## **JAVA**

| package j01\_출력; // 패키지명 (무조건 첫줄선언)  import java.util.Scanner; // 입력을 받으려면 필요함  public class TEST1\_1 (클래스명) {  public static void main(String[] args) { //시작할때 쓰는 문법?  내용  }  } |
| --- |

#### 

#### 

### **- [ 스프링툴 단축키 ]**

| **단축키** | **뜻** | |
| --- | --- | --- |
| sysout + space | System.out.println(); | |
| Ctrl + space | 자동완성 | |
| Ctrl + alt + ↓ | 카피 | |
| Alt + Shift + F | 자동정렬 | |
| Alt+Shift+S  > Generate Constructor using Fields | 생성자 자동생성 | |
| Alt+Shift+S  > Generate Getters and Setters | private 값 getter & setter | |
| Ctil + Shift + O | import 단축키 | |

### **- [ 문법 ]**

**클린코드** 참고하면 좋다.(자바가 주언어)

리팩토링2 참고해도 좋다. (자바가 주언어)

| **특수코드값** | **실제표현** | **비고** |
| --- | --- | --- |
| // | 주석 | 한줄 주석 |
| /\*  \*/ | 블록 주석 | /\* 부터 \*/ 까지 전부 주석 처리 됨 |
| Math.abs(); |  | 정수, 실수의 절대값을 구할 수 있다. |
| Math.min(a,b); | 최소값 | a 와 b 중 최소값 |
| Math.max(a,b); | 최대값 | a 와 b 중 최대값 |

#### 

## **출력**

| System.out.print(출력할데이터); > 줄바꿈이 없습니다.  System.out.println(출력할데이터); > 줄바꿈이 있습니다. |
| --- |

| **예제** | **뜻** | |
| --- | --- | --- |
| System.out.print(7); | print() 메소드는 줄 바꿈을 하지 않음. | |
| scan.nextLine() | 줄 단위로 받는 것 같다. 공백 포함 여지가 있을때는 이걸 쓰자 | |
| System.out.println(3); | 정수 출력 | |
| System.out.println(3.14); | 실수 출력 | |
| System.out.println("자바!"); | 문자열 출력  System.out.println("문자열끼리의 " + "연결도 가능.");  System.out.println("숫자" + 3 + "과 문자열의 연결도 가능."); | |

## **변수와 상수**

| **자료형** | **키워드** | **크기** | **기본값** | **표현범위** |
| --- | --- | --- | --- | --- |
| 논리형 | boolean | 1bit | false | true 또는 false(0과 1이 아니다) |
| 문자형 | char | 2byte | \u0000 (16진수) | 0~65,535 |
| 문자열 | String |  |  |  |
| 정수형 | byte | 1byte | 0 | -128 ~ 127 |
| sort | 2byte | 0 | -32,768 ~32,767 |
| int | 4byte | 0 | -2,147,483,648 ~ 2,147,483,648 |
| long | 8byte | 0 | -9,223,372,036,854,775,808  ~ 9,223,372,036,854,775,808 |
| 실수형 | float | 4byte | 0.0 | -3.4E38 ~ +3.4E38 |
| double | 8byte | 0.0 | -1.7E308 ~ +1.7E308 |

### **- [ 상수 ]**

| final | 상수, 변하며 안되는 기준값 (final int MONTH = 12;)  상수는 전부 대문자로 사용 + 스네이크 표기법 |
| --- | --- |

### **- [ 캐스팅 ]**

타입을 변환하는 것. 형 변환하기라고도 한다.

어떠한 자료형/클래스의 변수/객체를 만든다는 것은, 좌변의 자료형이 요구하는 정보를 모두 우변이 가졌을 때 가능하다.

캐스팅은 형변환을 의미하는데, 좌변의 자료형에 맞게 우변의 형을 변형하는 것이다.

좌변에 필요한 정보는 우변에 충분히 있어야한다.

| int num = 1; | int형에 해당하는 변수를 만들고자 한다는 것이 좌변에 주어졌으면, 우변에는 int형을 만들기에 충분한 정보인 1이 주어지면 옳은 변수 생성이 되는 것이다. |
| --- | --- |
| int num1 = 2.0;  // 에러 발생!  int num2 = (int) 2.0;  // 에러 해결! | 이 식의 경우, 컴파일 에러가 발생한다. 컴파일러가 알고 있는 int라는 기본형이긴 하지만, 우변의 데이터가 손실되는 것을 막기 위해 에러를 표시하는 것이다.  따라서 아래와 같이 형을 명시하여 캐스팅(강제형변환)을 하면 에러가 나지 않는다. 개발자가 그를 의도했다는 것이 분명하기 때문이다. |
| double num1 = 1; | 해당 식도 에러 없는 문장인데, 그 이유는 손실 될 데이터도 없고 기본형끼리의 캐스팅이기 때문에 JVM에서 알아서 처리해주기 때문이다. |

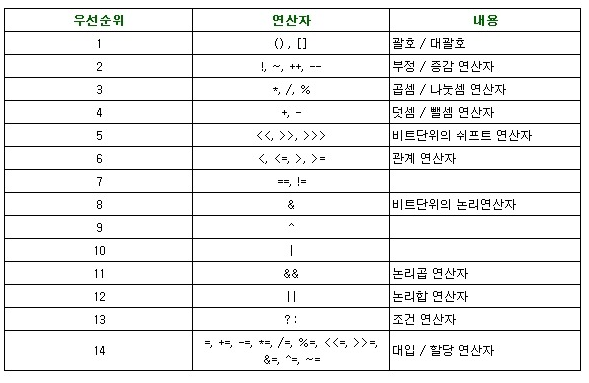
변에 정보가 불충분한 경우는 에러가 뜨기 마련이다.

업캐스팅 & 다운 캐스팅 : 물컵을 예시로 생각하면 이해하기 쉽다!

| 업캐스팅 | 클래스의 상속관계가 아래과 같은 상황에서  Parent  |  Child  Parent형의 객체를 생성하고자 할 때, Child형의 정보를 좌변에 제공하는 것이다.  Parent p = new Child();  상위클래스인 좌변이 요구하는 정보는 그의 하위클래스인 우변은 당연히 전부 가지고있기 때문에 위 문장은 옳은 문장이 된다.  말하자면 업캐스팅이란, **하위 클래스의 정보를 담을 수 있는 객체에 상위클래스의 자료형을 부여**해서, 상위클래스처럼 사용하게 하는 것이다. |
| --- | --- |
| 다운캐스팅 | 하위클래스(Child)의 정보를 담을 수 있는 객체의 자료형이 상위클래스 Parent)로 전환되어 있던 것(업캐스팅된 객체)을 다시 되돌리는 것을 의미한다. 업캐스팅 되었던 객체의 자료형을 다시 하위클래스의 정보를 담는 기능을 하도록 자료형을 Child로 바꾸어서 되돌려 놓는 것을 말한다.  Parent p = new Child(); //업캐스팅 -p는 Parent형.  Child c = (Child) p; //다운캐스팅! -p는 Child형.  원칙적으로 **다운캐스팅 혼자만 쓰면 우변이 좌변에서 필요한 정보를 모두 채워주지 못하기 때문에 불가능한 문장이 되고, 꼭 업캐스팅이 선행되어야 한다는 것을 명심**해야한다. 아래는 에러가 나는 코드이다.  Parent p = new Parent();  Child c = (Child) p;//에러!! |

| public static void main(String[] args) {  char a = '가';  int b = a;  double c = b;    System.out.println(b);  System.out.println(c);    int d = (int) c;  // double 자료형인 c를 int로 바꾸겠다고 명시하는 (int)  // 업캐스팅할때는 명시X 다운캐스팅할때는 명시해야함!  char e = (char) d;    System.out.println(d);  System.out.println(e);  //char>int 가능 char>double 불가능  //캐스팅 순서  //byte > sort > int > long > float > double  } |
| --- |

## **연산자**



### **- [ 부정(논리)·증감 연산자 ]**

| **연산자** | **내용** | |
| --- | --- | --- |
| !  NOT연산(부정)  **논리연산자** | 부정 연산자. !a 는 a가 아니다 라는 뜻  !true => false  !false => true  !(true && false) => true. | |
| ++ | ++num // 앞에 붙으면 지금 바로 num에 1이 더해진다.  num++ // 뒤에 붙으면 다음번에 num이 사용될 때 1이 더해진다.  num = 50  System.out.println(++num);  System.out.println(num);  System.out.println(num++); System.out.println(num);  를 연달아 출력해보면 차이를 알 수 있다. | |
| - - | 위와 동일 | |

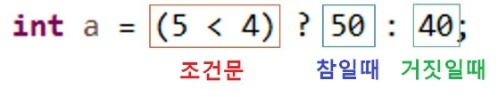
### **- [ 논리 연산자 ]**

| **연산자** | **내용 (우선순위: 논리곱 -> 논리합)** | |
| --- | --- | --- |
| AND연산 (곱) | && | true && true => true (1\*1=1)  true && false => false (1\*0=0)  false && false => false (0\*0=0) |
| OR연산(합) | || | true || true => true (1+1=1)  true || false => true (1+0=1)  false || false => false (0+0=0) |

### **- [ 조건(삼항) 연산자 ]**

#### 조건부 삼항 연산자는 세 개의 피연산자를 취할 수 있는 유일한 연산자.

#### 보통 if 명령문의 단축 형태



| 조건식? 식1:식2  조건식이 참이면 식1이 실행되고, 조건식이 거짓이면 식2가 실행된다.  result = score < 0 || score > 100 ? "계산불가"  : score > 89 ? "A학점"  : score > 79 ? "B학점"  : score > 69 ? "C학점"  : score > 59 ? "D학점"  : "F학점"; |
| --- |

### **- [ 대입 연산자 ]**

| **연산자** | **내용 (우선순위: 곱셈·나눗셈·나머지 -> 덧셈뺄셈)** | |
| --- | --- | --- |
| \*= | a\*=10 | a = a\*10 |
| %= | a%=10 | a = a / 10 의 나머지 |
| += | a+=10 | a = a+10 |

## **문자열**

### **- [ 추출 ]**

| " " + 문자열의 값은 문자열이 된다. |
| --- |

| **특수코드값** | **실제표현** | **비고** |
| --- | --- | --- |
| indexOf | int index = address.indexOf("동");  //처음부터 동을 찾아라  int index = address.indexOf("동", 1);  //1 이후부터 동을 찾아라 | 문자열의 처음에서 해당 문자열이 처음 나오는 값을 찾아 위치를 알려준다.  해당 문자열을 찾지 못하면 -1 출력된다 |
| lastIndexOf | int index = address.lastIndexOf("동"); | 문자열에서 해당 문자열이 마지막으로 나오는 값을 찾아 위치를 알려준다. |
| subString | String temp = address.substring(1, 5);  // 1부터 5전까지 = 0 ~ 4 | 해당 문자열에서 범위를 지정하여 문자열을 잘라줌  문자열로 반환 |
| replaceAll | String temp = address.replaceAll("-"," "); | 해당 문자열을 찾아서 다른 문자열로 바꿔주는 기능 |
| split | String str = "He-llo Wor@ld";  String[] ArraysStr = str.split(" ");  String[] ArraysStr = str.split(" |@|-"); | 구분자를 지정해서 문자열을 잘라 배열로 반환  " " 을 기준으로 단어를 잘라서  ArrayStr 배열에 하나씩 넣어준다.  이런 식으로 공백 @ - 전부를 기준으로 자를수 있다 |
| toLowerCase  toUpperCase |  | 전부 소문자 또는 대문자로 변환 |
| trim |  | 문자열의 양쪽 끝 공백을 제거해준다. |
| contains |  | 문자열에 해당 문자열이 포함되어 있는지 true/false |

### **- [ StringBuilder ]**

많은 문자열, String객체끼리 연결하면 많은 중간 문자열 객체가 생성되어 비효율적인 코드가 생성된다. ([관련자료](https://www.codejava.net/java-core/the-java-language/why-use-stringbuffer-and-stringbuilder-in-java))

자바에서 String 객체는 변경 불가능하기 때문이다. 한 번 생성되면 내용을 바꿀 수 없단 뜻이다. 따라서 하나의 문자열을 다른 문자열과 연결하면 새 문자열이 생성되고, 이전 문자열은 가비지 컬렉터로 들어간다.

이처럼 Stirng은 변경 불가능한 문자열을 생성하지만 StringBuilder는 변경 가능한 문자열을 만들어 주기 때문에, String을 합치는 작업 시 하나의 대안이 될 수 있다.

| **특수코드값** | **실제표현** | **비고** |
| --- | --- | --- |
| append | .add("내용") | 문자열 추가 |
| delete | .delete(인덱스 시작점, 인덱스 끝점+1) | 매개변수로 전달받은 인덱스 사이의 문자열 제거 |
| insert | .insert(2, "추가할 내용") | 특정 위치에 문자열 삽입 |
| toString | string에 담거나 출력할 때 사용한다. | String Builder로 처리한 문자열을 String으로 변환하기 위해서는 .toString()을 이용한다 |

| StringBuilder stringBuilder = new StringBuilder();  stringBuilder.append("문자열 ").append("연결");  //String str = stringBuilder;  // String에 StringBuilder를 그대로 넣을 순 없다. toString()을 붙여야 한다  String str = stringBuilder.toString();  // 두 println()은 같은 값을 출력한다  System.out.println(stringBuilder);  System.out.println(str); |
| --- |
| StringBuilder stringBuilder = new StringBuilder();  ArrayList<String> list = new ArrayList<>();  list.add("첫 번째, ");  list.add("두 번째, ");  list.add("세 번째, ");  list.add("네 번째, ");  list.add("다섯 번째");  for (int i = 0; i < list.size(); i++)  {  stringBuilder.append(list.get(i));  }  System.out.println(stringBuilder);  // >> 첫 번째, 두 번째, 세 번째, 네 번째, 다섯 번째 |

### **- [ 이스케이프 문자 ]**

제어문자들과 출력되지 않는 문자들을 가리킨다. 키보드로 표현할 수 없는 문자들이다.

보통 역슬래시(\)와 문자와 결합하여 사용한다.

| **이스케이프 문자** | **의미** | |
| --- | --- | --- |
| \\ | \ 출력 (back slash)  -> 폴더 경로 구분시 잘 사용됨 | |
| \n | 개행 (new Line)  -> 다음 줄로 이동됨 | |
| \t | 수평 탭(tab)  -> 탭 키만큼 수평으로 띄움 | |
| \' | ' 출력 (작은 따옴표) | |
| \" | " 출력 (큰 따옴표) | |

#### 

## **입력**

#### import java.util.Scanner; 자동완성 사용 권장

| package j05\_입력;  import java.util.Scanner;  public class Input1 {  public static void main(String[] args) {    Scanner scanner = new Scanner(System.in);  //Scan + ctrl + space  System.out.print("내용을 입력하세요: ");  String str = scanner.next(); // 문자열 입력  System.out.println("출력: " + str);  }  } |
| --- |

| Scanner를 사용하기 위해서 import를 통해 외부 클래스를 호출해야 한다.  public class 윗 줄에 사용하면 되는 듯.  **scanner.next(); 로 입력을 받음** |
| --- |

| **명칭** | **뜻** | |
| --- | --- | --- |
| x.next(); | 공백 문자 기준으로 받음. 공백 이전까지만. 띄워쓰기 줄바꿈 불포함 | |
| x.nextLine() | 줄 단위로 받는 것 같다. 공백 포함 여지가 있을때는 이걸 쓰자 | |
| x.nextInt() | 정수를 받을 때 사용 | |

### **- [ 버퍼 ]**

버퍼란 임시 저장 공간을 의미한다. A와 B가 서로 입출력을 수행하는데에 있어서 속도차이를 극복하기 위해 사용하는 임시 저장 공간이다.

nextInt()와 next() 사용 후에는 nextLine()을 넣어 주는 걸 버퍼 비운다는 표현을 쓴다.

|  |  |
| --- | --- |

nextInt() 는 입력값으로 들어온 값 중에 Enter나 공백을 기준, 그 앞의 Int형 값을 가져온다.  
위 코드의 경우 정수를 가져왔으며, 버퍼에 남아있는 값은 Enter입니다.

nextLine() 은 버퍼에 남은 것들을 문자열로 가져와버립니다.

Enter 값을 공백 문자열로 가져와버리고 끝나기 때문에 다음 단계(연락처)로 넘어간다.

위 문제를 해결하기 위해서는

|  |  |
| --- | --- |

nextInt() 아래에 nextLine()을 추가하여 버퍼를 털어내면 된다.

## **제어문**

조건문, 반복문

결과를 도출하기 위해 논리적 판단을 하는 역할.

논리적 판단에는 **조건을 비교**하거나 **일정 구간을 반복**시켜야 함.

