쉽게 배우는 Java
(V1.6)

박용준(redplus@redplus.net)

twitter.com/RedNuke

데브렉(http://www.devlec.com/) 온라인 강의

자바 시작하기

자바 강의 환경

- Windows Server 2008
- Java JDK 6
- Eclipse Classic 3.6.2

Java 소개

- Java
  - Sun 마이크로시스템의 제임스 고슬링에 의해서 만들어진 프로그래밍 언어
    - C/C++/C#과 함께 가장 많이 사용되는 언어 중 하나
  - 1996년 Java 1.0 버전
    - 1991년 개발 시작
  - Sun은 최근 Oracle에 합병됨
    - 제임스 고슬링은 2011년 3월 오라클 퇴사 후 구글로 입사

Java 버전

Version	Release Date
Java 1.0	1996
Java 1.1	1997
Java 1.2	1998
Java 1.3	2000
Java 1.4	2002
Java 5	2004
Java 6	2006
Java 7	*Expected mid 2011*
Java 8	*Expected mid-late 2012*

Table 1: Java timeline

프로그래밍 언어 순위(?)

Position Sep 2011	Position Sep 2010	Delta in Position	Programming Language	Ratings Sep 2011	Delta Sep 2010	Status
1	1	=	Java	18.761%	+0.85%	A
2	2	=	C	18.002%	+0.86%	A
3	3	=	C++	8.849%	-0.96%	A
4	6	↑↑	C#	6.819%	+1.80%	A
5	4	↓	PHP	6.596%	-1.77%	A
6	8	↑↑	Objective-C	6.158%	+2.79%	A
7	5	↓↓	(Visual) Basic	4.420%	-1.38%	A
8	7	↓	Python	4.000%	-0.58%	A
9	9	=	Perl	2.472%	+0.03%	A
10	11	↑	JavaScript	1.469%	-0.20%	A
11	10	↓	Ruby	1.434%	-0.47%	A
12	12	=	Delphi/Object Pascal	1.313%	-0.27%	A
13	24	↑↑↑↑↑↑↑↑↑↑↑↑	Lua	1.154%	+0.60%	A
14	13	↓	Lisp	1.043%	-0.04%	A
15	15	=	Transact-SQL	0.860%	+0.09%	A
16	14	↓↓	Pascal	0.845%	+0.06%	A-
17	20	↑↑↑	PL/SQL	0.720%	+0.08%	A--
18	19	↑	Ada	0.682%	+0.01%	B
19	17	↓↓	RPG (OS/400)	0.666%	-0.05%	B
20	30	↑↑↑↑↑↑↑↑↑↑	D	0.609%	+0.20%	B

Java 특징

- 완벽한 객체지향 언어(Object-Oriented)
- C/C++에서 유래
  - 기본 구문은 동일
- 분산 / 네트워크 / 웹 / 모바일 개발 환경에 유리
- 여러 환경에서 동일하게 실행됨
  - Windows
  - UNIX
  - MAC

Java 플랫폼

- Java 개발 환경은 운영체제 위에 Java 가상 머신 안에서 실행

자바(Java) 공식 설명서

- http://download.oracle.com/javase/6/docs/api/index.html
- 추천 서적
  - 자바 기초 입문 서적 모두
    - 강의는 책의 활자를 좀더 동적으로 쉽게 보여줄 수 있는 매력이 있음
    - 강의에서 미처 설명하지 못한 부분은 책을 통해서 보충
- 능력 업그레이드
  - 블로그 운영
    - 소스를 블로그 아티클로 작성해서 설명...

프로그래밍 학습

- 타이핑 및 결과확인
- 그리고, 소스를 이해하려고 노력
  - 이해가 안되면 다음 예제로 패스
    - 이런 과정을 100회 이상했는데도 이해가 안되면 프로그래밍을 포기...

Java 개발 환경 구축

- JDK(자바 엔진)
  - JDK 1.7 : Java SE Development Kit 7 Downloads
    - JDK 1.6.X : 처음 강의에 사용한 버전
  - http://java.sun.com/
    - 오라클 사이트 이동
      - http://www.oracle.com/technetwork/java/index.html
        - » Java SE 6 Update 24
- Eclipse(개발도구), EditPlus, UltraEditor, …
  - http://www.eclipse.org
    - Eclipse Classic 3.7.1
      - Eclipse Classic 3.6.2 : 처음 강의에 사용한 버전

실습 : Java 개발 환경 구축

Hello, World 프로그램 => File, New

```
public class HelloWorld
{
  public static void main(String [] args)
  {
    System.out.println("Hello, World");
  }
}
```

실습 : FileNew 프로젝트

- C:₩Java₩FileNew
  - File – New – JavaProject
  - File – New – Class
    - 클래스명과 파일명은 동일해야 함
    - FileNew.java
      - 기본 코드
      - 기본 출력문
        - » System.out.println() 메소드(메서드)

Java 컴파일러와 인터프리터

- javac.exe
  - 자바 컴파일러
- java.exe
  - 자바 인터프리터

데모 : 콘솔 기반으로 Java 프로그램 작성

- C:₩Java₩Hello₩Hello.java

```
C:\temp>notepad.exe Hello.java

C:\temp>"C:\Program Files\Java\jdk1.6.0_24\bin\javac.exe" Hello.java

C:\temp>"C:\Program Files\Java\jdk1.6.0_24\bin\java.exe" Hello
안녕

C:\temp>
```

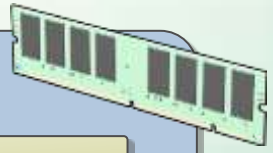
데이터 형식(DataType)
&
변수(Variable)

변수(Variable)

- 프로그래밍 언어에서의 변수는 프로그램 내에서 처리할 데이터를 저장하기 위한 메모리 상의 임시 데이터 저장 공간을 의미
- 변수를 생각함에 있어, 처리할 데이터를 잠시 보관해 놓는 그릇을 연상하면 됨
- 변수에는 종류에 따라서 여러 가지 데이터 유형(숫자 또는 문자 등)을 담아 놓을 수 있음
- 데이터 형식 = 주요 키워드
  - int
  - long
  - double
  - String
  - char
  - bool
  - 기타

변수의 필요 충분 조건

- 변수는 응용 프로그램에서 사용하는 메모리상의 임시 데이터 저장 공간

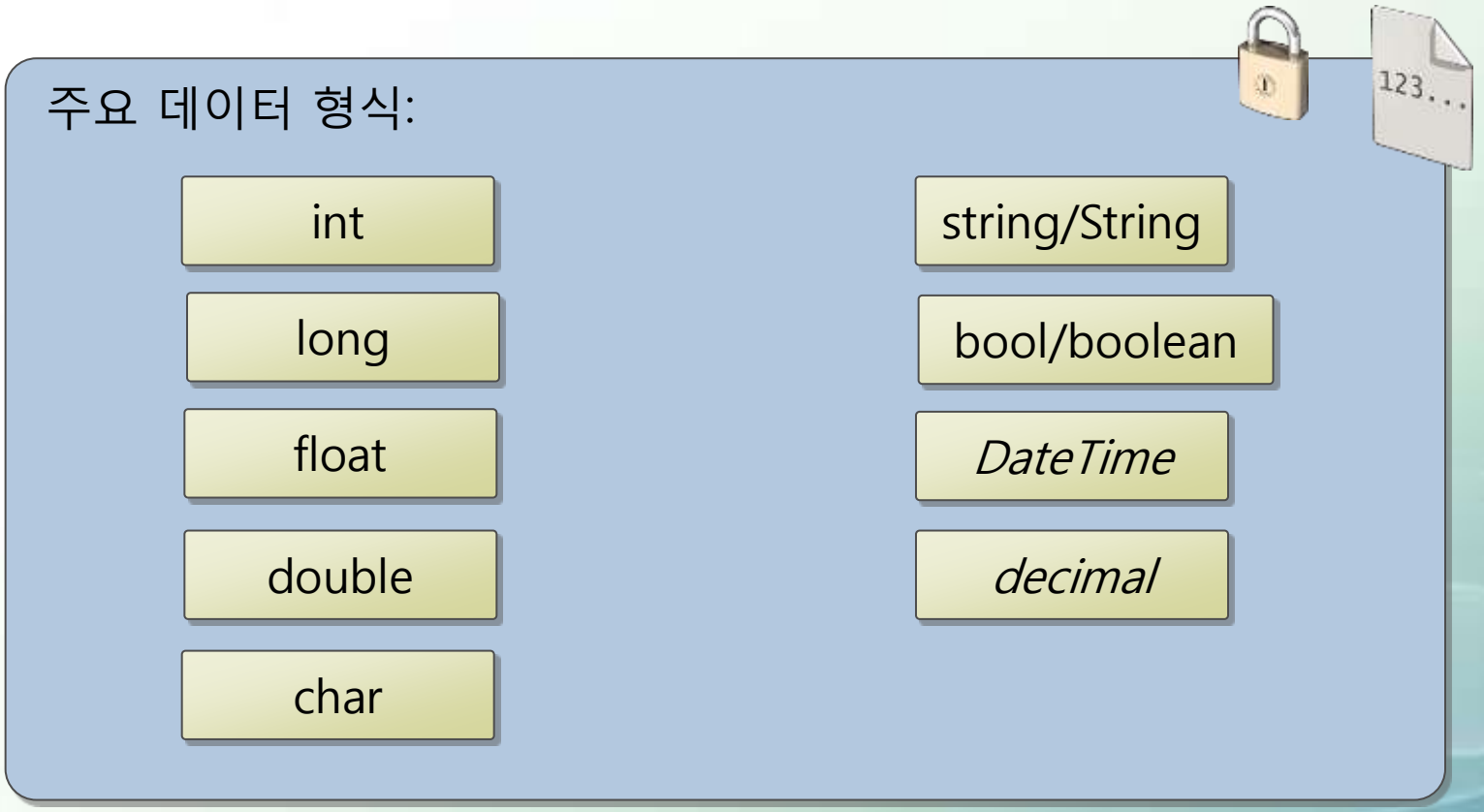


변수는 아래와 같은 특징을 가짐:

- Name
- Address
- Data type
- Value
- Scope
- Lifetime

데이터 형식(Data Type) : 타입, 유형, 형

- 변수는 반드시 적절한 데이터 형식을 설정하고 해당 데이터를 입력해야 함



지역 변수(Local Variables) 선언(Declaring)

- 데이터형과 변수명 사용해서 선언

```
int itemCount;
```

- 한번에 여러 개의 변수 선언 가능

```
int itemCount, employeeNumber;
```

변수를 사용하기전에는 반드시 초기화해야 함

```
DataType variableName;
…
DataType variableName1, variableName2;
…
DataType variableName = new DataType();
```

변수에 값 할당(Assigning)

- 선언된 변수에 값을 할당하는 방법

```
int employeeNumber;
employeeNumber = 23;
```

- 선언과 동시에 초기화

```
int employeeNumber = 23;
```

- 작은 따옴표를 사용한 문자 대입

```
char middleInitial = 'J';
```

변수를 선언한 후에는 값을 할당해야 함

```
variableName = Value;
...
DataType variableName = Value;
```

변수명 짓기의 규칙 및 추천방식

- Rules
  - 문자, 언더스코어(_), 숫자, 한글(유니코드) 사용 가능
- Recommendations
  - 모두 대문자 피할 것
  - 생략형 피할 것
  - 카멜/파스칼/헝가리언 표기법(PascalCasing naming) 사용 권장
    - employeeNumber : 카멜
    - employee_num : 파스칼
    - intEmployeeNum : 헝가리언

변수의 영역(Scope)

Block scope

```
if (length > 10)
{
    int area = length * length;
}
```

Procedure scope

```
void ShowName()
{
    string name = "RedPlus";
}
```

Module scope

```
private string message;

void SetString()
{
    message = "Hello World!";
}
```

Namespace/Package scope

```
public class CreateMessage
{
  public string message
    = "Hello";
}

public class DisplayMessage
{
   public void ShowMessage()
   {
       CreateMessage newMessage
         = new CreateMessage();
       MessageBox.Show(
         newMessage.message);

   }
}
```

데이터 형식 변환 : Casting / Converting

묵시적/암묵적 형변환(Implicit conversion)

큰 데이터형식에 작은 데이터 형식을 넣으면 자동으로 변환됨

```
int a = 4;
long b;
b = a;          // int 에서 long 으로 자동 변환
```

명시적 형변환(Explicit conversion)

타입이 맞지않을 경우에는 캐스팅 연산자를 사용하여 변환

```
DataType variableName1 = (castDataType) variableName2; // C Family
…
int count = Convert.ToInt32("1234"); // C#
...
int number = 0;
if (int.TryParse("1234", out number)) {// Conversion succeeded } // C#
```

변수(Variable)

- 프로그래밍 언어에서의 변수는 프로그램 내에서 처리할 데이터를 저장하기 위한 메모리 상의 임시 데이터 저장 공간을 의미
- 변수를 생각함에 있어, 처리할 데이터를 잠시 보관해 놓는 그릇을 연상하면 됨
- 변수에는 종류에 따라서 여러 가지 데이터 유형(숫자 또는 문자 등)을 담아 놓을 수 있음
- 하나의 이름으로 하나의 데이터 형식(Data Type)을 하나만 보관해 놓을 수 있는 그릇
  - 변수, 배열, 구조체, 클래스, …

데이터 형식(Data Type)

- 정수형
  - byte, short, int, long
- 실수형
  - float, double
- 문자형
  - char
- 논리형
  - boolean
- 문자열
  - String

이스케이프 시퀀스(Escape Sequence)

- 특수문자
  - ₩n
  - ₩b
  - ₩t
  - ₩r
  - ₩₩
  - ₩'
  - ₩"

