JAVA Tutorial

2017 winter GBC

Lee kyung su

KeyWord

Class

Interface

String

Method

DataStructure

Default Setting

JAVA 1.8⁻ 기준

~~Stream & Lamda~~

Ubuntu

sudo apt-get update

sudo apt-get install openjdk-8-jdk

$ java / $javac 로 확인

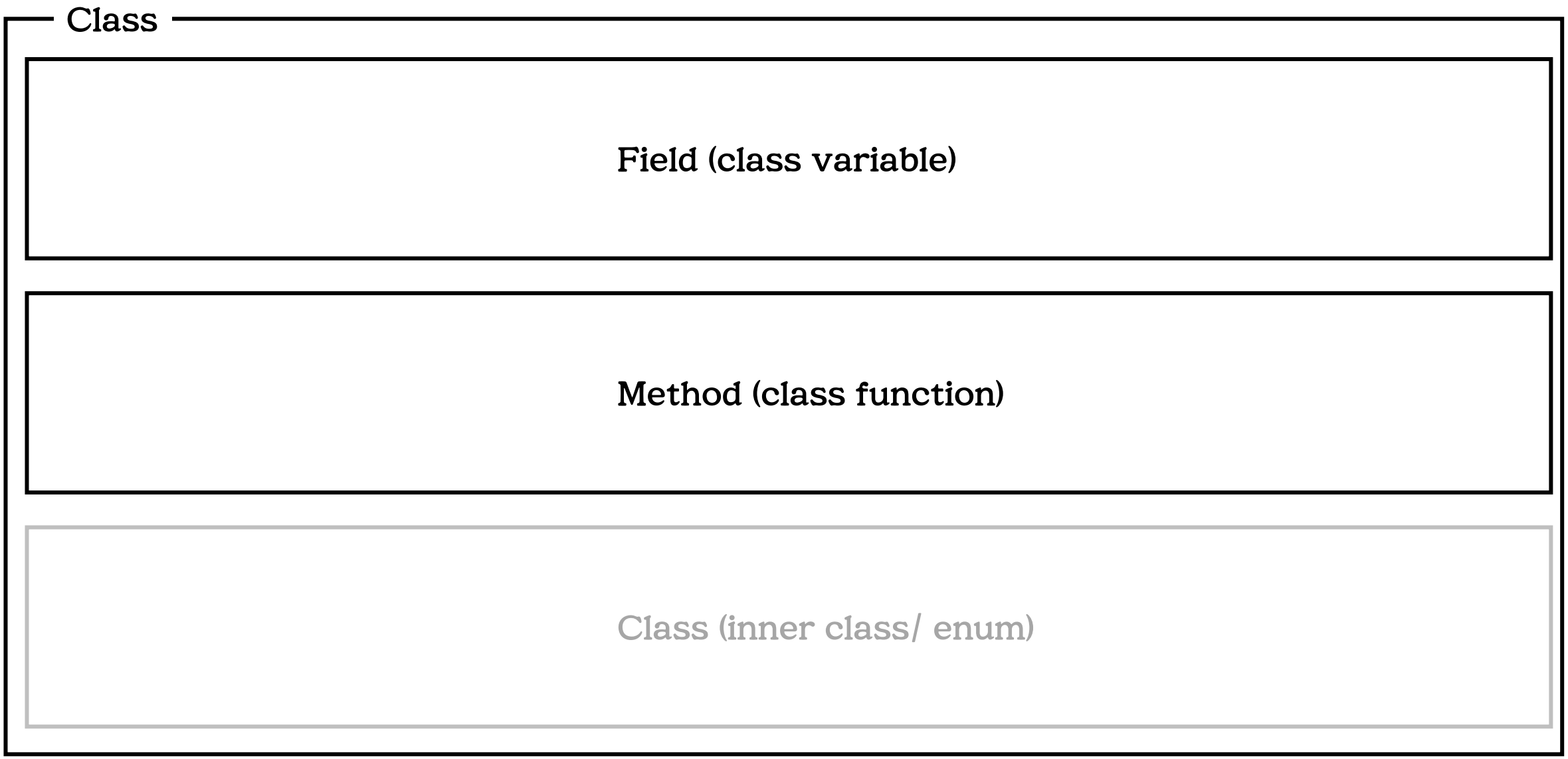
Windows

Java homepage에서 설치

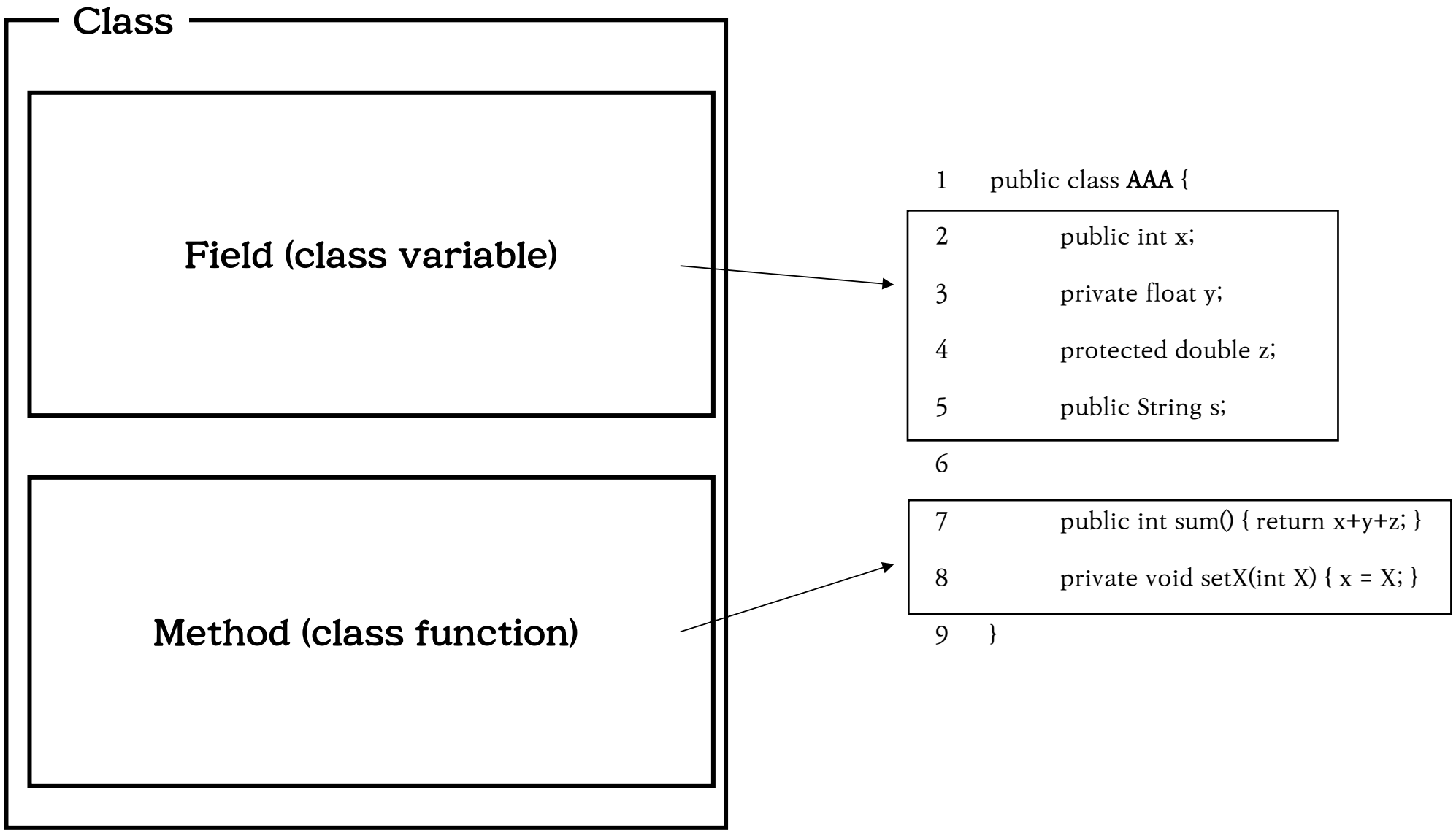
환경변수 설정

Cmd에서 java / $javac 로 확인

Structure of **Class**



Structure of **Class**




```
1   public class AAA {
2           public int x;
3           private float y;
4           protected double z;
5           public String s;
6
7           public int sum() { return x+y+z; }
8           private void setX(int X) { x = X; }
9   }
```

