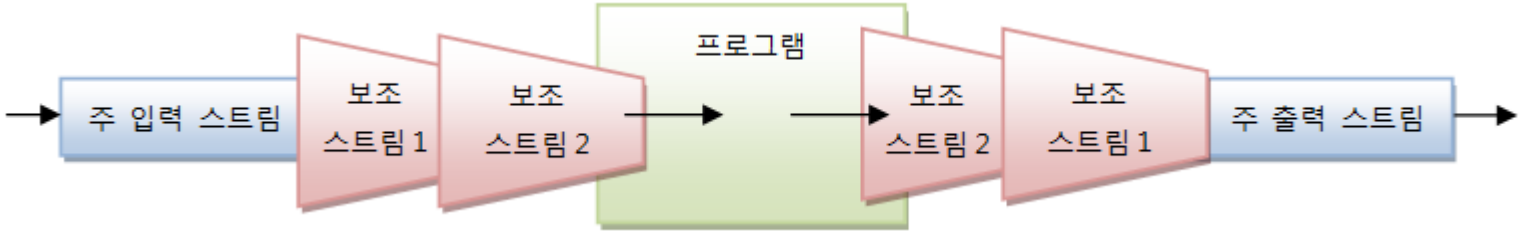


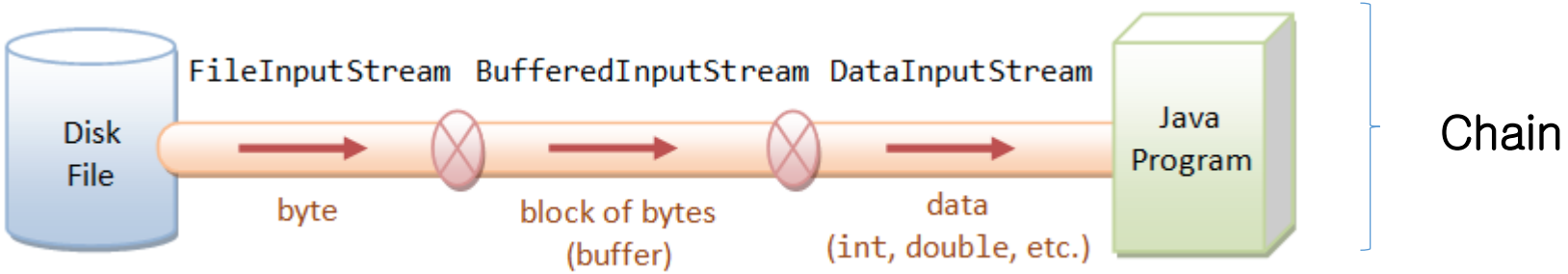
- **Writer 주요 메서드**

메서드	설명
void write(String s)	문자열 s를 출력
void write(char[] b)	char 배열 b를 출력
void write(char[] b, int off, int len)	b의 off 위치로부터 len만큼 문자를 출력
void write(String s, int off, int len)	s의 off 위치로부터 len만큼 문자를 출력

04. 보조 스트림

- **보조 스트림 개념**
  - 실제 스트림의 입출력의 편리를 위하여 사용
  - 다른 스트림과 연결 되어 여러 가지 편리한 기능을 제공해주는 스트림
  - 문자 변환, 입출력 성능 향상, 기본 데이터 타입 입출력, 객체 입출력 등의 기능을 제공

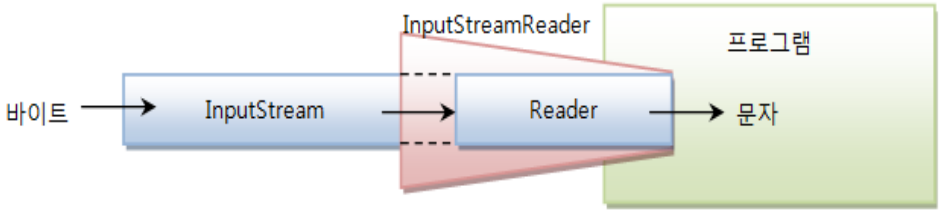




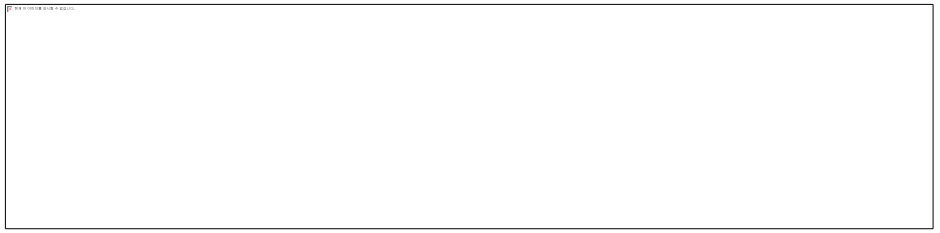
04. 보조 스트림

- **문자 변환 보조 스트림**
  - 소스 스트림이 바이트 기반 스트림이지만 데이터가 문자일 경우 사용
  - 문자셋의 종류를 지정할 수 있기 때문에 다양한 문자 입출력 가능

- **InputStreamReader**



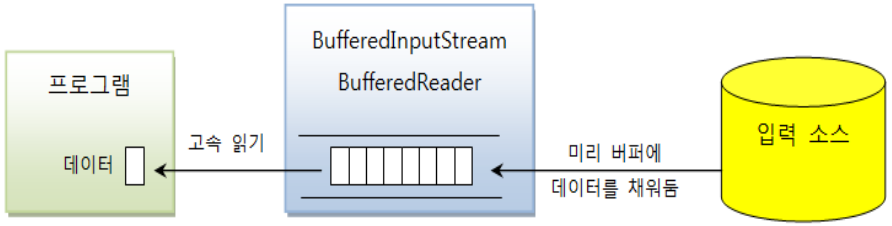
- **OutputStreamWriter**



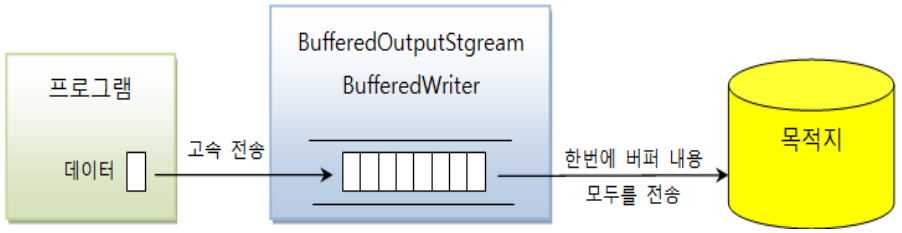
04. 보조 스트림

- **성능 향상 보조 스트림**
  - 소스에서 읽고 쓰는 모든 작업이 내부적인 버퍼를 대상으로 발생
  - 전체적인 입출력 성능이 동적으로 향상

