Java 8 -> 11

C++11/14 - Modern C++

C#5 - Modern C#

Java 8 - Modern Java

Effective Java

자바의 아버지: 제임스 고슬링

자바의 어머니: 조슈아 블로크 - 자바 라이브러리 - Java 6

두번째어머니: 브라이언 게츠 - Java 7

idioms(관용구)

package

: 이름 충돌을 방지하기 위해서

=> C++, C#: namespace

SOLID

: 객체지향 설계 5대 원칙

=> 정리한 아저씨

: Uncle Bob

로버트 C 마틴 - 클린 코드

SRP(Single Responsibility Principle)

: 단일 책임의 원칙

-> 모듈(변수, 함수, 클래스, 프로그램)은

단 하나의 책임을 가져야 한다.

OCP(Open Close Principle)

: 개방 폐쇄의 원칙

-> 모듈은 수정에는 닫혀 있고, 확장에는 열려 있어야 한다.

; 새로운 기능이 추가되어도, 기존 코드는 수정되면 안된다.

LSP(Liskov Substitution Principle)

: 리스코프의 치환 원칙

-> 자식은 부모 클래스로 대체할 수 있어야 한다.

: 자식의 공통된 기능은 부모로부터 비롯되어야 한다.

=> 상속의 목적은 다형성이다.

ISP(Interface Segregation Principle)

: 인터페이스 분리 원칙

MP3: play() / playOneMinute()

-> iPod: play() Person {

iPhone: play() void playMusic(MP3 mp3) { mp3.play(); }

}

: 범용 인터페이스 보다는 세분화된 인터페이스가 좋다.

DIP(Dependency Inversion Principle)

: 의존관계 역전 원칙

=> 클라이언트는 구체적인 타입에 의존하는 것이 아니라,

인터페이스나 추상 클래스에 의존해야 한다.

// C

// main()

// 함수: 동일한 기능이 있다면, 중복하는 것이 아니라

// 함수를 통해 재사용하기 위해서 사용한다.

// fA()

// fA() -> fA() { B }

// fA()

// fA()

// class {struct {} + function}

// Class -> Instance

// Mutable Instance -> 객체가 생성된 이후로 상태가 변경될 수 있다.

// Immutable Instance -> 객체가 생성된 이후로 상태가 변경되지 않는다.

// String(불변 객체)

// 비교

// 1. 객체 동등성(.equals)

// : 동일한 내용을 가지는가?

// 2. 참조 동등성(==)

// : 동일한 참조값을 가지는가?

// 자바 타입의 종류 2가지

// 1) Primitive Type(Built-in Type) - Scala Type(Value Type)

// int, double, float, char, boolean, byte

// 2) Reference Type(User-defined Type)

// 모든 class 타입

// : 대문자로 시작한다.

// 3가지

// 1. class

// 2. Array

// 3. enum

// 변수 앞

// final int n = 42;

// final Int n = new Int(42);

// 클래스 앞

// final class Car {}

// class Truck extends Car {} // Compile Error!!

// 메소드 앞

/\*

class Car {

public Car(int n) {

System.out.println("Car()");

initialize();

}

// final

private void initialize() {

// 초기화

}

final void go() {

System.out.println("car go");

}

}

class Truck extends Car {

Truck() {

super(42);

System.out.println("Truck()");

}

void initialize() {

// 초기화

}

void go() {

System.out.println("Truck go");

}

}

\*/

/\*

public class Sample {

public static void main(String[] args) {

// Truck is-a Car

Car car = new Truck();

// Upcasting은 암묵적(Implicit)으로 허용된다.

car.go();

// 바인딩(Binding)

// : 어떤 메소드를 호출할지 결정하는 것

// 1) static binding

// : 컴파일러가 결정하는 것

// 2) dynamic binding

// : 실행시간에 결정하는 것

}

}

\*/

/\*

public class Sample {

public static void main(String[] args) {

String s = "Hello";

// s = new String("Hello");

// 불변 객체는 상태가 변경되야 한다면,

// 그 상태를 가지는 새로운 객체를 생성한다.

s = s.replace("H", "X");

System.out.println(s);

}

}

\*/

import org.junit.Test;

import java.util.ArrayList;

import java.util.List;

import java.util.Vector;

// Annotation(주석) vs Comment(주석)

class Car {

void go() {

System.out.println("Car go"); // 1

}

}

// Compile, Runtime

class Truck extends Car {

// override void go()

// @FunctionalInterface

@Test

@Override

void go() {

System.out.println("Truck go"); // 2

}

}

// Binding

// : 어떤 메소드를 호출할지 결정하는 것

// 1) static binding

// 2) dynamic binding

/\*

public class Sample {

public static void main(String[] args) {

// Truck is a Car

// : 부모의 구현과 인터페이스를 물려받는다.

// '암묵적'인 허용: 업캐스팅

Car car = new Truck();

car.go();

}

}

\*/

// Array

// 정적 배열: 컴파일 타임에 크기가 결정된다.

// 동적 배열: 런타임에 크기가 계속 변경될 수 있다.

// 메모리 영역(Life cycle, 비용)

// 1. Text: 기계어 코드 저장되는 영역(read-only)

// : Segmentation Fault

// 2. Data: 정적(static) 변수

// 프로그램 시작 부터 끝까지

// : 프로그램의 용량이 커지고, 메모리 사용량이 높아진다.

// 정적 분석기: 코드를 분석해서, malloc, new 쌍이 맞는지 확인한다.

// 동적 분석기: 메모리가 실제로 해지되는지를 추적한다.

// 3. Heap(자유영역): malloc, new

// 원하는 시점에 생성하고, 원하는 시점에 파괴 가능한 메모리

// => 메모리 파괴의 책임: GC

// => 상대적으로 느리다.

// malloc(30)

// Free

// 32: [] -> [] -> []

// 64: [] -> []

// 128: []

// 4. Stack: 함수 실행, 함수 반환 주소값, 지역 변수

// 함수가 시작할 때 할당되고, 함수가 끝날 때 파괴된다.

// => 메모리 생성과 해지의 비용이 거의 없다.

/\*

public class Sample {

public static void main(String[] args) {

ArrayList<Integer> arr = new ArrayList<>(8);

arr.add(10);

}

}

\*/

/\*

class Int {

int value;

public Int(int value) {

this.value = value;

}

}

public class Sample {

public static void main(String[] args) {

// Collection's Generic

ArrayList<String> arrayList = new ArrayList<>();

// Object[] data = new Object[1024];

arrayList.add("Hello");

// arrayList.add(42);

// Object obj = arrayList.get(0); // String? Integer?