Field

Field = class variable

접근 제한자

public

Class 외부에서도 변수에 접근 가능

private

Class 내부에서만 변수에 접근 가능

protected

본인 또는 상속받은 class에서만 접근 가능

```
1  public class AAA {
2          public int x;
3          private float y;
4          protected double z;
5          public String s;
6
7          public int sum() { return x+y+z; }
8          private void setX(int X) { x = X; }
9  }
```

```
1  public static void main(String[] args) {
2          AAA aaa = new AAA();
3
4          aaa.x = 10;
5          ~~aaa.y = 20;~~
6          ~~aaa.z = 30;~~
7          int sum = aaa.sum();
8          ~~aaa.setX(10);~~
9  }
```

public을 제외한 private, protected는 class 외부(main)에서 사용 불가능하다!

Field

Field = class variable

Type

기본 번수형

int float, double boolean 등 사용 가능

기존 클래스

String, Math, Random 등 자바 라이브러리 사용 가능

생성 클래스

사용자가 만든 클래스 사용 가능

```
1  public class AAA {
2          public int x;
3          private float y;
4          protected double z;
5          public String s;
6          public AAA aaa;
7          private BBB bbb;
8  }
```

자기 자신클래스도 사용 가능

Method

Method = class function

Constructor (생성자)

setter/getter

Normal method

Method

Method = class function

Constructor (생성자) : class의 instance를 생성할 때 자동으로 실행되는 함수

```
1  public class AAA {
2          public AAA() {}  // 기본 생성자
3          public AAA(int x) {}
4          private AAA(int y) {}
5          protected AAA(int x, int y) {}
6
7
8
9  }
```

AAA aaa = new AAA();
AAA aaa = new AAA(1, 2);

- 인스턴스가 만들어졌을 때 기본적으로 실행 되어야 할 것들 (변수 초기화, 배열 길이 선언 등)을 해주면 좋다.
- 클래스는 기본적으로 인스턴스를 만들어야만 사용 가능하며 인스턴스를 통해서만 변수의 호출이나 메소드 호출이 가능하다.
- 생성자는 사용자가 만들어주지 않으면 기본 생성자가 생기며 이는 매개변수가 없는 꼴이다. 생성자는 여러개 존재할 수 있으며, 매개변수도 여러개 넣을 수 있다.
- 생성자는 **반드시!!** 반환 타입이 없이 선언 되어야 한다.

Method

Method = class function

Setter/ Getter : class의 내부 변수에 값을 할당할 때 사용하는 함수

```
1    public class AAA {
2        public AAA() {}  // 기본 생성자
3
4        private int x;
5        public void setX(int m) { x = m; }
6        public int getX() { return x; }
7
8        protected double y;
9        public void setY(double n) { y = n; }
10       public double getY() { return y; }
```

AAA aaa = new AAA();
aaa.setX(10) // aaa.setY(10.2) // aaa.getX() // …

- private 나 protected 변수는 클래스 밖에서 접근이 불가능하기 때문에 public으로 된 setter와 getter를 이용한다.
- Setter의 경우에는 **public void Set[변수명](type)**의 꼴이다.
- Getter의 경우에는 **public type get[변수명]()**의 꼴이다.
- 특별한 경우가 아니면 모든 변수는 private로 설정하고 setter와 getter를 만들어주는 것이 좋다.

Inheritance

Inheritance(상속) : 상속을 하면 부모클래스에 있는 변수와 메소드들을 사용할 수 있다.

```
1    public class AAA {
2        int x;
3    }
4    public class BBB extends AAA {
5        int y;
6    }
7
8    public static void main(String[] args) {
9        BBB b = new BBB();
10       b.x = 10;
11   }
```

상속을 할 때는 class [child class] extends [parent class] 로 작성한다.

이 경우에 BBB가 AAA를 상속 받았으므로, AAA가 부모 클래스, BBB가 자식 클래스가 된다.

BBB class에는 x라는 변수가 없지만 AAA에서 상속받은 x를 사용할 수 있다.