- **BufferedInputStream/BufferedReader**



- **BufferedOutputStream/BufferedWriter**



04. 보조 스트림

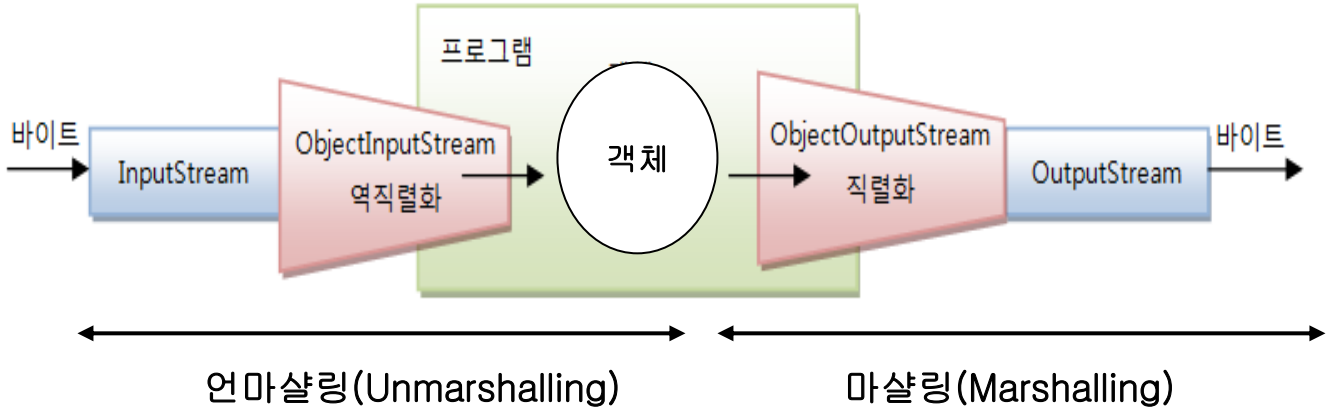
- **기본 타입 보조 스트림**
  - 기본형 데이터를 바이트 스트림으로 입출력하는 기능을 제공하는 ByteStream 클래스
  - 입출력 순서를 맞추어 사용
- **DataInputStream/DataOutputStream**

DataInputStream	DataOutputStream
boolean readBoolean()	void writeBoolean(boolean b)
byte readByte()	void writeByte(byte v)
char readChar()	void writeChar(char c)
double readDouble()	void writeDouble(double d)
float readFloat()	void writeFloat(float f)
int readInt()	void writeInt(int i)
long readLong()	void writeLong(long l)
short readShort()	void writeShort(int s)
String readUTF()	void writeUTF(String str)

04. 보조 스트림

- **객체 입출력보조 스트림**
  - 프로그램 메모리상에 존재하는 객체를 파일 또는 네트워크로 입출력할 수 있는 기능 제공
  - 현재 상태를 보존하기 위한 영속성을 지원
  - 객체 직렬화 필요: 객체는 문자가 아니므로 바이트 기반 스트림으로 데이터 변경 필요

- **ObjectInputStream, ObjectOutputStream**



04. 보조 스트림

- **직렬화가 가능한 클래스(Serializable)**
  - Serializable 인터페이스를 구현한 클래스만 직렬화, transient 필드는 제외
  - 객체 직렬화 할 때 private 필드 포함한 모든 필드를 바이트로 변환 가능

- **직렬화가 가능한 객체의 조건**
  - 기본형 타입(boolean, char, byte, short, int, long, float, double)은 직렬화가 가능하다.
  - Serializable 인터페이스를 구현한 객체여야 한다.
  - transient가 사용된 멤버는 전송되지 않는다. (보안 변수 : null 전송)