String s = arrayList.get(0);

// Collection에는 하나의 타입만 집어넣자.

// Generic: 컴파일 타임에 컬렉션에 하나의 타입만 들어갔는지

// 체크하는 목적으로 사용한다.

// arrayList.add(42);

// arrayList.add(new Integer(42));

// => Auto Boxing

// arrayList.add(new Int(42));

// Boxing

// : Collection에 Primitive Type을 저장하기 위해서는

// 'Reference Type'으로 변환할 필요가 있습니다.

}

}

\*/

/\*

class User {

// 같은 패키지 내에서는 접근이 가능하다.

// => 잘못된 설계!

private int age;

// protected int age;

}

public class Sample {

public static void main(String[] args) {

User user = new User();

// user.age = 42;

}

}

\*/

// Module 1

// kr.ac.ajou.User {

// int age;

// }

// Module 2

// kr.ac.ajou.Foo { age }

// io.thethelab.Program {}

// Java 8

// 1. private

// 2. default package

// 3. protected

// 4. public

// Data Structure - Stack, Queue

// : 우선순위 큐

// => 데이터가 들어간 순서에 상관없이, 우선순위가 가장 높은 요소가

// 먼저 나온다.

// Vector

// push()

// pop()

// elementData / elementCount

// => Vector의 protected 필드

// 자식 클래스가 부모의 기능을 이용할 때, 필드에 바로 접근하는 것이 아니라

// 메소드를 이용해야 한다.

/\*

class Stack extends Vector<Integer> {

public void push(Integer e) {

elementData[elementCount++] = e;

}

public Integer pop() {

return (Integer)elementData[--elementCount];

}

}

\*/

/\*

// 재사용은 포함이 좋다. (위임)

class Stack {

private List<Integer> data = new ArrayList<>();

public void push(Integer e) {

data.add(e);

}

public Integer pop() {

return data.get(data.size() - 1);

}

}

public class Sample {

public static void main(String[] args) {

Stack s = new Stack();

s.push(10);

s.push(20);

s.push(30);

System.out.println(s.pop());

}

}

\*/

public class Sample {

@Test

public void testcase2() {

StringBuilder sb = new StringBuilder();

for (int i = 0 ; i < 100000 ; ++i)

sb.append(i);

String s = sb.toString();

}

@Test

public void testcase1() {

String s = "";

for (int i = 0 ; i < 100000 ; ++i)

s += i;

// String s 는 불변 객체이다.

// 새로운 표현마다 새로운 객체가 만들어진다.

// s = ""

// s = "0"

// s = "01"

// 프로그램의 성능이 객체의 생성과 파괴에 큰 영향을 받는다.

}

}

package io.thethelab;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;

import java.util.Scanner;

// Design Pattern

// : 건축(크리스토퍼)

// 에릭 감마 외 3명Gangs of Four(GoF)'s Design Pattern

// : 잘 만들어진 소프트웨어에는 일련의 공통의 설계 원칙이 있다.

// 23 가지를 정리하였다.

// 1. Singleton

// 2. Builder

// 3. Abstract Factory

// 4. Factory Method

// 5. Prototype

// ...

// 객체 지향적인 도형 편집기 프로그램 만들기

// 1. 모든 타입은 클래스로 만들면 된다.

// 2. A와 B를 묶기 위해서는 공통의 부모가 필요하다.

// 3. 자식의 공통된 기능은 반드시 부모로부터 비롯되어야 한다.

// 부모의 레퍼런스를 통해서 자식의 기능을 온전히 이용할 수 있다.

// => LSP(Liskov Substitution Principle)

// : 자식은 부모 클래스로 대체 할 수 있어야 한다.

// => '다형성'

// => '다형성'은 OCP를 만족한다.

// => 리팩토링(Refactoring): 기존 코드의 동작을 변경하지 않고,

// 구조를 개선하는 작업(마틴 파울러)

// : 코드의 유지보수성을 떨어뜨리는 요소 - 냄새(smells)

// -> Replace type code with polymorphism

// 4. OCP

// : 새로운 기능이 추가되어도, 기존 코드는 수정되면 안된다.

// 5. '복제'를 다형적으로 수행하자.

// => Prototype Pattern

abstract class Shape {

// 자식은 반드시 기능을 제공해야 하고,

// 부모가 구현을 제공할 필요가 없다면, 추상 메소드가 좋다.

// public void draw() {

//

// }

public abstract void draw();

public abstract Shape copy();

}

class Rect extends Shape {

public void draw() {

System.out.println("Rect draw");

}

@Override

public Shape copy() {

return new Rect();

}

}

class Circle extends Shape {

public void draw() {

System.out.println("Circle draw");

}

@Override

public Shape copy() {

return new Circle();

}

}

class Triangle extends Shape {

@Override

public void draw() {

}

@Override

public Shape copy() {

return null;

}

}

public class Program {

public static void main(String[] args) {

// ArrayList<Rect> rects = new ArrayList<>();

// ArrayList<Circle> circles = new ArrayList<>();

// DIP(의존 관계 역전 원칙)

// : 구체적인 타입을 직접 사용하지 말고, 추상 클래스나 인터페이스에

// 의존해야 한다.

List<Shape> shapes = new LinkedList<>();

Scanner scanner = new Scanner(System.in);

while (true) {

int cmd = scanner.nextInt();

if (cmd == 1) {

shapes.add(new Rect());

} else if (cmd == 2) {

shapes.add(new Circle());

} else if (cmd == 9) {

for (Shape e : shapes) {

e.draw();

// 다형성

}

} else if (cmd == 0) {

int index = scanner.nextInt();

Shape shape = shapes.get(index);

shapes.add(shape.copy());

// shape? Rect? Circle?

// if (shape instanceof Rect) {

// // shapes.add() Rect 복제본 추가

// } else if (shape instanceof Circle) {

// // shapes.add() Circle 복제본 추가

// } else if (shape instanceof Triangle) {

// //....

// }

// shapes.add()

}

}

}

}

package io.thethelab;

import java.util.Arrays;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

// Comparator: 비교자

// => 내가 만든 객체가 아닐 경우 사용한다,

// Comparable: 비교 가능한

// => 내가 만든 객체일 경우 사용한다.

// Comparable

// 한계: 내가 만든거 아니면 사용할 수 없다.

