🔵

# [JAVA]

📌 프로그래머스 자바 입문, 자바 중급 강의 정리

## 입문

▼ 자바의 특징

1995년에 썬 마이크로시스템즈에서 발표한 객체지향 언어

- 플랫폼에 독립적
  JVM만 있으면 윈도우, 리눅스, 맥 등 어떤 플랫폼에서도 실행 가능
- 객체지향언어
- Garbage Collector로 사용되지 않는 메모리를 자동 정리

▼ 참조변수

참조변수끼리 `==` 으로 비교하면 서로 같은 것을 참조하는지를 비교함

▼ `String` 클래스

**인스턴스 생성 방법**

- new 연산자를 이용하지 않고 인스턴스를 만드는 경우
  문자열이 상수가 저장되는 영역에 저장되어 `str1`이 참조하는 인스턴스를 `str2`도 참조

```
String str1 = "hello";
String str2 = "hello";
```

- new 연산자를 이용해서 인스턴스를 만드는 경우
  인스턴스는 무조건 새롭게 만들어지므로 `str3`과 `str4`는 서로 다른 인스턴스를 참조

```
String str3 = new String("hello");
String str4 = new String("hello");
```

```
if(str1 == str2) {
  System.out.println("str1과 str2는 같은 레퍼런스입니다.");
}

if(str1 == str3) {
  System.out.println("str1과 str3는 같은 레퍼런스입니다.");
}

if(str3 == str4) {
  System.out.println("str3과 str4는 같은 레퍼런스입니다.");
}

// 실행 결과
str1과 str2는 같은 레퍼런스입니다.
```

### String 특징

- 불변 클래스이므로 String이 인스턴스가 될때 가지고 있던 값을 나중에 수정할 수 없음
- 메소드를 호출한다 하더라도 String은 내부의 값이 변하지 않으며, String을 반환하는 메소드는 모두 새로운 String을 생성해서 반환함

### String 클래스 메소드

- 문자열 길이 구하기

```
String str = "hello";
System.out.println(str.length()); // 5
```

- 문자열 붙이기

```
String str = "hello";
System.out.println(str.concat(" world"));  // hello world
System.out.println(str);  // hello
```

```
str = str.concat(" world");
System.out.println(str);  // hello world
```

- 문자열 자르기

```
System.out.println(str.substring(1, 3)); // el
System.out.println(str.substring(2));   // llo world
```

▼ 클래스(class), 필드(field), 메소드(method)

**클래스 선언**

객체를 만들기 위해서는 반드시 클래스를 생성해야 하며, 클래스는 필드와 메소드로 이루어짐

```
public class CarExam{
  public static void main(String args[]){
    Car c1 = new Car();
    Car c2 = new Car();
  }
}

public class Car{

}
```

**필드 선언**

```
public class CarExam{
  public static void main(String args[]){
    Car c1 = new Car();
    Car c2 = new Car();

    c1.name = "소방차";
    c1.number = 1234;
  }
}

public class Car{
    String name;
    int number;
}
```

### 메소드 선언

입력값을 매개변수, 결과값을 리턴값이라고 함
`인자(Argument)` : 어떤 함수를 호출시에 전달되는 값을 의미
`매개변수(Parameter)` : 그 전달된 인자를 받아들이는 변수를 의미

```
public class MyClassExam{
  public static void main(String args[]){
    MyClass my = new MyClass();
    my.method1();
    my.method2(10);
    int x = my.method3();
    my.method4(10,100);
    int x2 = my.method5(50);
  }
}

public class MyClass{
  public void method(){
    System.out.println("method1이 실행됩니다.");
  }

  public void method2(int x){
    System.out.println(x + " 를 이용하는 method2입니다.");
  }

  public int method3(){
    System.out.println("method3이 실행됩니다.");
    return 10;
  }

  public void method4(int x, int y){
    System.out.println(x + "," + y + " 를 이용하는 method4입니다.");
  }

  public int method5(int y){
    System.out.println(y + " 를 이용하는 method5입니다.");
    return 5;
  }
}
```

▼ 변수의 스코프(scope)와 `static`

### 변수의 스코프

프로그램상에서 사용되는 변수들은 사용 가능한 범위를 가지며, 그 범위를 변수의 스코프라고 함

```
public class VariableScopeExam {
  int globalScope = 10;

  public void scopeTest(int value){
    int localScope = 20;
    System.out.println(globalScope);
    System.out.println(localScope);
    System.out.println(value);
  }

  public static void main(String[] args) {
    System.out.println(globalScope);  //오류
    System.out.println(localScope);   //오류
    System.out.println(value);        //오류
  }
}
```