실습 : 데이터 형식과 변수

- C:₩Java₩DataTypeAndVariableDemo

실습 : 주석문

- C:₩Java₩CommentDemo

연산자(Operators)

연산자(Operator)

- 연산자
  - 어떤 데이터를 가지고 어떤 작업(연산)을 수행하고자 할 때 연산자라는 것을 사용
- 연산자의 유형
  - 단항 연산자
    - operator op
    - op operator
  - 이항 연산자
    - op1 operator op2
  - 삼항 연산자(?:) : 항이 3개
    - (expression) ? op1 : op2

주요 연산자

Common Operators	Example
• Equality operators	== !=
• Relational operators	< > <= >= is
• Conditional operators	&& || ?:
• Increment operator	++
• Decrement operator	- -
• Arithmetic operators	+ - * / %
• Assignment operators	= *= /= %= += -= <<= >>= &= ^= |=

산술 연산자

연산자	설 명
+	덧셈
-	뺄셈
*	곱셈
/	나눗셈
%	나머지 연산자(몫이 아닌 나머지 값을 구함)

산술 연산자 사용 예

- 산술연산자

더하기	a + b	빼기	a – b
곱하기	a * b	나누기	a / b
나머지	a % b		

- 연결연산자
  - '+' 기호를 사용해 문자열과 숫자를 연결
    - obj = "안녕" + "1234"; // 안녕1234

할당(대입) 연산자

연산자	설 명
=	오른쪽의 값을 왼쪽에 대입
+=	덧셈 연산 후 대입
-=	뺄셈 연산 후 대입
*=	곱셈 연산 후 대입
/=	나눗셈 연산 후 대입
%=	나머지 연산자 연산 후 대입

대입 연산자 사용 예

- 대입연산자(=, +=, -=, *=, /=, %=)
  - 우변의 값을 좌변의 변수에 할당하는 연산자
  - += : 앞에 변수에다 뒤의 변수 값을 누적
    - a = 5; b = 10;
    - a += b;
      - //a = a + b; //+=연산자와 같다.
    - Console.WriteLine(a); // a의 값은? 15

부호 연산자 : +, -

- 부호 연산자 : +, -
  - 현재 숫자형의 부호를 반전시킴
  - + 연산자
    - +1 곱하기
  - - 연산자
    - -1 곱하기

증감 연산자 : 증가(++) 또는 감소(--)

연산자	설 명
++	++a // 연산 전에 1 증가
	a++ // 연산 후에 1 증가
--	--a // 연산 전에 1 감소
	a-- // 연산 후에 1 감소

증감 연산자 사용 예

- 증가 연산자(++)
  - 피연산자의 값을 1 증가
- 감소연산자(--)
  - 피연산자의 값을 1 감소
- 전위 증감 연산자
  - 증감 연산자가 앞에 있으면 선 증감 후 대입
    - int a = 10; int r = ++a; // a ===11, r === 11
- 후위 증감 연산자
  - 증감 연산자가 뒤에 있으면 대입 후 증감
    - int a = 10; int r = a++; // r === 10, a === 11

관계(비교) 연산자 : 참 또는 거짓 값 반환

연산자	설 명
<	Less than
<=	Less than or equal
>	greater than
>=	Greater than or equal
==	Equal
!=	Not equal

관계(비교) 연산자 사용 예

- 두 피연산자를 조건에 따라 선택적으로 실행할 수 있도록 참이면 true, 거짓이면 false 값을 반환

==	a와 b가 같다	a == b	!=	a와 b가 같지 않다	a != b
>	a가 b보다 크다	a > b	<	a가 b보다 작다	a < b
>=	a가 b보다 크거나 같다	a >= b	<=	a가 b보다 작거나 같다	a <= b

논리 연산자

연산자	설 명	예
&&	논리 AND	if ((x < y) && (x > 0))
||	논리 OR	if ((x < y) || (x > 0))
!	논리 NOT	if (!(x < y))

논리 연산자 사용 예

- 논리연산자
  - true와 false 값을 가짐
  - 연산자 논리표

좌변(X)	우변(Y)	연산결과		
		||	&&	!X
false(0)	false(0)	0	0	1
false(0)	true(1)	1	0	1
true(1)	false(0)	1	0	0
true(1)	true(1)	1	1	0

계산	boolean 연산
OR = (3 > 6) || (4 < 8)	true = false || true
AND = (3 > 6) && (4 < 8)	false = false && true
NOT = !(3 > 6)	true = NOT(false)

3항 연산자 : 조건 연산자 : "?:"

- 조건연산자(3항연산자)
  - 조건에 따라 두 값 중에 하나를 연산결과로 전달
  - 뒤에서 배울 if~else문의 단축 표현
    - if (조건식)
    - 　　첫번째문장;
    - else
    - 　　두번째문장;
- 형식 : (조건) ? 수식1 : 수식2
  - 조건인 참이면 앞 문장 수행, 거짓이면 뒷 문장 수행
    - var result = i > 0 ? 0 : 1;

비트 연산자

연산자	설 명	예
&	AND	int result = bi & bi2; //result=00000000
|	OR	int result = bi | bi2; //result=00110011
^	XOR	int result = bi ^ bi2; //result=00110011

비트 연산자 사용 예

- 비트는 0과 1값을 갖는 최소의 단위로서 좌변과 우변의 비트값에 따라 or, and, xor이 결합되어 다음과 같은 논리값을 가짐.

좌변	우변	연산결과		
		|	&	^
false(0)	false(0)	0	0	0
false(0)	true(1)	1	0	1
true(1)	false(0)	1	0	1
true(1)	true(1)	1	1	0

시프트 연산자

연산자	설 명	예
<<	Left 시프트	왼쪽으로 비트 이동
>>	Right 시프트	오른쪽으로 비트 이동, 왼쪽 비트는 항상 0으로 채워짐

시프트 연산자 사용 예

- a << n
  - a * 2^n
    - 4 << 2 => 16
- a >> n
  - a / 2^n
    - 4 >> 2 => 1

연산자의 종류(1)

연산자 범주	연산자
산술 연산자	+ - * / %
논리 연산자(부울 및 비트)	& | ^ ! ~ && || true false
문자열 연결 연산자	+
증가, 감소 연산자	++ --
시프트 연산자	<< >>
관계 연산자	== != < > <= >=
대입 연산자	= += -= *= /= = %= &= |= ^= <<= >>=
멤버 액세스 연산자	.

연산자의 종류(2)

연산자 범주	연산자
인덱스 연산자	[ ]
캐스트(변환) 연산자	( )
조건(3항) 연산자	?:
대리자 연결 및 제거 연산자	+ -
개체 작성 연산자	new
형식 정보 연산자	is sizeof typeof
오버플로 예외 처리 연산자	checked unchecked
간접 참조 및 주소 연산자	* -> [] &

연산자 우선순위

- 연산자 우선순위
  - 연산자는 종류별로 먼저 실행되는 연산자가 있음
  - 우선순위를 모르면 잘못된 식이나 결과를 만들어 버그의 원인이 됨.
  - 우선 순위를 잘 모를 경우에는 괄호를 사용
  - C/C++/C#/Java/자바스크립트의 연산자는 C언어의 연산자와 비슷함
    - a = b + --c + (d – e++);

연산자 우선순위 : C Family(C언어 기준)

우선순위	연산자	우선순위	결합성
1	( ), [ ], ->, .	높음	->
2	sizeof, & (포인터), ++, ==, ~, !, * (포인터), -(부호)		<-
3	* (곱셈), /, %		->
4	+ (덧셈), - (뺄셈)		->
5	<<, >>		->
6	<, <=, >=, >		->
7	==, !=		->
8	& (비트 논리)		->
9	^		->
10	|		->
11	&&		->
12	||		->
13	? :		<-
14	=, +=, *=, /=, %=, ^=, !=, <<=, >>=		<-
15	, (콤마)	낮음	->

실습 : 산술 연산자

- C:₩Java₩ArithmeticOperatorDemo

실습 : 증감 연산자

- C:₩Java₩IncrementDecrementOperatorDemo

실습 : 관계 연산자

- C:₩Java₩RelationalOperatorDemo

실습 : 논리 연산자

- C:₩Java₩LogicalOperatorDemo

실습 : 대입 연산자

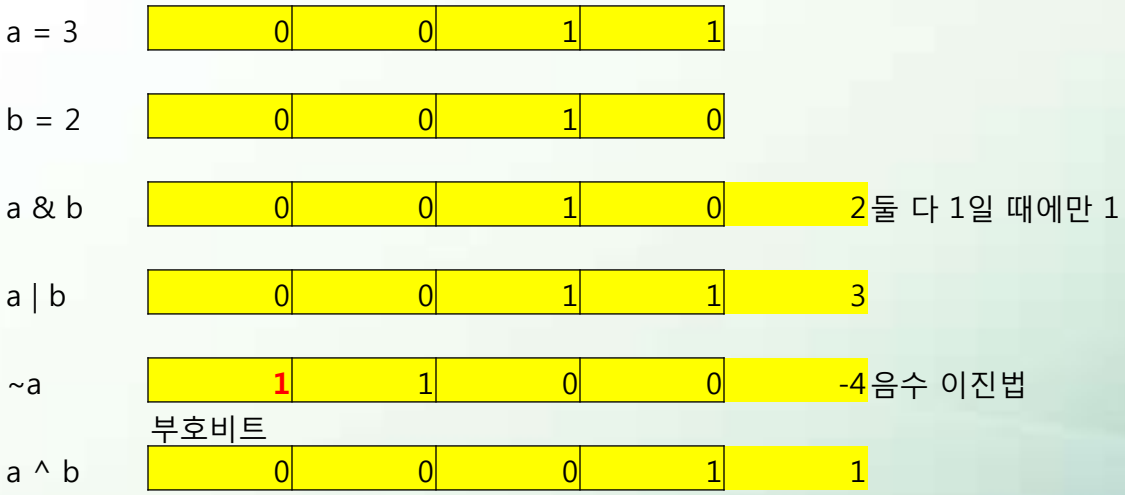
- C:₩Java₩AssignmentOperatorDemo

실습 : 조건(3항) 연산자

- C:₩Java₩ConditionalOperatorDemo

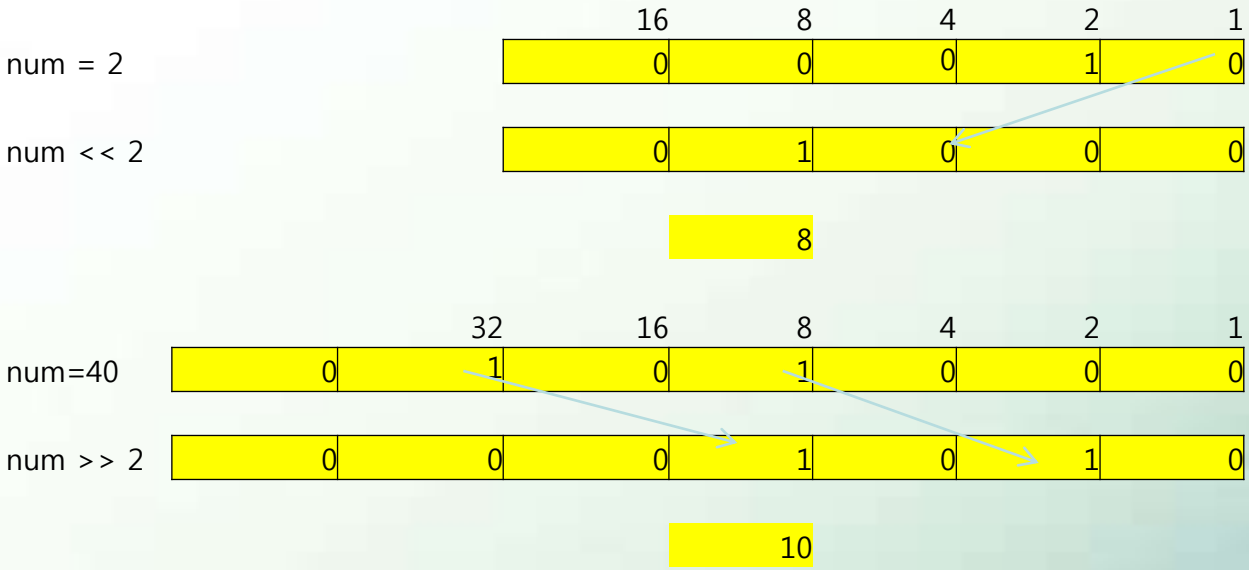
실습 : 비트 연산자

- C:₩Java₩BitOperatorDemo



실습 : 시프트 연산자

- C:₩Java₩ShiftOperatorDemo



제어문(조건문, 반복문)

제어문 : 조건문/반복문/분기문

- 제어문(Control Statement)
  - 제어의 흐름
  - 실행 순서
- 조건문(Decision Statements;선택문(Selection Statement))
  - if~else문
  - switch~case문
- 반복문(Iterative Statement)
  - for문
  - while문
  - do문
- 기타(점프문;Jumping)
  - continue문
  - break문
  - goto문
  - return문

블록(Block)이란? : scope 이해

단일 블록

```
{
  // code
  // 단일문
  // 복합문
  // null…
}
```

중첩된 블록내에서는 동일한 변수명 사용이 불가능

```
{
  int i;
  ...
  {
    int i;
    ...
  }
}
```

다른 블록내에서는 동일한 변수명 사용 가능

```
{
  int i;
  ...
}
...
{
  int i;
  ...
}
```

조건문(선택문 : Selection/Decision)