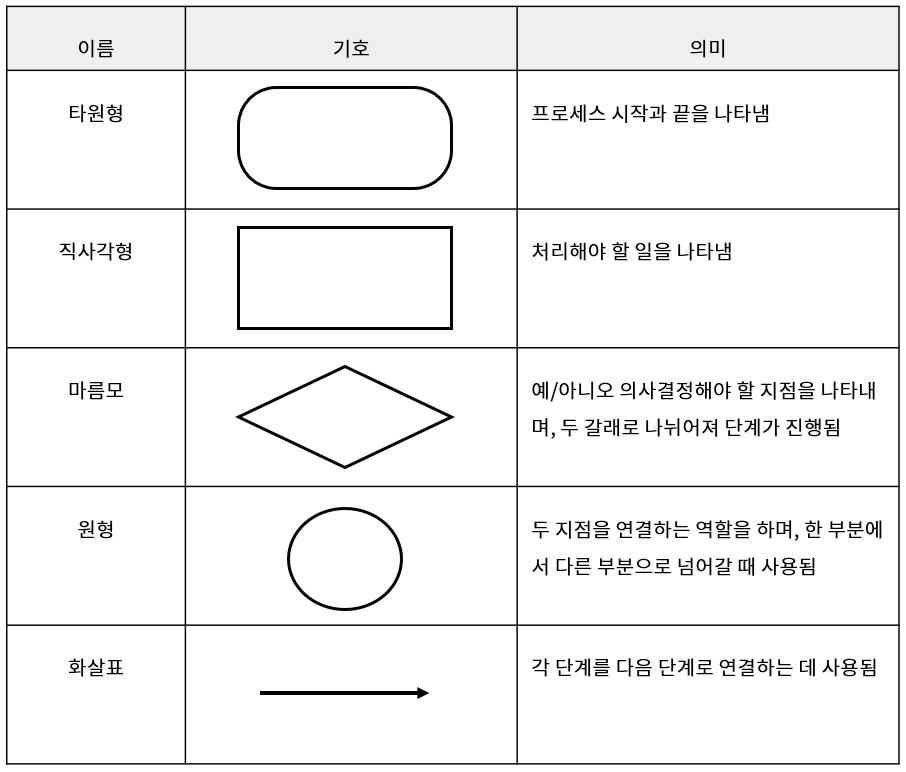
| if / switch / for / while / do~ while |
| --- |

### **- [ 흐름도 ]**

흐름도: https://app.diagrams.net/

프로그램에서 수행되어지는 흐름을 도식화하여 문서화 한 것

* 프로그래밍의 생산성 부분과 직결됨
* 다른 사람이 프로그램의 전체 흐름을 쉽게 파악할 수 있다
* 흐름도를 작성한 후 소스코드를 작성하면 실수를 줄일 수 있다



### **- [ 랜덤 ]**

random 메소드

| **Math** | | |
| --- | --- | --- |
| int i = 0;  while(i < 10) {  double rNum = Math.random();  System.out.println((int)(rNum \* 10));  i++;  } | 0 ~ 1 사이의 숫자를 실수 형태로 랜덤하게 값을 만들어 줌  \*10을 하고 int로 형변환 하면  1~10까지 숫자 추출 가능 | |

| **nextInt** | | |
| --- | --- | --- |
| ( ) 안의 숫자 까지의 정수를 랜덤하게 추출해낸다. | | |
| import java.util.Random;  public class Random3 {  public static void main(String[] args) {  Random random = new Random(); //변수 선언  int i = 0;  int a = 0;  int b = 0;  int c = 0;  int max = 0;    while(i < 20) { //20회 돌리겠다.  a = random.nextInt(50); // 1~50까지 랜덤 숫자 추출  b = random.nextInt(50); // 위와 같다  c = random.nextInt(50); // 위와 같다  System.out.println("a: " + a + " b: " + b + " c: " + c); // 각 a b c 값 확인    if(a > b && max<a) { // max는 이전 회차의 a b c 중 최대값  max = a; // 이전 최대값과 b보다 a 가 더 크다면 a = max  }else if(max < b) { // max가 b보다 작다면 (b는 a보다 큼!)  max = b; // b = max  } // 둘 다 아니라면 a 와 b 보다 전회차 max가 더 크다  // 여기까지 왔다면 a b 전회차max 의 최대값은 max이다.  if(max < c) {  max = c;  } // 현회차 max와 c를 비교하여 더 큰 값을 현 max로 지정한다.    System.out.println("max: " + max); // 현회차까지의 max를 알려준다  i++; // 다음 회차  }    System.out.println("최대값: " + max );  }  } | | |

| **UUID** | | |
| --- | --- | --- |
| 어떤 개체(데이터)를 고유하게 식별하는 데 사용되는 16바이트(128비트) 길이의 숫자. 이 숫자는 32개의 16진수로 구성되며, 5개의 그룹으로 표시되고 각 그룹은 하이픈으로 구분된다 | | |
| public static void main(String[] args) {  String uuid = UUID.randomUUID().toString();  System.out.println(uuid);  } | 300a5eea-ab3d-4de2-8d2f-c85fcf15fd19  출력된다. | |

### **- [ 조건문 ]**

#### **if 문**

if(조건식), if(조건식) else, if(조건식) else, if(조건식)

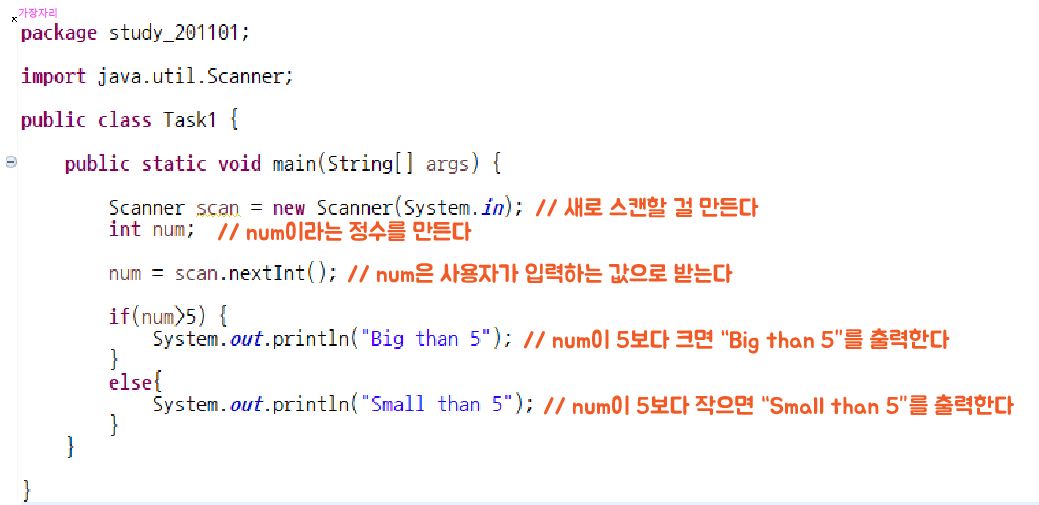
만약 조건식이 참이면 이것을 실행하세요.

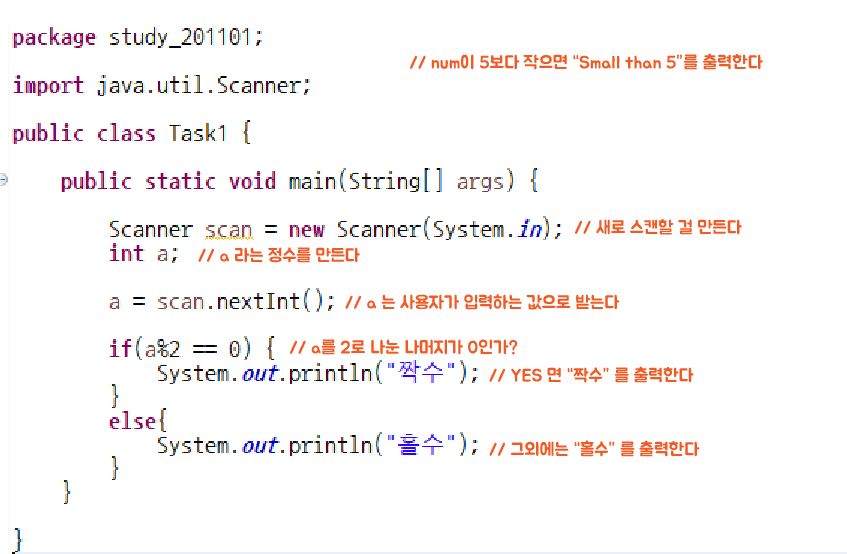
| if(조건식) -> 만약 조건식이 참(true) 이면 {} 안에있는 실행문을 실행  else if(조건식) -> 첫번째 if(조건식)이 거짓(false)이고 지금 else if(조건식) 이 참(true) 이면 {} 안에있는 실행문을 실행 (생략 가능)  else -> 위 조건식들이 모두 거짓(false) 이면 실행. (생략 가능) |
| --- |

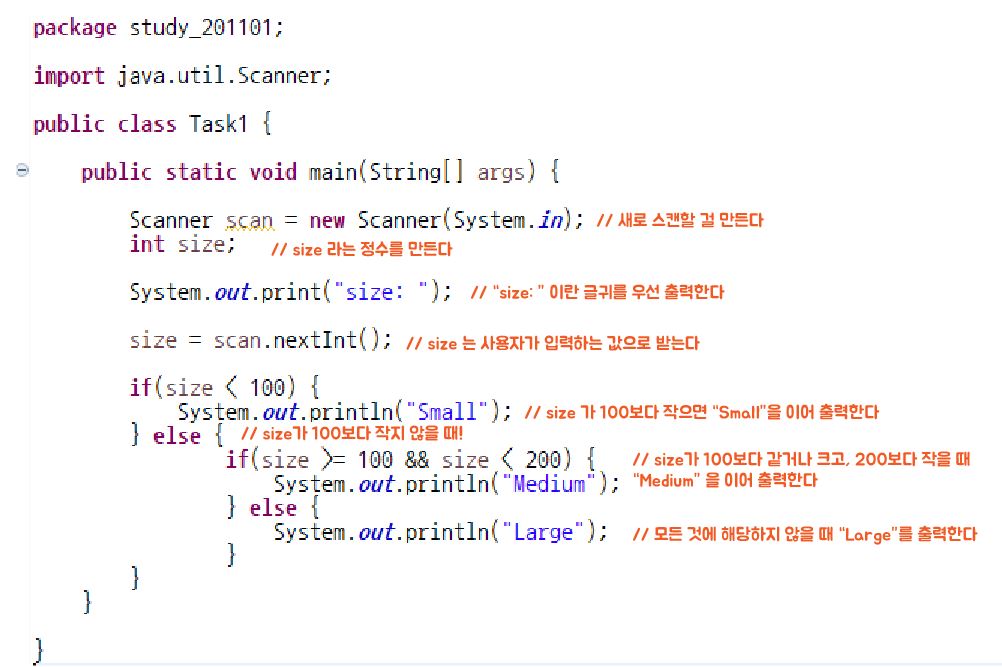
| **실제 표현** | **예문** | **비고** |
| --- | --- | --- |
| if(조건식){  System.out.println("**YES**")  } | 조건식이 True일 경우  **YES**가 출력된다 | 거짓일 경우 실행되지 않음 |

| **실제 표현** | **예문** | **비고** |
| --- | --- | --- |
| if(조건식){  System.out.println("**YES**")  } else {  System.out.println("**NO**")  } | 조건식이 True일 경우  **YES**가 출력된다  아닐 경우  NO가 출력된다 | 거짓일 경우 NO |

| **실제 표현** | **예문** | **비고** |
| --- | --- | --- |
| if(조건식1){  System.out.println("**YES**")  } else if(조건식2){  System.out.println("**NO**")  } else {  System.out.println("**몰랑**")  } | 조건식1이 True일 경우  **YES**가 출력된다  조건식2가 True일 경우 **NO**가 출력된다  아닐 경우  **몰랑**가 출력된다 | 조건식 둘 다 거짓일 경우 몰랑 |







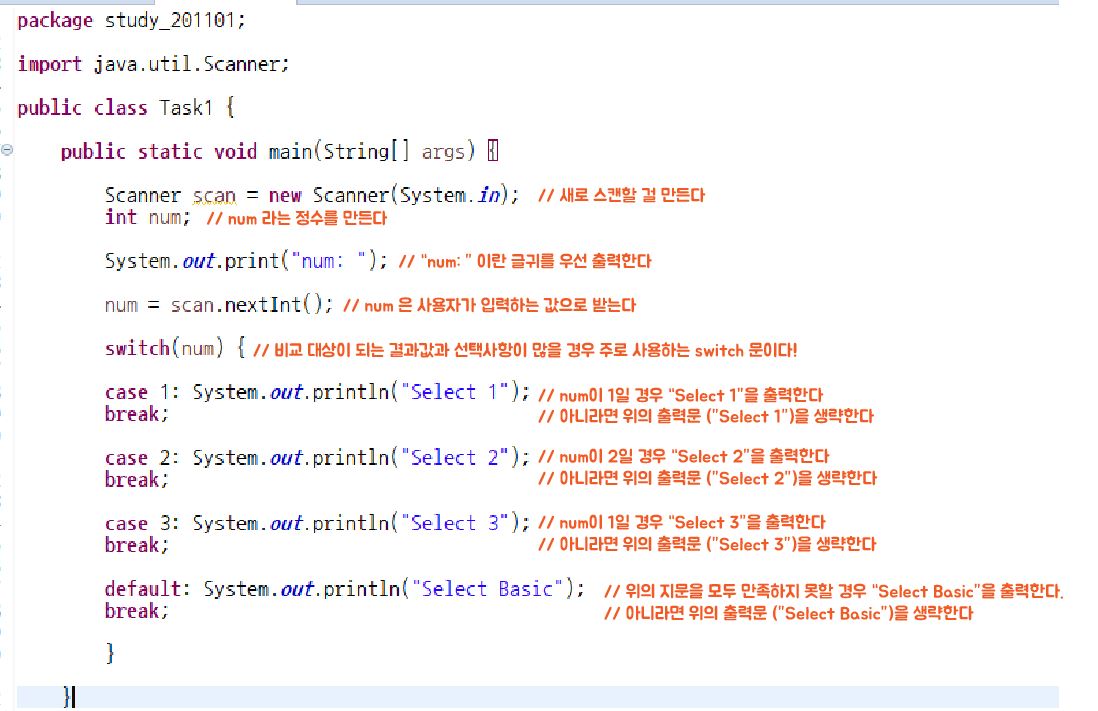
#### **switch 문**

다자택일

| 조건변수에 특정 값을 가진 변수가 들어감.  int count = 0; 이라는 변수가 있고 이 변수의 값에 따라 어떤 작업을 해야한다면,  switch(count) 로 작성하며 count가 어떤 값인지 case 문으로 비교, 확인.  case는 각각의 케이스를 말하는 것이며 '여기를 실행하세요' 라고 명시하는것으로 이해. (case1 : count 변수가 0입니까 ? case2 : count 변수가 1입니까? 등...)  break는 생략 가능하며 말그대로 '여기서 그만 실행하겠다'는 뜻.  break를 만나면 switch를 그만 실행(종료)하게 됨.  생략하게 되면 종료되지 않고 다음 case 로 이동하여 비교, 확인하며 switch문을 계속 실행,  default는 기본 값.  특정 case를 만족하지 못하면 실행문을 실행하지 못하고 break문도 못만났겠죠. 그때 마지막까지 내려와서 기본값을 실행하게 됩니다. case문 안에 break문이 없으면 조건에 부합하는지 여부를 따지지 않고 이어지는 case문을 실행합니다.  if 문의 else 구문과 비슷한 편! |
| --- |

| **실제 표현** | **예문** | **비고** |
| --- | --- | --- |
| switch(조건변수){  case 값1 : 실행문 1;  …  break;  case 값2 : 실행문 2;  …  break;  case 값3 : 실행문 3;  .....  break;  default :  실행문n;  ...  } | int test = prompt("숫자를 입력하세요.","")    switch (test)  {  case "1" : alert("하나");  break;  case "2" : alert("둘");  break;  case "3" : alert("셋");  break; default : alert("몰랑");  } | prompt: 문자열 입력할 때 사용.  첫번째 매개변수는 입력 창에 띄울 메세지, 두번째 메시지는 입력 부분의 기본 값  if문은  A가 아니면 B,  그게 아니면 C 느낌.  switch문은  A랑 B랑 C는 각자 a랑 b랑 c에  찾아서 들어가라는 느낌 |

| int score = 100;  boolean breakFlag = true;    switch(score / 10) {  case 10:  case 9:  System.out.println("A");  if(breakFlag) {  break; // 제어문: 탈출  }  case 8:  System.out.println("B");  break;  case 7:  System.out.println("C");  break;  case 6:  System.out.println("D");  break;  default:  System.out.println("F");  } | 해당 예문의 경우,  100점도 A등급에 포함되어야 하기 때문에  의도적으로 case10에는  break를 넣지 않았다. |
| --- | --- |



### **- [ 반복문 ]**

| **실제 표현** | **예문** | |
| --- | --- | --- |
| continue | 아래 실행하지 말고 다음 반복을 하라 | |
| break | 반복을 그만두라 | |

#### **while 문**

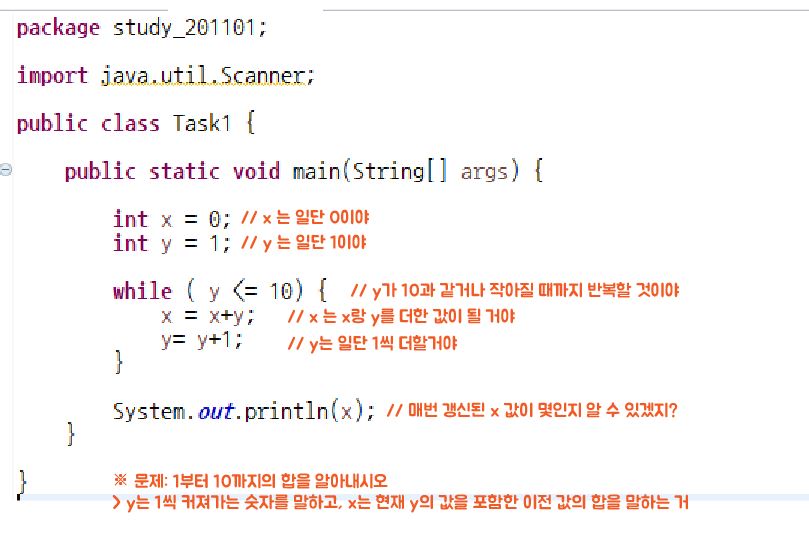
~ 하는 동안에. 조건식이 참(true)인 동안에 실행문을 반복.

for문처럼 초기값과 증감식은 없음.

while 안에 while. 중첩으로 사용이 가능하다.

| while |
| --- |

| **실제 표현** | | **비고** |
| --- | --- | --- |
| public static void main(String[] args) {  System.out.println("프로그램 시작");  int i = 0; //초기문  while(i < 5) { // 조건  System.out.println(i); //실행문  i++; //후처리  }  System.out.println("프로그램 종료");  } | | 프로그램 시작  0  1  2  3  4  프로그램 종료 |



| **" " 를 기준으로 문장 자르기** | | **출력 값** |
| --- | --- | --- |
| String address = "부산 동래구 사직동 중앙대로";  String a = null;  String b = null;  boolean startFlag = true;  while(true) {  int sliceIndex = address.indexOf(" ") + 1; // 가장 처음에 있는 " " 를 찾는다  int nextSliceIndex = address.indexOf(" ", sliceIndex);  // 두번째 있는 " " 를 찾는다    **if(nextSliceIndex == -1) {** // " " 가 더 이상 없을 때(마지막일때) 발생 EVENT  **b** = address.substring(sliceIndex);  // 마지막 단어를 변수 b에 담아서 출력한다  System.out.println(b);  break; // 반복중지  **}**  **if(startFlag) {** // 반복이 최초로 시작할때  **a** = address.substring(0, sliceIndex);  // 첫번째 단어를 변수 a에 담아서 출력한다.  System.out.println(a);  startFlag = false;  **}**    a = address.substring(sliceIndex, nextSliceIndex);  // 첫 " " ~ 두번째 " " 사이를 출력  System.out.println(a);    address = address.substring(nextSliceIndex);  // 이미 출력한 단어를 버리고 나머지만 남긴다.  } | | 부산  동래구  사직동  중앙대로 |

| **@ 뒤에 오는 문자열이 명령어**  **: 뒤에 오는 문자열은 내용** | | **출력 값** |
| --- | --- | --- |
| String data = "@Message:안녕하세요.@To:김준일@From:김준이@SendDate:2022/07/32";  while(true) {  int commandIndex = data.indexOf("@") + 1;  // commandIndex는 data에서 처음 나오는 @ 값 바로 뒤 인덱스 값이다.  int contentIndex = data.indexOf(":");  // contentIndex는 data에서 처음 나오는 : 인덱스 값이다.  int nextCommandIndex = data.indexOf("@" , 1);  // nextCommandIndex는 data에서 1 이후에 나오는 @의 인덱스 값이다.  String command = data.substring(commandIndex, contentIndex);  // command는 @ 부터 : 사이에 있는 단어다  // @뒤에오는 문자열 = 명령어  String content = nextCommandIndex != -1 ?  data.substring(contentIndex + 1, nextCommandIndex)  : data.substring(contentIndex + 1);  // content는 nextCommandIndex가 -1이 아니면 :와 @ 사이에 있는 단어이다.  // -1이라면 (@가 뒤에 더 없다면) 남은 남은 contentIndex 이후를 출력한다.  // : 뒤에 오는 문자열 = 내용  System.out.println(command + ": " + content);  // 명령어 : 문자열을 출력한다.    if(nextCommandIndex == -1) {  break; // 더 이상 @가 없다면 무한반복을 멈춘다  }    data = data.substring(nextCommandIndex);  // 이미 출력한 단어를 버리고 나머지만 남긴다.  } | | Message: 안녕하세요.  To: 김준일  From: 김준이  SendDate: 2022/07/32 |

#### **do while 문**

무조건 최초 1회는 실행하고 조건식을 검사.