### static

main 메소드는 `static` 키워드로 정의되어 있으며, 이런 메서드를 static 한 메소드라고 함
`static`한 필드나, `static`한 메소드는 Class가 인스턴스화 되지 않아도 사용 가능함

```
public class VariableScopeExam {
  int globalScope = 10;      // 인스턴스 변수
  static int staticVal = 7;  // 클래스 변수

  public void scopeTest(int value){
    int localScope = 20;
  }

  public static void main(String[] args) {
    System.out.println(staticVal);      //사용가능
  }
}
```

static으로 선언된 변수는 값을 저장할 수 있는 공간이 하나만 생성되어 인스턴스가 여러 개 생성되어도 static한 변수는 하나임(값을 공유)

```
public static void main(String[] args) {
  ValableScopeExam v1 = new ValableScopeExam();
  ValableScopeExam v2 = new ValableScopeExam();
  v1.golbalScope = 20;
  v2.golbalScope = 30;
```

```
  System.out.println(v1.golbalScope);  // 20
  System.out.println(v2.golbalScope);  // 30

  v1.staticVal = 10;
  v2.staticVal = 20;

  System.out.println(v1.statVal);  // 20
  System.out.println(v2.statVal);  // 20
}
```

▼ 생성자

**생성자의 특징**

- 생성자는 리턴타입이 없음
- 생성자를 만들지 않으면 컴파일 시 매개변수가 없는 기본 생성자가 자동으로 만들어짐
- 생성자를 만든 경우 컴파일 시 기본 생성자가 자동으로 만들어지지 않음

**생성자의 역할**

- 모든 클래스는 인스턴스화 될 때 생성자를 사용함
- 생성자는 인스턴스화 될 때 필드를 초기화하는 역할을 수행함

```
public class CarExam2{
  public static void main(String args[]){
    Car c1 = new Car("소방차");
    Car c2 = new Car("구급차");
    Car c3 = new Car(); // 컴파일 오류

    System.out.println(c1.name);
    System.out.println(c2.name);
  }
}

public class Car{
  String name;
  int number;

  public Car(String n){
    name = n;
  }
}
```

▼ `this`

- 객체 자신을 참조하는 키워드
- 필드 뿐만 아니라 클래스 안에서 자기 자신이 가지고 있는 메소드를 사용할 때도 `this.메소드명()`으로 호출 가능함

### `this`를 사용하지 않은 경우

가깝게 선언된 변수를 우선 사용하므로 매개변수의 `name` 값을 매개변수 `name`에 대입하게 되어 필드 값은 바뀌지 않음

```
public class Car{
  String name;
  int number;

  public Car(String name){
    name = name;
  }
}
```

### `this`를 사용한 경우

`this.name`은 필드 name을 의미, `name`은 매개변수를 의미하여 매개변수의 값을 필드에 대입함

```
public class Car{
  String name;
  int number;

  public Car(String name){
    this.name = name;
  }
}
```

▼ 메소드 오버로딩

매개변수의 유형(타입)과 개수를 다르게 하여 같은 이름의 메소드를 여러 개 가질 수 있게 함

```
public MethodOverloadExam{
  public static void main(String args[]){
    MyClass2 m = new MyClass2();
    System.out.println(m.plus(5, 10));              // 15
    System.out.println(m.plus(5, 10, 15));          // 30
    System.out.println(m.plus("hello" + " world")); // "hello world"
  }
}

class MyClass2{
  public int plus(int x, int y){
    return x + y;
  }

  public int plus(int x, int y, int z){
    return x + y + z;
  }

  public String plus(String x, String y){
    return x + y;
  }
}
```

▼ 생성자 오버로딩과 `this()`

**생성자 오버로딩**

매개변수의 유형(타입)과 개수를 다르게 하여 같은 이름의 생성자를 여러 개 가질 수 있게 함

```
public class CarExam4{
  public static void main(String args[]){
    Car c1 = new Car();
    Car c2 = new Car("소방차");
    Car c3 = new Car("구급차", 1234);
  }
}

public class Car{
  String name;
  int number;

  public Car(){

  }

  public Car(String name){
    this.name = name;
  }
```

```
  public Car(String name, int number){
    this.name = name;
    this.number = number;
  }
}
```