- 조건문(Conditional) : Branching
  - 조건을 판단한 후 조건에 따라 분기하여 수행
  - if 문
    - 조건을 만족하면 문장을 실행하고 다음 라인을 수행
    - if 문의 조건을 만족하지 않으면 수행을 한 번도 하지 않음.
  - 형식

```
if (조건)
{
    문장;
}
```

조건문

- if ~ else 문 : 3항 연산자와 비슷
  - 형식

```
if (조건)
{
    문장 1 ;
}
else
{
    문장 2 ;
}
```

  - 조건을 만족하면 조건 다음의 문장1을 수행하고 조건을 만족하지 않으면 else 다음의 문장 2를 수행

조건문

- 다중 if 문
  - 형식

```
if (조건1)
{
   문장 1 ;
}
else if (조건2)
{
   문장 2 ;
}
(else if를 붙여 연속적으로 만들 수 있음)
else
{
   기본문장;
}
```

switch 문 : 다중 선택자 : Switching

- 여러 개의 case 구문을 사용하여 여러 조건 처리

```
switch (trumps) { // 제어식
  case Suit.Clubs :
  case Suit.Spades :
    color = "Black"; break;
  case Suit.Hearts :
  case Suit.Diamonds :
    color = "Red"; break;
  default: // default 세그먼트 : 제어식에 포함되지 않는 값 처리
    color = "ERROR"; break;
}
```

반복문(Iteration Statements)의 종류

while

조건이 만족하는 동안 코드 블록을 0번 이상 수행

do

조건이 만족하는 동안 코드 블록을 1번 이상 수행

for

초기식부터 조건식(종료식)을 만족하는 동안 간격(Step)만큼 반복 실행

for 문

- for 문
  - [시작값], [최종값], [증가값]을 설정하여 반복적인 수행을 할 수 있는 문장
  - 규칙적인 증가를 하는 경우에 많이 사용
- 형식

```
for (초기값; 최종값; 증가식)
{
    문장 ;
}
```

```
for (초기식; 논리식; 증감식)
{
    명령문 ;
}
```

  - 한 문장 이상인 경우에는 '{'와 '}' 기호를 이용하여 문장을 묶어줌.

for 문 사용 예제

- 0부터 9까지 반복

```
for (int i = 0; i < 10; i++) {
    출력문(i);
}
```

```
0 1 2 3 4 5 6 7 8 9
```

- for문내에서 선언된 변수는 블록을 빠져나가면 소멸

```
for (int i = 0; i < 10; i++)
    출력문(i);
출력문(i); // 에러 발생
```

- 구문내에서 여러 개의 변수 선언 가능 : 추천 안 함

```
for (int i = 0, j = 0; ... ; i++, j++)
```

반복문 : 이중 for 문

- 다중 for 문
  - 안쪽의 for문과 바깥쪽의 for 문이 쌍을 이루어 반복 수행
  - 구구단 프로그램과 잘 어울림
- 형식

```
for (초기값; 최종값; 증가식)
{
        문장 ;
        for (초기값; 최종값; 증가식)
        {
                문장 ;
        }
}
```

반복문 : while문 : 조건반복

- 특징
  - 사전-검사 논리 제어 루프
  - [조건만족]이 true인 동안은 문장을 반복하고 false일 경우에는 while 문 다음의 문장을 수행
  - 조건을 먼저 확인하기 때문에 한 번도 수행하지 않는 경우도 있음.
- 형식

```
while (조건식)
{
        명령문;
}
```

while 문

- 조건이 만족하는 동안 반복 실행

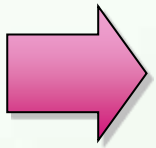
```
int i = 0;//초기식
while (i < 10)//논리식
{
    출력문(i);//명령문
    i++;//증감식
}
```

0 1 2 3 4 5 6 7 8 9

while문과 for문

- while문은 for문으로 대체해서 사용 가능

```
초기값;
while (조건식)
{
        명령문;
        증감식;
}
```



```
for (초기값; 최종값; 증감식)
{
        명령문;
}
```

do 문 : 실행반복

- 선 실행 후 조건 처리
- 특징
  - 사후-검사 논리 제어 루프
  - [반복문장]을 수행한 후에 [조건만족]을 확인
  - [조건 만족]이 맞던, 틀리던 [반복 문장]을 한 번 수행
- 형식

```
do
{
        문장 ;
} while(조건) ;
```

```
int i = 0;
do {
    출력문(i); // 구문을 먼저 실행
    i++;
} while (i < 10);
```

0 1 2 3 4 5 6 7 8 9

break 및 continue 문

- continue 문 : 외곽 루프(the smallest enclosing loop)의 제어로 전달
  - for문, while문, do while문의 반복중에 continue 문을 만나면 조건을 판단한 후 다음 구문을 수행
  - 다시 조건식으로 이동하는 명령문
  - continue문은 다음 반복위치로 이동한다.
- break문 : 탈출
  - for, while, do while 문의 반복중인 반복구문을 강제적으로 종료하는 경우에 사용
  - 반복문을 빠져나오는 명령문
  - break문은 반복문을 빠져나간다.

```
int i = 0;
while (true) {
    출력문(i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

foreach 문 : foreach … in : C#, JavaScript

- 배열 또는 컬렉션과 같은 데이터에 데이터가 있는만큼 반복

```
int[] numbers = {0, 1, 2, 3, …, 8, 9};

foreach (int number in numbers) {
    출력문(number);
}
```

0 1 2 3 4 5 6 7 8 9

goto 문 : 사용금지(?)

- 무조건 분기문
  - 레이블을 만들어 놓고 해당 레이블 위치로 이동시킬 수 있는 기능을 제공
  - 스파게티 코드를 만들 가능성이 있음
  - C/C++/C# 포함, Java는 goto 제거
  - 지난 20년간 프로그래머들은 goto문을 사용 안 함

```
if (number % 2 == 0) goto Even;
출력문("odd");
goto End;
Even:
출력문("even");
End:;
```

제어문

- if~else
- switch~case
- for
- while
- do~while
- break
- continue
- goto
- return

실습 : 조건문

- C:₩Java₩IfDemo

실습 : 선택문

- C:₩Java₩SwitchDemo

실습 : 반복문 : 구간반복

- C:₩Java₩ForDemo

실습 : 반복문 : 구간반복

- C:₩Java₩ForDemo

실습 : 반복문 : 조건반복

- C:₩Java₩WhileDemo

실습 : 반복문 : 실행반복

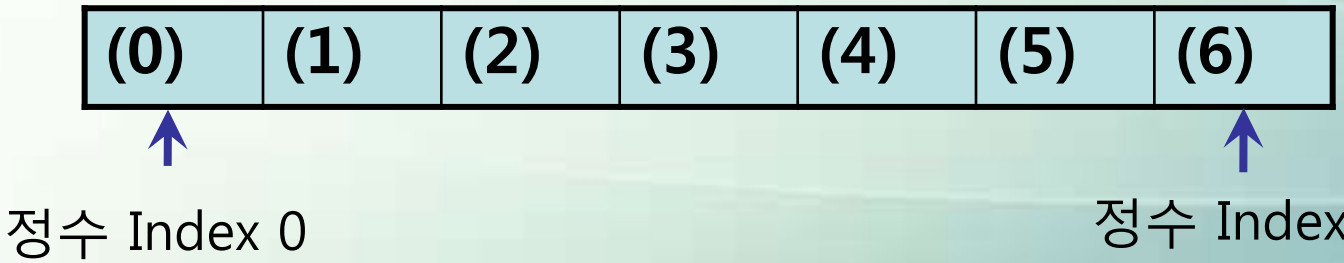
- C:₩Java₩DoWhileDemo

배열(Arrays)

(C/C++/C#/Java/JavaScript 공통)

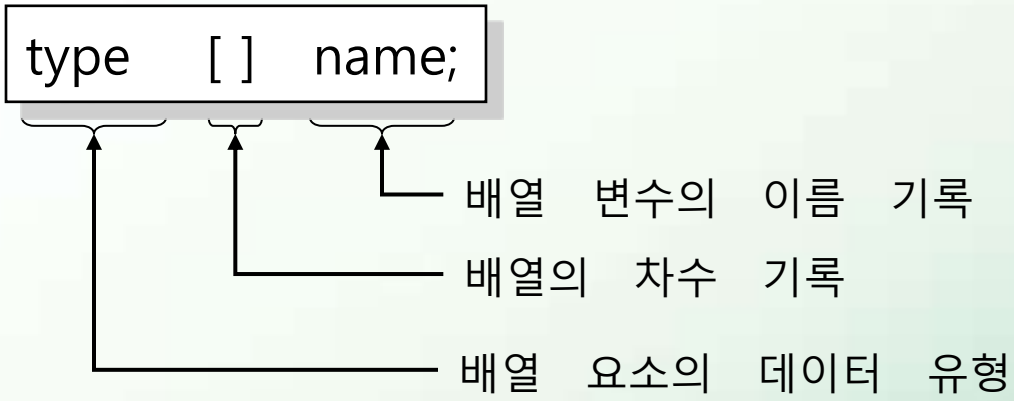
배열(Array)이란?

- 데이터 구조 관점
  - 하나의 이름으로 하나의 데이터 유형을 하나만 보관해 놓는 그릇 역할 => 변수
- 배열은 요소들의 집합이다.
  - 배열의 각 요소들은 모두 동일한 데이터 타입을 가짐
    - 구조체는 서로 다른 데이터 타입을 가짐
  - 각 요소들은 정수형 인덱스로 접근 가능



배열 표기법

- 배열 변수를 아래와 같이 선언할 수 있다.
  - 배열의 데이터 타입
  - 배열의 차원 수
  - 배열 변수의 이름



- 언어별 차이점 존재
  - C/C++/C#/Java/JavaScript 등에서 조금씩 표기법의 차이점이 존재(일단, 가장 나중에 나온 C#을 기준으로 설명)
    - 실습 예제를 통해서 비교

변수 선언 및 요소 수 생성

Single

```
Type[] arrayName = new Type[ Size ];
```

Multiple

```
Type[ , ] arrayName = new Type[ Size1, Size2];
```

Jagged

```
Type [][] JaggedArray = new Type[size][];
```

참고: C언어 1차원 배열 샘플 코드

```
/*
        6.1.1. 예제. 배열의 선언, 요소수 생성, 초기화, 참조 : 일차원배열1.c
*/
//사용자로부터 3개의 데이터를 입력받아,
//합계를 구하는 프로그램.
#include <stdio.h>
main(){
        //Init
        int i, intSum = 0;
        int intNum[3];//배열
        //Input
        for(i = 0;i < 3;i++){
                printf("\n%d번째 입력 : ", i + 1);
                scanf("%d", &intNum[i]);
        }
        //Process
        for(i = 0;i < 3;i++){
                intSum += intNum[i];
        }
        //Output
        printf("\n합계 : %d\n", intSum);
}
```

참고: C# 1차원 배열 샘플 코드

```
// 지역 변수
int k = 10;

//[1] 배열 선언
int[] j;

//[2] 배열 요소수 생성
j = new int[3];

//[3] 배열 초기화
j[0] = 10;
j[1] = 20;
j[2] = 30;

//[4] 배열 출력
for (int i = 0; i < j.Length; i++)
{
  Console.WriteLine(j[i]);
}
```

참고: Java 1차원 배열 샘플 코드

```
// 지역 변수
int v = 10;

// 배열 선언
int [] arr;

// 배열의 요소수 생성
arr = new int[3];

// 배열 초기화
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;

// 배열 사용
//System.out.println(arr[1]); // 20
for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
}
```

참고: JavaScript 1차원 배열 샘플 코드

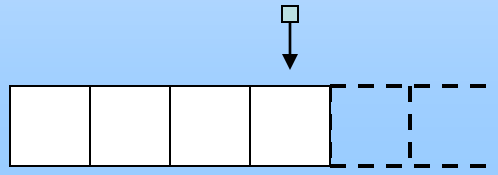
```
//[a] 배열 선언 : 배열의 인스턴스를 생성한다.
var eng = new Array(3);

//[b] 배열 초기화(할당)
eng[0] = 90;
eng[1] = 100;
eng[2] = 85;