*** 상속의 가장 큰 장점은 자식 클래스를 부모 클래스로 type casting이 가능하다는 점이다.**

Inheritance

```
class Calculation{
   int z;

   public void addition(int x, int y){
      z = x+y;
      System.out.println("The sum of the given numbers:"+z);
   }

   public void Substraction(int x,int y){
      z = x-y;
      System.out.println("The difference between the given numbers:"+z);
   }

}
```

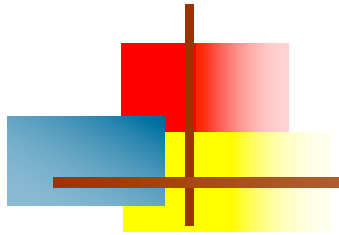
```
public class My_Calculation extends Calculation{

   public void multiplication(int x, int y){
      z = x*y;
      System.out.println("The product of the given numbers:"+z);
   }
```

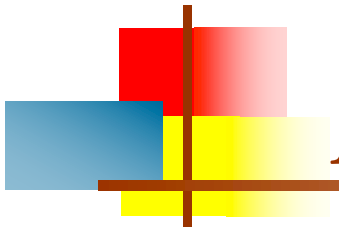
```
   public static void main(String args[]){
      int a = 20, b = 10;
      My_Calculation demo = new My_Calculation();
      demo.addition(a, b);
      demo.Substraction(a, b);
      demo.multiplication(a, b);
   }
```

result

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```



Abstract Classes and Interfaces



Abstract methods

- You can declare an object without defining it:

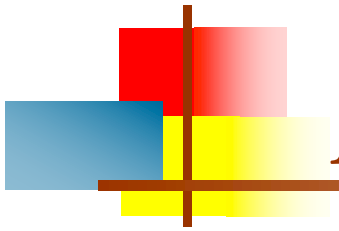
  Person p;

- Similarly, you can declare a method without defining it:

  public abstract void draw(int size);

  - Notice that the body of the method is missing

- A method that has been declared but not defined is an abstract method

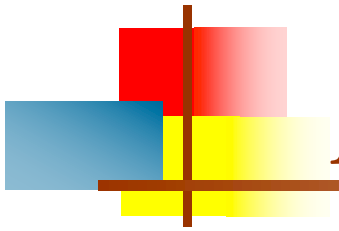


Abstract classes I

- Any class containing an abstract method is an abstract class
- You must declare the class with the keyword abstract:

  abstract class MyClass {...}

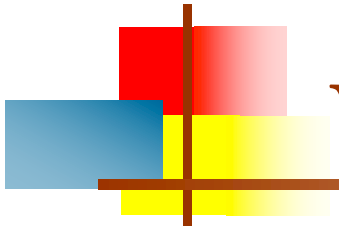
- An abstract class is incomplete
  - It has "missing" method bodies
- You cannot instantiate (create a new instance of) an abstract class



Abstract classes II

- You can extend (subclass) an abstract class
  - If the subclass defines all the inherited abstract methods, it is "complete" and can be instantiated
  - If the subclass does not define all the inherited abstract methods, it too must be abstract

- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated

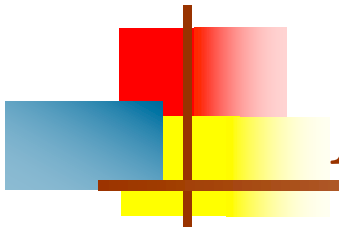


Why have abstract classes?

- Suppose you wanted to create a class Shape, with subclasses Oval, Rectangle, Triangle, Hexagon, etc.

- You don't want to allow creation of a "Shape"
  - Only particular shapes make sense, not generic ones
  - If Shape is abstract, you can't create a new Shape
  - You can create a new Oval, a new Rectangle, etc.

- Abstract classes are good for defining a general category containing specific, "concrete" classes



An example abstract class

- public abstract class Animal {

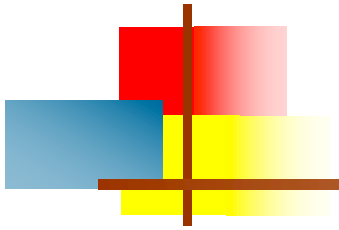
  abstract int eat();

  abstract void breathe();

  }

- This class cannot be instantiated

- Any non-abstract subclass of Animal must provide the eat() and breathe() methods

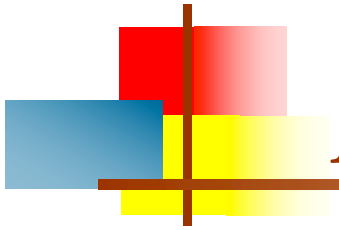


Why have abstract methods?