### 자신의 생성자를 호출하는 `this()`

자신의 생성자를 호출함으로써 비슷한 코드가 중복되는 것을 방지

```
public Car(){
  // this.name = "이름없음";
  // this.number = 0;
  this("이름없음", 0);
}
```

▼ 상속

`extends` 키워드로 부모 클래스를 상속 받을 수 있음

```
public class BusExam{
  public static void main(String args[]){
    Bus bus = new Bus();
    bus.run();
    bus.ppangppang();
  }
}

public class Car{
  public void run(){
    System.out.println("달리다.");
  }
}

public class Bus extends Car{
  public void ppangppang(){ // 메소드 확장
    System.out.println("빵빵");
  }
}
```

▼ 추상 클래스

**추상 클래스 정의**

- 클래스명 앞에 `abstract` 키워드 사용
- 미완성의 추상 메소드(구현되지 않은 메소드로, 리턴타입 앞에 `abstract` 키워드 사용) 포함 가능
- 인스터스화 할 수 없음

```
public abstract class Bird{
  public abstract void sing();
  public void fly(){
    System.out.println("날다.");
  }
}
```

### 추상 클래스를 상속받는 클래스 생성

추상 클래스를 상속받는 클래스는 추상 클래스가 가지고 있는 추상 메소드를 반드시 구현해야 하며, 추상 메소드를 구현하지 않는 경우 해당 클래스도 추상 클래스가 됨

```
public class Duck extends Bird{
  @Override
  public void sing() {
    System.out.println("꽥꽥!!");
  }
}
```

### 추상 클래스 사용

```
public class DuckExam {
  public static void main(String[] args) {
    Duck duck = new Duck();
    duck.sing();
    duck.fly();
    //Bird b = new Bird();
  }
}
```

▼ `super`와 부모 생성자

### 부모 생성자

- 클래스가 인스턴스화 될 때 생성자가 실행되며 객체를 초기화함
- 상속받은 클래스의 경우 자신의 생성자만 실행되는 것이 아니라 부모의 생성자부터 실행됨

```
public class BusExam{
  public static void main(String args[]){
    Bus b = new Bus();
  }
}

public class Car{
  public Car(){
    System.out.println("Car의 기본생성자입니다.");
  }
}

public class Bus extends Car{
  public Bus(){
    System.out.println("Bus의 기본생성자입니다.");
  }
}

// 실행결과
// Car의 기본생성자입니다.
// Bus의 기본생성자입니다.
```

`super`

- 자신을 가리키는 키워드가 `this`라면 부모를 가리키는 키워드는 `super`
- `super()`는 부모의 생성자를 의미함
- 부모의 생성자를 임의로 호출하지 않으면, 부모 클래스의 기본 생성자가 자동으로 호출됨

```
public class BusExam{
  public static void main(String args[]){
    Bus b = new Bus();
  }
}

public class Car{
  public Car(){
    System.out.println("Car의 기본생성자입니다.");
  }
}

public class Bus extends Car{
  super(); // 부모의 생성자가 기본 생성자이므로 생략 가능
```

```
  public Bus(){
    System.out.println("Bus의 기본생성자입니다.");
  }
}

// 실행 결과
// Car의 기본생성자입니다.
// Bus의 기본생성자입니다.
```

### 부모의 기본 생성자가 아닌 다른 생성자 호출

```
public class BusExam{
  public static void main(String args[]){
    Bus b = new Bus();
  }
}

public class Car{
  public Car(String name){
    System.out.println(name + " 를 받아들이는 생성자입니다.");
  }
}

public class Bus extends Car{
  public Bus(){
    super("소방차"); //  생략 불가능
    System.out.println("Bus의 기본생성자입니다.");
  }
}
```

▼ 오버라이딩

부모가 가지고 있는 메소드와 똑같은 모양의 메소드를 자식이 가지고 있는 것으로 오버라이딩은 메소드를 재정의 함

```
public class BusExam{
  public static void main(String args[]){
    Bus bus = new Bus();
    bus.run();
  }
}

public class Car{
  public void run(){
    System.out.println("Car의 run 메소드");
  }
}
```

```
public class Bus extends Car{

}

// 실행결과
// Car의 run 메소드
```

```
public class BusExam{
  public static void main(String args[]){
    Bus bus = new Bus();
    bus.run();
  }
}

public class Car{
  public void run(){
    System.out.println("Car의 run 메소드");
  }
}

public class Bus extends Car{
  public void run(){
    System.out.println("Bus의 run 메소드");
  }
}

// 실행결과
// Bus의 run 메소드
```

```
public class BusExam{
  public static void main(String args[]){
    Bus bus = new Bus();
    bus.run();
  }
}

public class Car{
  public void run(){
    System.out.println("Car의 run 메소드");
  }
}

public class Bus extends Car{
  public void run(){
    super.run();
    System.out.println("Bus의 run 메소드");
  }
}

// 실행결과
// Car의 run 메소드
// Bus의 run 메소드
```