= 조건식이 거짓(false)여도 실행문을 먼저 실행한 후 검사 (while 과 차이점)

| while |
| --- |

| **실제 표현** | **예문** | **비고** |
| --- | --- | --- |
| do{  실행문  ...  }while(조건식) | int a = 1;  while(a<=10){  document.write("안녕하세요(while)"  +" ");  a++;  }    do{  alert(a);  }  while(false) | 아래와 같이 작성하면 무한 반복문. break; 구문과 함께 작성.  while(true){  실행문  }  ▶ 안녕하세요 10줄 출력됨  + while이 실패할 때 alert(11)가 뜸 |

#### **for 문**

반복문입니다. 특정 조건만큼 변수를 증감하고 실행문을 반복.

| for문은 for문 내부에서 사용할 지역변수(숫자)를 선언하고 조건식이 참일때 계속 실행. 증감식은 초기화 변수나 다른 특정 변수를 증가(++ , +n)하거나 감소(--, -n) 또는 변수를 변화시킬 수식이 들어갈 수 있음.  for(int i = 0; i < 5; i++){  실행문  ...  }  i 라는 변수를 0으로 초기화 하고 i 가 5보다 작을 경우 실행.  실행문이 끝나면 i를 1씩 증가시키고 다시 반복 실행.  조건식이 거짓(false)가 되면 중지. |
| --- |

| **실제 표현** | **예문** | **비고** |
| --- | --- | --- |
| for(초기화 변수; 조건식; 증감식){  실행문  ...  } | int test = 0;  for ( int i = 0; i < 10; i++){  test = test + i;  }  System.out.println(i); | 증감식이  뒤에 있어야 나중에 실행 |

| **substring 활용해서 한 글자씩 재단하기** | | **출력 값** |
| --- | --- | --- |
| public static void main(String[] args) {  String str = "abcdefghijk";    for(int i = 0; i < str.length(); i++) {  String subStr = str.substring(i, i + 1);  System.out.println(subStr);  }  } | | a  b  c  d  e  f  g  h  i  j  k |

| **" " 를 기준으로 문장 자르기** | | **출력 값** |
| --- | --- | --- |
| String address = "부산 동래구 사직동 중앙대로";  for(int i = 0; i < address.length(); i++) { // address의 글자 개수만큼 돌려준다.  String subStr = address.substring(i, i + 1); // subStr에 자른 단어를 넣음  **if(subStr.equals(" ")) {** // 자른 단어가 공백(" ") 일 때 발생EVENT  int index = 0;  String temp = address.substring(0, i); // 공백 앞까지 단어를 다 넣는다    **if(temp.contains(" ")) {**  // temp에서 공백이 포함될때 발생EVENT2  index = temp.lastIndexOf(" ") + 1;  // temp에서 마지막 공백 위치를 찾아낸다  System.out.println(temp.substring(index, i));  // 마지막 공백 ~ 이번 i 까지 하면 그 사이의 단어가 추출됨  **}else {**  System.out.println(temp.substring(0, i));  // 이게 첫번째 단어(공백이 없다!)  **}**  **}**  **if(i == address.length() - 1) {** // i가 문자열의 갯수와 동일해질때  System.out.println(address.substring(address.lastIndexOf(" ") + 1));  // address의 마지막 공백(" ") 이후를 전부 출력하라  **}** | | 부산  동래구  사직동  중앙대로 |

#### 

#### **break / continue 문**

**break** : switch문과 반복문에서 사용.

반복문에서 이 명령을 만나면 반복문 다음 문장으로 빠져나감 (= 반복문 종료!)

**continue** : 반복문에서만 사용 가능.

반복문에서 이 명령을 만나면 반복문의 끝으로 이동합니다. (= 반복문 종료X !)

continue와 반복문 사이에 있는 실행문이 무시된다고 생각하면 됨.

| **continue 활용** | | **출력 값** |
| --- | --- | --- |
| for(int i = 0; i < 10; i++) {  if(i % 2 == 0) {  **continue;**  // 2로 나누어 나머지가 0이면 아래 무시하고 반복문의 끝으로 이동한다. (다음 반복회차 실행)  }  System.out.println("i: " + i);  } | | i: 1  i: 3  i: 5  i: 7  i: 9 |
| String address = "부산 동래구 사직동 중앙대로";  String tempAddress = null;  for(int i = 0; i < address.length(); i++) {  String subaddress = address.substring(i, i+1);  **if(subaddress.equals(" ")){**  continue;  // " " 이 나오는 구간은 아예 패스하고 다음 회차실행  **}**  **if(i == 0) {**  tempAddress = subaddress;  **} else {**  tempAddress += subaddress;  **}**  }  System.out.print(tempAddress); | | 부산동래구사직동중앙대로 |

## **메소드**

| 자바에서 클래스는 멤버(member)로 속성을 표현하는 필드(field)와 기능을 표현하는 메소드(method)를 가진다. 그중에서 메소드(method)란 어떠한 특정 작업을 수행하기 위한 명령문의 집합이라 할 수 있습니다. |
| --- |

| **class** Car {  **private** **int** currentSpeed;  **private** **int** accelerationTime;  ...  **①** **public** **void** **accelerate**(**int** *speed,* **int** *second*) { *// 선언부*  *// 구현부*  **System.**out**.**println(second **+** "초간 속도를 시속 " **+** speed **+** "(으)로 가속함!!");  }  …  } |
| --- |
| **public class Method01 {**  **public static void main(String[] *args*) {**  **Car myCar = new Car(); *// 객체 생성***  **myCar.accelerate(60, 3); *// 메소드 호출***  **}**  **}** |

위 예제의 ①번 라인에서는 accelerate() 메소드를 정의하고 있습니다.

| **실제 표현** | **예문** | **비고** |
| --- | --- | --- |
| 접근 제어자 | 해당 메소드에 접근할 수 있는 범위를 명시 | 이 메소드는 public 접근 제어자를 사용하여 선언되어 해당 객체를 사용하는 프로그램 어디에서나 직접 접근할 수 있습니다. |
| 반환 타입  (return type) | 메소드가 모든 작업을 마치고 반환하는 데이터의 타입을 명시 | 반환 타입에는 어떠한 값도 반환하지 않는다는 의미를 가진 void를 명시합니다. |
| 메소드 이름 | 메소드를 호출하기 위한 이름을 명시 | 그 다음으로 메소드의 이름을 명시하고, |
| 매개변수 목록  (parameters) | 메소드 호출 시 전달되는 인수의 값을 저장할 변수들을 명시 | 매개변수로 int형 변수인 speed와 second를 전달받습니다. |
| 구현부 | 메소드의 고유 기능을 수행하는 명령문의 집합 | 전달받은 매개변수를 가지고 메소드 구현부에서 고유한 작업을 수행할 수 있는 것입니다. |

### **- [ 종류 ]**

#### **반환X 메소드**

void가 포함된 식은 값(return)이 없다.

| **매개변수** | **예문** | **비고** |
| --- | --- | --- |
| X | public static **void** test1() { // void가 포함된거면 값이 없다!!!! return값이 없다!!!!  System.out.println("test1 메소드 호출");  } | test1();  test1 메소드 호출 |
| 1 | public static **void** test2(**int num**) { //정수밖에 못 받음. String num으로 고쳐야함.  System.out.println("num: " + **num**);  System.out.println("test2 메소드 호출");  } | test2(1234);  num: 1234  test2 메소드 호출 |
| 2 | public static **void** test3(**int num, int num2**) {  System.out.println("num: " + **num**);  System.out.println("num2: " + **num2**);  System.out.println("test3 메소드 호출");  } | test3(1111,2222);  num: 1111  num2: 2222  test3 메소드 호출 |

#### **반환O 메소드**

void가 포함되지 않은 식은 값(return)이 있다.

| **매개변수** | **예문** | **비고** |
| --- | --- | --- |
| X | public static **int test4**() {  System.out.println("test4 메소드 호출");  **return** 100;  } | int a = test4();  System.out.println(a);  System.out.println(test4() + "1");  test4 메소드 호출  100  test4 메소드 호출  1001 |
| O | public static **String test5**(**String name, int index**) {  System.out.println("name: " + **name**);  System.out.println("index: " + **index**);  **return** name + index;  } | String result = test5("김준일", 1);  System.out.println(result);  name: 김준일  index: 1  김준일1 |

#### **반환X에서 메소드 강제 탈출**

| public static **void** test6(String names){  **while(true) {**  int tokenIndex = names.indexOf(", ");  String name = tokenIndex != -1? names.substring(0, tokenIndex)  : names;  System.out.println(name);  **if(name.equals("김준삼")) {**  return; //void일때는 메소드를 탈출한다. void가 아닐때는 값을 리턴해줘야만한다.  // break; 반복만 멈춰서 아래 구문이 출력되므로 적절X.  **}** // 해당 if 구문이 없다면 김준사 까지 전부 출력되었을 것이다.  if(tokenIndex == -1) {  break;  }  names = names.substring(tokenIndex + 2);  **}**  System.out.println("메소드가 정상적으로 종료됐음.");  } | |
| --- | --- |
| String names = "김준일, 김준이, 김준삼, 김준사";  test6(names);  System.out.println("test6메소드 호출 후 출력"); | 김준일  김준이  김준삼  test6메소드 호출 후 출력 |

### **- [ 오버로딩 ]**

매개변수의 형태가 다르면 동일한 메소드명을 사용해서 메소드를 정의할 수 있다

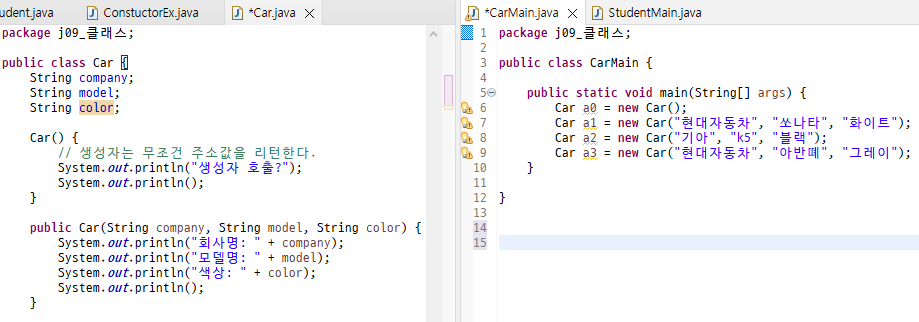
| public static void **test1()** {  System.out.println("**매개변수가 없는 test1**");  } |
| --- |
| public static void **test1(int num1)** {  System.out.println("**매개변수가 하나이고 int 자료형인 test1**");  } |
| public static void **test1**(**int num1, int num2**) {  System.out.println("**매개변수가 두개이고 둘다 int 자료형인 test1**");  } |
| public static void **test1**(**int num1, String num2**) {  System.out.println("**매개변수가 두개이고 첫번째 매개변수는 int 두번째 매개변수는 String인 test1**");  } |
| public static void **test1**(**String b, int a**) {  System.out.println("**매개변수가 두개이고 첫번째 매개변수는 String 두번째 매개변수는 int인 test1**");  } |

## **클래스**

클래스 안에는 null을 표기하지 않아도 된다. > 틀이라고 생각. 값이 X

클래스 값을 대입해야하면 (기본)생성자에서 값을 넣는게 더 좋다

생성자 or 메소드 안에는 null 등을 표기해야 한다. > 값을 초기화함



Car a1 = new Car(); // 생성(메모리 할당 = 주소를 받음)

클래스는 기본 뼈대이고

인스턴스(객체)는 구체적인 "상태(변수)" 와 "행위(기능)"의 집합이라고 생각하자.



위 카드를 생각해본다면

카드 클래스에 대표적으로 들어가야 할 것들이 있을 것이다.

### **- [ 생성자 ]**

생성자를 선언하는 방법은 아래와 같다. **클래스**라는 부분은 **생성자를 정의하는 클래스의 이름과 동일**하게 적어줘야 한다. 빨간색으로 표시한 부분은 필수로 적어야 하는 내용은 아니다. 생성자 앞에는 접근 제어자(public 등)만 올수 있다.

**반환값이 없으므로** void나 자료형을 작성할 수 없다.

| **public** **클래스**(**매개변수**) {  …  } |
| --- |

인스턴스를 생성할 때 반드시 생성자를 호출한다. 클래스를 정의할 때 생성자를 생략하면 컴파일러가 자동적으로 **기본 생성자(Default Constructor)**를 생성하여 준다.

\*\* 기본생성자 = NoArgsConstructor와 같지?

#### **기본 생성자**

| **NoArgsConstructor** | |
| --- | --- |
| public class ConstructorEx01 {  //public ConstructorEx01(){ } // Default Constructor 자동 생성    public static void main(String[] args) {  ConstructorEx01 ce = new ConstructorEx01(); // 인스턴스 생성 및 생성자 호출  }  } | |
| public class ConstructorEx01 {  public ConstructorEx01(){  System.out.println("생성자 호출 성공"); // 기본 생성자 활용!  }    public static void main(String[] args) {  ConstructorEx01 ce = new ConstructorEx01(); // 인스턴스 생성 및 생성자 호출  }  } | |

#### **필수 생성자**

| **RequiredArgsConstructor** | |
| --- | --- |
| public ConstuctorEx(String name) {  System.out.println("매개변수: " + name);  System.out.println("name을 매개변수로 받는 생성자");  **this**.name = name;  } | |
| public ConstuctorEx(int age) {  System.out.println("매개변수: " + age);  System.out.println("age을 매개변수로 받는 생성자");  } | |

#### **전체 생성자**

| AllArgsConstructor | |
| --- | --- |
| public ConstuctorEx(String name, int age) {  System.out.println("매개변수: " + name);  System.out.println("매개변수: " + age);  System.out.println("모든 변수를 받는 생성자");  } | |

#### **생성자 호출**

| Main 에서 호출해야 합니다. | |
| --- | --- |
| public class ConstructorEx01 {  //public ConstructorEx01(){ } // Default Constructor 자동 생성    **public static void main(String[] args) {**  ConstructorEx01 ce = new ConstructorEx01(); **// 인스턴스 생성 및 생성자 호출**  **}**  } | |

### **- [ 접근지정자 ]**

| **접근 지정자** | **접근 범위** | **동일**  **클래스** | **동일**  **패키지** | **다른**  **패키지의**  **자식**  **클래스** | **다른**  **패키지** |
| --- | --- | --- | --- | --- | --- |
| public | 접근 제한 없음 | O | O | O | O |
| protected | 동일 패키지와 상속 받은 클래스 내부 | O | O | O |  |
| default | 동일 패키지 내에서만 | O | O |  |  |
| private | 동일 클래스 내에서만 | O |  |  |  |

#### 

| **public** | |
| --- | --- |
| 공개 정도가 가장 높고 어디에서든 자유롭게 접근할 수 있다. | |

#### 

| **protected** | |
| --- | --- |
| 같은 패키지 내에서 접근이 가능하고 다른 패키지에서도 상속을 받은 클래스 내부에서는 사용 가능하다. public과 다른 점은 다른 패키지의 자식 클래스 외부에서는 접근할 수 없다는 것이다. | |

#### 

| **default (package private)** | |
| --- | --- |
| 같은 패키지 내에서만 접근이 가능하다. 아무런 접근 지정자도 없을 경우 이 옵션이 자동으로 적용된다. | |

#### 

| **private** | |
| --- | --- |
| 동일 클래스 내에서만 접근이 가능한 가장 낮은 단계의 자유도를 갖는다. | |

#### **private 값 주입**

private 변수에 값을 주입하는 방법 ( 단축키: Alt+Shift+S > Generate Getters and Setters )

#### 

| **생성자↓를 통한 값 주입** | |
| --- | --- |
| public Student(String name, int studentYear, int age, String addres, String phone) {  super();  this.name = name;  this.studentYear = studentYear;  this.age = age;  this.addres = addres;  this.phone = phone;  } | |

#### 

| **Setter↓를 통한 값 주입** | |
| --- | --- |
| public void setName(String name) {  this.name = name;  } | |

#### 

#### 

#### **private 값 가져오기**

private 변수의 값을 가져오는 방법

#### 

| **Getter↓를 통한 값 가져오기** | |
| --- | --- |
| public String getName() {  return name; // 이곳의 name을 외부로 전달해준다.  } | |

#### 

### **- [ 캡슐화 ]**

캡슐화는 관련이 있는 변수와 함수를 하나의 클래스로 묶고 외부에서 쉽게 접근하지 못하도록 은닉하는게 핵심이다. 객체에 직접적인 접근을 막고 외부에서 내부의 정보에 직접 접근하거나 변경할 수 없고, 객체가 제공하는 필드와 메소드를 통해서만 접근이 가능.

캡슐화의 가장큰 장점은 정보은닉(Information Hiding)에 있습니다.