14. 네트워크

01/ 네트워크 기초

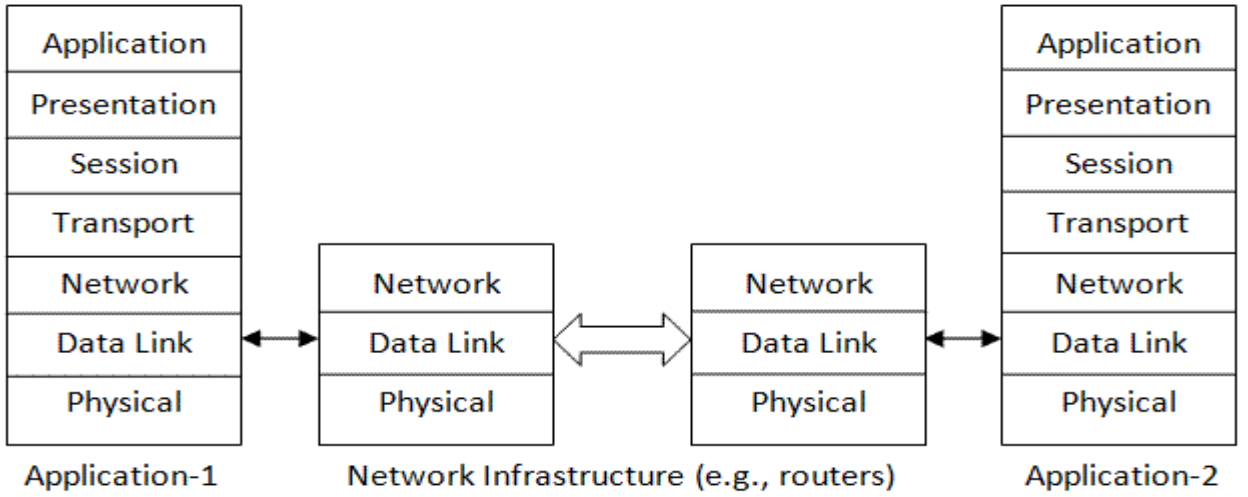
02/ URL

03/ TCP 네트워킹

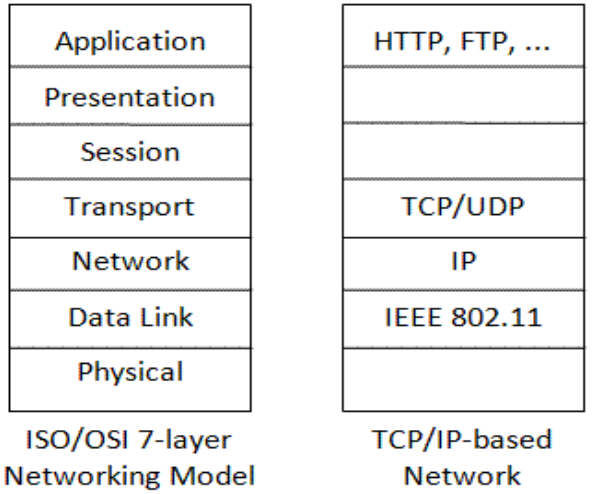
04/ UDP 네트워킹

01. 네트워크 기초

- **ISO OSI 7 Layer 네트워크 모델**



- **ISO OSI 7 Layer vs TCP/IP**



02. URL(Uniform Resource Locator)

- **URL 클래스**
  - http,ftp,등의 서비스를 지원하는 웹 서버들의 위치를 표현하는 체계를 나타내는 URL 정보 저장

메소드	설명
Object getContent()	컨텐츠 반환
String getFile()	파일명 반환
Stirng getHost()	호스트명 반환
String getPath()	경로 부분 반환
int getPort()	포트번호 반환
String getProtocol()	프로토콜명 반환
InputStream openStream()	URL 주소와 연결한 뒤 연결로부터 입력받을 수 있는 입력 스트림 객체 반환
URLConnection openConnection()	원격 객체에 접속한 뒤 통신할 수 있는 URLConnection 반환

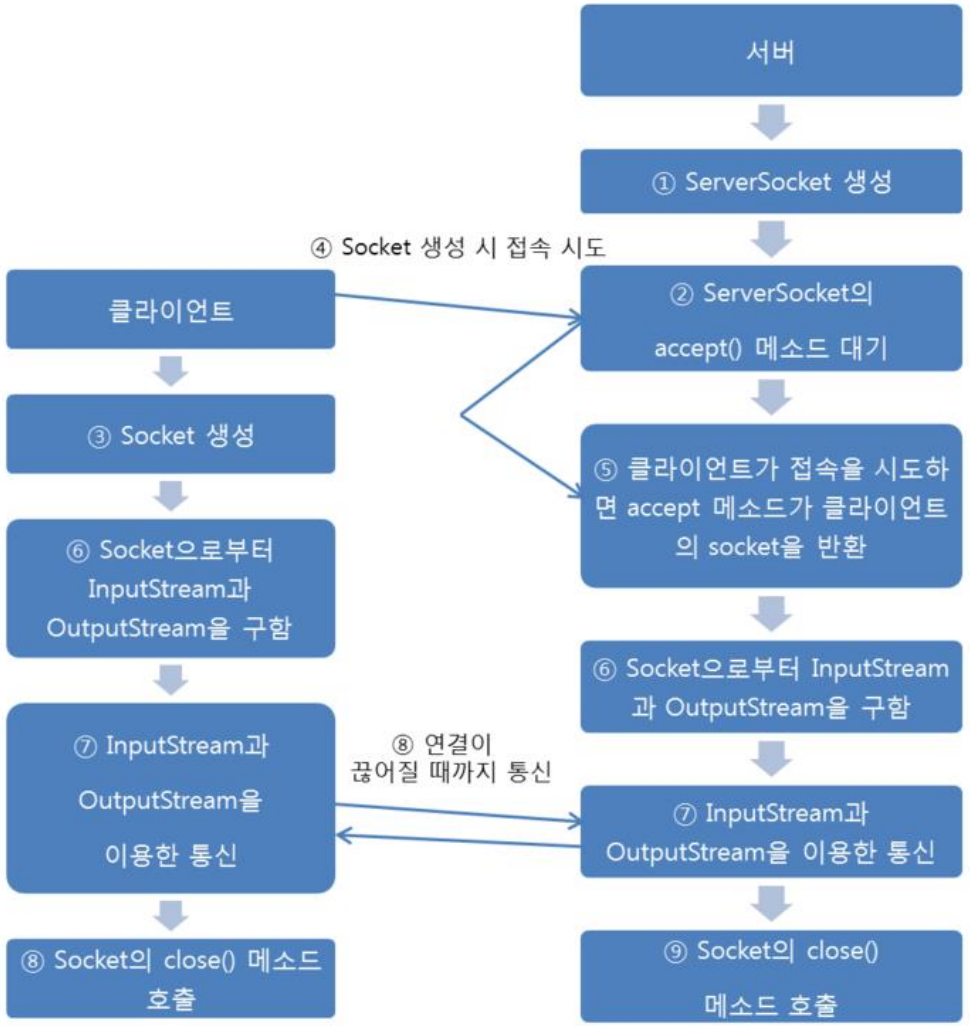
03. TCP 네트워킹

- **TCP 프로토콜 특징**
  - 신뢰성 있는 프로토콜, 송신 데이터가 수신 측에 차례대로 유실되지 않고 도착
  - 연결지향 방식은, 한번 연결되면 연결이 끊어질 때까지는 송신한 데이터가 차례대로 목적지의 소켓에 전달되는 신뢰성 있는 통신이 가능

- **TCP 관련 클래스**
  - ServerSocekt : 서버쪽에서 클라이언트의 접속을 대기하기 위해서 반드시 필요한 클래스
  - Socket : 서버와 클라이언트가 통신하기 위해서 반드시 필요한 클래스