▼ 클래스 형변환

- 상속 관계에 있는 경우 객체끼리도 형변환 가능
- 부모 타입으로 자식 객체를 참조하게 된다면 부모의 메소드만 사용 가능하므로 자식 객체가 가지고 있는 메소드나 속성을 사용하려면 형변환 필요
- 부모 타입으로 자식 타입의 객체를 참조할 때는 묵시적으로 형변환
- 부모 타입의 객체를 자식 타입으로 참조할 때는 명시적으로 형변환
  (단, 부모가 참조하는 객체가 형변환 하려는 자식 타입일 경우만 가능)

```
public class BusExam{
  public static void main(String args[]){
    Car car = new Bus();
    car.run();
    //car.ppangppang(); 컴파일 오류 발생

    Bus bus = (Bus)car;  //부모타입을 자식타입으로 형변환
    bus.ppangppang();
  }
}

public class Car{
  public void run(){
    System.out.println("Car의 run메소드");
  }
}

public class Bus extends Car{
  public void ppangppang(){
    System.out.println("빵빵.");
  }
}
```

**[예제]**

```
public class GasStation{
  public static void main(String[]args){
    GasStation gasStation = new GasStation();
    Suv suv = new Suv()
    Truck truck = new Truck();
    Bus bus = new Bus();

    gasStation.fill(suv);
    gasStation.fill(truck);
    gasStation.fill(bus);
```

```
    }

  public void fill(Suv suv){
    System.out.println("Suv에 기름을 넣습니다.");
    suv.gas += 10;
    System.out.println("기름이 "+suv.gas+"리터 들어있습니다.");
  }

  public void fill(Truck truck){
    System.out.println("Truck에 기름을 넣습니다.");
    truck.gas += 10;
    System.out.println("기름이 "+truck.gas+"리터 들어있습니다.");
  }

  public void fill(Bus bus){
    System.out.println("Bus에 기름을 넣습니다.");
    bus.gas += 10;
    System.out.println("기름이 "+bus.gas+"리터 들어있습니다.");
  }
}

public class Car{
  public int gas;
}

public class Suv extends Car{
}

public class Truck extends Car{
}

public class Bus extends Car{
}

// 실행결과
// Suv에 기름을 넣습니다.
// 기름이 10리터 들어있습니다.
// Truck에 기름을 넣습니다.
// 기름이 10리터 들어있습니다.
// Bus에 기름을 넣습니다.
// 기름이 10리터 들어있습니다.
```

```
public class GasStation{
  public static void main(String[]args){
    GasStation gasStation = new GasStation();
    Suv suv = new Suv()
    Truck truck = new Truck();
    Bus bus = new Bus();

    gasStation.fill(suv);
    gasStation.fill(truck);
    gasStation.fill(bus);

    }
```

```
  public void fill(Car car){
    System.out.println(car.getClass().getName()+"에 기름을 넣습니다.");
    car.gas += 10;
    System.out.println("기름이 "+car.gas+"리터 들어있습니다.");
  }
}

public class Car{
  public int gas;
}

public class Suv extends Car{
}

public class Truck extends Car{
}

public class Bus extends Car{
}

// 실행결과
// Suv에 기름을 넣습니다.
// 기름이 10리터 들어있습니다.
// Truck에 기름을 넣습니다.
// 기름이 10리터 들어있습니다.
// Bus에 기름을 넣습니다.
// 기름이 10리터 들어있습니다.
```

▼ 인터페이스

**인터페이스 정의(추상 메소드와 상수)**

인터페이스에서 변수를 선언하고 메소드를 정의하면 컴파일 시 자동으로 상수와 추상 메소드로 변경됨

```
public interface TV{
  public int MAX_VOLUME = 100;
  public int MIN_VOLUME = 0;

  public void turnOn();
  public void turnOff();
  public void changeVolume(int volume);
  public void changeChannel(int channel);
}
```

```
public interface TV{
  public static final int MAX_VOLUME = 100;
  public static final int MIN_VOLUME = 0;

  public abstract void turnOn();
```

```
  public abstract void turnOff();
  public abstract void changeVolume(int volume);
  public abstract void changeChannel(int channel);
}
```