// Object 타입이기 때문에 잘못된 타입이 올 수도 있다.

class User implements Comparable {

private String name;

private int age;

public User(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "User{" +

"name='" + name + '\'' +

", age=" + age +

'}';

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

// strcmp(s1, s2)

// s1 > s2: 0<ret

// s2 = s2: 0=ret

// s1 < s2: 0>ret

@Override

public int compareTo(Object o) {

User other = (User) o;

// return age - other.age;

return other.age - age;

// return name.compareTo(other.name);

}

}

public class Program {

public static void main(String[] args) {

List<String> names = Arrays.asList("Bobx", "Tomxxx", "Comxxx");

// Collections.sort(names);

Collections.sort(names, new Comparator<String>() {

@Override

public int compare(String o1, String o2) {

return o2.length() - o1.length();

}

});

System.out.println(names);

/\*

List<User> users = Arrays.asList(

new User("Tom", 32),

new User("Bom", 22),

new User("Com", 42),

new User("Kom", 72),

new User("Zom", 52)

);

// User u = new User("Tom", 42);

Collections.sort(users, new Comparator<User>() {

@Override

public int compare(User o1, User o2) {

return o1.getAge() - o2.getAge();

}

});

// 정렬 정책

// 1) 오름차순(ascending)

// 2) 내림차순(descending)

// Collections.sort(users);

System.out.println(users);

\*/

}

}

package io.thethelab;

// Reflection(투영)

// : Introspection

// 클래스의 특성을 의미한다.

// Class<User> -> class User -> User instance

// 얻는 방법 3가지

// 1. class

// 2. instance

// 3. 문자열 이름

import java.lang.reflect.Method;

// "io.thethelab.User"

class User {

void foo() {}

void goo() {}

void hoo() {}

}

class Car {

// void go() { System.out.println("Car go"); }

}

class Truck extends Car {

void go() { System.out.println("Truck go"); }

}

public class Program {

public static void main(String[] args) throws Exception {

// Class가 가지고 있는 정보를 다 알 수 있다.

Class clazz4 = User.class;

Method[] methods = clazz4.getDeclaredMethods();

for (Method e : methods) {

System.out.println(e.getName());

}

User obj = new User();

Class clazz1 = User.class;

Class clazz2 = obj.getClass();

Class clazz3 = Class.forName("io.thethelab.User");

// 1. 타입 정보를 확인할 수 있다.

Car car = new Truck();

foo(car);

// Intent intent = new Intent(this, LoginActivity.class);

// 2. 동적(Dynamic, Virtual) 생성

// : Runtime에 어떤 타입의 객체를 생성할지 결정하고 싶다.

Class clazz = Truck.class;

Object obj2 = clazz.newInstance();

}

private static void foo(Car car) {

// C++: RTTI(Run Time Type Information)

// 만약 car가 Truck 이면, go 하고 싶다.

if (car.getClass() == Truck.class) {

Truck truck = (Truck) car;

}

}

}

Web / Mobile Service Architecture

Program(Client) -----> Program(Server) -----> Database

Protocol

(HTTP)

------------------------------------------------------------------

Server Computer(고성능)

동시에 수많은 클라이언트의 요청을 처리할 수 있어야 한다.

Cloud

IaaS(임대해서 사용할 수 있다.)

- Amazon Web Service(75)

- Google Cloud Platform(10)

- Microsoft Azure(15)

EC2 인스턴스 생성

접속

설치

Apache(Web Server) - Tomcat(Web Application Server) : Java

: 정적 리소스 : 동적 리소스

nginx node.js(express)

spring

ASP.net

// Read

// open("hello.txt", O\_RDONLY);

// fopen("hello.txt", "r");

// => InputStream

// : FileInputStream

// Write

// open("hello.txt", O\_WRONLY);

// fopen("hello.txt", "w");

// => OutputStream

// : FileOutputStream

import java.io.\*;

// 아래의 클래스의 객체가 Try with Resource를 통해 자동으로 close 될수 있도록 하고 싶다.

// => AutoClosable 인터페이스를 구현하면 됩니다.

class MyResource implements AutoCloseable {

// File

@Override

public void close() {

System.out.println("MyResource closed");

}

}

// Java 7 = NIO(New IO)

// => 비메모리 자원에 대해서는 반드시 명시적인 종료 메소드를 호출해야 한다.

public class Program {

// Java 7 => Try with Resources

public static void main(String[] args) {

try (MyResource resource = new MyResource();

FileInputStream fis = new FileInputStream("hello.txt");

DataInputStream dis = new DataInputStream(fis)) {

String str = dis.readLine();

System.out.println(str);

} catch (IOException e) {

e.printStackTrace();

}

}

/\*

public static void main(String[] args) {

FileInputStream fis = null;

DataInputStream dis = null;

try {

fis = new FileInputStream("hello.txt");

dis = new DataInputStream(fis);

String str = dis.readLine();

System.out.println(str);

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (dis != null) {

dis.close();

}

} catch (IOException e) {

e.printStackTrace();

}

try {

if (fis != null) {

fis.close();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

\*/

// public static void main(String[] args) throws IOException {

// // FileOutputStream fos = new FileOutputStream("hello.txt");

// // fos.write("Show me the money".getBytes());

//

// FileInputStream fis = new FileInputStream("hello.txt");

// DataInputStream dis = new DataInputStream(fis);

//

// String str = dis.readLine();

//

//// byte[] buf = new byte[512];

//// int len = fis.read(buf);

//// String str = new String(buf, 0, len);

// System.out.println(str);

// }

}

Network Programming

=> Socket Programming

Process

= 프로그램(실행 가능한 파일)의 인스턴스

= 실행 중인 프로그램

Windows

Linux => Protected mode => 가상 메모리

macOS

Real mode

=> MS DOS

프로세스가 사용하는 메모리는 반드시 물리적으로 연속적이어야 한다.

가상 메모리

- 세그먼트

- 페이징

- 모든 프로세스는 자신만의 페이지 테이블을 가지고 있다.

=> 모든 프로세스는 자신만의 가상 주소를 가지고 있다.

=> 서로 다른 프로세스가 데이터를 교환하기 위해서는 IPC가 필요하다.

System V IPC

=> Message Queue

=> Semaphore

=> Signal

=> Shared Memory

IPC

Network Programming

=> Socket Programming

TCP/IP

File I/O

VFS(Virtual File System)

open / read / write / close

TCP/IP - IPC

다른 컴퓨터의 프로세스와 통신할 수 있다.

주소

컴퓨터의 주소 - IP Address

프로세스의 주소 - Port Address

TCP 통신

=> Connection 수립

Client Server

connect-> IP / PORT

www.naver.com(Domain Name)

--> DNS(Domain Name Server) -> IP 주소로 변환

210.89.160.88

서비스의 종류에 따라서 약속된 PORT 번호가 있습니다.