//[c] 배열의 값을 사용(참조) : 주로 반복문을 이용
var sum = 0;
for (var i = 0; i < 3; i++)
{
    sum += eng[i];
}

document.write("총점 : " + sum + "<br />");
```

배열의 랭크(차원 수)

- 배열의 종류
  - n열의 요소로 구성된 1차원 배열
  - n열의 요소가 m행으로 구성된 2차원 배열
    - C# : [,], C/C++/Java : [][]

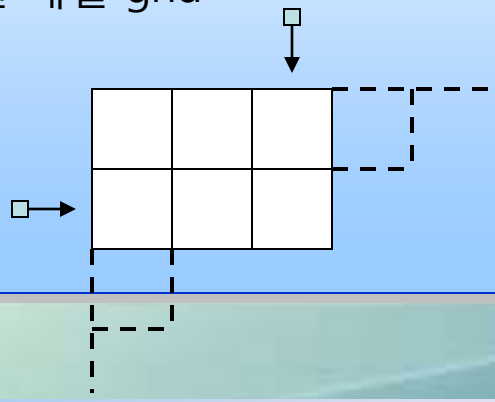
```
long[ ] row;
```

long형 데이터를 가지는 1차원 배열 row



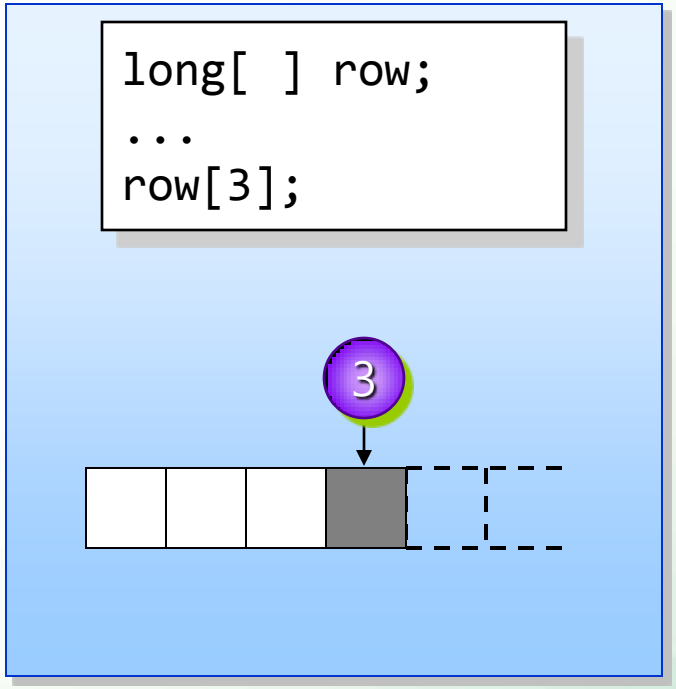
```
int[,] grid;
```

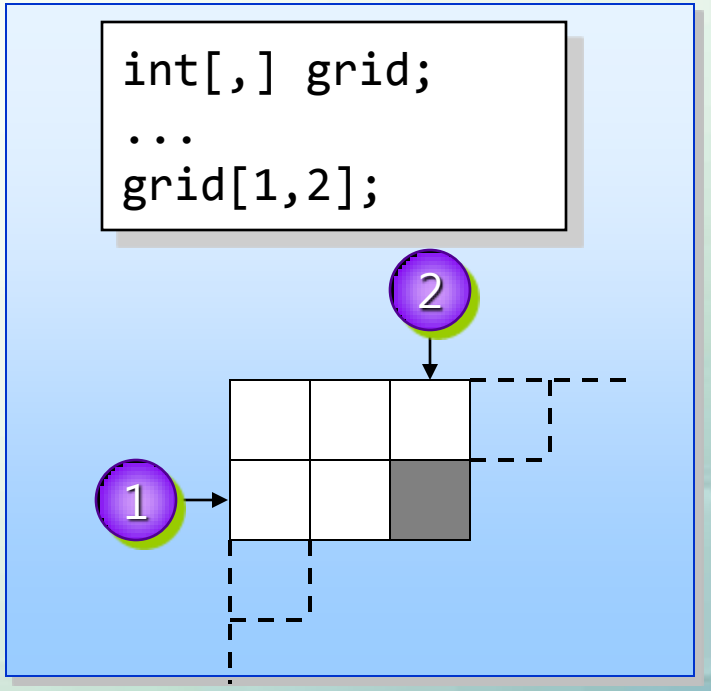
int형 데이터를 가지는 2차원 배열 grid



배열 요소에 접근하기

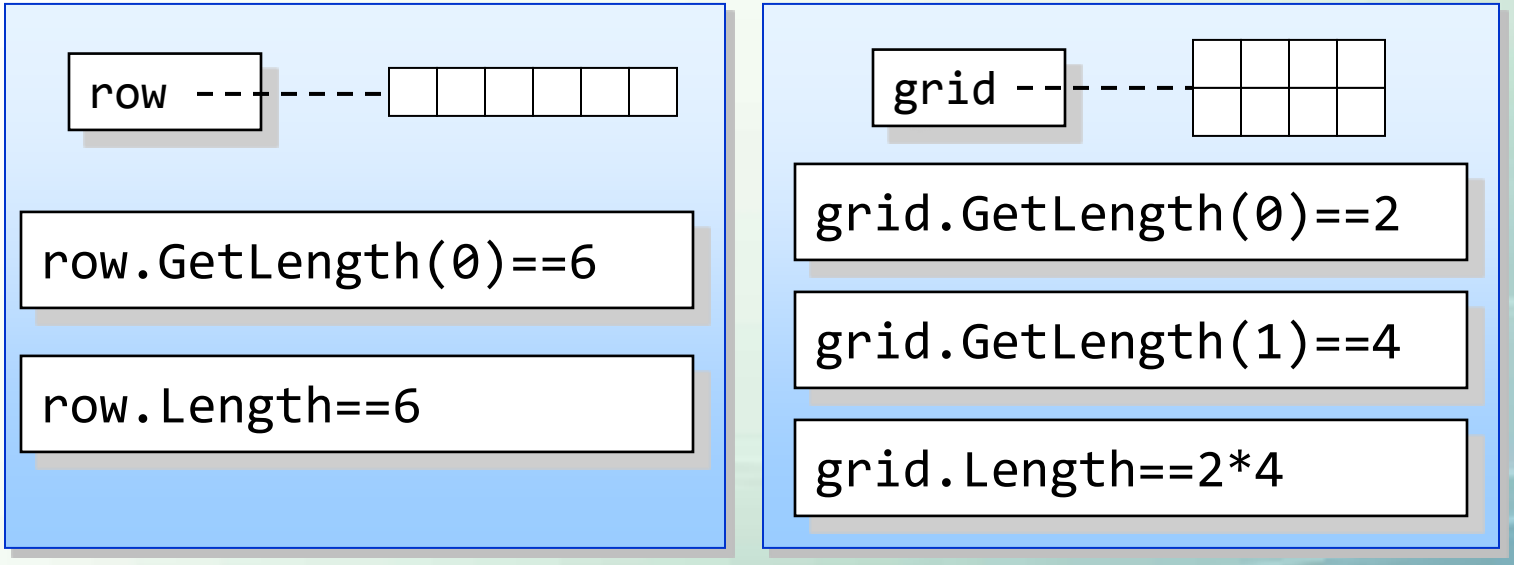
- 각 랭크에 정수형 인덱스를 지정함
  - 인덱스는 0부터 시작한다.





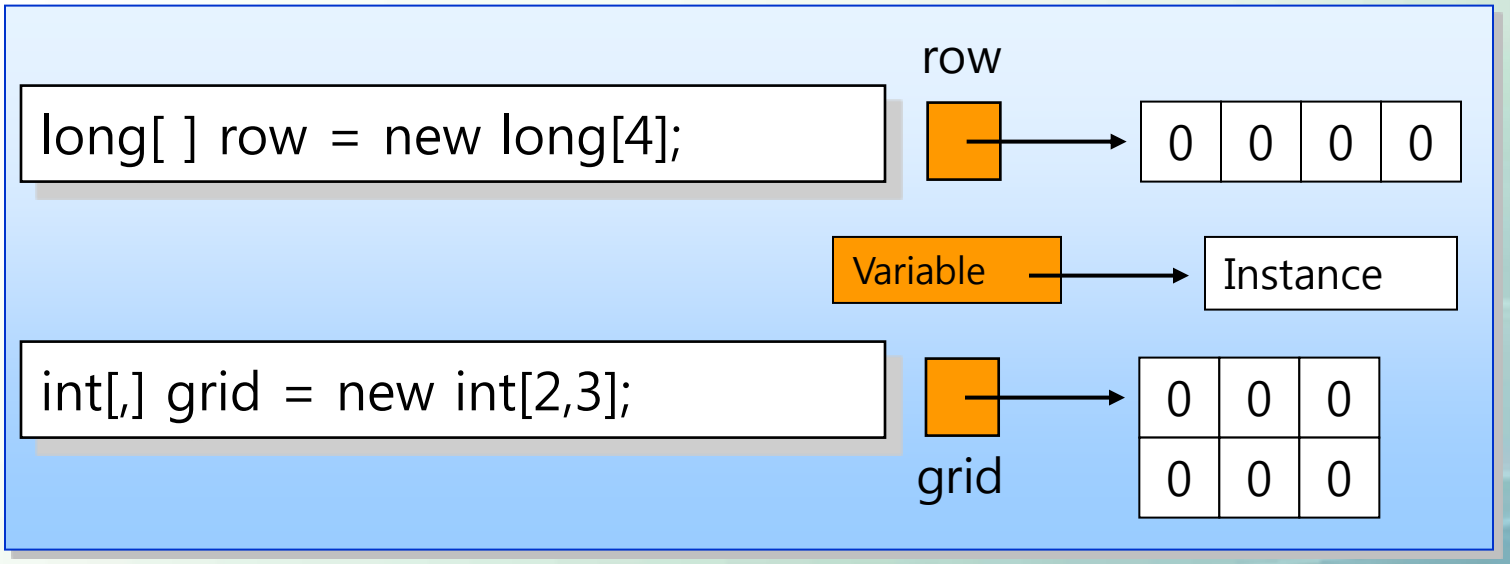
배열의 경계 체크하기

- 모든 배열의 경계를 체크 가능
  - 잘못된 인덱스 지정 : IndexOutOfRangeException
  - 배열의 요소수
    - 배열명.length
  - **Length** 속성과 **GetLength** 메서드 사용
    - C#, Java(length, arr[0].length), JavaScript(arr.length)



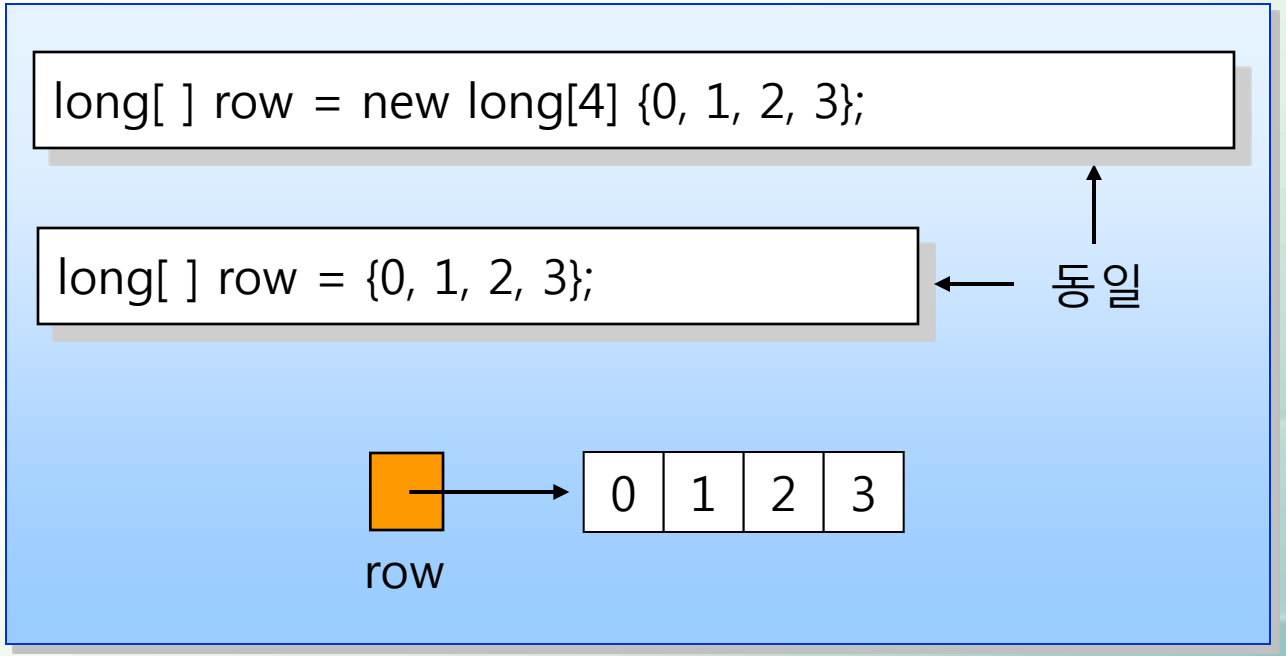
배열의 인스턴스 생성

- 배열을 선언한다고 배열요소가 만들어지는건 아님
  - 배열의 요소를 생성하려면 new 연산자 사용
    - var arr = new Array(n); // JavaScript
  - 배열 요소들은 기본값으로 0으로 초기화



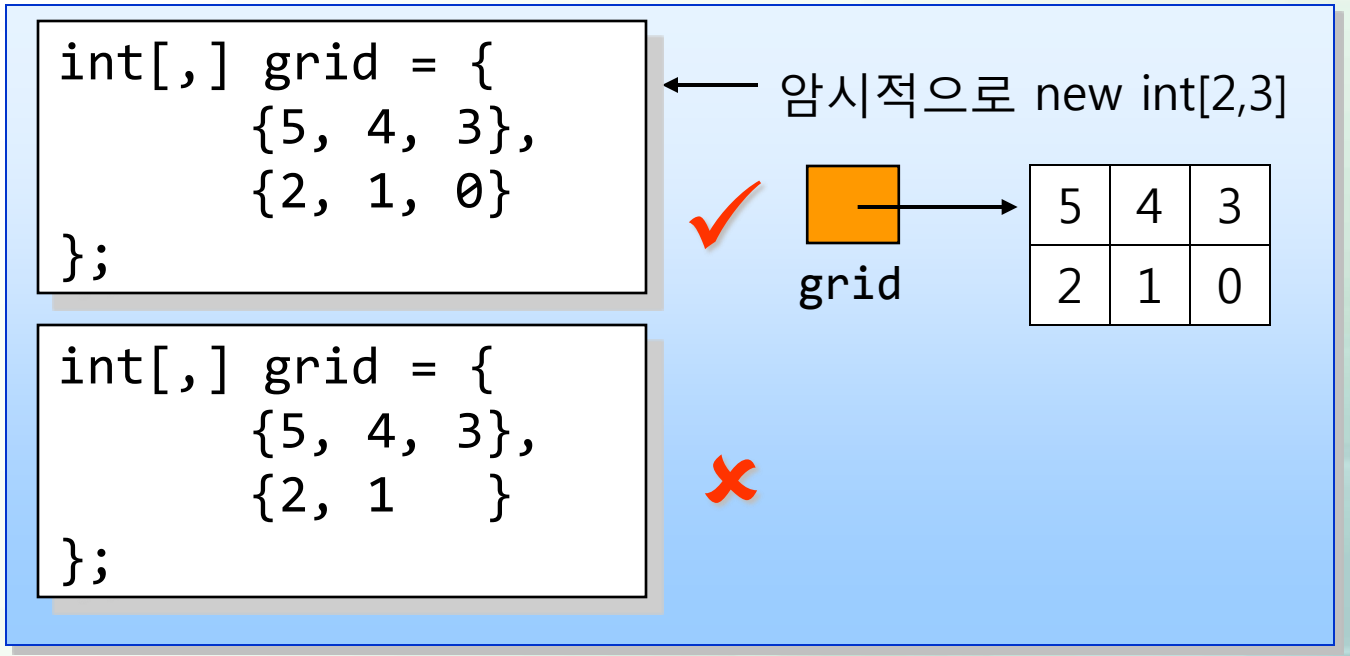
배열 요소의 초기화

- 배열 요소의 선언과 동시에 초기화
  - 배열의 요소를 초기화할 때 줄여 사용

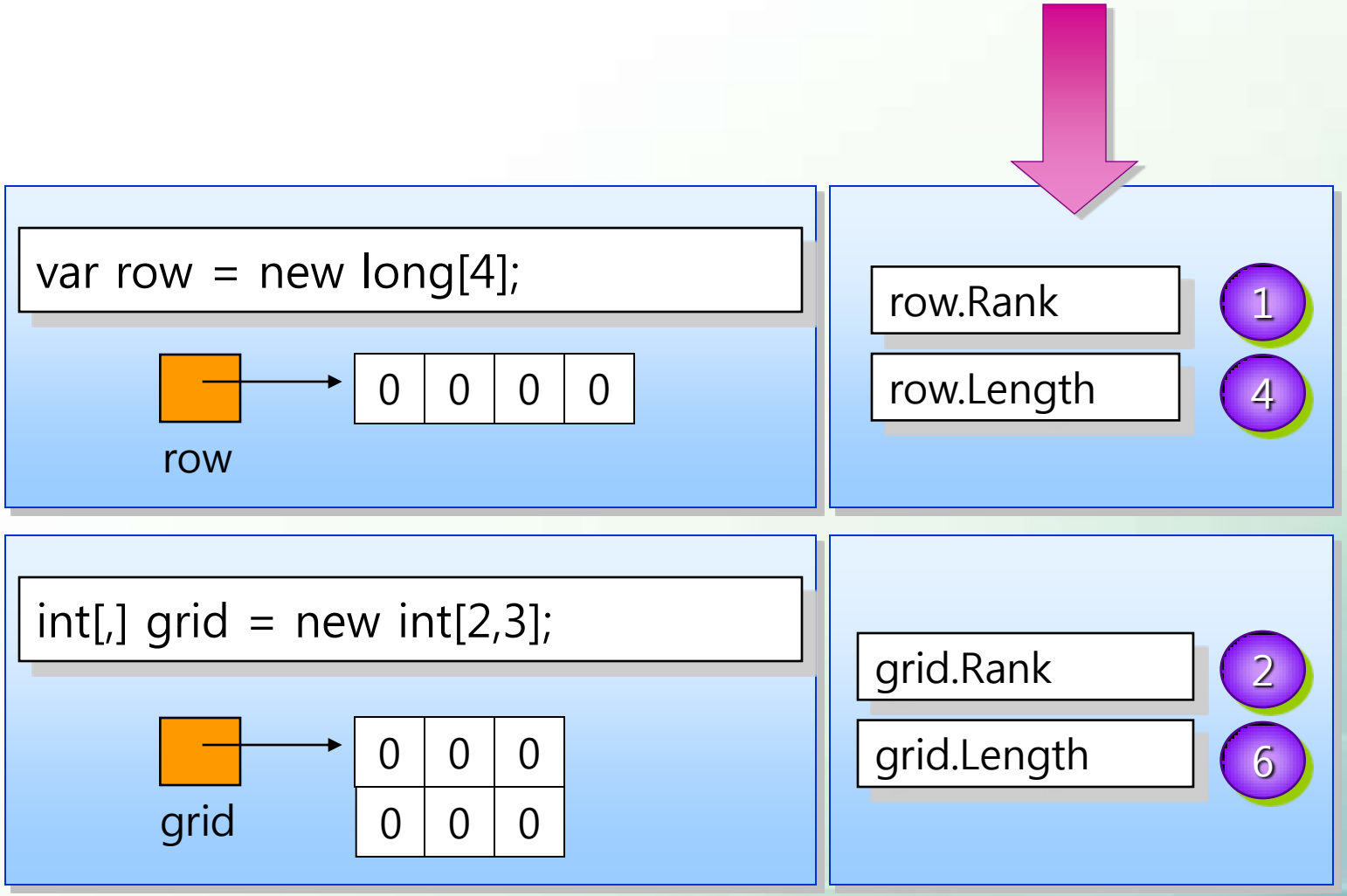


2차원 배열 요소의 초기화

- 2차원 배열 요소의 초기화도 아래와 같다.
  - 모든 요소가 명시되어야 함



배열 관련된 속성 : C#


var row = new long[4];
0 0 0 0
row
row.Rank
1
row.Length
4
int[,] grid = new int[2,3];
0 0 0
0 0 0
grid
grid.Rank
2
grid.Length
6

계산된 크기의 배열 선언

- 컴파일 시 배열 크기 지정
  - 컴파일시에 미리 배열의 크기를 지정해 놓는 게 다른 방법보다 속도가 향상된다.