- Suppose you have a class Shape, but it isn't abstract
  - Shape should not have a draw() method
  - Each subclass of Shape should have a draw() method

- Now suppose you have a variable Shape figure; where figure contains some subclass object (such as a Star)
  - It is a syntax error to say figure.draw(), because the Java compiler can't tell in advance what kind of value will be in the figure variable
  - A class "knows" its superclass, but doesn't know its subclasses
  - An object knows its class, but a class doesn't know its objects

- Solution: Give Shape an abstract method draw()
  - Now the class Shape is abstract, so it can't be instantiated
  - The figure variable cannot contain a (generic) Shape, because it is impossible to create one
  - Any object (such as a Star object) that is a (kind of) Shape will have the draw() method
  - The Java compiler can depend on figure.draw() being a legal call and does not give a syntax error



A problem

- class Shape { … }
- class Star extends Shape {

  void draw() { … }

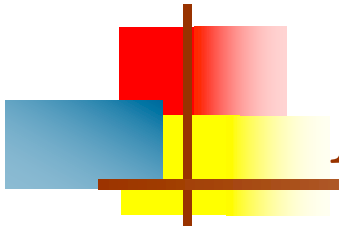
  …

  }
- class Crescent extends Shape {

  void draw() { … }

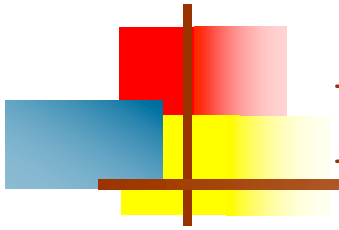
  …

  }
- Shape someShape = new Star();
  - This is legal, because a Star ***is*** a Shape
- someShape.draw();
  - This is a syntax error, because *some* Shape might not have a draw() method
  - Remember: ***A class knows its superclass, but not its subclasses***



A solution

- abstract class Shape {
  void draw();
  }
- class Star extends Shape {
  void draw() { … }
  …
  }
- class Crescent extends Shape {
  void draw() { … }
  …
  }
- Shape someShape = new Star();
  - This is legal, because a Star ***is*** a Shape
  - However, Shape someShape = new Shape(); is *no longer* legal
- someShape.draw();
  - This is legal, because every actual instance *must* have a draw() method

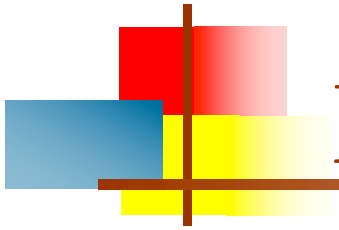


Interfaces

- An interface declares (describes) methods but does not supply bodies for them

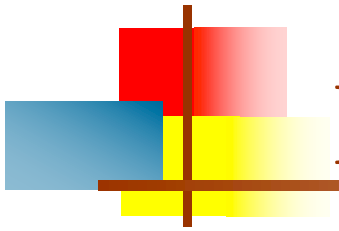
```
interface KeyListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

- All the methods are implicitly public and abstract
  - You can add these qualifiers if you like, but why bother?
- You cannot instantiate an interface
  - An interface is like a *very* abstract class—*none* of its methods are defined
- An interface may also contain constants (final variables)



Designing interfaces

- Most of the time, you will use Sun-supplied Java interfaces
- Sometimes you will want to design your own
- You would write an interface if you want classes of various types to all have a certain set of capabilities
- For example, if you want to be able to create animated displays of objects in a class, you might define an interface as:
  - public interface Animatable {
    install(Panel p);
    display();
    }
- Now you can write code that will display *any* Animatable class in a Panel of your choice, simply by calling these methods



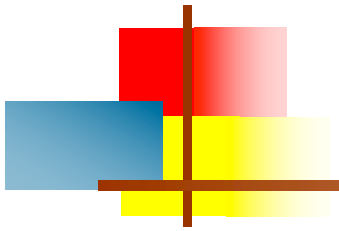
Implementing an interface I

- You extend a class, but you implement an interface

- A class can only extend (subclass) one other class, but it can implement as many interfaces as you like

- Example:

```
class MyListener
        implements KeyListener, ActionListener { … }
```



Implementing an interface II

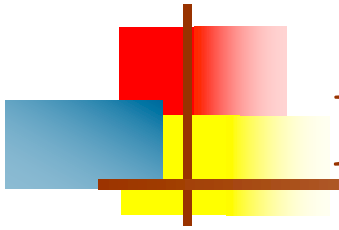
- When you say a class implements an interface, you are promising to *define* all the methods that were *declared* in the interface

- Example:

```
class MyKeyListener implements KeyListener {
    public void keyPressed(KeyEvent e) {...};
    public void keyReleased(KeyEvent e) {...};
    public void keyTyped(KeyEvent e) {...};
}
```

  - The "..." indicates actual code that you must supply

- Now you can create a new MyKeyListener



Partially implementing an Interface

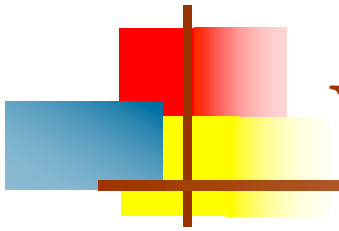
- It is possible to define some but not all of the methods defined in an interface:

```
abstract class MyKeyListener implements KeyListener {
    public void keyTyped(KeyEvent e) {...};
}
```

- Since this class does not supply all the methods it has promised, it is an abstract class

- You must label it as such with the keyword abstract

- You can even *extend* an interface (to add methods):
  - interface FunkyKeyListener extends KeyListener { ... }

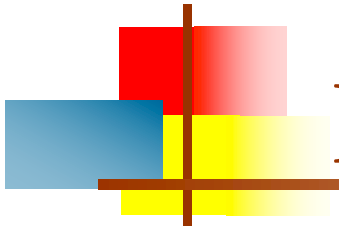


What are interfaces for?

- Reason 1: A class can only extend one other class, but it can implement multiple interfaces
  - This lets the class fill multiple "roles"
  - In writing Applets, it is common to have one class implement several different listeners
  - Example:

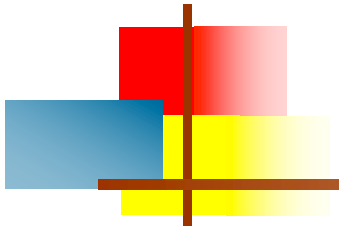
```
class MyApplet extends Applet
        implements ActionListener, KeyListener {
...
}
```

- Reason 2: You can write methods that work for more than one kind of class



How to use interfaces

- You can write methods that work with more than one class
- interface RuleSet { boolean isLegal(Move m, Board b);
  void makeMove(Move m); }
  - Every class that implements RuleSet must have these methods
- class CheckersRules implements RuleSet { // one implementation
  public boolean isLegal(Move m, Board b) { … }
  public void makeMove(Move m) { … }
  }
- class ChessRules implements RuleSet { … } // another implementation
- class LinesOfActionRules implements RuleSet { … } // and another
- RuleSet rulesOfThisGame = new ChessRules();
  - This assignment is legal because a rulesOfThisGame object *is* a RuleSet object
- if (rulesOfThisGame.isLegal(m, b)) { makeMove(m); }
  - This statement is legal because, *whatever* kind of RuleSet object rulesOfThisGame is, it *must* have isLegal and makeMove methods



instanceof

- instanceof is a keyword that tells you whether a variable "is a" member of a class or interface

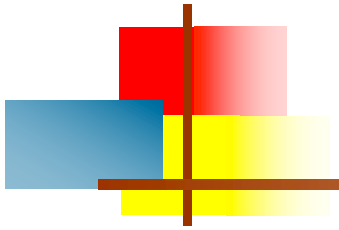
- For example, if

  ```
  class Dog extends Animal implements Pet {...}
  Animal fido = new Dog();
  ```

  then the following are all true:

  ```
  fido instanceof Dog
  fido instanceof Animal
  fido instanceof Pet
  ```

- instanceof is seldom used
  - When you find yourself wanting to use instanceof, think about whether the method you are writing should be moved to the individual subclasses



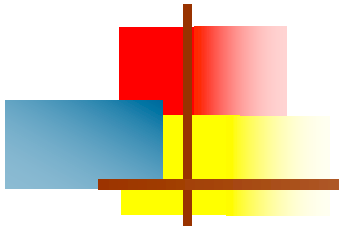
Interfaces, again

- When you implement an interface, you promise to define *all* the functions it declares

- There can be a *lot* of methods

```
interface KeyListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

- What if you only care about a couple of these methods?