03. TCP 네트워킹

- **TCP 프로그래밍**



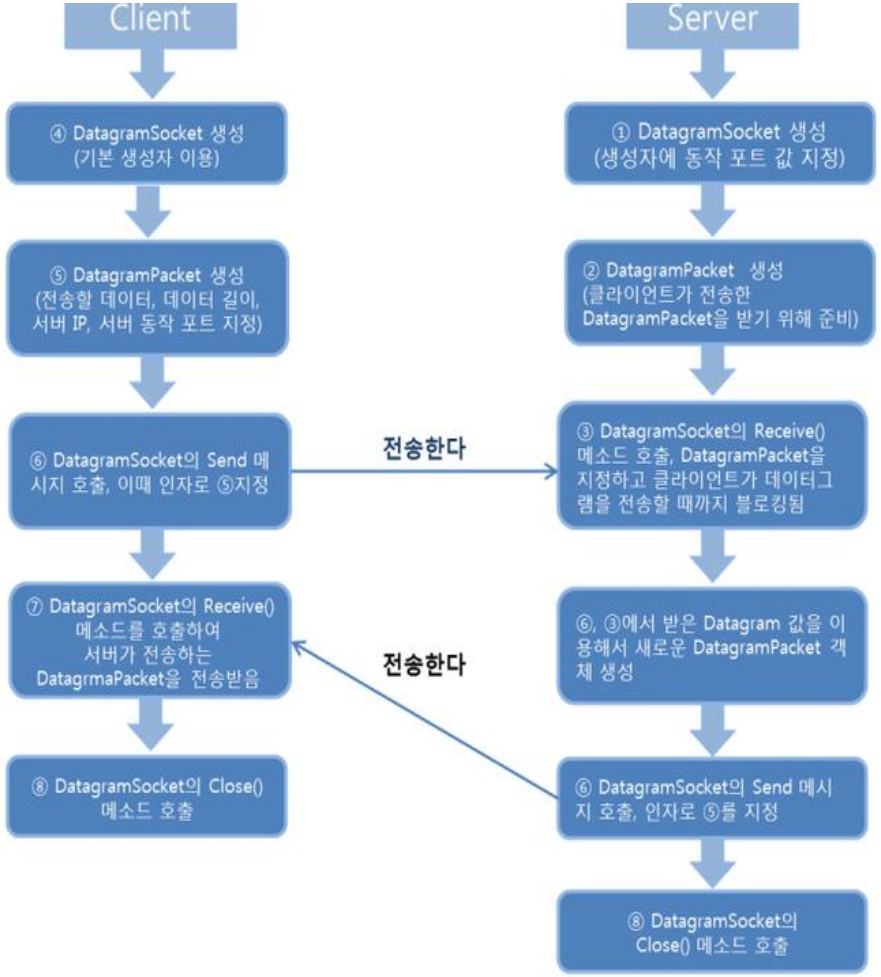
04. UDP 네트워킹

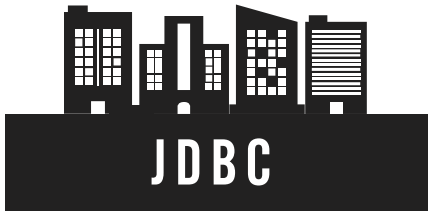
- **UDP 프로토콜**
  - 패킷을 보낼 때마다 수신 측의 주소와 로컬 파일 설명자를 함께 전송
  - 비연결성이기 때문에 TCP보다 신뢰성이 떨어지나 속도는 빠름

- **UDP 관련 클래스**
  - DatagramSocket : 클라이언트는 데이터를 전송하기 위한 소켓 생성
  - DatagramPacket : 전송할 데이터, 데이터 길이, 서버 IP, 포트 지정 등의 데이터를 서버로 전송