## **배열**

동일한 자료형(Data Type)의 데이터를 연속된 공간에 저장하기 위한 자료구조이다. 즉, 연관된 데이터를 그룹화하여 묶어준다고 생각하면 된다.

| 자료형[] 변수 = {데이터1, 데이터2, 데이터3, ... };  int[] numbers2 = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  int[] numbers3 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; |
| --- |
|  |
| **배열 안의 값 확인**  for(int i = 0; i<nums.length; i++) {  System.out.println(numbers[i]);  } |

| **정의** | **예시** | |
| --- | --- | --- |
| 정수 | int[] numbers = new int[5]; | |
| 문자열 | String[] strArray = new String[5]; | |
| 객체 | Scanner[] scanArray = new Scanner[5]; | |

#### 

#### 

#### 

### **- [ 복사 ]**

#### **얕은 복사**

| **int[] numbers = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };** // 조건  int[] numbers2 = numbers; 를 하면,// numbers2라는 새 배열 생성  **int[] numbers2 = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };** // 생성 결과 값  위의 두 식의 값 **주소는 동일**하므로  **numbers2[0] = 100;** 를 한다면,  **int[] numbers = { 100, 9, 2, 8, 3, 7, 4, 6, 5 };**  **int[] numbers2 = { 100, 9, 2, 8, 3, 7, 4, 6, 5 };**  이렇게 둘 다 [0]번째 인덱스가 바뀐다. 이걸 **얕은 복사**라고 부른다. |
| --- |

#### **깊은 복사**

| **int[] numbers = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };** // 조건  **int[] numbers3 = new int[numbers.length];** // numbers와 크기가 동일한 새 배열생성  for(int i = 0; i<numbers.length; i++) {  numbers3[i] = numbers[i];  }  위의 두 식의 값 **주소가 다르**므로  **numbers3[0] = 100;** 를 한다면,  **int[] numbers = { 1, 9, 2, 8, 3, 7, 4, 6, 5 };**  **int[] numbers3 = { 200, 9, 2, 8, 3, 7, 4, 6, 5 };**  이렇게 numbers3만 [0]번째 인덱스가 바뀐다. 이걸 **깊은 복사**라고 부른다. |
| --- |

#### 

#### 

### **- [ Arrays 클래스 사용법 ]**

배열을 관리해 주는 객체

| **binarySearch(찾고자 하는 값)** | |
| --- | --- |
| 배열의 내용을 검색 후 인덱스를 반환받는 방법 | |
| Arrays.sort(numbers); //binarySearch는 꼭 sort후에 써야한다.  int index = Arrays.binarySearch(numbers, 5); | |

#### 

| **copyOf(복사하고자 하는 배열, 새로 생성할 배열의 크기)** | |
| --- | --- |
| 배열을 복사할 때 사용하는 메소드 | |
| int[] numbers4 = Arrays.copyOf(numbers, numbers.length + 10); | |

#### 

| **copyOfRange(복사하고자 하는 배열, 시작 인덱스, 마지막 인덱스)** | |
| --- | --- |
| 배열을 복사할 때 원하는 범위만 복사해주는 메소드 | |
| int[] numbers5 = Arrays.copyOfRange(numbers, 1, 5); | |

#### 

| **fill(초기화하고자 하는 배열, 채울 값)** | |
| --- | --- |
| 배열을 초기화하고 원하는 값으로 모두 채우는 메소드 | |
| Arrays.fill(numbers5, 10); // numbers5을 비우고 10으로 채워라 | |

#### 

| **sort(정렬하고자 하는 배열)** | |
| --- | --- |
| 오름차순 정렬  Arrays.sort(numbers4); | |
| 내림차순 정렬  Integer[] numArray = {1,2,3,4,5}; // 역방향은 int 말고 Integer를 써야함  Arrays.sort(numArray, Collections.reverseOrder()); | |

## **상속**

부모 클래스(상위 클래스)와 자식 클래스(하위 클래스)가 있으며, 자식 클래스는 부모 클래스를 선택해서, 그 부모의 멤버를 상속받아 그대로 쓸 수 있게 된다. 이미 마련되어 있던 클래스를 재사용해서 만들 수 있기 때문에 효율적이고, 개발 시간을 줄일 수 있다.

| public Class Parent{ .... }; // 부모 클래스  public Class Child extends Parent { .... }; // 자식 클래스 |
| --- |

1. 기존에 작성된 클래스를 재활용할 수 있습니다.

2. 자식 클래스 설계 시 중복되는 멤버를 미리 부모 클래스에 작성해 놓으면, 자식 클래스에서는 해당 멤버를 작성하지 않아도 된다

3. 클래스 간의 계층적 관계를 구성함으로써 다형성의 문법적 토대를 마련합니다.

| **부모 클래스** | public class Animal {  public Animal() { // Animal 기본생성자  System.out.println("Animal 객체 생성");  }  public void move() {  System.out.println("동물이 움직입니다.");  }  } |
| --- | --- |
| **자식 클래스** | public class Human extends Animal {  public Human() { // Human 기본생성자  // super(); 이걸 삭제하지는 못함.  //부모의 주소가 생략되어있다. 부모가 먼저 생성  // System.out.println("Animal 객체 생성"); 이 출력됨  System.out.println("Human 객체 생성");  }    @Override //오버라이드: 재정의, @ -> 어노테이션  public void move() {  System.out.println("사람이 두발로 걷습니다.");  super.move(); // System.out.println("동물이 움직입니다."); 출력해줌  }  public void readBooks() {  System.out.println("사람이 책을 읽습니다.");  }  } |

### **- [ 오버라이드 ]**

| **@Override** // 오버라이드: 재정의, @ -> 어노테이션 | |
| --- | --- |
| 선언한 메서드가 오버라이드 되었다는 것을 나타낸다.  만약 상위 클래스에서 해당 메서드를 찾을 수 없다면 컴파일 에러를 발생 시킨다 | |
| 오버라이드란, 상위 클래스에 정의된 메소드의 이름, 반환형, 매개변수 선언까지 완전히 동일한 메소드를 하위 클래스에서 다시 정의한다면, 하위 클래스의 해당 메소드가 상위 클래스의 메소드를 덮어버리는(가려버리는, 지워버리는) 것이다. | |
| public class Animal {  public void speak() {  }  }  public class Cat extends Animal {  @Override  public void speak() {  System.out.println("Meow.");  }  } | |

### **- [ 캐스팅 ]**

**자료형이 다르면 캐스팅이 되었다는 걸 눈치채야 한다!**

예시

| ConnectionTerminal connectionTerminal = new Monitor(); |
| --- |

#### **업캐스팅**

자바의 상속관계는 상위클래스의 모든 특성을 하위 클래스가 받아서 사용할 수 있다. 그 말은 서브 클래스는 수퍼클래스 취급을 받을수 있다는 말인데 여기서 업캐스팅을 통해 객체가 슈퍼클래스 타입으로 형변환을 시킨다.

즉, 자식 클래스를 부모 클래스로 바꾸는 것이다.

| **묵시적** 형변환 : 캐스팅이 자동으로 발생 (업캐스팅)  Parent p = new Child();  // (Parent) new Child()할 필요가 없음  Parent를 상속받은 Child는 Parent의 속성을 포함하고 있기 때문  작은 컵에 가득 담긴 물을 큰 컵에 담아도 넘치지 않는다! 고 이해하면 편함 |
| --- |

업캐스팅 된 (구)자식클래스는 자식 요소만 가졌던 성질을 사용할 수 없다.

#### **다운캐스팅**

업캐스팅과 반대로 캐스팅 하는 것을 다운캐스팅(Downcasting)이라고 한다.

자신의 고유한 특성을 잃은 서브클래스의 객체를 다시 복구시켜주는 것을 말하는데, 다시말하자면 **업캐스팅 된 것**을 **다시 원상태로 돌리는 것**을 말한다.

// 변수와 상수에서 나왔던 다운캐스팅은 이런 제한이 없다

| 명시적 형변환 : 캐스팅할 내용을 적어줘야 하는 경우 (다운캐스팅)  Parent p = new Child(); // 업캐스팅 발생  Child c = (Child) p; // 다운캐스팅 함  // 다운캐스팅은 업캐스팅이 발생한 이후에 작용한다. |
| --- |

#### **instaceof**

객체의 원래 형태를 비교하는 함수이다.

instanceof 연산자의 결과 값은 boolean 타입으로 true, false를 반환 값으로 가진다.

상속에서 다운캐스팅하며 자주 사용됨.

(자식만 가진 속성은 업캐스팅 상태로는 사용할 수 없기 때문이다)

| [객체 레퍼런스] instanceof [클래스타입] |
| --- |
| Person jee = new Student();  Person kim = new Professor();  Person lee = new Researcher();  if(jee instanceof Person) // jee는 Person타입을 상속 받았으므로 true  if(jee instanceof Student) // jee는 Student타입이므로 true  if(kim instanceof Student) // kim은 Student타입이 아니므로 false  if(kim instanceof Professor) // kim은 Professor타입이므로 true  if(kim instanceof Researcher) // kim은 Researcher타입을 상속 받았으므로 true  if(lee instanceof Professor) // lee는 Professor타입이 아니므로 false  if("java" instanceof String) // "Java"는 String 타입의 인스턴스이므로 true;  if(3 instanceof int) // 문법 오류. instanceof는 객체에 대한 레퍼런스에만 사용 |

| **활용 예문** | |
| --- | --- |
| for(int i = 0; i < factories.length; i++) {  if(factories[i] instanceof SamsungFactory) { // factories의 다운캐스팅이 삼성팩토리  SamsungFactory samsungFactory = (SamsungFactory) factories[i];  samsungFactory.produceSmartPhone();  }else if(factories[i] instanceof LGFactory) { // 팩토리즈의 다운캐스팅 LG  LGFactory lgFactory = (LGFactory) factories[i];  lgFactory.produceSmartTv();  }else {  System.out.println("형변환 불가");  }  } | |

#### 

## **다형성**

상위 클래스가 동일한 메시지로 하위 클래스들을 서로 다르게 동작시키는 객체 지향 원리다. 다형성을 활용하면 부모 클래스가 자식 클래스의 동작 방식을 알수 없어도 오버라이딩을 통해 자식 클래스를 접근할 수 있다.

| 응집도를 높이고 결합도를 낮추기 위함이다. |
| --- |

= 하나의 객체가 여러 가지 타입을 가질 수 있는 것을 의미한다. 자바에서는 이러한 다형성을 부모 클래스 타입의 참조 변수로 자식 클래스 타입의 인스턴스를 참조할 수 있도록 하여 구현하고 있다.

상속, 추상화와 더불어 객체 지향 프로그래밍을 구성하는 중요한 특징 중 하나이다.

| **참조 변수의 다형성** | |
| --- | --- |
| class Parent { ... }  class Child extends Parent { ... }  ...  Parent pa = new Parent(); // 허용  Child ch = new Child(); // 허용  Parent pc = new Child(); // 허용  Child cp = new Parent(); // 오류 발생. | |

#### 

**특정 타입의 참조 변수**로 **같은 타입의 인스턴스를 참조** **가능**

- 참조 변수가 사용할 수 있는 멤버의 개수가 실제 인스턴스의 멤버 개수와 같기 때문

**부모 클래스 타입의 참조 변수**로 **자식 클래스 타입의 인스턴스를 참조** **가능**

- 참조 변수가 사용할 수 있는 멤버의 개수가 실제 인스턴스의 멤버 개수보다 적기 때문

**자식 클래스 타입의 참조 변수**로는 **부모 클래스 타입의 인스턴스를 참조** **불가능**

- 참조 변수가 사용할 수 있는 멤버의 개수가 실제 인스턴스의 멤버 개수보다 많기 때문

### **- [ 추상메소드 ]**

자식 클래스에서 반드시 오버라이딩해야만 사용할 수 있는 메소드를 의미한다.

추상 메소드가 포함된 클래스를 **상속받는 자식 클래스**가 반드시 **추상 메소드를 구현**하도록 하기 위함입니다. 설계도와 비슷한 느낌이다.

이러한 추상 메소드는 선언부만이 존재하며, 구현부는 작성되어 있지 않다.

바로 이 작성되어 있지 않은 구현부를 자식 클래스에서 오버라이딩하여 사용한다.

| 문법: **abstract** 반환타입 메소드이름(); |
| --- |

위와 같이 선언부만 있고 구현부가 없다는 의미로 선언부 끝에 바로 세미콜론(;)을 추가한다.

#### **추상 클래스**

하나 이상의 추상 메소드를 포함하는 클래스이다. **반드시 사용되어야 하는 메소드**를 추상 클래스에 **추상 메소드로 선언**해 놓으면, 이 클래스를 **상속받는 모든 클래스**에서는 이 **추상 메소드를 반드시 재정의**해야 한다.

| **abstract** class 클래스 이름 {  … // 필드 or 메소드 or 생성자  **abstract** 반환타입 메소드이름();  …  } |
| --- |
| public **abstract** class Animal {  public **abstract** void move(); // 이렇게 사용도 가능하다.  } |
| 추상 클래스를 상속받는 자식 클래스  public class Human **extends** Animal {  **@Override** // 추상메소드를 상속받았기 때문에 꼭 포함되어 있어야 한다  public void move() {  System.out.println("사람이 두 발로 걷습니다.");  }  } |

동작이 정의되어 있지 않은 추상 메소드를 포함하고 있으므로, 인스턴스를 생성할 수 없다.

추상 클래스는 먼저 **상속을 통해 자식 클래스를 만들고**, 만든 **자식 클래스에서 추상 클래스의 모든 추상 메소드를 오버라이딩**하고 나서야 **자식 클래스의 인스턴스를 생성**할 수 있다.

1. 추상 메소드가 하나라도 포함되어 있는 클래스는 무조건 추상 클래스로 정의해야 한다.

2. 기본 메소드이나 @Override 할 거라 생랴 가능한 메소드일 경우에도 추상메소드 사용

3. 상속은 1개씩만 가능하다 (부모n명 자식1명은 불가능)

4. 객체를 직접 생성해서 사용할 수 없다.

(즉, 추상 클래스는 new 연산자를 사용해서 인스턴스를 생성시키지 못한다)

| Animal animal = **new** Animal (); // 불가능  Animal animal = **new** **Human** (); // 가능! 부모 대신 자식 이용 |
| --- |

5. 생성자를 가질 수 있다 (자식 클래스의 생성자에 포함되기 위함)

6. 오버라이딩을 전제로 하기 때문에 private으로 선언할 수 없다(publc이 기본이다)

### **- [ 인터페이스 ]**

자식 클래스가 여러 부모 클래스를 상속받을 수 있도록 **다중 상속**을 지원한다. 다른 클래스를 작성할 때 **기본이 되는 틀을 제공**하면서, 다른 클래스 사이의 **중간 매개 역할**까지 담당하는 일종의 추상 클래스를 의미한다.

단, 추상 클래스(생성자, 필드, 일반 메소드도 포함하는)와 달리 추상 메소드와 상수만을 포함할 수 있다.

| 접근제어자 **interface** 인터페이스이름 {  public static final 타입 상수이름 = 값;  ...  public **abstract** 메소드이름(매개변수 목록);  // 추상메소드만 가질 수 있으므로 abstract 생략 가능하다.  ...  } |
| --- |
| public **interface** USB {  public String VERSION = "2.0"; // 상수    public void connect(); // 추상메소드. abstract 생략!  public void disConnect(); // 추상메소드,  } |

인터페이스는 상속이라 하지 않고 **구현**이라 한다

| 인터페이스 > 인터페이스 | extends | 상속 |
| --- | --- | --- |
| 클래스 > 클래스 | extends | 상속 |
| 인터페이스 > 클래스 | implements | 구현 |

1. 모든 메소드는 기본적으로 "추상 메소드"로 정의한다. (abstract를 안 붙여도 되는 이유!)

2. 생성을 할 수 없다.

3. 일반 변수는 선언할 수 없다. (무조건 스태틱 상수를 사용한다.)

4. 인터페이스의 조상은 인터페이스만 가능하다. (Object가 최고 조상이 아니다)

5. 일반메소드를 정의하기 위해서는 반환 자료형(void) 앞에 default 를 입력해야한다.

6. 인터페이스는 다중 구현이 가능하다. (추상메소드와 충돌해도 문제없다)

## **최상위 클래스**

모든 클래스는 Object 클래스를 상속받고 있다. extends Object 가 생략되었다. 즉, 자바의 클래스는 Object 클래스의 모든 메소드를 바로 사용할 수 있으며 @Override 가 가능하다.

이러한 Object 클래스는 필드를 가지지 않으며, 총 11개의 메소드만으로 구성되어 있다.

| 메소드 | 설명 |
| --- | --- |
| protected Object **clone**() | 해당 객체의 복제본을 생성하여 반환함. |
| boolean **equals**(Object obj) | 해당 객체와 전달받은 객체가 같은지 여부를 반환함. |
| protected void **finalize**() | 해당 객체를 더는 아무도 참조하지 않아 가비지 컬렉터가 객체의 리소스를 정리하기 위해 호출함. |
| Class<T> **getClass**() | 해당 객체의 클래스 타입을 반환함. |
| int **hashCode**() | 해당 객체의 해시 코드값을 반환함. |
| void **notify**() | 해당 객체의 대기(wait)하고 있는 하나의 스레드를 다시 실행할 때 호출함. |
| void **notifyAll**() | 해당 객체의 대기(wait)하고 있는 모든 스레드를 다시 실행할 때 호출함. |
| String **toString**() | 해당 객체의 정보를 문자열로 반환함. |
| void **wait**() | 해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행할 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출함. |
| void **wait**(long timeout) | 해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행하거나 전달받은 시간이 지날 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출함. |
| void **wait**(long timeout, int nanos) | 해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행하거나 전달받은 시간이 지나거나 다른 스레드가 현재 스레드를 인터럽트(interrupt) 할 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출함. |

### **- [ toString ]**

이 메서드는 인스턴스에 대한 정보를 문자열(String)로 제공할 목적으로 정의한 것이다. 이때 반환되는 문자열은 클래스 이름과 함께 구분자로 '@'가 사용되며, 그 뒤로 16진수 해시 코드 (hash code)가 추가된다.

16진수 해시 코드 값은 주소를 가리키는 값으로, 인스턴스마다 모두 다르게 반환된다.

| **예시** | **결과** | |
| --- | --- | --- |
| Car car01 = new Car();  Car car02 = new Car();  System.out.println(car01.**toString()**);  System.out.println(car02.**toString()**); | Car@15db9742  Car@6d06d69c | |

#### 

.**toString()** 은 **생략이 가능**하다.

| **예시** | **결과** | |
| --- | --- | --- |
| Car car01 = new Car();  Car car02 = new Car();  System.out.println(car01);  System.out.println(car02); | Car@15db9742  Car@6d06d69c | |

인스턴스의 정보를 제공한다는 것은 대부분의 경우 인스턴스 변수에 저장된 값들을 문자열로 표현한다는 뜻이다.

### **- [ equals ]**

두 객체의 같고 다름을 참조변수의 값으로 판단한다. 그렇기 때문에 서로 다른 두 객체를 equals 메서드로 비교하면 항상 false를 결과로 얻게 된다. 즉 eqauls는 **주소값**으로 비교한다.