### 인터페이스 사용하기

- 인터페이스를 구현하는 클래스에서 `implements` 키워드 이용
- 인터페이스가 가지고 있는 메소드를 하나라도 구현하지 않는다면 해당 클래스는 추상 클래스가 되며, 추상 클래스는 인스턴스화 할 수 없음
- 참조변수 타입으로 인터페이스 사용 가능하며, 이 경우 인터페이스가 가지고 있는 메소드만 사용 가능

```
public class LedTVExam{
  public static void main(String args[]){
    TV tv = new LedTV();
    tv.on();
    tv.volume(50);
    tv.channel(6);
    tv.off();
  }
}

public class LedTV implements TV{
  public void on(){
    System.out.println("켜다");
  }

  public void off(){
    System.out.println("끄다");
  }

  public void volume(int value){
    System.out.println(value + "로 볼륨조정하다.");
  }

  public void channel(int number){
    System.out.println(number + "로 채널조정하다.");
  }
}
```

### 인터페이스의 메소드(JAVA 8)

- `default` 메소드
  인터페이스가 `default` 키워드로 선언되면 메소드 구현 가능하며, 이를 구현하는 클

래스는 `default` 메소드를 오버라이딩 할 수 있음

```
public class MyCalculatorExam {
  public static void main(String[] args){
    Calculator cal = new MyCalculator();
    int value = cal.exec(5, 10);
    System.out.println(value);
  }
}

public interface Calculator {
  public int plus(int i, int j);
  public int multiple(int i, int j);
  default int exec(int i, int j){
    return i + j;
  }
}

public class MyCalculator implements Calculator {
  @Override
  public int plus(int i, int j) {
    return i + j;
  }

  @Override
  public int multiple(int i, int j) {
    return i * j;
  }
}
```

- `static` 메소드
  인터페이스에 `static` 메소드를 선언하면 인터페이스를 이용하여 간단한 기능을 가지는 유틸리티성 인터페이스를 만들 수 있으며, `static` 메소드는 반드시 `인터페이스명.메소드명` 형식으로 호출해야 함

```
public class MyCalculatorExam
  public static void main(String[] args){
    Calculator cal = new MyCalculator();

    int value = cal.exec(5, 10);
    System.out.println(value);

    int value2 = Calculator.exec2(5, 10);
    System.out.println(value2);
  }
}

public interface Calculator {
  public int plus(int i, int j);
  public int multiple(int i, int j);
```

```
  default int exec(int i, int j){
    return i + j;
  }

  public static int exec2(int i, int j){
    return i * j;
  }
}
```

# 중급

▼ `java.lang` 패키지

`java.lang` 패키지의 클래스는 `import` 하지 않고도 사용 가능

**`java.lang` 패키지의 종류**

- Wrapper 클래스
  `Boolean`, `Character`, `Byte`, `Short`, `Intger`, `Long`, `Float`, `Double`
- Object 클래스
- 문자열과 관련된 클래스
  `String`, `StringBuffer`, `StringBuilder`
- System 클래스
- Math 클래스
- Thread 클래스

▼ 오토박싱과 오토언박싱

**오토박싱(Auto Boxing)**

기본 타입의 데이터를 Wrapper 클래스의 인스턴스로 변환하는 과정

**오토언박싱(Auto Unboxing)**

Wrapper 클래스의 인스턴스에 저장된 값을 다시 기본 타입의 데이터로 꺼내는 과정

```
public class WrapperExam {
  public static void main(String[] args) {
    int i = 5;
    // Auto Boxing
    // Integer i2 = new Integer(5);
    Integer i3 = 5;
    // Auto Unboxing
    // int i4 = i2.intValue();
    int i5 = i2;
  }
}
```

▼ `int`와 `Integer`

**자료형 - 기본형(primitive type)**

- `null`값을 가질 수 없음
- 변수의 선언과 동시에 메모리 생성
- 메모리의 스택에 저장되며, 저장공간에 실제 자료 값을 저장
- 산술연산 가능

| 자료형 | 데이터 | 메모리 크기 | 표현 가능 범위 |
|---|---|---|---|
| boolean | 참/거짓 | 1byte | true, false |
| char | 문자 | 2byte | 모든 유니코드 문자 |
| byte | 정수 | 1byte | -128~127 |
| short | | 2byte | -32768~32767 |
| int | | 4byte | -2147483648~2147483647 |
| long | | 8byte | -9223372036854775808~9223372036854775807 |
| float | 실수 | 4byte | 1.4E-45~3.4028235E38 |
| double | | 8byte | 4.9E-324~1.7976931348623157E308 |