04. UDP 네트워킹

- **UDP 프로그래밍**





01/ JDBC

02/ 디자인 패턴

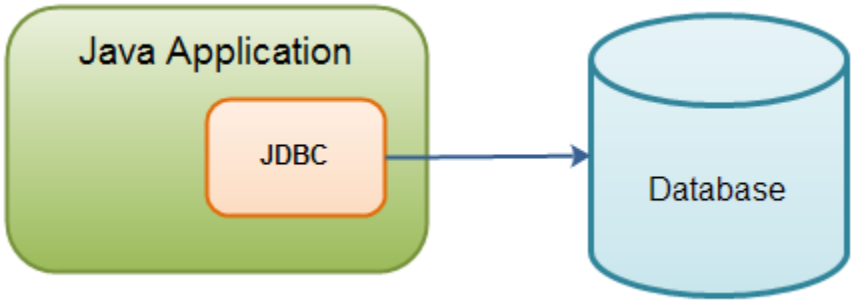
01. JDBC

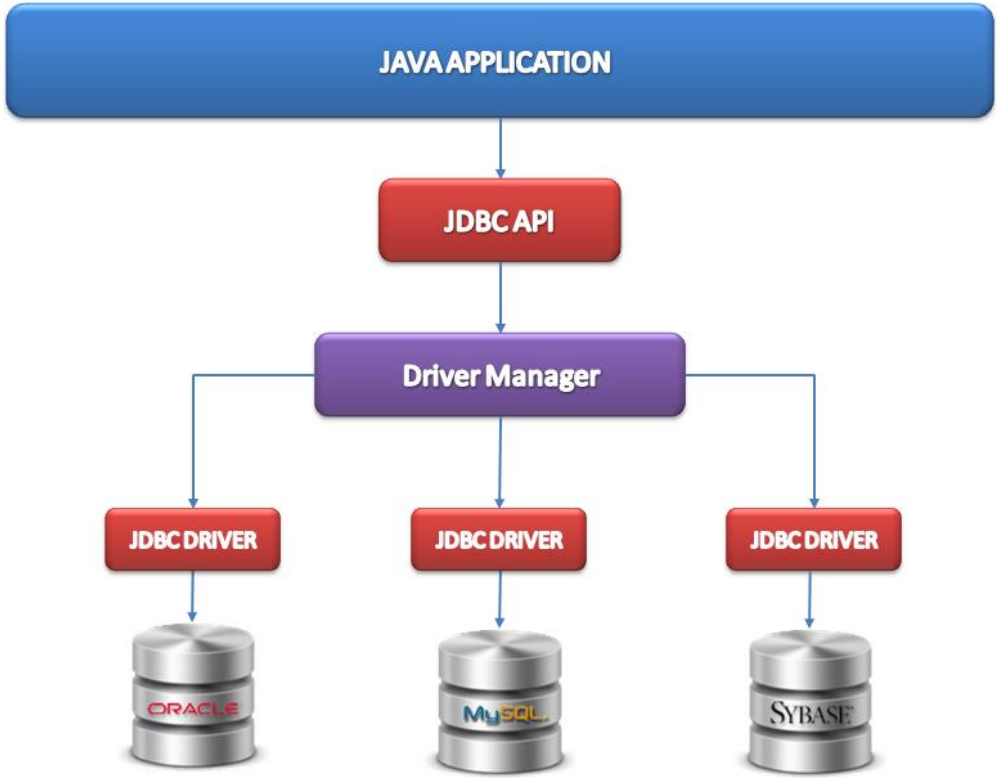
01/ JDBC 개요

02/ JDBC 프로그래밍

01. JDBC(Java DataBase Connectivity)

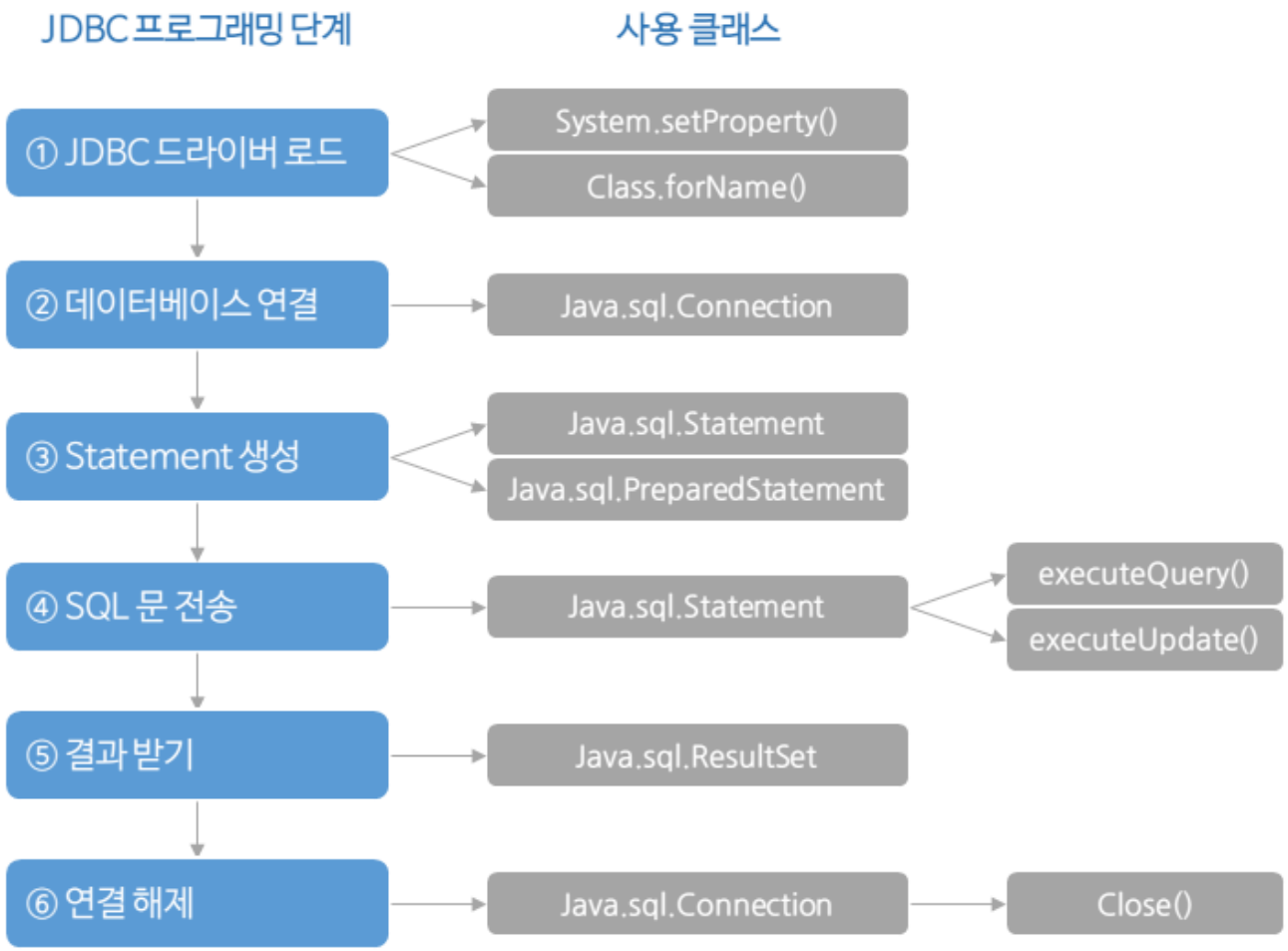
- **JDBC 개요**
  - 자바에서 데이터베이스에 접속할 수 있도록 하는 자바 API
  - 데이터베이스에서 자료를 쿼리하거나 업데이트하는 방법을 제공
  - JDBC를 사용하려면 해당 데이터베이스 제품에 맞는 JDBC Driver가 필요





02. JDBC 프로그래밍

- **JDBC 프로그래밍 과정**



02. JDBC 프로그래밍

- **JDBC 드라이버 로드**

```
Class.forName("JDBC 드라이버 클래스명");
```

  - 오라클 클래스명: oracle.jdbc.driver.OracleDriver

- **데이터베이스 연결**

```
Connection con = DriverManager.getConnection('연결URL", "사용자계정", "비밀번호");
```

02. JDBC 프로그래밍

- **Statement 생성**

Statement	PreparedStatement
• 쿼리문과 인자를 컴파일 후 실행 • 쿼리문이 동적인 경우 적합	• 쿼리문을 컴파일해서 저장후 호출시 인자 대입해 실행 • 쿼리문이 정적인 경우 적합
Statement stmt = con.createStatement();	PreparedStatement pstmt = con.prepareStatement("SQL문");

- **SQL문 전송 및 결과받기**

	Statement	PreparedStatement
insert/update/delete	int result = stmt.executeUpdate("SQL문");	int result = pstmt.executeUpdate();
select	ResultSet rs = stmt.executeQuery("SQL문");	ResultSet rs = pstmt.executeQuery();

- **연결 해제**

```
rs.close();
stmt.close();     // pstmt.close();
con.close();
```