```
long[ ] row = new long[4];
```

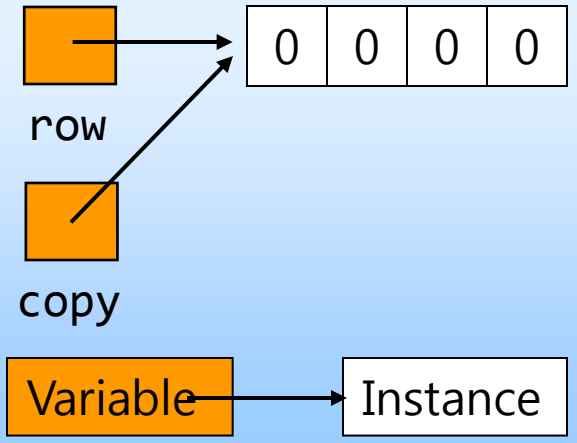
- 런타임 시 배열 크기 지정

```
string s = Console.ReadLine();
int size = int.Parse(s);
long[ ] row = new long[size];
```

배열 변수의 복사

- 배열 변수의 복수는 배열 변수 자체만 복사
  - 배열의 인스턴스는 복사되지 않음
  - 2개의 배열 변수는 동일한 배열 인스턴스를 참조

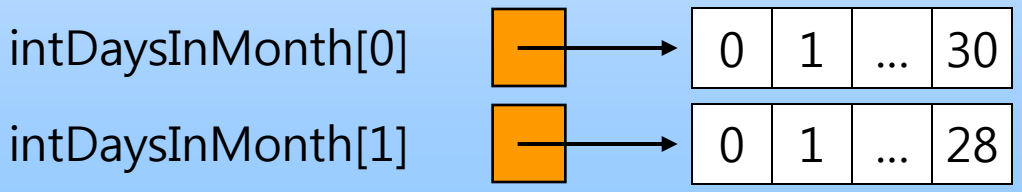
```
long[ ] row = new long[4];
long[ ] copy = row;
...
row[0]++;
long value = copy[0];
Console.WriteLine(value);
```



배열의 배열

- 배열의 배열(Jagged Arrays)
  - 배열의 요소가 일정하지 않은 배열
  - 달력의 12개월 각각의 날짜수가 다름

```
int [] [] intDaysInMonth;
intDaysInMonth = new int [12][];
intDaysInMonth[0] = new int[31];
intDaysInMonth[1] = new int[28];
...
intDaysInMonth[11] = new int[31];
```



배열의 특징 정리

- 배열의 각 요소는 값을 갖는다.
- 배열은 0부터(zero-indexed) 인덱스를 설정한다.
- 배열의 크기(length)는 배열의 총 요소수를 나타낸다.
- 배열은 1차원(Single-Dimensional), 다차원(multidimensional), 배열의 배열(톱니형;jagged) 중 하나로 구성될 수 있다.
- 배열의 첨자 범위의 하한(lower bound)은 0부터 시작한다.
- 배열은 고정된 크기를 갖는다. 새로운 요소를 추가할 수 없다.

배열(Array)과 컬렉션(Collection)의 비교

- 배열은 꽉 찼을 때 크기 조절이 안 됨
  - ArrayList와 같은 컬렉션 클래스는 크기 조정 가능
- 배열은 하나의 데이터 타입만 지정/저장 가능
  - 컬렉션은 서로 다른 요소 저장 가능
- 배열의 각 요소에 읽기 전용 설정 안됨
  - 컬렉션은 읽기 전용 가능
- 일반적으로, 배열이 컬렉션보다 빠르다. 하지만 유연성은 떨어진다.
  - 컬렉션은 조금 느리지만 유연성은 크다.
    - 빠르고 느리다는 표현은 거의 의미가 없다. 고로 컬렉션을 사용하자.

배열 정리

- C/C++/C#/Java
  - 1차원 배열
  - 2차원 배열
  - 3차원 배열
- JavaScript
  - 1차원 배열
  - 2차원 배열
    - 크게 의미는 없음

실습 : 일차원배열

- C:₩Java₩ArrayOneDemo

실습 : 최댓값 알고리즘

- C:₩Java₩MaxDemo

실습 : 이차원배열

- C:₩Java₩ArrayTwoDemo

부 프로그램(함수)
(메서드와 파라미터)
(Methods and Parameters)

부 프로그램(subprogram) : 함수, 프로시저

- 부 프로그램(메서드, 함수, 프로시저, ...)이란?
  - 메서드(함수)란 어떤 값을 받아서(매개변수) 그 값을 가지고 가공을 거쳐 어떤 결과 값(리터값)을 반환시켜주는 코드이다.
  - 동일한 코드를 반복하여 사용할 수 있도록 하나의 이름으로 만들어 놓은 코드의 집합이라고 볼 수 있다.
  - 메서드는 프로그램 코드 내에서 특정한 기능을 처리하는 독립적인 하나의 단위 또는 모듈을 가리킨다.
  - 메서드에게 어떤 값을 처리하라고 넘겨주는 값은 문자열, 숫자 등이 있으며, 이를 가리켜 인수, 인자, 파라미터(Parameter), 매개 변수 등으로 부르는데 현재 시점에서는 모두 같은 개념으로 보면 된다.
  - Visual Basic과 같은 프로그래밍 언어에서는 C#/Java에서 말하는 메서드를 함수라고도 말한다. 하지만 C#/Java에서는 함수라고 부르지 않고 메서드(Method)라고 부른다. 필자는 경우에 따라서 같은 클래스 내의 메서드를 함수라고 하고 클래스와 클래스간의 함수를 메서드라고 부를 예정이다.
  - 지금까지 사용해 온 Main() 메서드가 메서드의 전형적인 예라고 볼 수 있다.

함수 : 메서드(Method)

- 부 프로그램의 종류
  - 프로시저(procedure)
  - 함수(function)
- 함수(메서드)
  - 어떤 특정한 로직을 모아놓은 프로그램 코드
  - C#/Java에서는 함수라는 이름을 사용하지 않는다.
  - 같은 클래스 안에서의 메서드를 함수라 한다.
- 함수의 종류
  - 내장 함수
    - 자주 사용하는 기능을 미리 만들어서 제공하는 메서드로, 내장 메서드도 그 사용 용도에 따라서 문자열 관련 메서드, 날짜 및 시간 관련 메서드, 수학 관련 메서드, 형식 변환 관련 메서드 등으로 나눌 수 있다.
    - ToString()
  - 사용자 정의 함수
    - 사용자가 필요할 때마다 새롭게 기능을 추가시켜 사용하는 메서드이다.
    - 매개변수가 없는 함수
    - 매개변수가 있는 함수
    - 반환값이 있는 함수

메서드 선언

- Main도 메서드이다.
  - 새로운 메서드도 Main과 같은 형식으로 정의됨

```
// C# 코드 샘플
using System;

class 샘플클래스
{
    static void 샘플메서드( )
    {
        Console.WriteLine(“샘플 메서드");
    }
    static void Main( )
    {
        // ...
    }
}
```

메서드 호출(Call)

- 메서드 선언 후에 할 수 있는 사항들
  - 같은 클래스 안에 있는 메서드 호출
    - 괄호안에 파라미터 목록을 사용한 메서드 이름 사용
  - 다른 클래스 안에 있는 메서드 호출
    - 반드시 컴파일러한테 해당 메서드를 포함하는 클래스의 이름을 지정해주어야 함!
    - 호출되는 메서드는 반드시 **public** 키워드로 선언되어야 함
  - 중첩 호출 가능
    - 메서드는 중첩 호출이 가능하다. 즉, 메서드가 또 다른 메서드를 호출 가능하다.

return 구문 사용

- 즉시 리턴
- 조건문을 사용한 리턴

```
static void ExampleMethod( )
{
    int numBeans;
    //...