그러나 우리가 오버라이딩을 하여 사용하기 때문에 값(문자)를 비교하는게 된다.

| String name2 = "김준일";  String name3 = new String("김준일"); | |
| --- | --- |
| 주소의 비교 | name2 == name3 |
| 문자의 비교  (오버라이딩 기본적용) | name2.**equals**(name3) |

| **예시** | **결과** | |
| --- | --- | --- |
| Car car01 = new Car();  Car car02 = new Car();  // 새로운 개체를 두 번 생성한것(주소가 다름)  System.out.println(car01.**equals**(car02));  car01 = car02; // 참조 변수가 같은 주소를 가리킴.  System.out.println(car01.**equals**(car02)); | false  true | |

#### 

구현된 객체가 다르면 false로 판단한다.

| **예시** | **결과** | |
| --- | --- | --- |
| Student s1 = new Student(20220001, "김준일");  Student s2 = new Student(20220001, "김준일");  Student s3 = new Student(20220002, "김준이");  Student s4 = new Student(20220002, "김준일");  System.out.println(s1 == s2);  System.out.println(s1.equals(s2));  System.out.println(s1.equals(s3));  System.out.println(s3.equals(s4));  System.out.println(s1.getClass() == s2.getClass());  System.out.println(s1 instanceof Student);  System.out.println(s1.getClass() == Student.class);  //instaceof 비교보단 이걸 많이 쓰는 추세이다  System.out.println(s1.getClass());  System.out.println(s2.getClass()); | false  true  false  false  true  true  true  class j13\_최상위클래스.Student  class j13\_최상위클래스.Student | |

#### 

### **- [ HashCode ]**

객체를 식별하는 Integer 값이다. 객체가 갖고 있는 데이터를 어떤 알고리즘에 적용하여 계산된 정수 값을 hashcode라고 할 수 있다.

주소가 다르더라도 값이 같을 수 있다

만일 hashCode메서드를 오버라이딩하지 않는다면 Object 클래스에 정의된 대로 모든 객체가 서로 다른 해시코드 값을 가진 것이다.

### **- [ Class ]**

Class 클래스는 자바에서 사용되는 클래스들에 대한 구조를 가지고 있는 Class이다.

| 클래스와 인스턴스의 메타 데이터를 얻을 수 있다(=리플렉션)   * 메타 데이터 : 클래스이름, 생성자, 필드, 메서드 정보   객체로부터 얻는법, 클래스 이름으로부터 얻는 법(패키지명.class) |
| --- |

| **예시** | |
| --- | --- |
| Student student = new Student(20220001, "김준일");  Class c = student.getClass();  Class c2 = Student.class;  System.out.println("클래스명(모든 경로 포함): " + c.getName());  System.out.println("클래스명(클래스 명만): " + c.getSimpleName());  //메소드 클래스의 배열을 가져온다 //클래스의 메소드 관리  Method[] methods = c.getMethods();  for(int i = 0; i<methods.length; i++) {  Method m = methods[i];  Parameter[] prameters = m.getParameters();    for(int j = 0; j < prameters.length; j++) {  System.out.println("\t" + prameters[j].getType());  }  } | |

| **결과** | |
| --- | --- |
| 클래스명(모든 경로 포함): j13\_최상위클래스.Student  클래스명(클래스 명만): Student  class java.lang.Object  int  class java.lang.String  long  int  long  최상위클래스.Student | |

## **lombok**

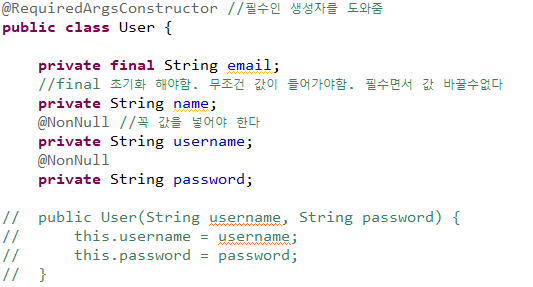
반복되는 getter, setter, toString 등의 메서드 코드를 줄여주는 코드 다이어트 라이브러리이다.

Lombok은 여러가지 어노테이션을 제공하고 이를 기반으로 코드를 컴파일과정에서 생성해 주는 방식으로 동작한다. 즉 코딩 과정에서는 롬복과 관련된 어노테이션만 보이고 getter와 setter 메서드 등은 보이지 않지만 실제로 컴파일된 결과물(.class)에는 코드가 생성되어 있다.

| **특수코드값** | **뜻** | **비고** |
| --- | --- | --- |
| @NoArgsConstructor | 기본생성자 | 해당 클래스에 빈 생성자를 자동으로 생성 |
| @AllArgsConstructor | 전체생성자 | 해당 클래스의 모든 변수를 등록하는 생성자를 자동으로 생성 |
| @EqualsAndHashCode |  | equals 메소드와 hashCode 메소드를 생성.   * equals : 두 객체의 내용이 같은지, 동등성을 비교하는 연산자. * hashCode : 두 객체가 같은 객체인지, 동일성을 비교하는 연산자입니다. |
| @Setter |  | 선언 되어진 변수의 Setter를 자동으로 생성 |
| @Getter |  | 선언 되어진 변수의 Getter를 자동으로 생성 |
| @ToString |  | toString 을 자동으로 생성 |
| @Data | 위 4개를 통합 | @Setter @Getter @EqualsAndHashCode @ToString |
| @NonNull |  | 어노테이션을 변수에 붙이면 자동으로 null 체크를 해줍니다. 즉, 해당 변수가 null로 넘어온 경우, NullPointerException 예외를 일으켜 줍니다 |

### **- [ @RequiredArgsConstructor ]**

final이나 @NonNull 이 붙은 필드의 생성자를 자동 생성한다.



위에서 필수라고 표기된 email, username, password의 생성자만 생성된다.

### **- [ @Builder ]**

@AllArgsConstructor 어노테이션을 붙인 효과를 발생시켜 모든 멤버 필드에 대해서 매개변수를 받는 기본 생성자를 만든다.

| @Builder  public class 클래스명 {  멤버변수…  } |
| --- |

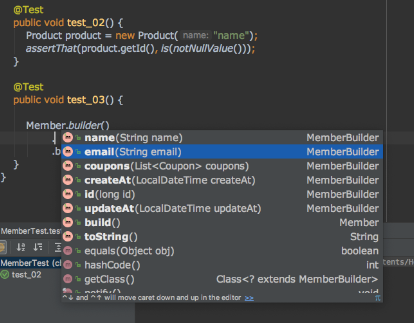
| **예시** | |
| --- | --- |
| ▶ Member 클래스 | import lombok.Builder;  import lombok.Data;  @Data  **@Builder**  public class Member {  private Long id;  private String name;  private String email;  private String birthDate;  } |
| ▶ ApplicationMain 클래스  .builder()  ~  .build() | public class ApplicationMain {  public static void main(String[] args) {  Member member = Member**.builder()**  **.id**(1L)  **.name**("피카츄")  **.email**("pikachu@naver.com")  **.birthDate**("2021-05-21")  **.build();**  System.out.println(member);  }  } |
| 실행 결과 | Member(id=1, name=피카츄, email=pikachu@naver.com, birthDate=2021-05-21) |

Builder 사용시 매개변수를 최소화 하는게 좋다.

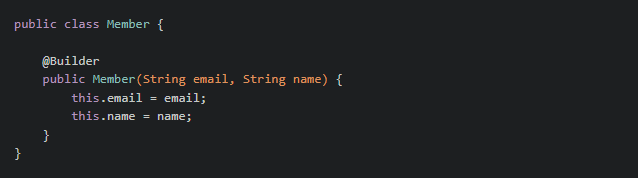


클래스 위에 @Builder를 사용 시 @AllArgsConstructor 어노테이션을 붙인 효과를 발생시켜 모든 멤버 필드에 대해서 매개변수를 받는 기본 생성자를 만든다.

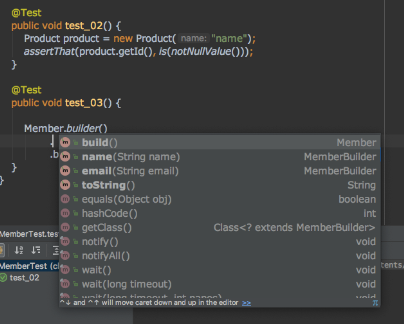
이때, Builder AllArgsConstructor는 무슨 문제는 아래와 같다..



위 그림처럼 모든 멤버필드에 대한 매개변수를 허용하게 된다.



이렇게 **받아야 하는 생성자**를 **필요조건에 따라 지정**하고 그 위에 **@Builder**를 붙이는게 바람직하다.



위 그림처럼 매개변수 name, email만 넘겨 받을 수 있게 된다. 코드의 안전성을 높임.

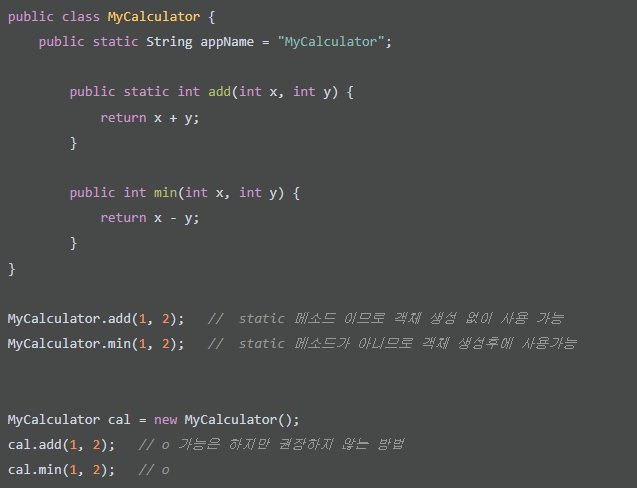
## **스태틱**

일반적으로 우리가 만든 **Class**는 **Static 영역에 생성**되고, **new 연산을 통해 생성한 객체**는 **Heap영역에 생성**된다. 객체의 생성시에 할당된 Heap영역의 메모리는 Garbage Collector를 통해 수시로 관리를 받는다.

하지만 Static 키워드를 통해 Static 영역에 할당된 메모리는 **모든 객체가 공유하는 메모리라는 장점**을 지니지만, Garbage Collector의 관리 영역 밖에 존재하므로 Static을 자주 사용하면 프로그램의 종료시까지 메모리가 할당된 채로 존재하므로 자주 사용하게 되면 시스템의 퍼포먼스에 악영향을 주게 된다.

| Static 변수는 클래스 변수이다.  객체를 생성하지 않고도 Static 자원에 접근이 가능하다. |
| --- |

Static 변수와 static 메소드는 Static 메모리 영역에 존재하므로 객체가 생성되기 이전에 이미 할당이 되어 있다. 그렇기 때문에 객체의 생성없이 바로 접근(사용)할 수 있다.



### **- [ Static 변수 ]**

일반적으로 상수들만 모아서 사용하며 상수의 변수명은 대문자와 \_를 조합하여 이름짓는다. 또한 상속을 방지하기 위해 final class로 선언을 한다.

| **public final class** AppConstants {  **public static final** String APP\_NAME = "MyApp";  **public static final** String PREF\_NAME = "MyPref";    } |
| --- |

Java에서 Static 변수는 메모리에 한번 할당되어 프로그램이 종료될 때 해제되는 변수로, 메모리에 한번 할당되므로 여러 객체가 해당 메모리를 공유하게 된다.

*\* 그런 이유로 일반 메소드 안에는 static 변수 선언을 할 수 없다? 이미 할당됐으므로? 호출될 때 생성되는 메소드안에는 안 맞다?*

| 메모리에 고정적으로 할당되어, 프로그램이 종료될 때 해제되는 변수 |
| --- |

예를 들어, 세상 모든 사람의 이름이 'MangKyu'인 세상에 살고있다고 가정을 하겠습니다. 이럴때면 우리는 아래와 같이 객체를 만들 수 있습니다.

Java에서 Static 변수는 메모리에 한번 할당되어 프로그램이 종료될 때 해제되는 변수로, 메모리에 한번 할당되므로 여러 객체가 해당 메모리를 공유하게 됩니다.

| **예시** | |
| --- | --- |
| 세상 모든 사람의 이름이 'MangKyu'인 세상에 살고있다고 가정한다.  이럴때면 우리는 아래와 같이 객체를 만들 수 있다.  public class Person {  private String name = "MangKyu";    public void printName() {  System.out.println(this.name);  }  } | |

위와 같은 클래스를 통해 100명의 Person 객체를 생성하면, "MangKyu"라는 같은 값을 갖는 메모리가 100개나 중복해서 생성되게 된다. 이러한 경우에 static을 사용하여 여러 객체가 하나의 메모리를 참조하도록 하면 메모리 효율이 더욱 높아질 것이다.

또한 "MangKyu"라는 이름은 결코 변하지 않는 값이므로 final 키워드를 붙여주며, 일반적으로 Static은 상수의 값을 갖는 경우가 많으므로 public으로 선언을 하여 사용한다.

이러한 이유로, 일반적으로 static 변수는

**public** 및 **final**과 함께 사용되어 **public static final로 활용 되는 경우가 많다.**

| **예시 수정안** | |
| --- | --- |
| public class Person {  public static final String name = "MangKyu";    **public static void** printName() {  System.out.println(this.name);  }  } | |

### **- [ Static 메소드 ]**

변수와 마찬가지로 상속을 방지하고자 final class로 선언하고, 유틸관련 함수들을 모아둔다.

| **import** java.text.SimpleDateFormat;  **import** java.util.Date;  **import** android.util.Patterns;    **public final class** CommonUtils {    **public static** **String** getCurrentDate() {  Date date = new Date();  SimpleDateFormat dateFormat = new SimpleDateFormat("yyMMdd");  **return** dateFormat.format(date);  }    **public static boolean** isEmailValid(String email) {  **return** Patterns.EMAIL\_ADDRESS.matcher(email).matches();  }    } |
| --- |

Static Method는 객체의 생성 없이 호출이 가능하며, 객체에서는 호출이 가능은 하지만 지양한다. 일반적으로는 유틸리티 관련 함수들은 여러 번 사용되므로 static 메소드로 구현을 하는 것이 적합하며, static 메소드를 사용하는 대표적인 Util Class로는 java.uitl.Math가 있다.

| **예시** | |
| --- | --- |
| public class Test {  **private** String name1 = "MangKyu";  **private static** String name2 = "MangKyu";    **public static void** printMax(int x, int y) {  System.out.println(Math.max(x, y));  }    **public static void** printName(){  // System.out.println(name1); 불가능한 호출  System.out.println(name2);  }  } | |

두 수의 최대값을 구하는 경우에 Math클래스를 사용하는데, static 메소드로 선언된 max 함수를 **초기화 없이 사용**한다.

하지만 static 메소드에서는 static이 선언되지 않은 변수에 접근이 불가능한데, 메모리 할당과 연관지어 생각해보면 당연합니다. 우리가 Test.printName() 을 사용하려고 하는데, **name1**은 new **연산을 통해 객체가 생성된 후에 메모리가 할당**됩니다. 하지만 **static 메소드는 객체의 생성 없이 접근하는 함수**이므로, 할당되지 않은 메모리 영역에 접근을 하므로 **문제가 발생**하게 됩니다.

그러므로 **static 메소드에서 접근하기 위한 변수**는 반드시 **static 변수로 선언**되어야 한다.

### **- [ 싱글톤 ]**

자기 자신을 생성?

어떤 클래스가 최초 한번만 메모리를 할당하고(Static) 그 메모리에 객체를 만들어 사용하는 디자인 패턴을 의미한다. 즉 **생성자의 호출이 반복**적으로 이뤄져도 **실제로 생성되는 객체는 최초 생성된 객체를 반환** 해주는 것이다.

| public class Singleton {    **private static Singleton** instance = new **Singleton**(); //Instance 스태틱 변수    private **Singleton**() {  // 생성자는 외부에서 호출못하게 private 으로 지정해야 한다.  }  **private static Singleton** getInstance() { //getinstance 스태틱 메소드  return instance;  }  } |
| --- |

위 코드에서는 instance라는 전역 변수를 선언하는데 static을 줌으로써 인스턴스화 하지 않고 사용할 수 있게 하였지만 접근 제한자가 private 로 되어 있어 직접적인 접근은 불가능하다. 또한 생성자도 private으로 되어 있어 new 를 통한 객체 생성도 불가능하다.

결국 getInstance 메서드를 통해서 해당 인스턴스를 얻을 수 있게 된다. 이렇게 생성된 객체는 무조건 **한번 생성**으로 **전역성**을 띄기에 다른 객체와 공유가 용이하다

| 유일한 객체를 만들어서 공유하는 생성패턴  1. 생성자는 접근지정자가 private이다.  2. 자기 자신의 유일한 객체를 담을 수 있는   * insntance 라는 변수명을 가진 스태틱 변수가 필요하다.   3. instance 변수를 리턴(공유)해줄 getInstance 스태틱 메소드가 필요하다. |
| --- |

## **제네릭**

데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있게 해준다. 클래스 내부에서 지정하는 것이 아닌 외부에서 사용자에 의해 지정되는 것을 의미한다. 한마디로 특정(Specific) 타입을 미리 지정해주는 것이 아닌 필요에 의해 지정할 수 있도록 하는 일반(Generic) 타입이라는 것이다.

| **타입** | **설명** | |
| --- | --- | --- |
| <T> | Type | |
| <E> | Element | |
| <K> | Key | |
| <V> | Value | |
| <N> | Number | |
| <Student> //사용자가 정의한 클래스 | Student //사용자가 정의한 클래스 | |
| <?> | 문법상 정해져있는 것 | |

**객체<타입> 객체명 = new 객체<타입>();** 의 형태로 생성할 때 **<> 괄호** 안에 들어가는 타입을 지정해준다.

| **public class** ClassName **<T>** { ... }  **public class** ClassName **<T>** {  private String message;  private **T** data;  }  **public Interface** InterfaceName **<T>** { ... } |
| --- |
| **public class** ClassName **<T, K>** { ... }  **public Interface** InterfaceName **<T, K>** { ... }    // HashMap의 경우 아래와 같이 선언되어있을 것이다.  **public class** HashMap **<K, V>** { ... } |

위처럼 생성된 제네릭 클래스를 사용하기 위해 **객체를 생성**할 때, 구체적인 타입을 명시를 해주어야 하는 것이다.

| **public class** ClassName **<T, K>** { ... }    **public class**  Main {  **public static void** main(String[] args) {  ClassName**<String, Integer>** a = new ClassName**<String, Integer>**();  }  } |
| --- |

이때, 타입 파라미터로 명시할 수 있는 것은 참조 타입(Reference Type)밖에 올 수 없다. 즉, int, double, char 같은 primitive type은 올 수 없다는 것이다. int형 double형 등 primitive Type의 경우 Integer, Double 같은 Wrapper Type으로 쓰는 이유이다.