02. 디자인 패턴

01/ 디자인 패턴 소개

02/ 싱글턴 패턴

03/ MVC 패턴

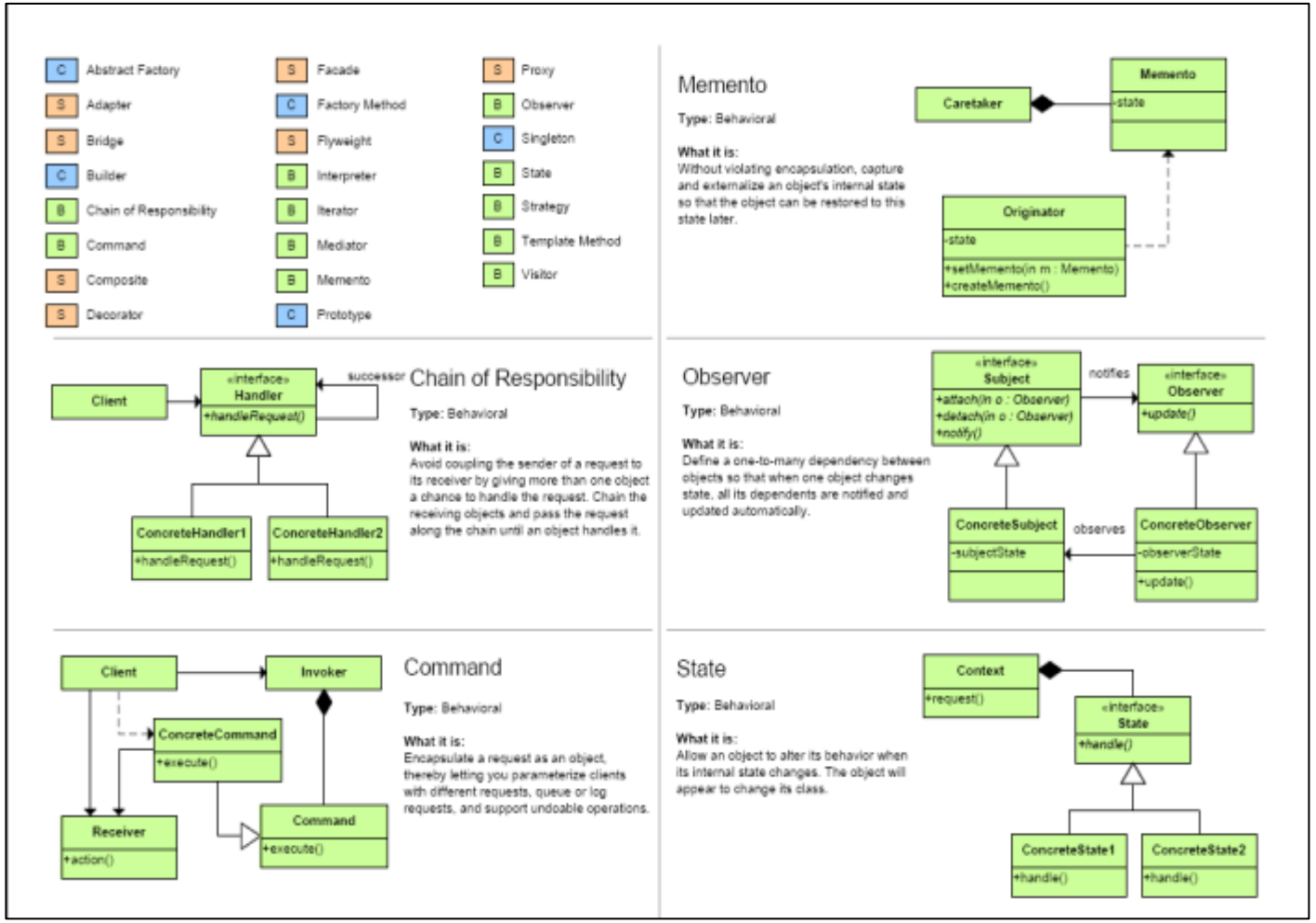
01. 디자인 패턴(Design Pattern)

- **디자인 패턴이란?**
  - 프로그램을 개발하는 과정에서 빈번하게 발생하는 디자인 상의 문제를 정리해서, 상황에 따라 간편하게 적용해서 쓸 수 있는 패턴 형태로 만든 것

- **디자인 패턴 유형**
  - 생성 관련 패턴(Creational Pattern)
    - 객체 인스턴스 생성을 위한 패턴
    - 클라이언트와 그 클라이언트에서 생성해야 할 객체 인스턴스 사이의 연결을 끊어주는 패턴
    - 싱글턴, 팩토리 메소드, 추상 팩토리, 프로토타입, 빌더 패턴
  - 행동 관련 패턴(Behavioral Pattern)
    - 클래스와 객체들이 상호작용하는 방법 및 역할을 분담하는 방법과 관련된 패턴
    - 스트래티지, 옵저버, 스테이트, 커맨드, 이터레이터, 템플릿 메소드, 인터프리터, 미디에이터, 역할 변경, 메멘토, 비지터
  - 구조 관련 패턴(Structural Pattern)
    - 클래스 및 객체들을 구성을 통해서 더 큰 구조로 만들 수 있게 해 주는 것과 관련된 패턴
    - 데코레이터, 어댑터, 컴포지트, 퍼사드, 프록시, 브리지, 플라이웨이트