    출력문("Hello");
    if (numBeans < 10)
        return;
    출력문("World");
}
```

로컬 변수 사용

- 로컬 변수(Local variables)
  - 메서드가 시작될 때 생성
  - 메서드에 대해서 Private
  - 메서드 종료와 함께 소멸
- 공용 변수(Shared variables) : 전역(Global) 변수
  - 공유 용도로 사용되는 클래스 변수
- 범위 충돌(Scope conflicts)
  - 컴파일러는 로컬/클래스 변수의 충돌에 대해서 경고를 발생시키지 않음

반환 값

- 반환값이 있는 메서드 선언
- return 키워드를 구문에 추가
  - 반환값 설정
  - 호출한 곳으로 반환값 반환
- 반환값이 있는 메서드는 반드시 값을 리턴해야 함

```
static int TwoPlusTwo( ) {
    int a,b;
    a = 2;
    b = 2;
    return a + b;
}
```

```
int x;
x = TwoPlusTwo( );
출력문(x);
```

파라미터 정의 및 호출

- 파라미터 정의
  - 메서드명 다음에 나오는 괄호안에 위치
  - 각 파라미터에 데이터형과 이름 지정
- 파라미터를 사용한 메서드 호출
  - 각 파라미터에 값이 전달됨

```
static void MethodWithParameters(int n, String y)
{ ... }

MethodWithParameters(2, "Hello, world");
```

매개변수(파라미터) 전달 기법

- 파라미터 전달의 3가지 방법

in	입력 모드 값에 의한 전달(Pass by value)
in out	입출력 모드 참조에 의한 전달(Pass by reference)
out	출력 모드 Output 매개변수(Output parameters)

값에 의한 전달(Pass by Value)

- 파라미터 전달의 기본 기법
  - 파라미터 값이 복사됨
  - 변수는 메서드 안에서 바뀜
  - 메서드 밖에 있는 값에는 영향을 미치지 않음
  - 파라미터는 반드시 같거나 호환 가능해야 함

```
static void AddOne(int x)
{
    x++; // Increment x
}
static void Main( )
{
    int k = 6;
    AddOne(k);
    Console.WriteLine(k); // Display the value 6, not 7
}
```

참조에 의한 전달(Pass by Reference)

- 참조형 매개변수란?
  - 참조란 메모리 상의 위치(포인터)이다.
- 참조형 매개변수 사용
  - 타입과 변수값이 맞아야 함
  - 호출자에 의해서 원본 데이터에 영향
  - 메서드를 호출하기 전에 매개변수를 초기화해야 함

Output 매개변수(Output Parameters) : C#

- Output 매개변수란?
  - 매개변수값이 메서드 안에서 밖으로 전달
- Output 매개변수 사용
  - 기본적을 ref 매개변수와 같다.
    - 그렇지만 값이 메서드 안으로 전달되지는 않음
  - 메서드 선언과 호출시 out 키워드 사용

```
static void OutDemo(out int p)
{
    // ...
}
int n;
OutDemo(out n);
```

가변형 매개변수 : C#

- params 키워드 사용
- 값 전달
- 한 번에 여러 개의 매개변수(배열형) 전달 가능

```
static long AddList(params long[ ] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( )
{
    long x = AddList(63,21,84);
}
```

오버로드된 메서드 선언(Overloaded Methods)

- 클래스 안에서 메서드 이름 공유 : 다중 정의
  - C++/C#/Java
    - 컴파일러는 매개변수의 타입으로 구분해서 호출

```
class OverloadingExample
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        출력문(Add(1,2) + Add(1,2,3));
    }
}
```

부프로그램 사용 정리

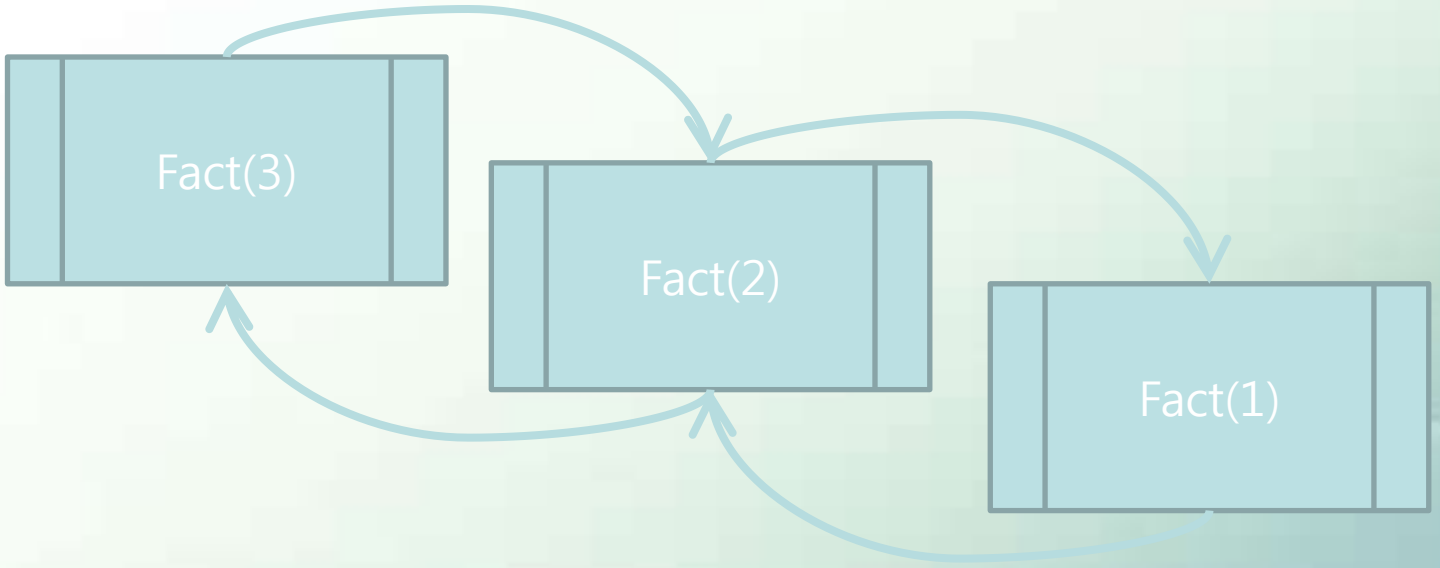
- 프로그래밍 언어에서 프로세스 추상화는 부프로그램으로 표시함
- 재 사용의 목적이 강함

실습 : 함수

- C:₩Java₩FunctionDemo

재귀(Recursion)

- 재귀함수
  - 함수는 자기 자신을 다시 호출할 수 있음
    - 재귀호출 : 함수가 자기 자신을 다시 호출



실습 : 재귀(Recursion)

- C:₩Java₩RecursionDemo

```
function factorial(n){
    if (n === 0 || n === 1)
    {
        return 1;
    }
    return n * factorial(n - 1);
}

factorial(3);
```

객체 지향 프로그래밍

- Object Oriented Programming(OOP)
  - 현실 세계를 모델링(추상화)하여 프로그래밍화 한 것
- 객체 지향 프로그래밍 언어
  - C++/C#/Java
- OOP의 3가지 큰 개념
  - 캡슐화
  - 상속
  - 다형성
- 특징
  - 재 사용성(ReUse)
    - Component Based Development

클래스(Class)

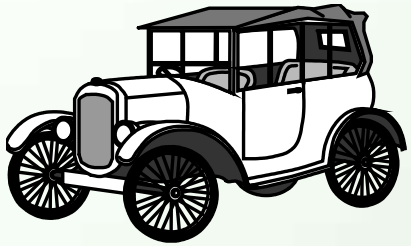
클래스

- 데이터 보관 관점
  - 하나의 이름으로 여러 개의 데이터형식을 여러 개 보관해 놓는 그릇 역할 + 기능 추가

클래스의 의미

- 현실 세계의 자동차(객체)를 만들어내는 설계도(클래스)의 의미
- 프로그램 세계의 데이터 저장 공간(필드)과 기능(메서드)을 하나의 이름(클래스명/객체명)으로 관리

클래스(Class)와 객체(Object)의 비교



클래스	객체
객체를 생성해 주는 형틀	어떤 클래스의 인스턴스
객체들의 특성들을 기술	객체마다 각각의 상태를 가진다
개념적, 가상적	물리적, 실제적
프로그램 텍스트상에서만 존재	실행시에만 존재
디자인 단계에서의 관심사	실행 단계의 관심사

클래스: 개체를 만들어내는 설계도(틀)

Class

A symbolic representation of an object.

Analogy: A blueprint.

Object

An instance of a class.

Analogy: A building based on the blueprint.

Example

Each form in a C# project is an object. Each form is an instance of the Form class.

클래스(Class)

- 데이터(멤버변수;필드)와 메서드(기능;함수)를 결합
- 접근 권한의 통제
  - 메서드는 public, 외부로부터 접근이 가능
  - 데이터(멤버)는 private, 내부에서만 접근 가능
  - public, protected, internal, private
- 상속 (Inheritance)
  - 상속은 클래스간의 관계
  - 기존의 클래스를 상속 받아 새로운 클래스를 만든다.
  - 단일상속 및 다중상속 가능 (Interface에 의해)
- 인터페이스 (Interface)
  - 단지 operation만으로 구성되어져 있음
  - 실제 구현을 가지지 않음

개체(Object) 생성 순서

- Step 1: 메모리 할당
  - 힙(heap)에 메모리를 할당하기 위해서는 new 키워드를 사용
    - new : 참조 연산자
- Step 2: 생성자를 통한 개체 초기화
  - 클래스의 이름과 괄호 기호를 사용

```
// Date 클래스의 인스턴스(실체) 생성 => when 객체 생성
Date when = new Date( );
```

Object LifeCycle

- 객체 생성 시기
  - new 연산자를 이용한 메모리를 할당하고 객체를 생성
  - 생성자를 이용해 메모리 상의 객체 초기화
- 객체 사용 시기
  - 객체에 대한 메서드, 멤버 호출 / 접근
- 객체 소멸 시기
  - 객체에게 할당된 메모리는 실행 환경으로 회수
  - 메모리 해제
  - 소멸 시기가 비결정적임
  - Garbage Collector (GC)가 객체 소멸을 관장한다

클래스(Class)와 객체(개체;Object)

- Class
  - 현실 세계의 자동차 설계도
- Object
  - 자동차 설계도로부터 만들어진 하나의 자동차

실습 : 클래스(Class)

- C:₩Java₩ClassDemo

실습 : 필드(Field)

- C:₩Java₩FieldDemo

실습 : 스태틱과 인스턴스

- C:₩Java₩StaticAndInstanceDemo
  - 정적(static) 멤버 : static 키워드가 붙은 멤버
    - 정적 필드
    - 정적 메서드
    - 접근
      - "클래스명.멤버명"
        - » Math 클래스
  - 인스턴스 멤버
    - 클래스내에서 static 키워드가 붙지않은 멤버
    - 클래스의 인스턴스 생성 후 멤버에 접근
    - static 키워드 제거
    - 접근
      - 클래스 객체 = new 클래스();
      - "객체.멤버명"

생성자(Constructor)

생성자(Constructor)란?

- 클래스(객체)를 초기화 시켜주는 메서드
  - void 키워드를 붙이지 않고도 반환값이 없음
- 생성자는 개체를 초기화(클래스내의 필드를 초기화)하는데 사용하는 구문이다.
- 클래스 이름과 동일한 이름을 사용하는 메서드이다.
- 현실 세계에서 자동차의 시동을 거는 동작에 비유할 수 있다.
- 클래스 실행시 제일 먼저 실행되는 메서드

기본 생성자 사용

- 기본 생성자의 특징
  - public 접근수식자
  - 클래스명과 동일한 이름 사용
  - **반환값도 없고, void도 아님**
  - 매개변수가 없음
  - 모든 필드를 **zero**, **false** 또는 **null로 초기화**
- 생성자 문법