바꿔 말하면 참조 타입이 올 수 있다는 것은 **사용자가 정의한 클래스**도 타입으로 올 수 있다.

| **public class** ClassName **<T, K>** { ... }  **public class** Student { ... }    **public class**  Main {  **public static void** main(String[] args) {  ClassName**<Student>** a = new ClassName**<Student>**();  }  } |
| --- |

1. 제네릭을 사용하면 잘못된 타입이 들어올 수 있는 것을 **컴파일 단계에서 방지**할 수 있다.

2. 클래스 **외부에서 타입을 지정**해주기 때문에 따로 타입을 체크하고 변환해줄 필요가 없다. 즉, 관리하기가 편하다.

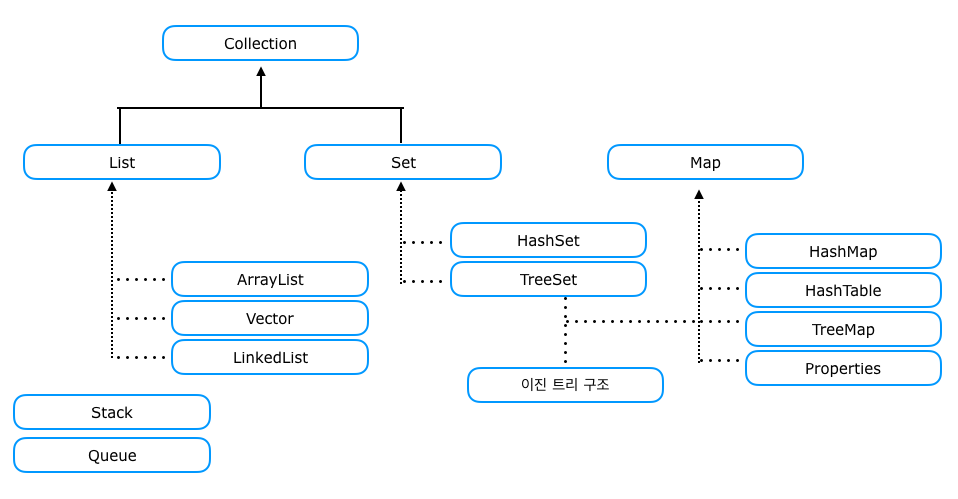
3. 비슷한 기능을 지원하는 경우 **코드의 재사용성**이 높아진다.

## **컬렉션 클래스**

다수의 데이터를 쉽고 효과적으로 처리할 수 있는 표준화된 방법을 제공하는 클래스의 집합이다 컬렉션 프레임워크의 모든 컬렉션 클래스는 **List와 Set, Map 인터페이스** 중 하나의 인터페이스를 구현한다.

클래스 이름에도 구현한 인터페이스의 이름이 포함된다.

| **인터페이스** | **설명** | **구현 클래스** |
| --- | --- | --- |
| List<E> | 순서가 있는 데이터의 집합으로, **데이터의 중복을 허용**함 | Vector, **ArrayList**, LinkedList, Stack, Queue |
| Set<E> | 순서가 없는 데이터의 집합으로, 데이터의 **중복을 허용하지 않음**. | **HashSet**, TreeSet |
| Map<K, V> | **키와 값의 한** 쌍으로 이루어지는 데이터의 집합으로, **순서가 없음**.  이때 **키는 중복을 허용**하지 않지만, **값은 중복될 수 있음**. | **HashMap**, TreeMap, Hashtable, Properties |



### **- [ Collection 인터페이스 ]**

컬렉션을 다루는데 가장 기본적인 동작들을 정의하고, 그것을 메소드로 제공한다. LIST 인터페이스. SET인터페이스는 이를 상속받는다. (두 인터페이스는 Collection에서 제공하는 메소드를 사용할 수 있다.)

| **메소드** | **설명** | |
| --- | --- | --- |
| boolean add(E e) | 해당 컬렉션(collection)에 전달된 요소를 추가. (선택적 기능) | |
| void clear() | 해당 컬렉션의 모든 요소를 제거함. (선택적 기능 | |
| boolean contains(Object o) | 해당 컬렉션이 전달된 객체를 포함하고 있는지를 확인함 | |
| boolean equals(Object o) | 해당 컬렉션과 전달된 객체가 같은지를 확인함. | |
| boolean isEmpty() | 해당 컬렉션이 비어있는지를 확인함. | |
| Iterator<E> iterator() | 해당 컬렉션의 반복자(iterator)를 반환함. | |
| boolean remove(Object o) | 해당 컬렉션에서 전달된 객체를 제거함. (선택적 기능) | |
| int size() | 해당 컬렉션의 요소의 총 개수를 반환함. | |
| Object[] toArray() | 해당 컬렉션의 모든 요소를 Object 타입의 배열로 반환함. | |

#### **Iterator**

자바의 컬렉션(Collection)에 저장되어 있는 요소들을 순회하는 인터페이스이다. 컬렉션 프레임워크에 대해 공통으로 사용이 가능하고 사용법이 간단하다.

| **Iterator**<T> iterator = **Collection.iterator**(); |
| --- |

| **메소드** | **설명** | |
| --- | --- | --- |
| hasNext() | 다음 값이 존재하는지 확인한다.  읽어올 요소가 있으면 true, 없으면 false | |
| next() | 다음 데이터를 반환한다. | |
| remove() | next()로 읽어온 요소를 삭제한다. | |

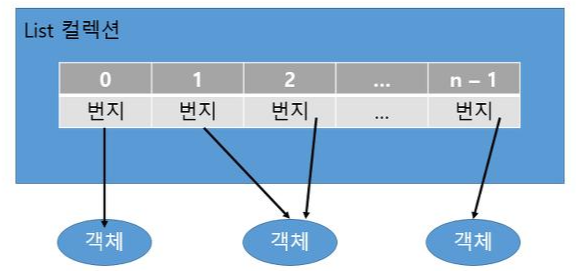
메소드 호출 순서는 hasNext() -> next() -> remove()이다.

| **예시** | |
| --- | --- |
| System.out.println("List 생성");  **ArrayList** list = new**ArrayList**();  list.add("일");  list.add("월");  list.add("수");  **Iterator** iter = list.**Iterator**();  while (iter.hasNext() == true) {  String day = (String) iter.next();  if (day == "수") {  iter.remove();  }  System.out.println("Day : " + day);  }  iter = list.**Iterator**();  while(iter.hasNext() == true) {  String day = (String)iter.next();  System.out.println("Day : " + day);  } |  |

### **- [ List 컬렉션 ]**

List 컬렉션은 객체를 일렬로 늘어놓은 구조로 이루어져 있다. 객체를 **인덱스로 관리**하기 때문에 List 컬렉션에 객체를 추가하면 자동 인덱스가 부여된다. 인덱스는 객체를 검색, 삭제할 때 사용한다. List 컬렉션은 객체 자체를 저장하는 것이 아닌 객체의 번지를 참조한다.

동일한 객체를 저장하는 것을 허락(이 점은 Set 인터페이스와 다른점)하는데, 이 경우에는 동일한 객체의 번지를 참조한다. 그리고 null도 저장할 수 있다.



즉, List 인터페이스를 구현한 모든 List 컬렉션 클래스는 다음과 같은 특징을 가진다.

1. 요소의 **저장 순서가 유지**된다.

2. 같은 요소의 **중복 저장을 허용**한다.

| **메서드** | **설명** |
| --- | --- |
| boolean add(E e) | 주어진 객체를 맨 끝에 추가합니다. |
| void add(int index, E element) | 주어진 인덱스에 객체를 추가합니다. |
| set(int index, E element) | 주어진 인덱스에 저장된 객체를 주어진 객체로 바꿉니다. |
| boolean contains(Object o) | 주어진 객체가 있는지에 대한 여부를 검색합니다. |
| E get(int index) | 주어진 인덱스에 저장된 객체를 리턴합니다. |
| isEmpty() | 컬렉션이 비어있는지 여부를 확인합니다. |
| int size() | 저장되어 있는 전체 객체 수를 리턴합니다. |
| E remove(int index) | 주어진 인덱스에 저장된 객체를 삭제합니다. |
| void clear() | 주어진 인덱스에 저장된 객체를 삭제합니다. |
| boolean remove(Object o) | 주어진 객체를 삭제합니다. |

#### **Arrays.asList()**

일반 배열을 ArrayList로 변환한다.

| **List**<String> list = **Arrays.asList**(arr) |
| --- |

Arrays.asList()는 Arrays의 private 정적 클래스인 ArrayList를 리턴한다.

java.util.ArrayList 클래스와는 다른 클래스이다.

java.util.Arrays.ArrayList 클래스는 set(), get(), contains() 메서드를 가지고 있지만 원소를 추가하는 메서드(.add())는 가지고 있지 않기 때문에 사이즈를 바꿀 수 없다.

asList()로 List를 생성하면 원소를 새롭게 추가할 수 없다.

List는 내부 구조가 배열로 만들어져 있다. 따라서 asList()를 사용해서 반환되는 List도 배열을 갖게 된다. 이때, asList()를 사용해서 List 객체를 만들 때 새로운 배열 객체를 만드는 것이 아니라, **원본 배열의 주소값**을 가져오게 된다.

따라서 asList()를 사용해서 내용을 수정하면 원본 배열도 함께 바뀌게 되고 원본 배열을 수정하면 그 배열로 만들어뒀던 asList()를 이용한 List 내용도 바뀌게 된다.

이러한 이유 때문에 Arrays.asList()로 만든 List에 **새로운 원소를 추가하거나 삭제 할 수 없다**. 따라서 Arrays.asList()는 배열의 내용을 수정하려고 할 때 **List로 바꿔서 편리하게 사용**하기 위해 쓰인다.

만약 진짜 ArrayList를 받기 위해서는 다음과 같이 변환하면 된다.

| **List**<String> list = new **ArrayList**<String>(**Arrays.asList**(arr)); |
| --- |

ArrayList 생성자는 java.util.Arrays.ArrayList의 상위(super) 클래스인 Collection Type도 받아들일 수 있다.

#### **ArrayList 클래스**

**List 컬렉션 인터페이스를 구현한 클래스**이다. List 컬렉션처럼 인덱스로 객체를 관리한다. 일반 배열과 ArrayList는 인덱스로 객체를 관리한다는 점에서 동일하지만, **크기를 동적으로 늘릴 수 있다**는 점에서 차이점이 있다.

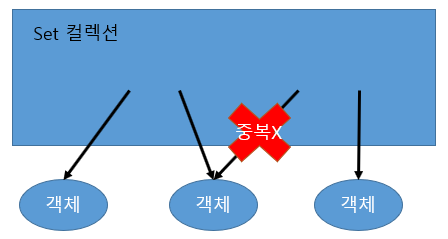
| // String 객체를 관리하는 ArrayList 생성  // 기본 저장 용량은 10  **List**<String> list = new **ArrayList**<>();  // 저장 용량을 100으로 설정해서 String 객체를 관리하는 ArrayList 생성  **List**<String> list = new **ArrayList**<>(100);  **List**<**Map**<String, Object>> objectList = new **ArrayList**<String,String>()  // LIST 안에 MAP 넣는 것도 가능하다 |
| --- |

| **실제 표현** | **예문** | |
| --- | --- | --- |
| **값 추가**  - **.add**(index, value) | list **.add**("java");  list **.add**("python");  list **.add**("null"); // null 값도 add 가능  list**.add**(null);  list**.add**(1, 10) // index 1에 10 삽입. 원하는 위치에 가능  // 기존의 값 index는 1씩 밀림  **List**<Student> members = new **ArrayList**<Student>();  **Student** student = new **Student** (name,age);  members **.add**(student);  members **.add**(new **Student** ("홍길동",15)); | |
| **값 삭제**  - **.remove**(index)  - **.remove**(value)  - **.removeAll**(Collection)  - **.clear**() | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  **List**<String> test = new **ArrayList**<String>(Arrays.asList(1,2,6));  list**.remove**(1); // index 1 제거  list**.remove**("1"); // 값 1을 찾아서 제거  list**.removeAll**(test); // list에서 test와 겹치는 값 제거  list**.clear**(); //모든 값 제거 | |
| **크기 구하기**  - **.size**() | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  list **.size**()  // list 크기 : 3  // 배열과 달리 공간 크기를 지정하지 않는다. 자동확장. | |
| **값 찾기(출력)**  - **.get**(index) | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  list **.get**(0)  // 0번째 index 찾기  System.out.println(list);  // list의 전체 출력 | |
| **인덱스 검색**  - **.indexOf**(value) | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  list **.indexOf**(1)  // 1이 있는 index반환. 값이 없으면 -1. | |
| **인덱스 위치의 값 수정**  - **.set**(index, value) | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  list **.set**(1, 6)  // 첫번째 인덱스 값을 6로 바꾼다. | |
| **값 검색**  - **.contains**(value)  - **.containsAll**(Collection) | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  **List**<String> test = new **ArrayList**<String>(Arrays.asList(1,2,6));  list **.contains**(1)  // list에 1을 포함한 값이 있는지 검색 : true  list **.containsAll**(test)  // list가 test 요소를 전부 포함하는지 검색 : false | |
| **다른 리스트값 모두 추가**  - **.addAll**(Collection) | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  **List**<String> test = new **ArrayList**<String>(Arrays.asList(1,2,6));  list **.addAll**(test)  // list가 test 요소를 전부 포함하는지 검색 : false | |
| **리스트가 비었는지 확인**  - **.isEmpty**(Collection) | **List**<Integer> list = new **ArrayList**<Integer>(Arrays.asList(1,2,3));  list **.isEmpty**()  // list 에 아무것도 없을 경우 true | |
| **배열을 리스트로 변경**  - **.isEmpty**(Collection) | **List**<String> list = Arrays.asList(new String[] {1,2,3}); | |

### **- [ Set 컬렉션 ]**

Set컬렉션의 경우에는 **저장 순서가 유지되지 않는다**. 그렇기에 Set컬렉션은 순서 자체가 없으므로 인덱스로 객체를 검색해서 가져오는 get(index) 메서드도 없다. 대신 전체 객체를 대상으로 한 번씩 반복해서 가져오는 반복자(Iterator)를 제공한다.

반복자(Iterator)는 iterator() 메소드를 호출하면 얻을 수 있다.



즉, Set 인터페이스를 구현한 모든 Set 컬렉션 클래스는 다음과 같은 특징을 가진다.

1. 요소의 **저장 순서가 없다**.

2. 같은 요소의 **중복 저장을 비허용**한다.

| **메서드** | **설명** |
| --- | --- |
| boolean add(E e) | 주어진 객체를 저장 후 성공적이면 true를 중복 객체면 false를 리턴합니다. |
| boolean contains(Object o) | 주어진 객체가 저장되어있는지 여부를 리턴합니다. |
| Iterator<E> iterator() | 저장된 객체를 한번씩 가져오는 반복자를 리턴합니다. |
| isEmpty() | 컬렉션이 비어있는지 조사합니다. |
| int Size() | 저장되어 있는 전체 객체 수를 리턴합니다. |
| void clear() | 저장된 모든 객체를 삭제합니다. |
| boolean remove(Object o) | 주어진 객체를 삭제합니다. |

#### **Hashset 클래스**

**Set 인터페이스에서 지원하는 구현 클래스**이다. 순서대로 입력되지 않고, 일정하게 유지되지 않는게 특징이다. HashSet은 **null 요소도 허용**하나, **중복은 비허용**한다.

| **Set**<데이터타입> 변수명 = new **HashSet**<데이터타입>(); |
| --- |

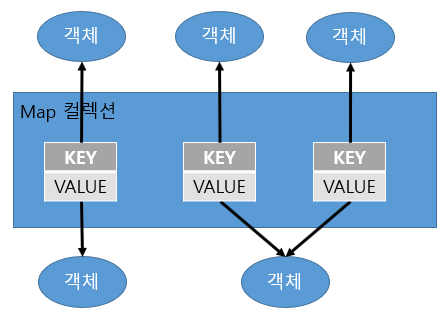
HashSet은 객체를 저장하기 전에 먼저 객체의 hashCode()메소드를 호출해서 해시 코드를 얻어낸 다음 저장되어 있는 객체들의 해시 코드와 비교한 뒤 같은 해시 코드가 있다면 다시 equals() 메소드로 두 객체를 비교해서 true가 나오면 동일한 객체로 판단하고 중복 저장을 하지 않는다.

| **Set**<Integer> set = new **HashSet**<Integer>();  **Set**<String> set2 = new **HashSet**<String>(); |
| --- |

| **실제 표현** | **예문** | |
| --- | --- | --- |
| **값 추가**  - **.add**(value) | set **.add**(1);  set2 **.add**("python");  set2 **.add**("null"); // null 값도 add 가능 | |
| **값 삭제**  - **.remove**(value)  - **.removeAll**(Collection)  - **.clear**() | set **.remove**("1"); // 값 1을 찾아서 제거  set**.removeAll**(test); // list에서 test와 겹치는 값 제거  set**.clear**(); //모든 값 제거 | |
| **크기 구하기**  - **.size**() | list **.size**()  // set 크기 : 3  // 중복값은 자동으로 제거된다. | |
| **값 조회**  - **Iterator** 사용 | **Iterator**<String> ir = set.**Iterator**();  while(ir.**hasNex**t()){ // hasNext 다음 값이 있는지 없으면 false  String str = ir.**next**(); // 다음값이 어떤건지 알려준다.  if(ir.**next**().equals("java")) {  System.out.println(str);  }  } | |

### **- [ Map 컬렉션 ]**

**키(key)와 값(value)으로 구성된 객체**를 저장하는 구조를 가지고 있는 자료구조입니다. **키는 중복으로 저장할 수 없고 값은 중복으로 저장할 수 있으며** 중복된 key값이 들어온다면 기존의 값은 없어지고 **새로운 값으로 대치**됩니다.



즉, Map 인터페이스를 구현한 모든 Map 컬렉션 클래스는 다음과 같은 특징을 가진다.

1. 요소로 **Key**와 **Value** 를 가진다.

2. 순서 대신 **Key로 데이터를 관리**한다. (Key로 Value를 알아낼 수 있다.)