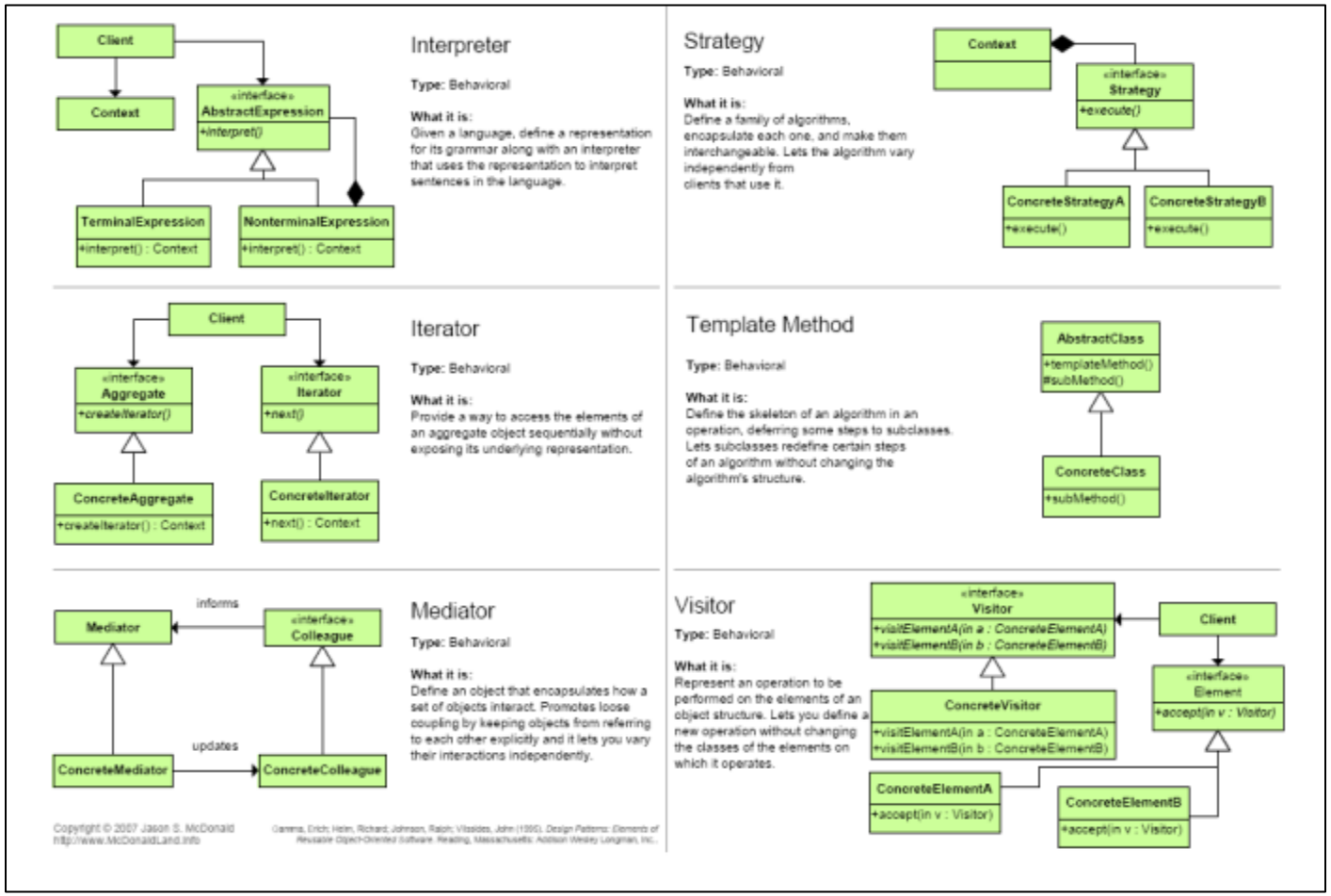
01. 디자인 패턴(Design Pattern)

- **GoF의 디자인 패턴**



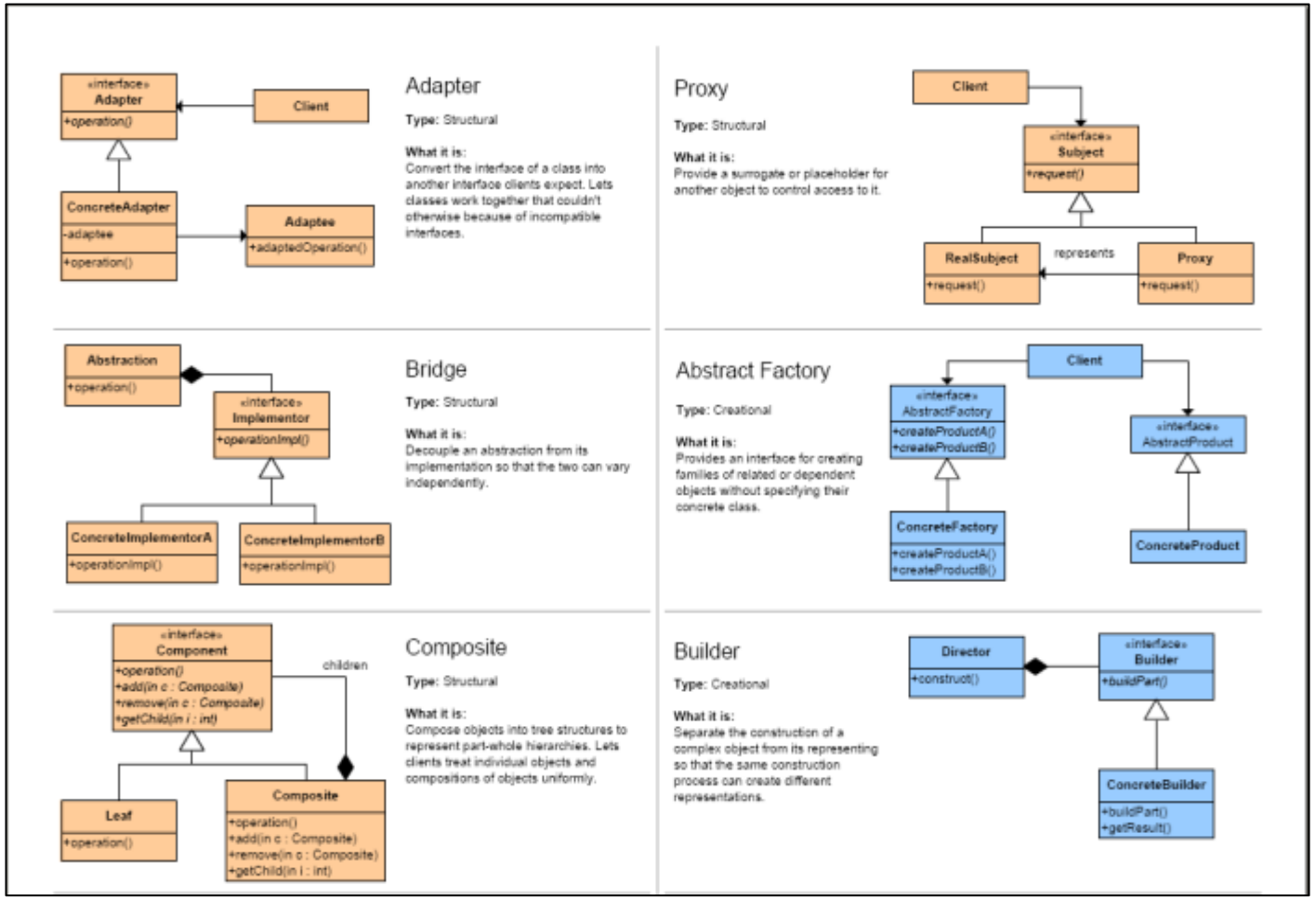
01. 디자인 패턴(Design Pattern)