Adapter classes

- Solution: use an adapter class

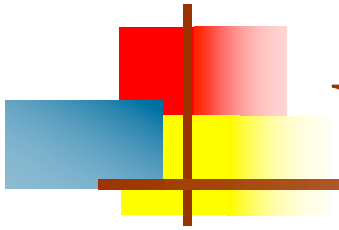
- An adapter class implements an interface and provides empty method bodies

```
class  KeyAdapter implements KeyListener {
    public void keyPressed(KeyEvent e) { };
    public void keyReleased(KeyEvent e) { };
    public void keyTyped(KeyEvent e) { };
}
```

- You can override only the methods you care about

- This isn't elegant, but it does work

- Java provides a number of adapter classes



Vocabulary

- abstract method—a method which is declared but not defined (it has no method body)
- abstract class—a class which either (1) contains abstract methods, or (2) has been declared abstract
- instantiate—to create an instance (object) of a class
- interface—similar to a class, but contains only abstract methods (and possibly constants)
- adapter class—a class that implements an interface but has only empty method bodies

프로그램 내에 자료(data)의 구분

- 프로그램 내부에서 데이터는 '변하지 않는 값'과 '변하는 값'으로 구분한다.
- '변하지 않는 값'을 **상수(Constant)**라고 부르며, 상수는 별도의 메모리 공간을 차지하지 않는다.
- '변하는 값'은 **변수(variable)**이라는 메모리 공간을 마련하고, 해당 공간 내에 특정 시점의 값을 보관 한다.

상수 (Constant)

```
final int CONST_3 = 3;
int x = 1;
x = 2 + 3;
```

- 1, 2, 3, CONST_3 은 모두 상수 혹은 값이다.
- 상수가 메모리는 차지하지 않는 이유는 해당 값을 사용(참조)하는 명령어에 값(value)이 포함되어 있기 때문이다.
- 예를 들어, 'x = 1' 이라는 수식은 'x라는 변수에 값 1을 저장한다'는 하나의 기계어 명령을 만들어내는데, 명령어 자체에 1이라는 값이 포함된다. 달리 말해, 명령어 내에 포함된 1라는 값은 프로그램 실행 중에 변할 수 없다.

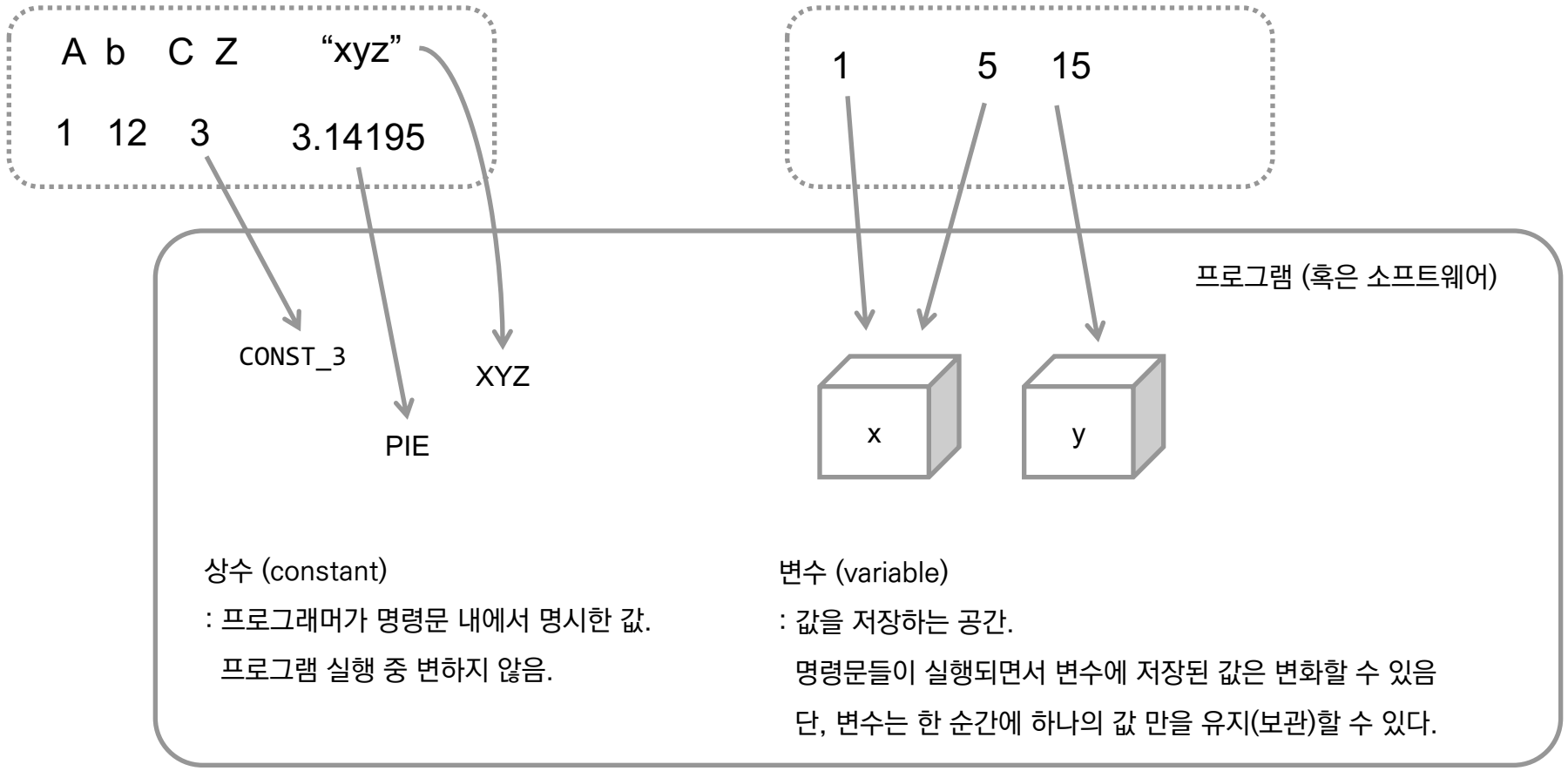
변수 (Variable)

```
int y = 0;
y = x * CONST_3;
```