3. Key는 중복이 안 된다. 마지막에 들어간 게 덮어쓴다.

| **메소드** | **설명** |
| --- | --- |
| V put(K Key, V value) | 주어진 키와 값을 추가하여 저장되면 값을 리턴합니다. |
| boolean containsKey(Object Key) | 주어진 키가 있는지 확인합니다. |
| boolean containsValue(Object value) | 주어진 값이 있는지 확인합니다. |
| Set<Map.Entry<K,V>> entrySet() | 모든 Map.Entry 객체를 Set에 담아 리턴합니다. |
| Set<K> keySet() | 모든 키를 Set객체에 담아서 리턴합니다. |
| V get(Object key) | 주어진 키에 있는 값을 리턴합니다. |
| boolean isEmpty() | 컬렉션이 비어있는지 조사합니다. |
| int Size() | 저장되어 있는 전체 객체의 수를 리턴합니다. |
| Collection<V> values() | 저장된 모든 값을 Collection에 담아서 리턴합니다. |
| void clear() | 저장된 모든 Map.Entry를 삭제합니다. |
| V remove(Object Key) | 주어진 키와 일치하는 Map.Entry를 삭제하고 값을 리턴합니다. |

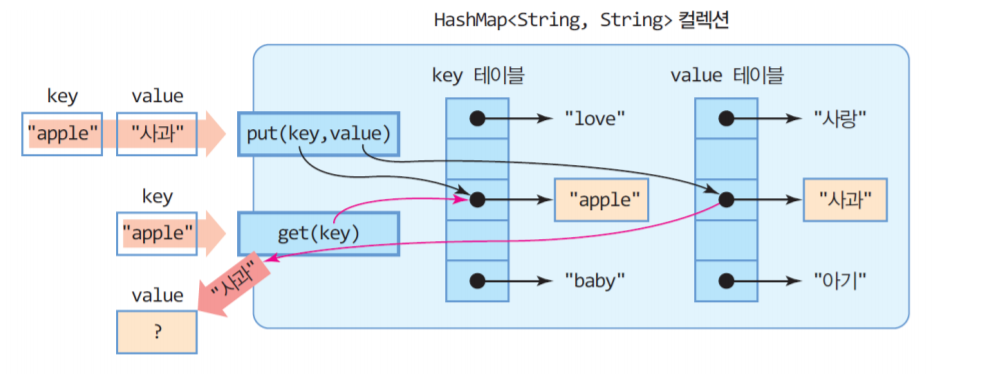
#### 

#### **HashMap 클래스**

**Map 인터페이스를 구현**한다. 키와 값으로 구성된 Entry객체를 저장하는 구조를 가지고 있는 자료구조이다. 여기서 키와 값은 모두 객체이다. **값은 중복 저장될 수 있지만 키는 중복 저장될 수 없다**.

| **Map**<데이터타입,데이터타입> 변수명 = new **HashMap**<데이터타입,데이터타입>() |
| --- |

만약 기존에 저장된 키와 동일한 키로 값을 저장하면 기존의 값은 없어지고 새로운 값으로 대치된다. 많은 양의 데이터를 검색하는 데 있어서 뛰어난 성능을 보인다.



HashMap은 해시 함수를 통해 '키'와 '값'이 저장되는 위치를 결정하므로, 사용자는 그 위치를 알 수 없고, 삽입되는 순서와 들어 있는 위치 또한 관계가 없다.

| **Map**<String,String> map1 = new **Hash Map**<String,String>();  // HashMap 생성  **Map**<String,String> map2 = new **Hash Map**<>;  // new에서 타입 파라미터 생략가능  **Map**<String,String> map3 = new **Hash Map**<>(map1);  // map1의 모든 값을 가진 HashMap생성  **Map**<String,String> map4 = new **Hash Map**<String,String>(){{  put( " a " , " b " );  }}; // 초기값 지정 |
| --- |

| **실제 표현** | **예문** | |
| --- | --- | --- |
| **값 추가**  - **.put**(key, value) | **Map**<Integer,String> map = new **Hash** <>();  map .put(1,"사과"); // 값 추가  map .put(2,"바나나");  map .put(3,"포도"); | |
| **값 삭제**  - **.remove**(key)  - **.clear**() | map **.remove**("1"); // 키값 1을 찾아서 제거  map **.clear**(); //모든 값 제거 | |
| **크기 구하기**  - **.size**() | map **.size**()  // set 크기 : 3  // 중복값은 자동으로 제거된다. | |
| **값 가져오기**  - **.get**(key) | map **.get**(1)  // "사과" | |
| **내용 변경하기**  - **.replace**(key, value) | map **.replace**(1, value);  // key값을 찾아서 value로 바꿔준다 | |
| **특정 Key, Value**  **들었는지 확인**  - **.containsKey**(key)  - **.containsValue**(value) | map **.containsKey**(1);  // key값이 있으면 T  map **.containsValue**("사과");  // value값이 있으면 T | |
| **크기가 0인지 확인**  - **.isEmpty**() | map **.isEmpty**()  // 크기가 0이면 T | |
| **Key가 있으면 Value**  **없으면 default값 가져오기**  - **.getOrDefault**(key, default); | 찾는 key가 존재하면 해당 key에 매핑되어 있는 값을 반환하고, 그렇지 않으면 디폴트 값이 반환된다. | |
| **값 순환**  - **Iterator** 사용 | **Iterator**<Integer> **irKey** = map **.keySet()** .**Iterator**();  while(irKey .**hasNex**t()){  // hasNext 다음 값이 있는지 없으면 false  String str = **irKey** .**next**();  } // value값을 전부 출력한다 | |
| **Set으로 반환**  - **.entrySet**() | System.out.println(strMap.**entrySet**());  // [1=java3, 2=java2, 3=java1]  System.out.println(strMap);  // {1=java3, 2=java2, 3=java1}  strMap.**entrySet**().forEach(arg -> { System.out.println(arg.getKey());  System.out.println(arg.getValue());  }); | |

## **JSON**

JavaScript Object Notation의 약자이며 자바스크립트의 객체 형식을 기반으로 만들어졌다. 인터넷과 통신 네트워크에서 데이터를 전달할 때 사용되는 데이터 형식이다

| **JsonObject** jsonObject = new **JsonObject**();  **Gson** gson = new **Gson**(); |
| --- |

JSON은 특별한 통신 문법이나 필요한 스펙 그리고 라이브러리를 설치해야 되는게 아니며 단지 데이터를 표현하는 방식일 뿐이다. JSON 데이터 형식은 순수하게 데이터만 표현하는 형식이라 심플하고 가벼워서 최근 대부분의 송신과 수신 간에 데이터 교환 시 JSON 형식이 많이 채택되고 있다

| public class **Student** {  private int id;  private String name;  public **Student** (int id, String name) {  this.id = id;  this.name = name;  }  @Override  public String toString() {  return "**Student**  [id=" + id + ", name=" + name + "]";  }  }  **Student** student = new **Student** (1, "Anna");  **Gson** gson = new **Gson**(); |
| --- |

| **메소드** | **설명** | |
| --- | --- | --- |
| **.addProperty**(key, value);  key, value 추가 | jsonObject **.addProperty**("id", 1); | |

### **- [ Gson ]**

Gson은 Java에서 Json을 파싱하고, 생성하기 위해 사용되는 구글에서 개발한 오픈소스이다. Java Object를 Json 문자열로 변환할 수 있고, Json 문자열을 Java Object로 변환할 수 있다.

| 아파치 메이븐(mvn) 에서 Gson 검색하여 설치할 수 있다.  <dependency>  <groupId>com.google.code.gson</groupId>  <artifactId>gson</artifactId>  <version>2.8.7</version>  </dependency> |
| --- |

Gson() 생성자를 통해 생성할 수 있다.

| **Gson** gson1 = new **Gson**(); // 출력될 때 비교적 덜 예쁘다  **Gson** gson2 = new **GsonBuilder**() ~ **.create(**);  // 몇 가지 옵션을 겸해서 생성하는 방법. create가 함께 사용되어야 한다. |
| --- |

| **메소드** | **설명** | |
| --- | --- | --- |
| **.addProperty**(key, value);  key, value 추가 | **Gson** gson= new **GsonBuilder**() **.create(**);  **JsonObject** jsonObject = new **JsonObject**();  jsonObject **.addProperty**("id", 1);  String json = gson.**toJson**(jsonObject); | |
| **.serializeNulls**()  null인 field도 Json의 property로 생성 | **Student** student **=** new **Student**(1, null);  **1. default**  **값이 null인 field는 Json에 포함시키지 않는다.**  **Gson** gsonWithoutNull = new **GsonBuilder**() **.create(**);  String studentJson = gsonWithoutNull .**toJson**(student);  System.out.println(studentJson );  // {"id":1}  **2. serializeNulls 옵션**  **값이 null인 field도, Json에 포함시킨다.**  **Gson** gsonWithoutNull = new **GsonBuilder**().**serializeNulls** ()**.create(**);  String studentJson = gsonWithoutNull .**toJson**(student);  System.out.println(studentJson);  // { "id" : 1, "name" : null } | |
| **transient**  Json 문자열 생성시 특정 필드를 제외하고 싶을 떄 사용한다 | public class **Student** {  private **transient** int id;  private String name;  public **Student** (int id, String name) {  this.id = id;  this.name = name;  }  @Override  public String toString() {  return "**Student**  [id=" + id + ", name=" + name + "]";  }  }  **Student** student **=** new **Student**(1, Anna);  **Gson** gson= new **Gson(**);  String studentJson = gson.**toJson**(student);  System.out.println(studentJson);  // {"name":"Anna"}  // id인 1은 표시되지 않는다. | |
| **.toJson**()  객체를 Json 문자열  로 변환 | String jsonStr= gson .**toJson**(student);  출력: System.out.println(jsonStr);  // {"id":1,"name":"Anna"} | |
| **.fromJson**(Json 문자열, 변환할 객체 클래스)  객체를 Json 문자열  로 변환 | **Student** student = gson **.fromJson**(jsonStr, **Student** .class)  출력: System.out.println(student);  // Student [id=1, name=Anna] | |

### **- [ Gson ~ Map ]**

| **Map**<String, String> map = new **HashMap**<>();  map.put("id", "1");  map.put("name", "Anna"); |
| --- |

Map을 JSON 문자열로 변환한다. Object를 문자열로 변환하는 방법과 동일.

| **메소드** | **설명** | |
| --- | --- | --- |
| **.toJson**()  객체를 Json 문자열  로 변환 | **Gson** gson = new **Gson** ();  String jsonStr= gson .**toJson**(map);  출력: System.out.println(jsonStr);  // { "id":1, "name":"Anna" } | |

JSON 문자열을 Map으로 변환한다. Json 문자열을 Object로 변환하는 방법과 같다.

| **메소드** | **설명** | |
| --- | --- | --- |
| **.fromJson**(Json 문자열, 변환할 객체 클래스)  객체를 Json 문자열  로 변환 | String jsonStr = { "id":1, "name":"Anna" }  **Gson** gson = new **Gson** ();  **Map**<String, String> map = gson .**fromJson**(jsonStr , **Map**.class);  출력:  for (Map.Entry<String, Object> entry : map.entrySet()) {  System.out.println(entry.getKey() + "=" + entry.getValue());  }  // id=1  // name=Anna | |

### **- [ @Expose Annotation ]**

객체가 Json 문자열로 변환되거나(Serialized), Json 문자열이 객체로 변환될 때(Deserialized) Gson의 **@Expose** 어노테이션을 이용하여 **특정 필드가 포함될지, 포함되지 않을지** 결정 할 수 있다.

| **@Expose** | **설명** | |
| --- | --- | --- |
| @**Expose**(serialize = true); | 특정 필드의 serialize 값이 true이면, 객체가 Json 문자열로 변환될 때, 해당 필드가 포함됩니다. | |
| @**Expose**(serialize = false); | 특정 필드의 serialize 값이 false이면, 객체가 Json 문자열로 변환될 때, 해당 필드가 제외됩니다. | |
| @**Expose**(deserialize = true); | 특정 필드의 deserialize 값이 true이면, Json 문자열이 객체로 변환될 때, 필드에 값이 세팅됩니다. | |
| @**Expose**(deserialize = false); | 특정 필드의 deserialize 값이 false이면, Json 문자열이 객체로 변환될 때, 필드에 값이 세팅되지 않습니다. | |
| @**Expose**(serialize = true , deserialize = false); | @Expose 어노테이션은, excludeFieldsWithoutExposeAnnotation 옵션이 적용된 Gson 객체로,Json을 생성하거나(toJson()), 객체를 생성할 때만(fromJson()) 유효합니다. | |
| @**Expose**(serialize = false, deserialize = true); | excludeFieldsWithoutExposeAnnotation 옵션이 적용된 Gson 객체로,Json을 생성하거나(toJson()), 객체를 생성할 때(fromJson())@Expose 어노테이션이 없으면, 해당 필드는 무시됩니다. | |

| **예시** | |
| --- | --- |
| public class **Student** {  **@Expose(serialize = false, deserialize = true)**  private int id;  **@Expose(serialize = true, deserialize = false)**  private String name;  private String name;  public **Student** (int id, String name, String major) {  this.id = id;  this.name = name;  this.major = major;  }  @Override  public String toString() {  return "**Student**  [id=" + id + ", name=" + name + "]";  }  } | |
| **@Expose(serialize = false, deserialize = true) private int id;**  **'id' field**는  객체 -> Json 변환시에는 제외되고,  Json -> 객체 변환시에는 포함됩니다.    **@Expose(serialize = true, deserialize = false) private String name;**  **'name' field**는  객체 -> Json 변환시에는 포함되고,  Json -> 객체 변환시에는 제외됩니다.    private String major;  **@Expose 어노테이션이 명시되지 않은 'major' field**는,  객체 -> Json 변환시에도 제외되고,  Json -> 객체 변환시에도 제외됩니다. | |

#### **객체 -> Json 변환**

| **Gson** gson = new GsonBuilder()**.excludeFieldsWithoutExposeAnnotation()**.create(); |
| --- |

@Expose 어노테이션을 적용하기 위해서 **excludeFieldsWithoutExposeAnnotation** 옵션을 적용하여 Gson 객체를 생성한다.

| import com.google.gson.Gson;  import com.google.gson.GsonBuilder;  public class **ExcludeExposeSerialize** {  public static void main(String[] args) {  **Student** student= new **Student** (1, "Anna", "CS"); // Student 객체 생성  // Gson 객체 생성  **Gson** gson = new GsonBuilder()**.excludeFieldsWithoutExposeAnnotation()**.create();  String studentJson = gson .**toJson**(student); // Student 객체 -> Json 문자열    System.out.println(studentJson ); // Json 문자열 출력  // { "name" : "Anna" }  }  } |
| --- |

객체 -> Json으로 변환하는 경우

**serialize가 true로 선언**된 'name' 필드만 Json 문자열에 포함된다.

**serialize가 false로 선언**된 'id' 필드와

**@Expose 어노테이션이 없는** 'major' 필드는 제외된다.

#### **Json -> 객체 변환**

| **Gson** gson = new GsonBuilder()**.excludeFieldsWithoutExposeAnnotation()**.create(); |
| --- |

@Expose 어노테이션을 적용하기 위해서 **excludeFieldsWithoutExposeAnnotation** 옵션을 적용하여 Gson 객체를 생성한다.

| import com.google.gson.Gson;  import com.google.gson.GsonBuilder;  public class **ExcludeExposeSerialize** {  public static void main(String[] args) {  String jsonStr = "{ "id":1 , "name" : "Anna" , "major" : "CS" }"; // Json 문자열  // Gson 객체 생성  **Gson** gson = new GsonBuilder()**.excludeFieldsWithoutExposeAnnotation()**.create();  // Json 문자열 -> Student 객체  String student = gson .**fromJson**(jsonStr.**Student.**class);    System.out.println(student); // Json 문자열 출력  // Student [id=1, major=null, name=null]  }  } |
| --- |

Json -> 객체로 변환하는 경우

**deserialize가 true로 선언**된 'id' 필드만 객체 필드에 값이 세팅되었습니다.

**deserialize가 false로 선언**된 'name' 필드와

**@Expose 어노테이션이 없는** 'major' 필드는 값이 세팅되지 않았습니다.

## **익명클래스**

프로그램에서 **일시적으로 한번만 사용되고 버려지는 객체**이다. 일시적으로 사용된다는 것은 **나중에 재사용이 되지 않는다는 것**이며, 재사용이 될 필요가 없다는 뜻은 확장성이 그렇게 좋지 못하다는 뜻이다.

1. 프로그램 내에서 일시적으로(단발성으로) 한번만 사용되어야 하는 객체일 경우

-> UI 이벤트처리, 스레드 객체 등 (단발성 이벤트 처리)

2. 재사용성이 없고, 확장성을 활용하는 것이 유지보수에서 더 불리할 때

-> 비즈니스 로직이 정말 재각각이며, 재사용성이 전혀없어 매번 클래스를 생성해야하는 비용이 더 많을때

클래스 파일을 따로 만들지 않고 일회용으로 { }에서만 사용된다.

| **익명 클래스** | **일반 클래스** | |
| --- | --- | --- |
| **Test** t1 = new **Test**(){  public int num = 10;  @Override  public int getNum(){  return this.num;  }  }; | public class **Test** {  private int num = 1;  public int getNum(){  return this.num;  }  public void setNum(int num){  this.num = num;  }  }  **Test** t1 = new **Test** (); | |

### **- [ 람다식 ]**

람다 표현식(lambda expression)이란 "선언없이 실행가능한 함수" 이다. 간단히 말해 코드한줄에 함수를 써서 그것을 호출하는 방식으로 표현했다. 자바 8부터 람다식(Lambda Expressions)을 지원하며, 간결한 코딩만큼 가독성 면에서 장점을 가진다.

| **장점** | |
| --- | --- |
| 1. 불필요한 **코드를 제거하여 간결**하게 만들 수 있다.  2. 코드가 간결하고 식에 개발자의 의도가 명확히 드러나므로 **가독성이 향상**된다.  3. 함수를 만드는 과정없이 한번에 처리할 수 있기에 코딩시간이 단축된다.  4. 다중 cpu를 활용하는 형태로 구현되어 병렬 처리에 유리하다. | |

| **단점** | |
| --- | --- |
| 1. 람다를 사용하면서 만드는 무명함수는 **재사용이 불가능**하다.  2. 디버깅 시 함수 콜 스택 추적이 다소 어려움.  3. 재귀로 만들어 완전탐색하는 경우, 느릴수 있다.  4. 한 클래스내에 많은 람다식을 사용하는 경우, 오히려 가독성이 떨어진다. | |

1. 자바에서는 **화살표(->) 기호를 사용**하여 **매개변수부와 선언부**를 나눈다.

| ( 매개변수, … ) -> { 실행문 } | |
| --- | --- |
| **public** int sum ( int a, int b) {  return a + b  } | // 왼쪽과 동일한 익명 클래스  **new Object** {  int sum ( int a, int b) {  return a + b  }  } |
| // 왼쪽과 동일한 람다식(익명함수)  ( a, b ) -> a + b |

2. 람다식 사용을 위해서는, **함수형 인터페이스에 접근**하여 사용된다. (함수형 인터페이스란 한개의 추상메소드를 가지는 인터페이스로, 동적인 함수구현후 사용하기 위해 정의한다.)

| **@FunctionalInterface 함수형 인터페이스 선언** | |
| --- | --- |
| **@FunctionalInterface** // @FunctionalInterface 어노테이션을 반드시 명시하여 정의함  **interface Calc** { // 함수형 인터페이스의 선언  public int min(int x, int y);  } | |

람다형 인터페이스는 메소드가 하나여야한다.