--> Well-known Port Address

HTTP : 80

HTTPS: 443

SSH : 22

FTP : 21

Redis: 6379

KT

168.126.63.1

168.126.63.2

Google

8.8.8.8

8.8.4.4

Server 과정

1. Socket

Server Socket: 연결을 수립하기 위한 Socket

- socket, bind, listen, accept

=> ServerSocket

Socket : 데이터를 교환하기 위한 Socket

- read / write

=> Socket

package xyz.ourguide;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.Socket;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

try {

Scanner scanner = new Scanner(System.in);

Socket socket = new Socket("127.0.0.1", 5000);

try (InputStream is = socket.getInputStream();

OutputStream os = socket.getOutputStream()) {

byte[] buf = new byte[512];

while (scanner.hasNext()) {

String line = scanner.next();

os.write(line.getBytes());

int len = is.read(buf);

if (len == -1) {

break;

}

System.out.println("From server: " + new String(buf, 0, len));

}

} catch (IOException e) {

e.printStackTrace();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

// Concurrent: 병행

// 1) Process - Unix(Linux)

// 2) Thread - Windows, Java

// Parallel: 병렬(하드웨어)

// Concurrent Programming

// => 하나의 프로그램에서 동시에 여러개의 코드를 수행할 수 있도록 만들어야 한다.

// Thread per Connection Model Server

// Accept

// => Socket => new Thread() => Echo

// 문제점?

// => 연결이 적을 경우, 매우 잘 동작하나, 연결이 매우 많아지면, 점점 성능이 떨어진다.

// => Multiplexing Model을 사용하는 것이 좋다.

// Linux: poll, epoll

// BSD: kqueue

// Windows: IOCP

// Java: NIO2

class SessionThread extends Thread {

private Socket socket;

public SessionThread(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try (OutputStream os = socket.getOutputStream();

InputStream is = socket.getInputStream()) {

while (true) {

byte[] buf = new byte[512];

int len = is.read(buf);

if (len == -1) {

System.out.println("Disconnected");

break;

}

os.write(buf, 0, len);

}

} catch (IOException e) {

e.printStackTrace();

}

System.out.println("Thread 종료되었습니다.");

}

}

// Echo Server

public class Main {

public static void main(String[] args) {

try {

// ServerSocket

// socket()

// bind()

// listen(32)

ServerSocket serverSocket = new ServerSocket(5000);

while (true) {

Socket socket = serverSocket.accept();

SessionThread sessionThread = new SessionThread(socket);

sessionThread.start();

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

// Concurrent: 병행

// 1) Process - Unix(Linux)

// 2) Thread - Windows, Java

// Parallel: 병렬(하드웨어)

// Concurrent Programming

// => 하나의 프로그램에서 동시에 여러개의 코드를 수행할 수 있도록 만들어야 한다.

// Thread per Connection Model Server

// Accept

// => Socket => new Thread() => Echo

// 문제점?

// => 연결이 적을 경우, 매우 잘 동작하나, 연결이 매우 많아지면, 점점 성능이 떨어진다.

// => Multiplexing Model을 사용하는 것이 좋다.

// Linux: poll, epoll

// BSD: kqueue

// Windows: IOCP

// Java: NIO2

class SessionThread extends Thread {

private Socket socket;

public SessionThread(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try (OutputStream os = socket.getOutputStream();

InputStream is = socket.getInputStream()) {

while (true) {

byte[] buf = new byte[512];

int len = is.read(buf);

if (len == -1) {

System.out.println("Disconnected");

break;

}

os.write(buf, 0, len);

}

} catch (IOException e) {

e.printStackTrace();

}

System.out.println("Thread 종료되었습니다.");

}

}

// Echo Server

public class Main {

public static void main(String[] args) {

try {

// ServerSocket

// socket()

// bind()

// listen(32)

ServerSocket serverSocket = new ServerSocket(5000);

while (true) {

Socket socket = serverSocket.accept();

SessionThread sessionThread = new SessionThread(socket);

sessionThread.start();

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

package xyz.ourguide;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.Socket;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

try {

Scanner scanner = new Scanner(System.in);

Socket socket = new Socket("127.0.0.1", 5000);

try (InputStream is = socket.getInputStream();

OutputStream os = socket.getOutputStream()) {

byte[] buf = new byte[512];

while (scanner.hasNext()) {

String line = scanner.next();

os.write(line.getBytes());

int len = is.read(buf);

if (len == -1) {

break;

}

System.out.println("From server: " + new String(buf, 0, len));

}

} catch (IOException e) {

e.printStackTrace();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.ArrayList;

import java.util.List;

class SessionThread extends Thread {

private Socket socket;

private List<SessionThread> sessions;

public SessionThread(Socket socket, List<SessionThread> sessions) {

this.socket = socket;

this.sessions = sessions;

this.sessions.add(this);

}

private void broadcast(String message) throws IOException {

for (SessionThread e : sessions) {

OutputStream os = e.socket.getOutputStream();

os.write(message.getBytes());

}

}

@Override

public void run() {

try (InputStream is = socket.getInputStream()) {

while (true) {

byte[] buf = new byte[512];

int len = is.read(buf);

if (len == -1) {

System.out.println("Disconnected");

break;

}

String message = new String(buf, 0, len);

broadcast(message);

}

} catch (IOException e) {

e.printStackTrace();

}

System.out.println("Thread 종료되었습니다.");

sessions.remove(this);

}

}

// Echo Server

public class Main {

public static void main(String[] args) {

List<SessionThread> sessions = new ArrayList<>();

try {

ServerSocket serverSocket = new ServerSocket(5000);

while (true) {

Socket socket = serverSocket.accept();

SessionThread sessionThread = new SessionThread(socket, sessions);

sessionThread.start();

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

package xyz.ourguide;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.Socket;

import java.util.Scanner;

// Processing => GUI(Client) => 산성비

// Server(1초에 한개씩 연결된 클라이언트에게 단어를 전달한다.)