**자료형 - 참조형(referece type)**

- 기본형 타입을 제외한 나머지

- 문자열(String), 배열(Array), 클래스(Class), 인터페이스(Inteface) 등
- `null` 값으로 초기화 가능
- 기본형과 달리 실제 값이 저장되지 않고 자료가 저장된 공간의 주소를 저장
- 메모리의 힙에 실제 값을 저장하고, 주소값을 갖는 변수는 스택에 저장
- 언박싱 하지 않는 경우 산술연산 불가

**`Integer`은 `int`의 Wrapper 클래스**

## Wrapper Classes

| Primitive Data Type | Wrapper Class |
|---|---|
| *double* | *Double* |
| *float* | *Float* |
| *long* | *Long* |
| *int* | *Integer* |
| *short* | *Short* |
| *byte* | *Byte* |
| *char* | *Character* |
| *boolean* | *Boolean* |

7

▼ `StringBuffer`

**`StringBuffer`와 `String` 차이**

- `String`은 한 번 선언되면 문자를 추가하거나 삭제할 수 없는 불변(immutable) 타입
- `StringBuffer`는 변경(mutabl) 가능한 타입

### `StringBuffer` 객체 생성 및 사용

```
StringBuffer sb = new StringBuffer();
sb.append("hello");
sb.append(" ");
sb.append("world");
String str = sb.toString();
```

### 메소드 체이닝(Method Chaining)

자기 자신을 리턴하여 계속해서 자신의 메소드를 호출하는 방식

```
StringBuffer sb2 = new StringBuffer();
StringBuffer sb3 = sb2.append("hello");
if(sb2 == sb3){
  System.out.println("sb2 == sb3");
}

// 실행결과
// sb2 == sb3
```

### String 클래스의 문제점

- `String` 클래스는 불변 클래스이므로 문자열과 문자열을 더하게 되면 내부적으로 `StringBuffer` 객체를 생성하여 문자열이 더해짐
- 문자열을 반복문 안에서 더하는 것은 성능상 문제가 생길 수 있음

```
String str1 = "hello world";
String str2 = str1.substring(5);
String str3 = str1 + str2;
// 내부적으로 실행되는 코드
// String str3 = new StringBuffer().append(str1).append(str2).toString();
System.out.println(str3);

// 실행결과
// hello world world
```

```
public class StringBufferPerformanceTest{
  public static void main(String[] args){
    // #1 String
    // 시작시간을 기록(millisecond단위)
    long startTime1 = System.currentTimeMillis();
```

```
    String str="";
    for(int i=0;i<10000;i++){
      str=str+"*";
    }
    // 종료시간을 기록(millisecond단위)
    long endTime1 = System.currentTimeMillis();

    // #2 StringBuffer
    // 시작시간을 기록(millisecond단위)
    long startTime2 = System.currentTimeMillis();
    StringBuffer sb = new StringBuffer();
    for(int i=0;i<10000;i++){
      sb.append("*");
    }
    // 종료시간을 기록(millisecond단위)
    long endTime2 = System.currentTimeMillis();

    long duration1 = endTime1-startTime1;
    long duration2 = endTime2-startTime2;

    System.out.println("String을 이용한 경우 : "+ duration1);
    System.out.println("StringBuffer를 이용한 경우 : "+ duration2);
  }
}

// 실행결과
// String을 이용한 경우 : 22
// StringBuffer를 이용한 경우 : 1
```

▼ `Math` 클래스

- `Math` 클래스는 생성자가 `private`으로 되어 있기 때문에 `new` 연산자를 이용하여 객체를 생성할 수 없음
- 객체를 생성할 수는 없지만 모든 메소드와 속성이 `static`으로 정의되어 있기 때문에 객체를 생성하지 않고도 사용할 수 있으며, `클래스명.메소드명` 형식으로 호출하여 사용

▼ 제네릭(Generic)

- JAVA 5부터 인스턴스화할  사용하는 타입을 지정하는 Generic (`<>`)문법이 추가됨
- Generic을 사용함으로써 선언할 때는 가상의 타입으로 선언하고, 사용할 때 구체적인 타입을 설정하여 다양한 타입의 클래스를 이용할 수 있음