- **GoF의 디자인 패턴**



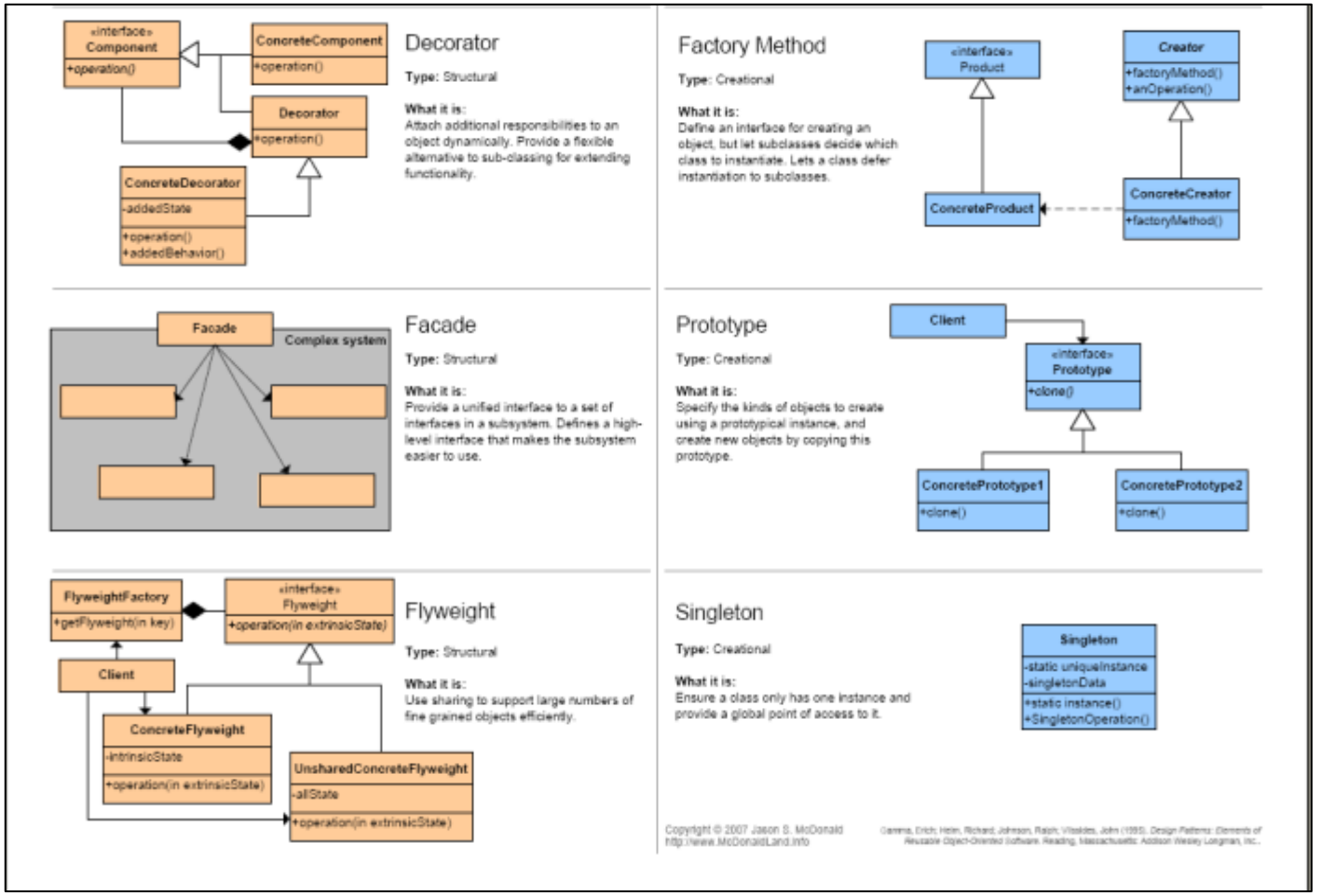
01. 디자인 패턴(Design Pattern)

- **GoF의 디자인 패턴**



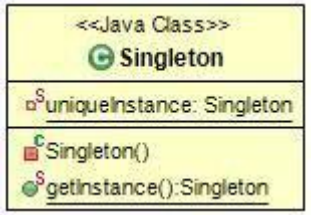
01. 디자인 패턴(Design Pattern)

- **GoF의 디자인 패턴**



02. 싱글턴 패턴(Singleton Pattern)

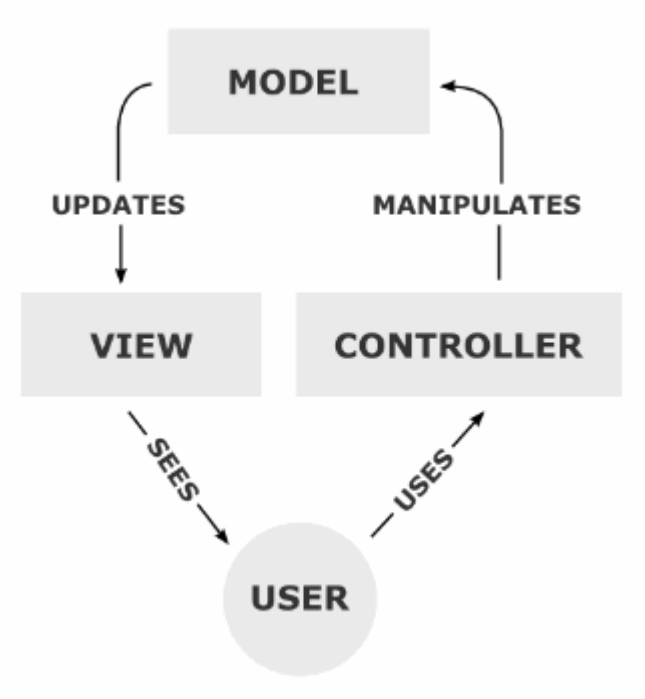
- **싱글턴 패턴**
  - 해당 클래스의 인스턴스가 하나만 만들어지고, 어디서든지 그 인스턴스에 접근할 수 있도록 하기 위한 패턴




```
public class Singleton {
        private static Singleton uniqueInstance = new Singleton();

        private Singleton(){}

        public static synchronized Singleton getInstance(){
                return uniqueInstance;
        }
}
```

03. MVC(Model View Controller) 패턴

- **MVC 패턴**
  - 3가지의 역할로 구분
  - 사용자가 Controller를 조작하면 Controller는 Model을 통해서 데이터를 가져오고 그 정보를 바탕으로 시각적인 표현을 담당하는 View를 제어해서 사용자에게 전달
  - 주로 웹에서 많이 사용



8253jang@daum.net
장희정 강사

THANK YOU

장희정(8253jang@daum.net)