// Clean Code

// : 로버트 C 마틴(엉클 밥)

// 최소 지식의 원칙: 디미터의 법칙

// => 객체를 설계할 때, 최소한의 필요한 정보만 전달 받는 것이 좋다.

class ReceiveThread extends Thread {

private InputStream is;

ReceiveThread(InputStream is) {

this.is = is;

}

@Override

public void run() {

byte[] buf = new byte[512];

try {

while (true) {

int len = is.read(buf);

if (len == -1) {

break;

}

System.out.println("From server: " + new String(buf, 0, len));

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

is.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

public class Main {

public static void main(String[] args) {

try {

Scanner scanner = new Scanner(System.in);

Socket socket = new Socket("127.0.0.1", 5000);

ReceiveThread receiveThread = new ReceiveThread(socket.getInputStream());

receiveThread.start();

try (OutputStream os = socket.getOutputStream()) {

while (scanner.hasNext()) {

String line = scanner.next();

os.write(line.getBytes());

}

} catch (IOException e) {

e.printStackTrace();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

package xyz.ourguide;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.Socket;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

// Vector: Java의 잘못된 동적 배열 라이브러리

// 상속

// => 부모의 구현과 인터페이스를 물려 받는다.

// : 깨지기 쉬운 기반 클래스 문제

// 해결방법

// 1. 반드시 부모 클래스의 모든 필드는 private 이어야 한다.

// 2. 부모 클래스의 메소드를 통해 부모 클래스를 사용해야 한다.

// 클래스: 재사용

// => 잘못 사용하기 어렵게 만들어야 한다.

// 재사용을 사용하기 위해서는 '포함'이 좋다.

class Stack<E> {

private List<E> list = new ArrayList<E>();

public void push(E e) {

list.add(e);

}

public E pop() {

return list.remove(list.size() - 1);

}

public static void main(String[] args) {

Stack<Integer> s = new Stack<>();

s.push(10);

s.push(20);

s.push(30);

System.out.println(s.pop());

System.out.println(s.pop());

System.out.println(s.pop());

}

}

/\*

class Stack<E> extends Vector<E> {

public void push(E e) {

elementData[elementCount++] = e;

}

public E pop() {

return (E)elementData[--elementCount];

}

public static void main(String[] args) {

Stack<Integer> s = new Stack<>();

s.push(10);

s.push(20);

s.push(30);

System.out.println(s.pop());

System.out.println(s.pop());

System.out.println(s.pop());

}

}

\*/

// Processing => GUI(Client) => 산성비

// Server(1초에 한개씩 연결된 클라이언트에게 단어를 전달한다.)

// Clean Code

// : 로버트 C 마틴(엉클 밥)

// 최소 지식의 원칙: 디미터의 법칙

// => 객체를 설계할 때, 최소한의 필요한 정보만 전달 받는 것이 좋다.

class ReceiveThread extends Thread {

private InputStream is;

ReceiveThread(InputStream is) {

this.is = is;

}

@Override

public void run() {

byte[] buf = new byte[512];

try {

while (true) {

int len = is.read(buf);

if (len == -1) {

break;

}

System.out.println("From server: " + new String(buf, 0, len));

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

is.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

public class Main {

public static void main(String[] args) {

try {

Scanner scanner = new Scanner(System.in);

Socket socket = new Socket("127.0.0.1", 5000);

ReceiveThread receiveThread = new ReceiveThread(socket.getInputStream());

receiveThread.start();

try (OutputStream os = socket.getOutputStream()) {

while (scanner.hasNext()) {

String line = scanner.next();

os.write(line.getBytes());

}

} catch (IOException e) {

e.printStackTrace();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

package xyz.ourguide;

// Concurrent Package

// => 더그 리 교수

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

class Producer extends Thread {

private final BlockingQueue<String> queue;

Producer(BlockingQueue<String> queue) {

this.queue = queue;

}

@Override

public void run() {

for (int i = 0; i < 100; ) {

try {

queue.put("hello - " + i);

System.out.println("Prod - " + i);

i++;

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer extends Thread {

private final BlockingQueue<String> queue;

Consumer(BlockingQueue<String> queue) {

this.queue = queue;

}

@Override

public void run() {

for (int i = 0; i < 100; ++i) {

try {

String e = queue.take();

System.out.println("Consume - " + e);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

}

}

}

// Queue

// => BlockingQueue

public class Main {

public static void main(String[] args) throws Exception {

BlockingQueue<String> queue = new ArrayBlockingQueue<String>(5);

Thread[] threads = new Thread[] {

new Producer(queue),

new Producer(queue),

new Producer(queue),

new Producer(queue),

new Consumer(queue),

new Consumer(queue),

new Consumer(queue),

new Consumer(queue),

};

for (Thread thread : threads) {

thread.start();

}

for (Thread thread : threads) {

thread.join();

}

}

}

/\*

// Java 에서는 Mutex를 별도의 객체로 제공하지 않는다.

// => 모든 객체는 Mutex가 될 수 있다.

import java.util.concurrent.atomic.AtomicInteger;

// CAS(Atomic Operations)

class FooThread extends Thread {

// static int n = 0;

static AtomicInteger n = new AtomicInteger(0);

static final Object mutex = new Object();

@Override

public void run() {

for (int i = 0; i < 1000000; ++i) {

n.incrementAndGet();

}

}

}

/\*

class FooThread extends Thread {

static int n = 0;

static final Object mutex = new Object();

@Override

public void run() {

for (int i = 0; i < 1000000; ++i) {

synchronized (mutex) {

n++;

}

}

}

}

public class Main {

public static void main(String[] args) throws Exception {

Thread[] threads = new Thread[]{

new FooThread(),

new FooThread(),

};

for (int i = 0; i < 2; ++i) {

threads[i].start();

}

for (int i = 0; i < 2; ++i) {

threads[i].join();

}

System.out.println(FooThread.n);