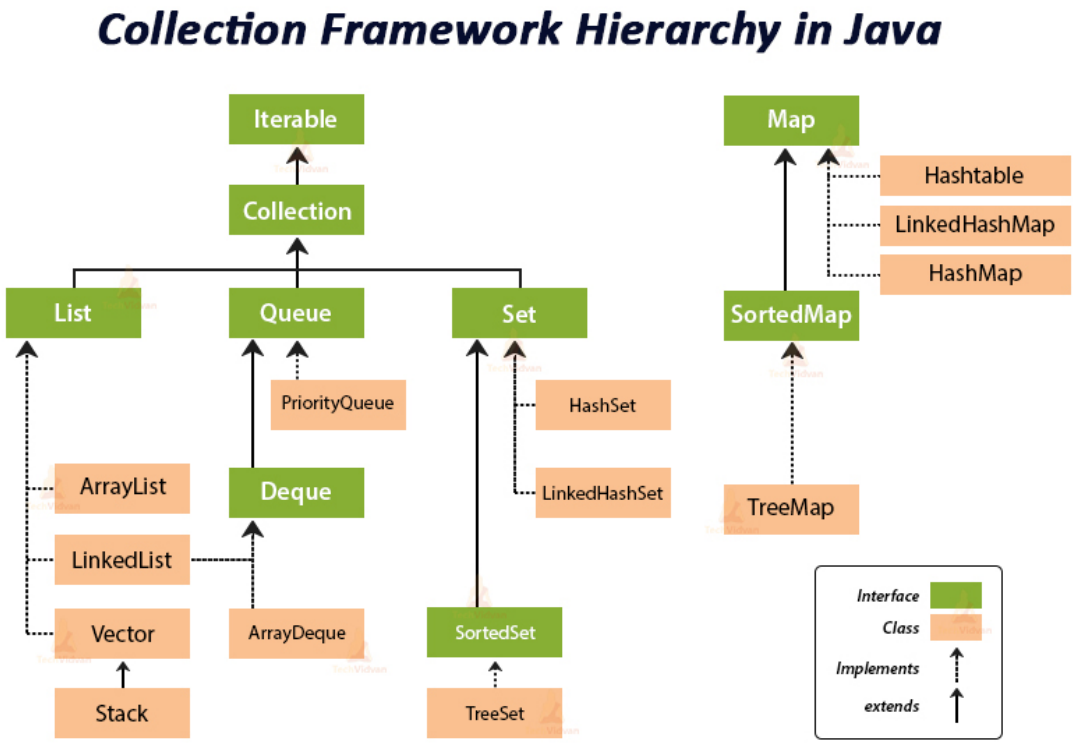
```
public class BoxExam {
  public static void main(String[] args) {
    Box box = new Box();
```

```
    box.setObj(new Object());
    Object obj = box.getObj();

    box.setObj("hello");
    String str = (String)box.getObj();
    System.out.println(str);

    box.setObj(1);
    int value = (int)box.getObj();
    System.out.println(value);
  }
}

public class Box {
  private Object obj;
  public void setObj(Object obj){
    this.obj = obj;
  }
  public Object getObj(){
    return obj;
  }
}
```

```
public class BoxExam {
  public static void main(String[] args) {
    Box<Object> box = new Box<>();
    box.setObj(new Object());
    Object obj = box.getObj();

    Box<String> box2 = new Box<>();
    box2.setObj("hello");
    String str = box2.getObj();
    System.out.println(str);

    Box<Integer> box3 = new Box<>();
    box3.setObj(1);
    int value = (int)box3.getObj();
    System.out.println(value);
  }
}

public class Box<E> {
  private E obj;
  public void setObj(E obj){
    this.obj = obj;
  }
  public E getObj(){
    return obj;
  }
}
```

▼ 컬렉션 프레임워크(Collection Framework)

다수의 요소를 하나의 그룹으로 묶어 효율적으로 저장하고, 관리할 수 있는 기능을 제공




### Collection 인터페이스

- 중복 허용
- 자료가 저장된 순서를 기억하지 못하기 때문에 저장된 자료를 하나씩 꺼낼 수 있는 Iterator 인터페이스를 반환함
- `add()`, `size()` , `iterator()`