@FunctionalInterface 는 메소드가 하나가 아닐 경우 오류가 뜨게 한다.

| **함수형 인터페이스에 접근하여 람다식 사용** | |
| --- | --- |
| **public class** LambdaExam1 {  **public static void** main(String[] args){  **Calc minNum** = ( x, y ) -> x < y ? x : y; // 추상 메소드의 구현  // 무조건 리턴값이면 중괄호와 리턴 생략 가능하다.  // 중괄호가 있다면 로직이 있다고 생각해야 함.. 리턴 꼭 있어야한다.  System.out.println(**minNum**.min(3, 4)); // 함수형 인터페이스의 사용  }  } | |

3. 매개변수의 타입을 추론할 수 있는 경우에는 타입을 생략할 수 있다.

4. 매개변수 작성부에서 변수가 하나인 경우에는 괄호(())를 생략할 수 있습니다.

| ( a, b ) -> a + b | 가능  함수의 몸체가 **하나의 명령문만으로**  **이루어진 경우**에는 **중괄호({})를 생략**할 수 있다.  (이때 세미콜론(;)은 붙이지 않음) |
| --- | --- |
| ( a, b ) -> { return a > b ? a : b } | 가능 |
| ( a, b ) -> return a > b ? a : b | 에러  함수의 몸체에 return 문이 있는 경우에는  중괄호({})를 생략할 수 없다. |
| ( a, b ) -> a > b ? a : b | 가능  return 문 대신 표현식을 사용할 수 있으며,  이때 반환값은 표현식의 결과값이 된다.  (이때 세미콜론(;)은 붙이지 않음) |

#### **함수형 인터페이스**

람다식은 익명 객체라고 했다. 익명 객체는 메소드의 인자로 넘겨 줄 수도 있고, 메소드의 리턴 값으로 넘겨 받을 수도 있다. 따라서 **익명 객체인 람다를 다룰 수 있는 인터페이스**가 필요하다.

람다식을 저장할 수 있는 변수는 '함수형 인터페이스(functional interface)' 타입이어야 한다.

| **FunctionalInterface** myLambda = (a, b) -> a + b; |
| --- |

**FunctionalInterface는 하나의 추상 메소드만을 갖는 인터페이스다**. 그 추상 메소드의 시그니처(매개변수 개수와 타입, 리턴타입)와 동일한 시그니처를 갖는 람다함수를 할당해서 사용할 수 있다. (Functional Interface의 static 메소드와 default 메소드의 개수에는 제약이 없다)

함수형 인터페이스를 정의하고 '@FunctionalInterface' 애너테이션을 붙여주면 자바 컴파일러가 함수형 인터페이스의 정의를 검증해준다.

| **@FunctinalInterface**  **interface** MySum {  public int sum(int a, int b);  } |
| --- |

이런 인터페이스를 정의하면 두 숫자를 더하는 람다를 다음과 같이 사용할 수 있다.

| public static void main(String []args) {  MySum func = (a, b) -> a + b;  System.out.println(func.sum(10, 11));  } |
| --- |

람다를 이용해서 짧게 작성할 수 있는 코드 중에 가장 흔한 종류가 **컬렉션과 같이 사용**하는 경우다. **List<> 인터페이스에 추가된 forEach() 메소드**를 이용하면 리스트 컬렉션에 들어있는 각 엘리먼트들의 내용을 입력받은 람다식을 수행할 수 있다.

| List<String> list = new ArrayList();  list.add("Element1");  list.add("Element2");  list.forEach(x -> System.out.println(x)) // = list.forEach(System.out::println) 으로 축약가능 |
| --- |

리스트의 system.out.println()을 호출하는 코드가 굉장히 간단해졌다.

#### **기본 함수형**

가장 **기본적인 함수형 인터페이스**는 다음과 같다.

= 파라미터가 없거나 하나 (T: 데이터 타입, R: 리턴 타입 [참조URL](https://inma.tistory.com/151))

| **함수형 인터페이스** | **메서드** | **매개변수** | **반환값** |
| --- | --- | --- | --- |
| java.lang.Runnable | void run() | X | X |
| Supplier<T> | T get() | X | O |
| Consumer<T> | void accept(T t) | O | X |
| Function<T, R> | R apply(T t) | O | O |
| Predicate<T> | boolean test(T t) | O | O |

| **메소드** | **설명** | |
| --- | --- | --- |
| **Runnable**  매개변수X 반환X | **Runnable** runnable = () -> {  System.out.println("runnable 인터페 활용");  System.out.println("여러 실행문을 작성가능.");  }; | |
| **Supplier**<T>  ~ .**get**  매개변수X 반환O | **Supplier**<Integer> supplier = () -> 100;  **Supplier**<Integer> supplier2 = () -> {  System.out.println("다른 실행문도 작성 가능");  return 200;  };  System.out.println(supplier .**get**()); // 100  System.out.println(supplier2 .**get**());  // 다른 실행문도 작성 가능 // 200 | |
| **Consumer**<T>  ~ .**accept**  매개변수O 반환X | **Consumer**<String> consumer = (name) -> {  System.out.println("이름: " + name);  };  consumer .**accept**("김준일"); // 이름: 김준일 | |
| **Function**<Integer, Integer>  ~ .**apply**  매개변수O 변수O | **Function**<Integer, Integer> function = (num) -> num\*num;  System.out.println(function.**apply**(10)); | |
| **Predicate**<String>  ~ .**test**  매개변수O 변수O | **Predicate**<String> predicate = (str) -> str == null || str.isBlank();  System.out.println(predicate.**test**(""));  // true | |

#### **파라미터가 두 개**

**파라미터가 두개인 함수형 인터페이스**는 다음과 같다. (T/U: 데이터 타입, R: 리턴 타입)

| **함수형 인터페이스** | **메서드** | **매개변수** | **반환값** |
| --- | --- | --- | --- |
| **BiConsumer**<T, U> | void **accept**(T t, U u) | O | X |
| **BiFunction**<T, U, R> | R **apply**(T t, U u) | O | O |
| **BiPredicate**<T, U> | boolean **test**(T t, U u) | O | O (boolean) |

| // BiConsumer  **BiConsumer**<Integer, Integer> biConsumer = (num1, num2) -> System.out.println(num1 + num2);  biConsumer .**accept**(1, 2); // 3  // BiFunction  **BiFunction**<Integer, Integer> biFunction = (num1, num2) -> num1 + num2;  int result1 = biFunction .**apply**(1, 2);  System.out.println(result1 ); // 3  // BiPredicate  **BiPredicate**<Integer, Integer> biPredicate = (num1, num2) -> num1 > num2;  boolean result2 = biPredicate .**test**(1, 2);  System.out.println(result2 ); // false |
| --- |

#### **파라미터 타입이 반환 타입과 일치**

**파라미터 타입이 반환타입과 일치**하는 인터페이스

| **함수형 인터페이스** | **메서드** | **매개변수** | **반환값** |
| --- | --- | --- | --- |
| **UnaryOperator**<T> | T **apply**(T t) | O | O |
| **BinaryOperator**<T> | T **apply**(T t, T t) | O | O |

| // UnaryOperator  **UnaryOperator**<Integer> unaryOperator = (num) -> num;  int result1 = unaryOperator .**apply**(1);  System.out.println(result1 ); // 1  // BinaryOperator  **BinaryOperator**<Integer> binaryOperator = (num1, num2) -> num1 + num2;  int result2 = binaryOperator .**apply**(1, 2);  System.out.println(result2 ); // 3 |
| --- |

### **- [ forEach 메서드 ]**

foreach 반복문은 오직 **Array 객체에서만 사용가능**한 메서드이다. (ES6부터는 Map,Set 지원)

| **for** (type var : iterate) { // iterate 부분에 들어가는 타입은 Array 혹은 Collections  body-of-loop // 반복되는 내용  } |
| --- |

배열의 요소들을 **반복하여 작업을 수행**할수 있다. foreach구문의 인자로 callback함수를 등록할수 있고, 배열의 각 요소들이 반복될 때 이 callback 함수가 호출됩니다. callback 함수에서 배열요소의 인덱스와 값에 접근할수 있다.

배열의 첫 번째부터 마지막까지 반복하면서 item을 꺼낼수 있다.

#### 

| **예문** | **해당 for문과 같다.** |
| --- | --- |
| String[] temp = { "aa", "bb", "cc" };  **for** (String el : temp) {  System.out.println(el);  } | String[] temp = { "aa", "bb", "cc" };  **for** (int i = 0; i < temp .length; i++) {  System.out.println(temp[i]);  } |

#### 

| **asList 활용 예문** | |
| --- | --- |
| **Arrays.asList**(roleArray)**.forEach**(arg -> {  System.out.println("role: " + arg);  }); | |

#### 

| **Set 활용 예문** | |
| --- | --- |
| **Set**<String> strSet = new **HashSet**<String>();   | for(int i =0; i<26; i++) {  strSet .add(Character.toString((char) (97 + i)));  } | | --- | | for(String str : strSet) {  System.out.println(str );  } | | strSet.**forEach**(str -> {System.out.println(str );}); | | |

#### 

| **Map 활용 예문** | |
| --- | --- |
| **Map**<String, Integer> map = new **HashMap**<String, Integer>();   | for(int i =0; i<26; i++) {  map .put(Character.toString((char) (97 + i)), 97 + i);  } | | --- | | map .**forEach**((k, v) -> {  System.out.print("key: " + k);  System.out.print(", ");  System.out.println("value: " + v);  }); | | |

#### 

## **예외처리**

예외가 발생하면 프로그램을 중단하고 오류메시지를 보여준다. 오류가 날 때 그에 맞는 적절한 처리를 하고 싶을 때, try ... catch, throw 구문을 이용해 오류를 처리 한다.

| **try** {  ...  } **catch**(예외1) {  ...  } **catch**(예외2) {  …  } **finally** { // 이건 예외에 대한 소스코드이다라는 명시용.  … // 어떤 예외가 발생하던 말던 반드시 실행한다  } |
| --- |
| try {  if(1 == 1) throw new BadAttributeValueExpException(null);  // throw는 예외를 강제적으로 생성    System.out.println("예외 이후에 정의된 실행문"); // 예외 발생시 실행되지 않음  } **catch**( **ArrayIndexOutOfBoundsException** e ) {  System.out.println( "배열의 범위를 초과하여 참조하였습니다");  } **catch** ( **Exception** e ) { //모든 예외를 다 받습니다.  System.out.println("무슨 예외인지는 모르겠지만 예외가 발생하면 무조건 처리");  **e.printStackTrace();** // 무슨 로그인지 알려줌  } **finally**  {  System.out.println("무조건 실행"); // 보통 실행된 부분까지의 저장을 알려줌.  } |

#### 

finally 블록 : 예외가 발생하건 발생하지 않건 **공통으로 수행되어야할 코드**가 쓰여진다. 임시 파일의 삭제 등 뒷정리 코드가 쓰인다.

| **Exeption 발생!** | Try 블록 수행 -> Catch 블록 수행 -> Finally 블록 수행 (생략가능) |
| --- | --- |
| **Exeption 미발생!** | Try 블록 수행 -> Finally 블록 수행 (생략가능) |

| **예외** | **설명** |
| --- | --- |
| NullPointerException | 실제 참조할 대상이 null인데 참조할때 발생하는 예외 |
| ClassNotFoundException | 해당 클래스가 존재하지 않으면 발생하는 일반 예외 |
| ArrayIndexOutOfBoundsException | 배열의 범위를 초과할 때 실행된다. |
| Exception | 모든 예외를 다 받는다 |

### **- [ throw new ]**

throw의 경우 개발자가 **의도적으로 예외를 발생**시키는 것이다. 주로 비즈니스 로직을 구현하는 과정 중 컴파일에는 문제가 없지만 해당 비지니스 로직이 개발자가 의도한 대로 통과하지 못했을 경우 고의로 예외를 발생시켜야 할 때 사용한다.

예외가 발생할 수 있는 코드가 있다는 것을 인지시키고 예외처리를 강요하며, 여러 가지 발생 가능한 예외들을 호출한 메서드에서 한 번에 처리할 수 있어 관리를 용이하게 해 줄 수 있다.

| try {  if(1 == 1) **throw new** BadAttributeValueExpException(null);  // throw는 예외를 강제적으로 생성  } **catch** ( **Exception** e ) { //모든 예외를 다 받습니다.  System.out.println("무슨 예외인지는 모르겠지만 예외가 발생하면 무조건 처리");  } **finally**  {  System.out.println("무조건 실행"); // 보통 실행된 부분까지의 저장을 알려줌.  } |
| --- |

throws 는 메서드 내에서 예외처리를 하지 않고 **해당 메서드를 사용한 곳**에서 예외 처리를 하도록 예외를 전가하는 것이다.

| **public void** printArray() **throws** Exception {  //throws Exception 예외가능성이 있음. 강제적으로 메인에서 예외처리 하게 해줌  int[] numArray = new int[] { 1 , 2 , 3 , 4 , 5 };  if(0==0) **throw new** CustomException(**"내가만든 예외 강제로 발생"**);    for(int i = 0; i < 6; i++) {  System.out.println(numArray[i]);  }  } |
| --- |

### **- [ CustomException ]**

사용자 정의 예외 클래스는 컴파일러가 체크하는 **일반 예외로 선언**할 수도 있고, 컴파일러가 체크하지 않는 **실행 예외로 선언**할 수도 있다. **일반 예외로 선언할 경우 Exception을 상속**하면 되고, **실행 예외로 선언할 경우에는 RuntimeException을 상속**하면 된다.

| **public class** CustomException extends **RuntimeException**{  1. 매개 변수가 없는 기본 생성자  CustomException () {  }  2. 예외 발생 원인(예외 메시지)을 전달하기 위해 String 타입의 매개변수를 갖는 생성자  CustomException(String message) {  super(message); // RuntimeException 클래스의 생성자를 호출합니다.  }  } |
| --- |

사용자 정의 예외 클래스 이름은 Exception으로 끝나는 것을 권장한다.

사용자 정의 예외 클래스 작성 시 생성자는 두 개를 선언하는 것이 일반적이다.

- 매개 변수가 없는 기본 생성자

- 예외 발생 원인(예외 메시지)을 전달하기 위해 String 타입의 매개변수를 갖는 생성자

### **- [ 예외정보 얻기 ]**

try 블록에서 예외가 발생되면 예외 객체는 catch 블록의 매개 변수에서 참조하게 되므로 매개 변수를 이용하면 예외 객체의 정보를 알 수 있다. **모든 예외 객체는 Exception 클래스를 상속**하기 때문에 Exception이 가지고 있는 메소드들을 모든 예외 객체에서 호출할 수 있다.

Exception는 Throwable를 상속받고 Throwable에 객체의 정보를 얻을 수 있는 메소드를 가지고 있다. 주로 자주 사용되는 메소드는 getMessage()와 printStackTrace()입니다.

| **public class** **Exception** extends **Throwable** {  public **Exception** () {  super();  }  public **Exception** (String message) {  super(message);  }  // (. . .) 생략  }  **public class** **Throwable** implements **Serializable** {  public String **getMessage()** {  return detailMessage;  }    public void **printStackTrace()** {  **printStackTrace**(System.err);  }    // (. . .) 생략  } |
| --- |

| **getMessage()** | String 타입의 매개변수 메시지를 갖는 생성자를 이용하였다면, 메시지는 자동적으로 예외 객체 내부에 저장되게 되는데 이 메시지를 리턴하는 함수이다.  예외 메시지의 내용에는 왜 예외가 발생했는지에 대한 간단한 설명이 포함된다. |
| --- | --- |
| **printStackTrace()** | 예외 발생 코드를 추적해서 모두 콘솔에 출력한다. 어떤 예외가 어디에서 발생했는지 상세하게 출력해주기 때문에 프로그램을 테스트하면서 오류를 찾을 때 활용한다. |

## **기본자료형\_형변환**

### **- [ 문자(열) ~ 숫자 ]**

Double, Float, Long, Short 등의 자료형도 valueOf 함수로 모두 변환이 가능하다.

| (Casting 하려는 자료형) **.valueOf**(Casting 할 Value) |
| --- |

| **형 변환** | **설명** |
| --- | --- |
| String > Int | String s = "12345";  int i = Integer.**parseInt**(s);  String s = "12345";  int i = Integer.**valueOf**(s); |
| Int > String | int i = 12345;  String s = Integer.**toString**(i);  int i = 12345;  String s = String.**valueOf**(i); |
| Char > Int | char ch = '5';  int i = **(int)**(ch - '0'); |
| Int > Char | int i = 5;  char ch = **(char)**(i + '0');  단순 형 변환을 할 경우 해당 아스키코드 값으로 출력된다. ex) 65->'A' / 'A'->65  원하는 값을 얻으려면 '0' (48) 에 해당하는 값을  더하거나 빼 주어야 한다. |

### **- [ 문자열 ~ 문자 ]**

| **형 변환** | **설명** |
| --- | --- |
| String > Char | char ch1 = '5';  char[] ch2 = {'a','b', 'c'};  String s1 = String.**valueOf**(ch1); // '5'  String s2 = String.**valueOf**(ch2); // 'abc' |
| Char > String | String s1 = "1";  String s2 = "1234";  char ch1 = s1.**charAt**(0); // '1'  char[] ch2 = s2.**toCharArray**(); // 1234 |

ch2는 Char형 '1','2','3','4'를 가지고 있는 Char 배열이므로 ch2[0], ch2[1], ch2[2]... 와 같이 접근이 가능하다.

s2의 length() 만큼 돌려서 Char를 하나씩 꺼내는것, s2.toCharArray()로 Char배열 만든 후 length 만큼 돌려서 하나씩 꺼내는 것 두가지 방법이 있다.

### **- [ 정수 ~ 실수 ]**

| **형 변환** | **설명** |
| --- | --- |
| Double, Float > Int  이 때, 소수점 뒷 자리들은 버려진다. (반올림 되지 않음.) | double d = 1010.10101010;  // double타입은 64비트로 실수를 표현  float f = 1010.101010f;  // float타입은 32비트로 실수를 표현,  // 리터럴에 f를 붙혀 실수임을 표기해야함.  int i;  i = (int) d; // Double > Int  i = (int) f; // Float > Int |
| Int > Double, Float  기본적으로 소수점 첫 째 자리까지 출력된다. (1234.0) | int i= 1234;    double d = (double)i; // Int To Double  float f = (float)i; // Int To Float |