}

}

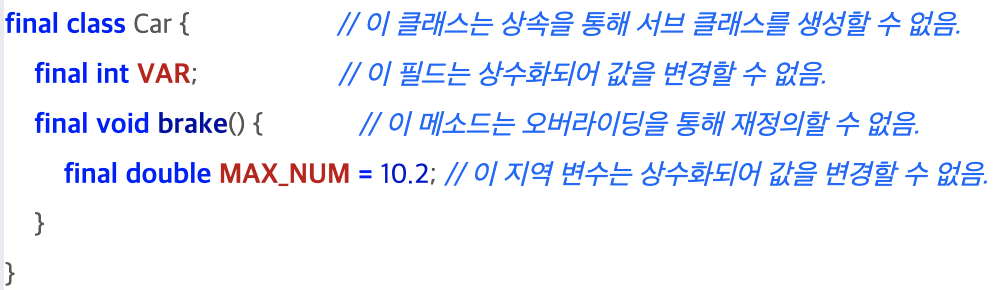
\*/

1. 접근 제어자 접근 범위

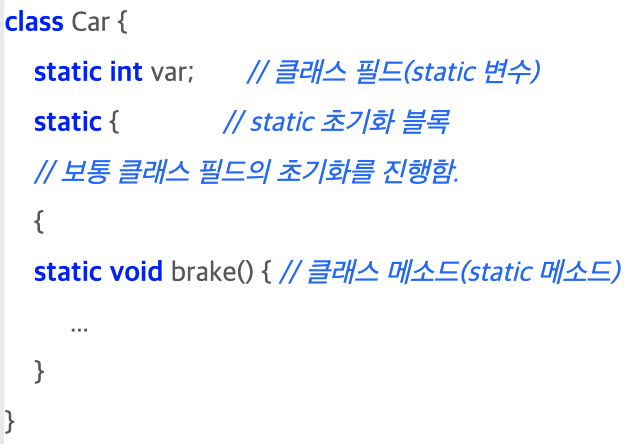
스크린샷이(가) 표시된 사진

자동 생성된 설명

2. final



3. static



4. abstract

스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

5. 필드

스크린샷이(가) 표시된 사진

자동 생성된 설명

지역변수는 컴파일 하지 않으면 자바 컴파일러가 오류 발생

스크린샷이(가) 표시된 사진

자동 생성된 설명

클래스변수 = 공유변수

나이프, 테이블이(가) 표시된 사진

자동 생성된 설명

6. 메소드

스크린샷이(가) 표시된 사진

자동 생성된 설명

7. 필드 초기화

조류이(가) 표시된 사진

자동 생성된 설명

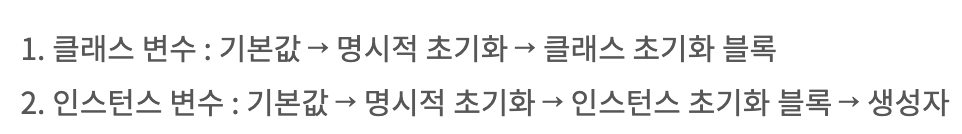
초기화 블록은 {}를 이용한 초기화를 말한다.

스크린샷, 닫기, 테이블이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명



8. 상속

테이블, 나이프이(가) 표시된 사진

자동 생성된 설명

조류이(가) 표시된 사진

자동 생성된 설명

나이프이(가) 표시된 사진

자동 생성된 설명

9. Object 클래스

스크린샷이(가) 표시된 사진

자동 생성된 설명

10. 변수 = 기본형+참조형

스크린샷이(가) 표시된 사진

자동 생성된 설명

new로 생성자를 통해 생성하는 변수가 참조 변수이다. 16진수 참조값이 할당되며 참조값과 연결된 시스템 내부이 인덱스 테이블을 검색하고, 해당 메모리의 변수 자리를 찾아 할당한다. + 가져온다.

11. super와 super() this와 this()

나이프, 테이블이(가) 표시된 사진

자동 생성된 설명

나이프, 테이블이(가) 표시된 사진

자동 생성된 설명 

나이프, 테이블이(가) 표시된 사진

자동 생성된 설명

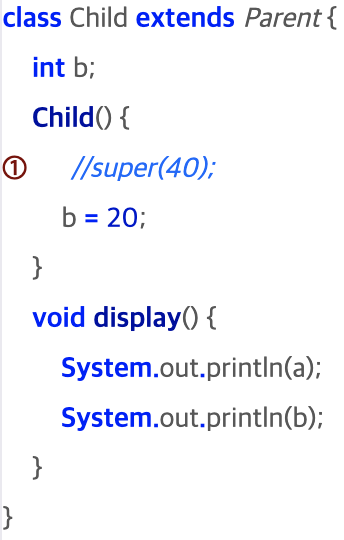
즉, 둘 다 인스턴스의 주소를 가리키는 참조 변수이므로, 인스턴스가 없어도 실행잉 되어야 하는 static에선 쓸수 없다.

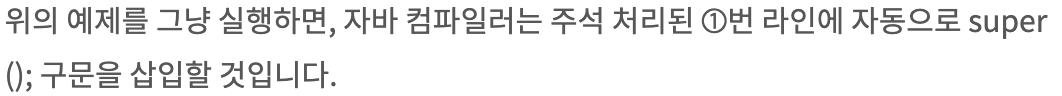
나이프, 테이블이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