### Iterator 인터페이스

- `hasNext()`, `next()`

### Set 인터페이스

- 중복을 허용하지 않음
- Collection 인터페이스를 상속받음

- `add()` : 같은 자료가 있으면 `false`, 없으면 `true` 반환

### List 인터페이스

- 중복 허용
- 자료가 저장된 순서 기억
- Collection 인터페이스를 상속받음
- `get(n)` : `n`번째 자료 반환

### Map 인터페이스

- key와 value를 가짐
- key 값은 중복될 수 없음
- `put()`, `get()`, `keySet()`

▼ `Set`

### Set 클래스

- `HashSet` : 데이터를 중복 저장할 수 없으며, 순서를 보장하지 않는 Set의 대표 클래스
- `TreeSet` : HashSet 특성을 가지며, 오름차순으로 데이터 정렬하는 클래스
- `LinkedHashSet` : 데이터를 중복 저장할 수 없으며, 입력 순서대로 데이터를 저장하는 클래스

### Set 메소드

- `add()` : 객체(데이터) 추가
- `iterator()` : 검색을 위한 반복자 생성
- `size()` : 저장된 객체(데이터) 수 반환
- `clear()` : 저장된 객체(데이터)를 모두 삭제
- `remove()` : 해당 객체(데이터) 삭제

```
import java.util.HashSet;
import java.util.Iterator;
```

```
import java.util.Set;

public class SetExam {
  public static void main(String[] args) {
    Set<String> set = new HashSet<>();

    boolean flag1 = set.add("kim");
    boolean flag2 = set.add("lee");
    boolean flag3 = set.add("kim");

    System.out.println(set.size());  // 2
    System.out.println(flag1);       // true
    System.out.println(flag2);       // true
    System.out.println(flag3);       // false

    Iterator<String> iter = set.iterator();

    while (iter.hasNext()) {
      String str = iter.next();
      System.out.println(str);
    }
  }
}
```

▼ `List`

### List 클래스

- `ArrayList` : 데이터를 순차적으로 추가하여 인덱스로 접근이 가능한 List의 대표 클래스로, 배열과 유사하지만 배열과 달리 크기를 동적으로 늘릴 수 있음
- `LinkedList` : 인덱스를 가지고 있지 않으며, 다음 리스트와 이전 리스트의 주소들이 노드로 연결되어 있는 클래스

### List 메소드

- `add()` : 객체(데이터) 추가
- `add(index, value)` : 해당 `index`에 `value` 추가
- `set(index, value)` : 해당 `index`에 `value` 대체
- `get(index)` : 해당 `index`의 값 반환
- `isEmpty()` : 데이터의 유무 반환
- `iterator()` : 검색을 위한 반복자 생성
- `size()` : 저장된 객체(데이터) 수 반환

- `clear()` : 저장된 객체(데이터)를 모두 삭제
- `remove()` : 해당 객체(데이터) 삭제
- `remove(index)` : 해당 `index`의 객체(데이터) 삭제
- `contains(value)` : List에 `value` 포함 유무 반환

```
import java.util.ArrayList;
import java.util.List;

public class ListExam {
  public static void main(String[] args) {
    List<String> list = new ArrayList<>();

    list.add("kim");
    list.add("lee");
    list.add("kim");

    System.out.println(list.size()); // 3
    for(int i = 0; i < list.size(); i++){
      String str = list.get(i);
      System.out.println(str);
    }
  }
}
```

▼ `Map`

### Map 클래스

- `HastMap` : key와 value를 묶어 하나의 entry로 저장하는 Map의 대표 클래스
- `TreeMap` : key와 value를 한 쌍으로 하는 데이터를 이진 검색 트리 형태로 저장하는 클래스

### Map 메소드

- `put(key, value)` : `key`와 `value` 추가
- `get(key)` : 해당 `key`의 값 반환
- `isEmpty()` : 데이터의 유무 반환
- `size()` : 해당 `key`의 개수 반환
- `clear()` : 저장된 모든 `key`와 `value` 삭제

- `keySet()` : 저장된 모든 `key` 반환
- `entrySet()` : 저장된 모든 `key`와 `value` 반환
- `remove(key)` : 해당 `key`의 값 삭제
- `containsKey(key)` : 해당 `key` 지정 여부 반환
- `containsValue(value)` : 해당 `value` 지정 여부 반환

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class MapExam {
  public static void main(String[] args) {
    Map<String, String> map = new HashMap<>();

    map.put("001", "kim");
    map.put("002", "lee");
    map.put("003", "choi");
    map.put("001", "kang");

    System.out.println(map.size());  // 3

    System.out.println(map.get("001"));
    System.out.println(map.get("002"));
    System.out.println(map.get("003"));

    Set<String> keys = map.keySet();
    Iterator<String> iter = keys.iterator();
    while (iter.hasNext()) {
      String key = iter.next();
      String value = map.get(key);
      System.out.println(key + " : " + value);
    }
  }
}
```