```
class Date { public Date( ) { ... } }
```

실습 : 생성자(Constructor)

- C:₩Java₩ConstructorDemo

가비지 컬렉션(Garbage Collection)

- 자동차(Car) 개체와 관련하여
  - 소멸자 : 좋은 호텔 앞에서의 주차 요원 역할
    - 왜?
      - 키를 받고 알아서 시동을 꺼주니까?
- 개체의 라이프 사이클(The Object Life Cycle)
- 값형과 참조형의 할당된 자원 관리 비교

개체의 라이프 사이클(Object Life Cycle)

- Block of memory allocated
  - new 키워드
- Memory converted into an object
  - 생성자(Constructor)에 의한 객체 생성/초기화
- Object used in application
  - 속성, 메서드 및 기타 멤버 사용
- Object tidied up
  - 소멸자에 의한 객체 정리
- Memory reclaimed
  - 메모리 반환

값형과 참조형 메모리 자원 관리 비교

- 값형(Value Type)
  - Stack에 저장됨
    - LIFO : 나중에 만들어진 변수부터 먼저 소멸됨
  - 해당 범위(Scope)를 벗어나면 자동으로 소멸됨
  - 메모리 단편화(memory fragmentation)가 없음
- 참조형(Reference Type)
  - Heap에 저장됨
    - FIFO
  - 동일 개체를 여러 번 참조 가능
  - 하나라도 참조하고 있으면 소멸되지 않음
    - 메모리(memory fragmentation)의 원인이 될 수 있음
  - 가비지 컬렉터(Garbage Collector)에 의해 관리됨

값형과 참조형 코드 비교

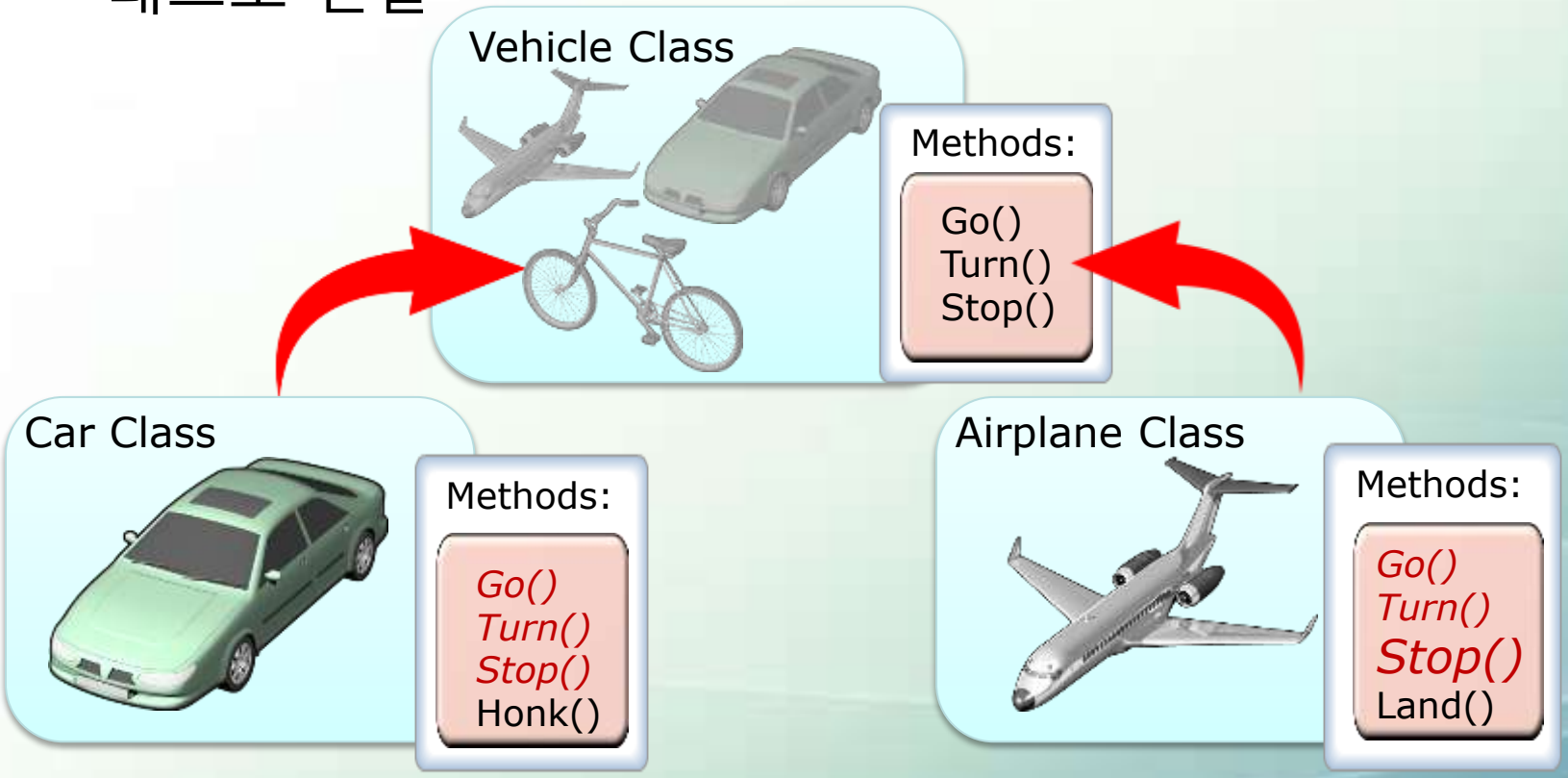
```
private void MyMethod(…)
{
    double y = …;
    int x = …;
    …

    // x, y, z, and a disappear here
}
```

```
String s = …;
MyClass c = …;

private void MyMethod2(…)
{
    String t = s;
    MyClass d = c;
    …
    // References t and d disappear here
    // Objects s and c remain on the heap
}
```

상속(Inheritance)

상속(Inheritance)과 추상화(Abstraction)

- 상속
  - 기본(base) 클래스의 기능 및 구조를 파생(derived) 클래스로 전달

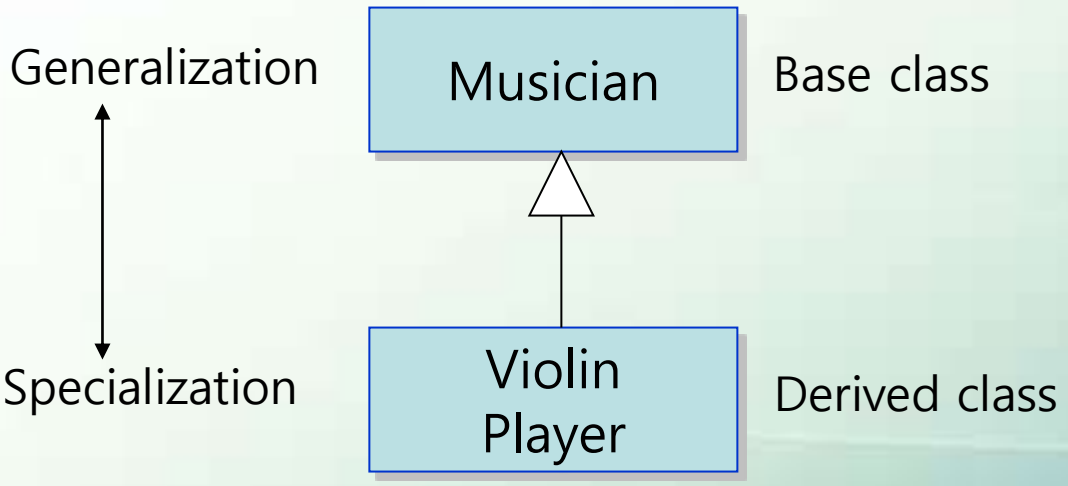


상속(Inheritance) 용어

- 상속은 이미 만들어져 있는 타입(클래스/인터페이스)를 바탕으로 새로운 타입을 만드는 기능
  - 예를 들어, Manager 및 ManualWorker 클래스는 Employee 클래스로부터 상속받을 가능성이 있음
  - Employee 클래스의 필드와 메서드는 Manager 및 ManualWorker에게 상속(재사용)될 수 있음
  - Manager 및 ManualWorker는 자신만의 멤버를 만들 수 있음
  - 기본(base) 클래스 멤버의 기능을 상속하려면 protected 접근 한정자를 붙임
- 부모 클래스와 자식 클래스
  - 부모 클래스 : 수퍼 클래스, 베이스(base) 클래스, 기본클래스
    - 특정 클래스에게 상속을 부여해줄 클래스
    - 공통 기능을 모아 놓음
  - 자식 클래스 : 서브 클래스, 파생(derived) 클래스
    - 특정 타입에게로부터 상속을 부여받은 타입

현실 세계의 부모로부터 상속(Inheritance)?

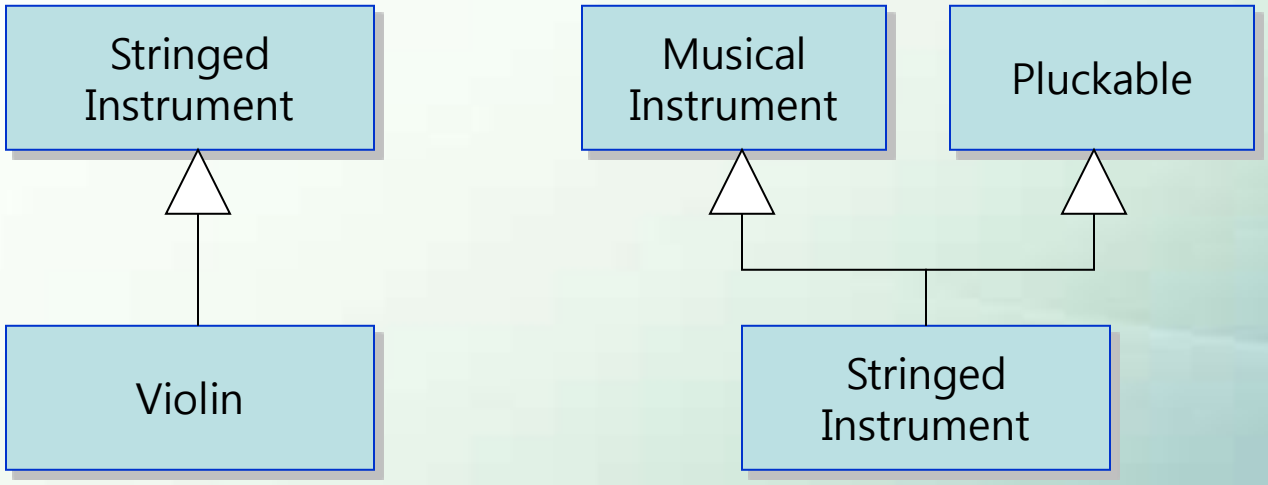
- A car “is a” type of vehicle
- An airplane “is a” type of vehicle
- Inheritance specifies an “is a kind of" relationship
  - Inheritance is a class relationship
  - New classes specialize existing classes



다중 상속 불가?

단일(Single) 상속과 다중(Multiple) 상속

- Single inheritance:
  - C#/Java는 클래스로부터 단일 상속만 지원
  - deriving from one base class
- Multiple inheritance:
  - C++ : 클래스로부터 다중 상속 지원
  - deriving from two or more base classes



기본 클래스 확장(상속) 방법

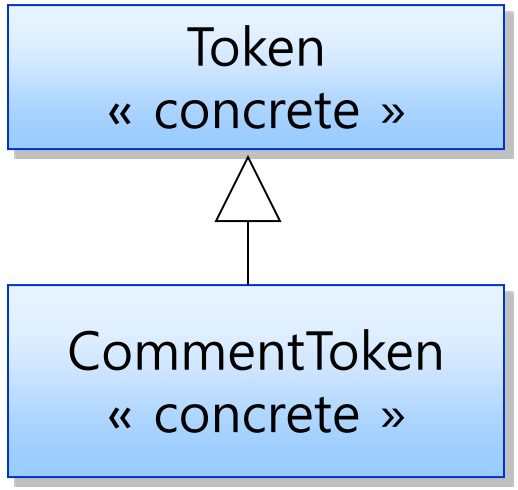
- 기본 클래스를 상속하는 문법

```
class Token
{
    ...
}
class CommentToken : Token
{
    ...
}
```

Derived class

Base class

Colon, extends



- 파생 클래스는 기본 클래스의 대부분의 멤버를 상속받음
  - public, protected 접근 한정자만 상속

기본 클래스와 파생 클래스

- C#은 콜론(:), 자바는 extends 키워드 사용

```
// Base class
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}

// Inheriting classes
class Manager : Employee
{
    public void DoManagementWork()
    { ... }
}

class ManualWorker : Employee
{
    public void DoManualWork()
    { ... }
}
```

```
// Base class
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}

// Inheriting classes
class Manager extends Employee
{
    public void DoManagementWork()
    { ... }
}

class ManualWorker extends Employee
{
    public void DoManualWork()
    { ... }
}
```

protected 접근 한정자

- 파생 클래스에서 기본 클래스의 protected 멤버 이상에만 접근 가능
  - 기본 클래스의 private 멤버에는 접근 불가
  - 상속 받지 않는 클래스에서는 protected 멤버에 접근 불가

그렇다면, 상속의 장점은?

- 개발 생산성 증가
  - 코드 중복 감소
  - 코드 재사용성 증가
- 동적 실행(Dynamic execution)
  - 다형성(polymorphism) 구현

상속 및 접근 한정자

- 상속(Inheritance)
  - 부모 타입의 기능을 자식 타입에서 재 사용
- 접근 한정자 scope 비교
  - public > protected > default > private
  - public : 공용
  - protected : 다른 패키지에서도 상속 가능
  - 생략 : 같은 패키지에서 상속 가능
  - private : 상속 불가

실습 : 상속(Inheritance)

- C:₩Java₩InheritanceDemo

```
class Parent {
    protected void Hi() { System.out.println("안녕하세요."); }
}
class Child extends Parent // Parent 클래스의 멤버 상속
{
    public void Hello() { System.out.println("반갑습니다."); }
}
public class InheritanceDemo {
    public static void main(String[] args) {
            Parent p = new Parent(); p.Hi();
            Child c = new Child(); c.Hi(); c.Hello();
    }
}
```

자식 클래스에서 부모 클래스의 멤버에 접근

- super 키워드
  - 필드
    - super.필드명
  - 메서드
    - super.메서드명
  - 생성자
    - super(???)

실습 : 부모 클래스에 접근 : super

- C:₩Java₩SuperDemo