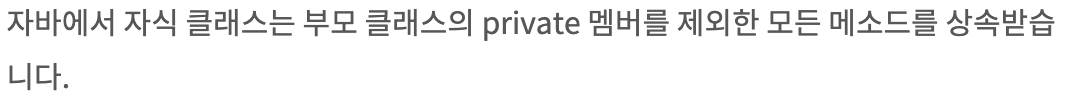




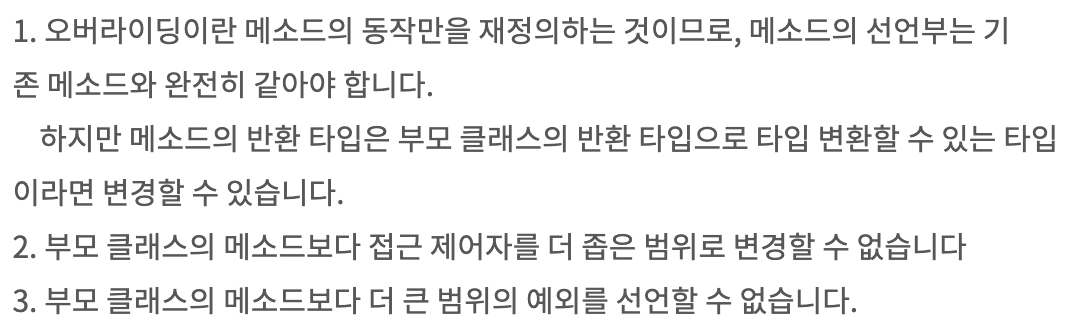
12. 오버라이딩

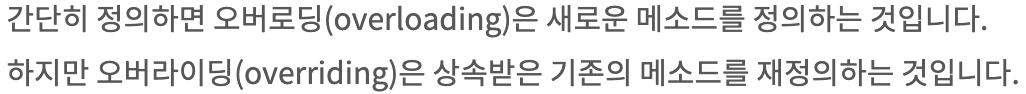
조류이(가) 표시된 사진

자동 생성된 설명



조건





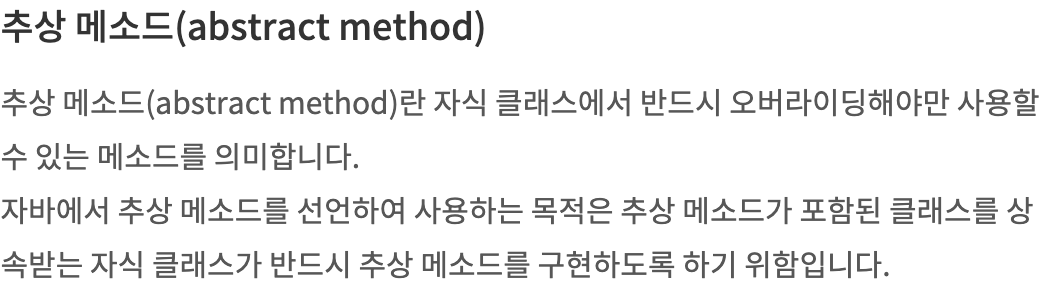
13. 다형성

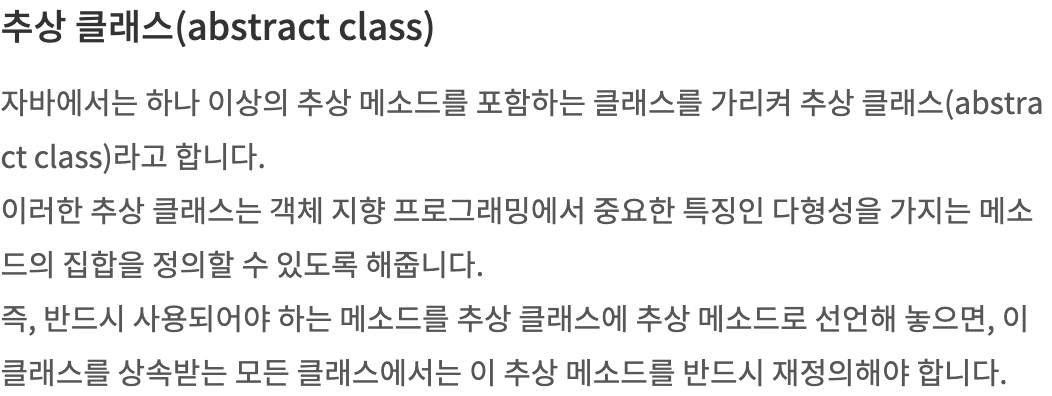
스크린샷이(가) 표시된 사진

자동 생성된 설명



13. 추상 메소드





14. interface

나이프, 테이블이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명



나무이(가) 표시된 사진

자동 생성된 설명

조류, 나무이(가) 표시된 사진

자동 생성된 설명

15. 이너 클래스

스크린샷이(가) 표시된 사진

자동 생성된 설명

나무, 조류이(가) 표시된 사진

자동 생성된 설명

주의!!!!!!외부에서는 내부 클래스에 접근할 수 없다!!!!!!!!!!

조류, 나무, 꽃이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

16. equals

같은 주소를 가리켜야 true를 반환한다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

17. 래퍼 클래스

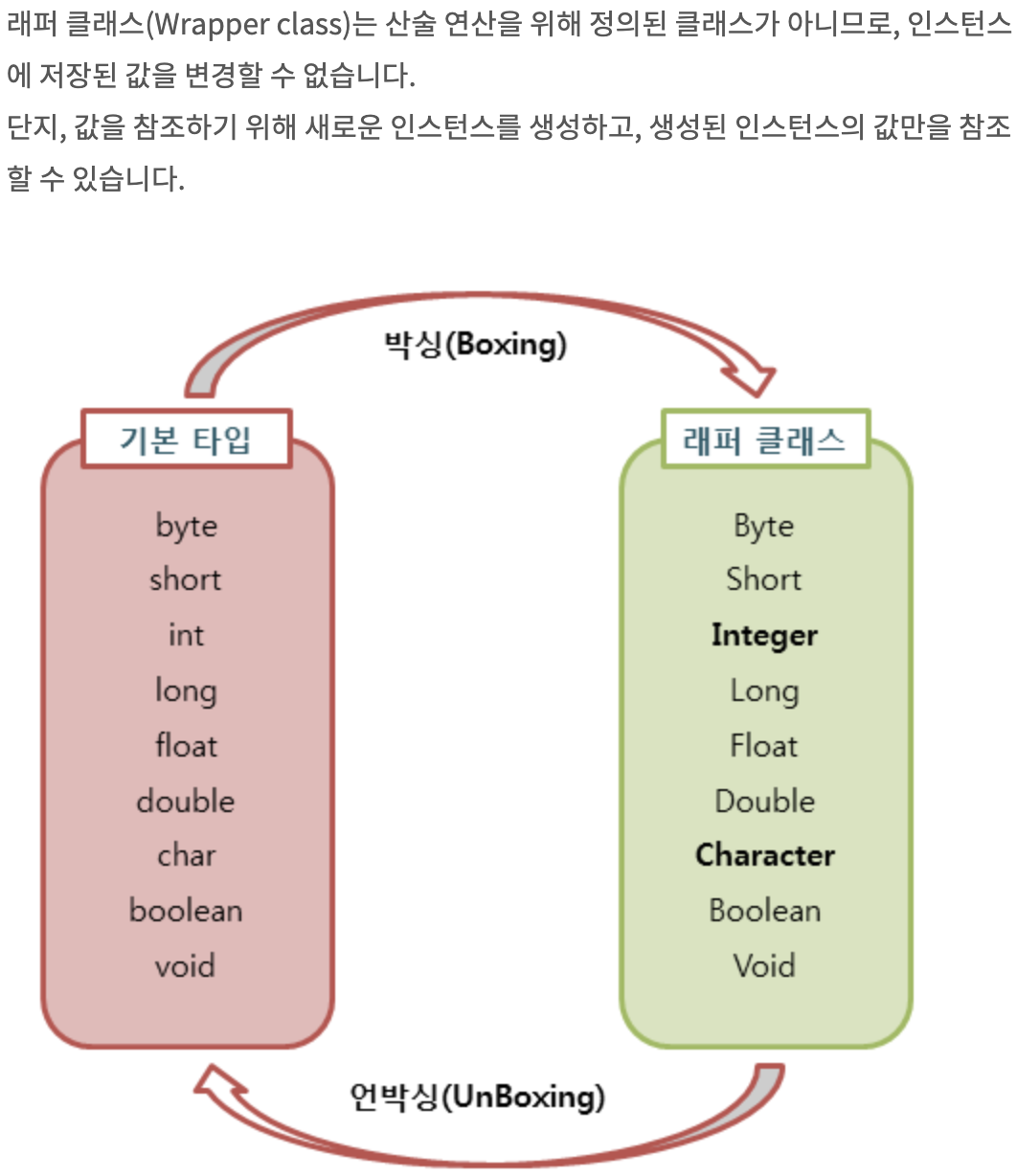
스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

18. 박싱과 언박싱



스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

19. enum

스크린샷이(가) 표시된 사진

자동 생성된 설명

20. 제네릭(generic)

나이프, 테이블이(가) 표시된 사진

자동 생성된 설명­

스크린샷이(가) 표시된 사진

자동 생성된 설명

제너릭은 컴파일 시 컴파일러에 의해 자동으로 변환되며, 모두 제거되어 어떤 제너릭 타입도 포함되지 않게 된다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

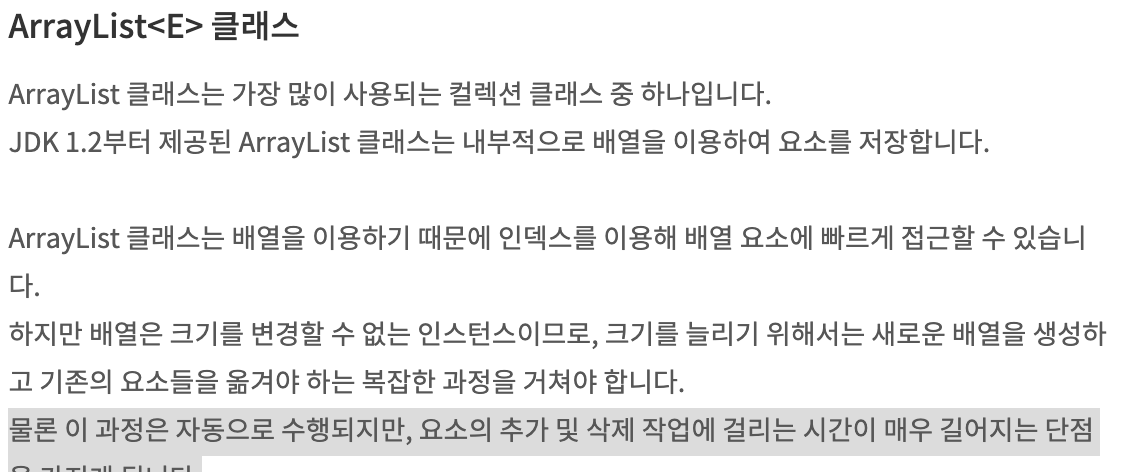
자동 생성된 설명

21컬렉션 프레임워크

스크린샷이(가) 표시된 사진

자동 생성된 설명

22. 어레이 리스트와 링크드 리스트



어레이 리스트는 배열기반 링크드 리스트는 이중 연결 리스트 기반

조류이(가) 표시된 사진

자동 생성된 설명

조류이(가) 표시된 사진

자동 생성된 설명

나이프이(가) 표시된 사진

자동 생성된 설명

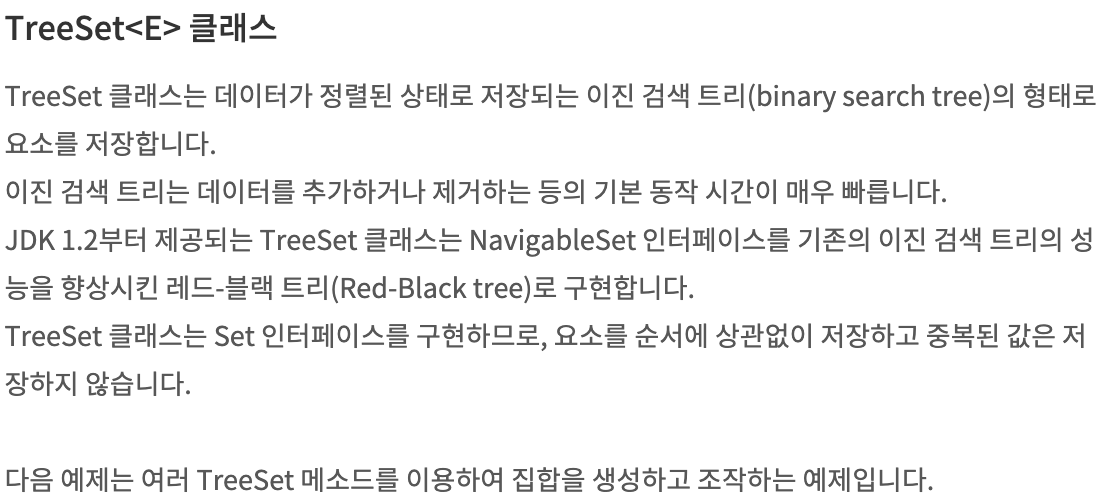
23. 해시와 해시셋

스크린샷, 조류, 나무이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

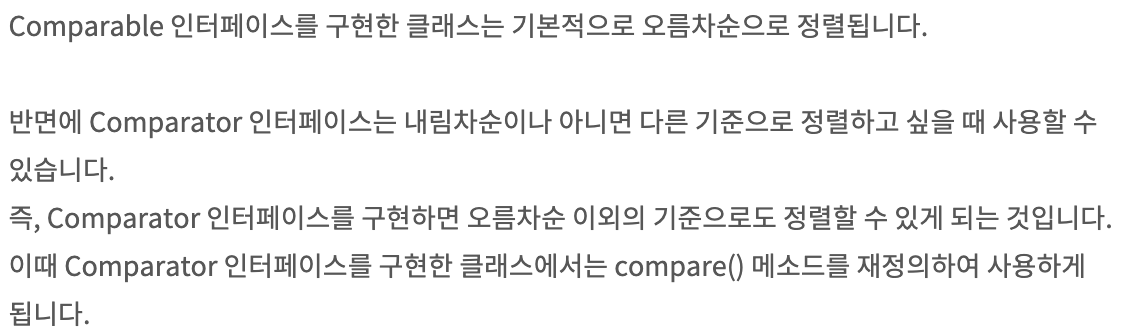


24. 맵 컬렉션

나이프, 테이블이(가) 표시된 사진

자동 생성된 설명

25. compare



26. thread

나무, 조류, 꽃이(가) 표시된 사진

자동 생성된 설명

27.데몬 스레드

스크린샷, 나무, 조류이(가) 표시된 사진

자동 생성된 설명

28. 가비지 컬렉터

스크린샷이(가) 표시된 사진

자동 생성된 설명

29. 람다와 함수형 인터페이스

스크린샷이(가) 표시된 사진

자동 생성된 설명

30. 메소드 참조

스크린샷이(가) 표시된 사진

자동 생성된 설명

31. 배열과 컬렉션에서 반복자를 사용하지 않기 위한 스트림

나무, 조류이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

32. optional

실내, 조류이(가) 표시된 사진

자동 생성된 설명