```
class P {
	public String word; // 필드
	public P() {
		this.word = "안녕하세요.";
	}
	public P(String word) {
		this.word = word; // this.필드 = 매개변수;
	}
}
class C extends P {
	public C(){}
	public C(String word) {
		super(word); // 부모의 생성자 접근 : super();
	}
	public void say() {
		System.out.println(super.word); // 부모의 멤버에 접근 super.멤버명
	}
}
```

메서드 오버라이드(override)

- 오버라이드(override) : 부모의 메서드를 자식 클래스에서 다시 정의
  - 다시 정의
  - 재 정의
  - 덮어쓰기
  - 새로 정의
- 오버로드(overload)
  - 다중 정의 : 여러 번 정의
    - 동일한 이름의 메서드가 여러 번 정의

실습 : 메서드 오버라이드

- C:₩Java₩MethodOverrideDemo

추상 클래스(Abstract Classes)

추상 메서드와 추상 클래스

- 추상 메서드(Abstract Method)
  - 부모 클래스의 메서드 중 메서드의 프로토타입만 정의한 메서드
    - 실행 내용은 없이 메서드명만 정의됨
      - abstract void 추상메서드명(매개변수);
- 추상 클래스(Abstract Classs)
  - 추상 메서드를 하나 이상 포함한 클래스
  - abstract 키워드로 추상 클래스/메서드 표현
  - 인스턴스화될 수 없고 반드시 상속되어야 함
  - 일반 메서드 및 필드, 속성 그리고 다른 멤버를 포함할 수 있음
  - 클래스의 사양을 정해주는 사양서 역할
- 구현(Implement)
  - 추상 클래스를 상속 받은 클래스는 반드시 추상 메서드를 오버라이드 해야 함
    - 프로토타입만 정의된 메서드의 실제 내용을 구현

추상 클래스 선언

- 클래스 정의시 abstract 키워드를 사용

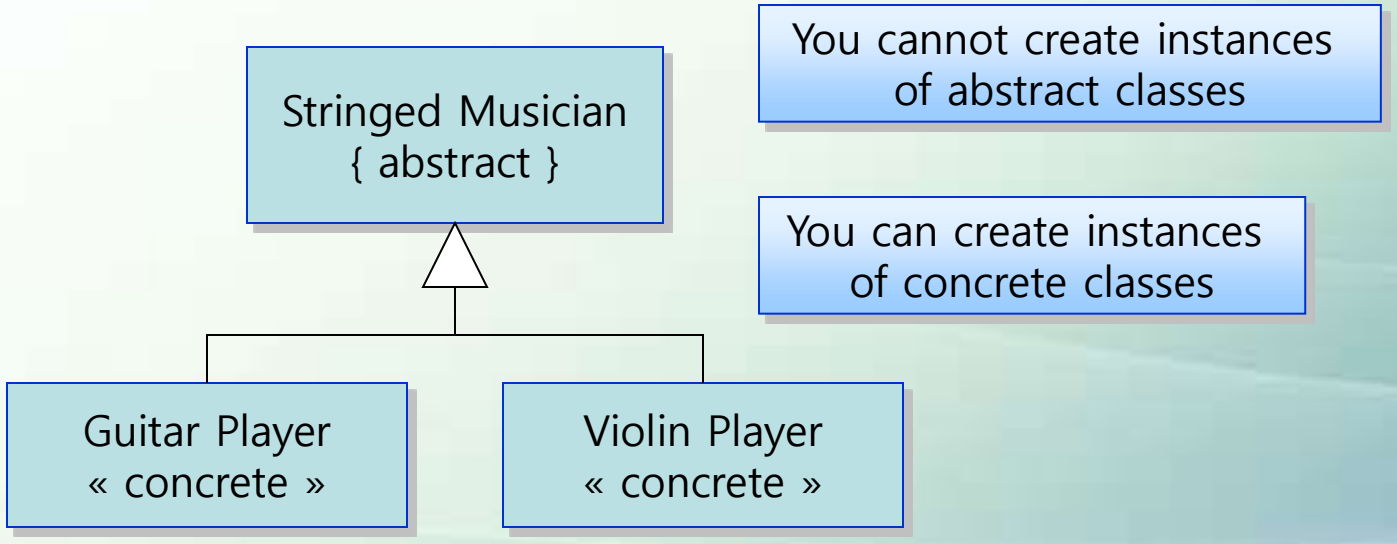
```
abstract class Token
{
    ...
}
class Test
{
    static void Main( )
    {
        new Token( );
    }
}
```

Token
{ abstract }

abstract class는 인스턴스화 될 수 없음

추상 클래스 상속

- 추상 클래스는 반드시 상속되어야 함
  - 클래스명 앞에 abstract 키워드 붙임
  - 인스턴스화 될 수 없음



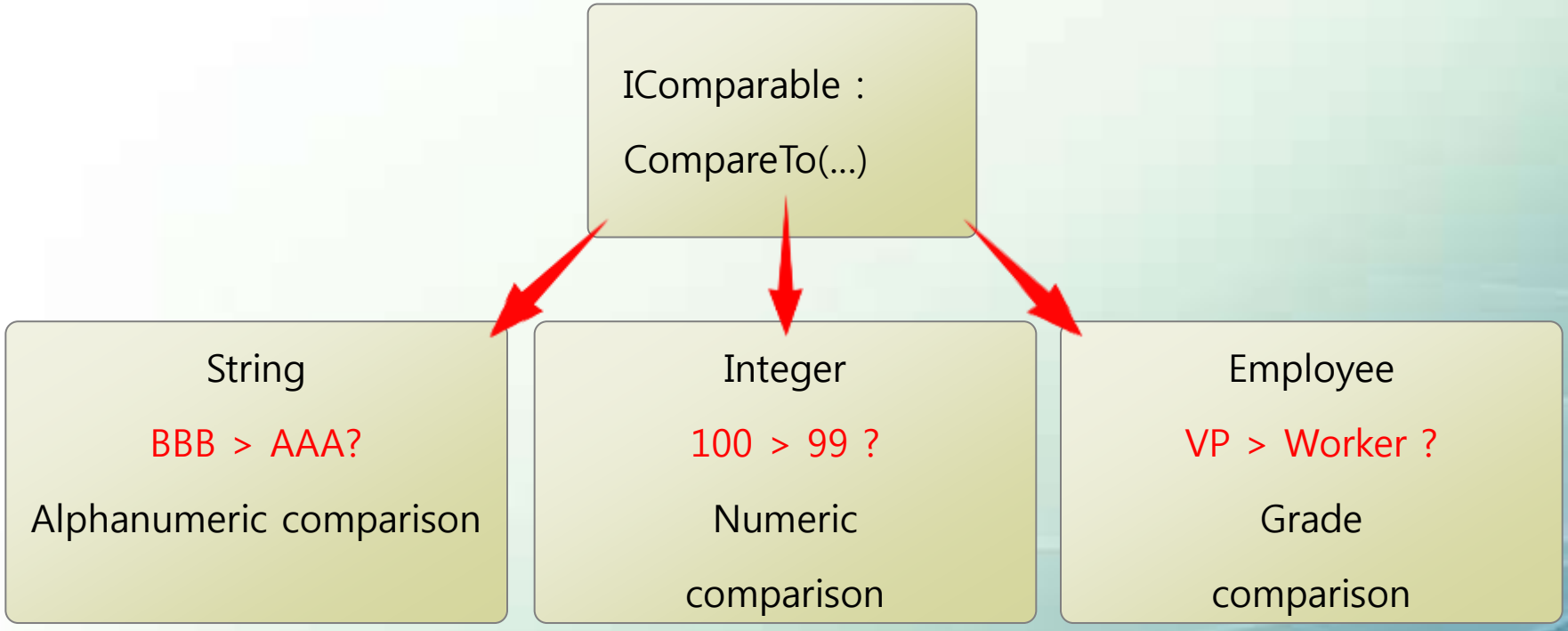
실습 : 추상 클래스와 추상 메서드

- C:₩Java₩AbstractClassDemo

인터페이스(Interface)

인터페이스(Interface)란?

- 인터페이스
  - 클래스에서 어떤 메서드를 구현해야하는가를 정의해 놓은 약속/규약/법규
  - 현실세계의 표준과 같은 개념

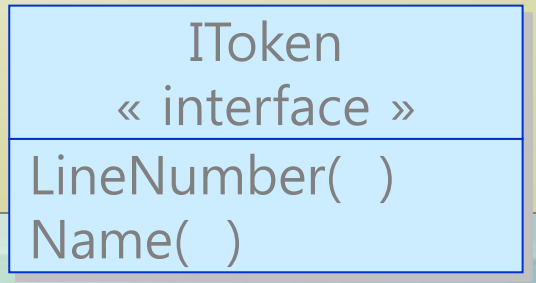


Interface 란?

- 일종의 보장 : 클래스내에 구현해라~
- 문법
  - interface 키워드 사용
  - 관례적으로 interface 이름으로 대문자 I 접두어를 사용
  - 인터페이스 메서드는 모두 public
  - 인터페이스 메서드는 메서드 시그너쳐(프로토타입)만을 기술

```
interface IToken
{
    int LineNumber( );
    string Name( );
}
```

관례적으로 접두어 I를 사용한다.



접근 한정자 없음

메서드 본문이 없음

실습 : 인터페이스 구현

- C:₩Java₩InterfaceDemo

패키지(Package)

- 현실 세계
  - 관련 있는 자동차를 묶어놓는 자동차회사와 같은 의미
- 프로그래밍 세계
  - 관련 있는 클래스 또는 인터페이스를 그룹으로 묶어 놓은 집합
    - jar 압축 파일에 포함
  - package 패키지명;
    - 소문자
  - 패키지내의 클래스 사용
    - import 문 사용
    - java.lang 패키지는 기본적으로 Import가 되어 있음

패키지 유형

- 패키지
  - Java 클래스 라이브러리(여러 클래스...)를 패키지라는 이름으로 묶어서 관리
- Java의 주요 패키지
  - Java.lang
  - Java.applet
  - Java.io
  - Java.util
  - Java.net
  - Java.awt
  - Java.awt.event

실습 : String 클래스

- C:₩Java₩StringDemo

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

            //String str = new String(" Abc Def Fed Cba ");
            String str = " Abc Def Fed Cba ";

            System.out.println(str);
            System.out.println(str.length());
            System.out.println(str.charAt(5)); // D
            ...
```

예외 처리(Exception Handling)

예외의 의미

- 프로그램 실행도중 발생하는 예상치 못한 오류
  - 오류(Error) === 예외(Exception)
- 처리되지 않은 예외는 프로그램의 실행을 중단시키는 원인
  - 강제 종료
- 신뢰도 및 안정성 측면에서 매우 중요
  - 예외 처리(Exception Handling)를 통한 강제 종료 방지

예외(오류)의 종류

- 문법 오류 :
  - 잘못된 명령어를 입력했거나, 타이핑의 실수로 발생되는 오류이다. 문법 오류를 방지하려면 많은 예제를 접해가면서 C#의 기초문법을 확실하게 이해하여야 할 것이다.
- 런타임 오류 :
  - 런타임 오류는 프로그램 작성 후 실행할 때 발생하는 오류인데, 데이터베이스를 연결할 때 데이터베이스 서버 위치, 데이터베이스 이름, 데이터베이스에 연결할 로그인 사용자 이름 및 암호 등이 잘못된 경우에 발생할 수 있다. 웹 사이트 프로그래밍에서는 데이터베이스 처리가 주된 구문이기 때문에 상당히 많은 런타임 오류가 발생할 수 있다.
- 알고리즘 오류 :
  - 주어진 문제에 대한 잘못된 해석으로 잘못된 결과를 초래하는 에러를 알고리즘 오류 또는 로직 오류라 한다. 문법 오류나 런타임 오류는 쉽게 발견해 낼 수 있지만, 알고리즘 오류는 처리 결과가 틀리게 나왔는데도 알 수 없는 경우가 많기 때문에 이 알고리즘 오류를 해결하기가 가장 어렵다. 알고리즘 오류를 해결하기 위해서는 많은 책을 통해서 코드 분석 및 많은 코드를 직접 만들어 보는 등 오류를 찾아내는 능력을 키워야 한다.

예외 처리 구문 : Try/Catch/Finally Block

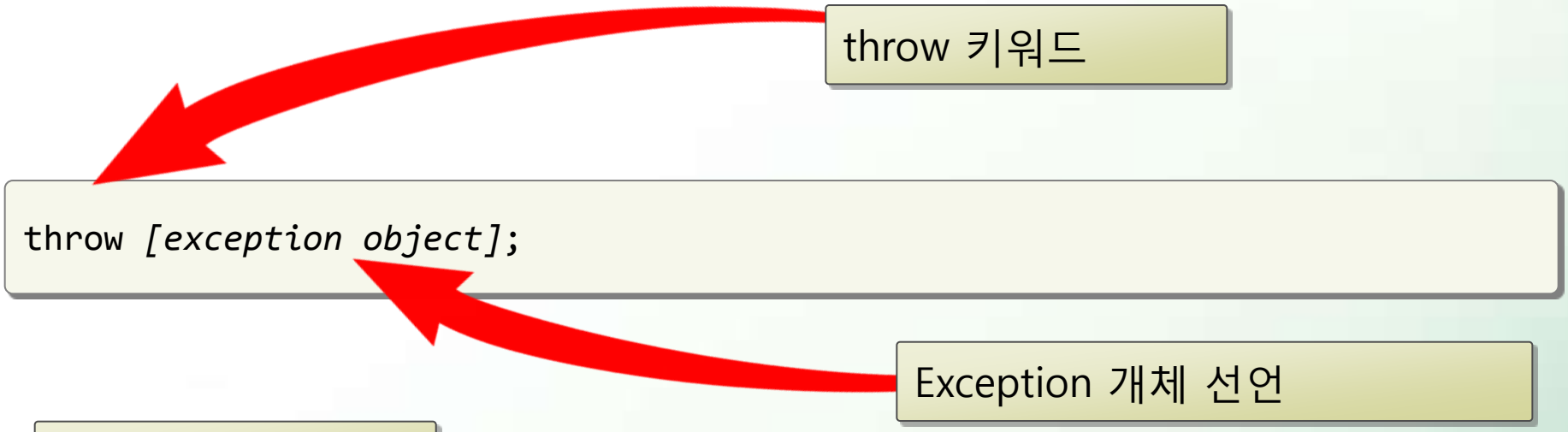
```
try
{
    // Try block.
}
catch (XXXException ex)
{
    // Catch block, can access XXXException
    // exception in ex.
}
finally
{
    //
}
```

예외가 발생할만한 구문 들어오는 곳

특정 예외가 발생시 그에 대한 처리하는 코드가 들어오는 곳

예외가 발생하던 하지 않던 항상 실행되는 구문

예외 던지기(Throwing)



예

```
public int GetTest()
{
    ...
    if (root != (int)root)
    {
        throw new Exception("예외가 발생됨.");
    }
    return (int)root;
}
```

실습 : 예외 처리

- C:₩Java₩ExceptionHandlingDemo