- x, y 는 변수이다. 변수는 어떤 값을 저장하는 공간이다.
- 코드 예제에서 x 변수의 값은 각 행(line)이 실행되면서 1에서 5로 변한다.
- 변수는 다양한 값을 가질 수 있지만 시간의 흐름에 따라 특정 시점에 단 하나의 값을 저장한 후 다시 할당되기 전까지 그 값을 유지한다.

값, 상수 그리고 변수

값 (value) : 기록할 수 있는 모든 형태의 숫자와 문자 조합. 단, 컴퓨터 내부에서는 이진수(binary)로 기록됨



상수 (constant)

: 프로그래머가 명령문 내에서 명시한 값.

프로그램 실행 중 변하지 않음.

변수 (variable)

: 값을 저장하는 공간.

명령문들이 실행되면서 변수에 저장된 값은 변화할 수 있음

단, 변수는 한 순간에 하나의 값 만을 유지(보관)할 수 있다.

타입 (Type)

- 타입(type)이라는 개념은 데이터, 객체 등 조작이 가능한 대상(넓은 의미의 object)들을 동일한 성질(속성)로 분류한 것이다.
- 데이터 타입은 데이터의 다양한 유형들을 분류해서 식별한 것이다.
  (classification identifying one of various types of data : 위키피디아 참조)
- 다양한 프로그래밍 언어에서 수치형(number type), 문자 혹은 문자열형(character or string type), 논리형(logical type) 등의 데이터 타입을 정의하고 있다. 수치형은 정수(integer), 실수(real), 문자 및 문자열 형은 문자(character), 문자열(string), 논리형은 부울린(boolean) 등이 있으며, 수치, 문자, 문자열, 참/거짓 등의 이진수 형태로 표현된 데이터를 담을 수 있는 타입을 데이터 타입이라 한다.
- 자바의 타입(type)은 기본형 타입(primitive type)과 나머지 타입으로 분류된다.

기본형 (primitive type)	나머지 타입
- '단순한' 혹은 '순수한' 데이터를 저장하는 타입을 의미한다. - boolean, character, byte, short, int, long, float, double 등의 타입을 제공한다. - 기본형 타입은 정해진 (혹은 고정된) 크기의 공간을 사용한다.	- 기본형 이외의 모든 타입이라고 생각하자. 배열, 객체형(클래스) 등이 있다. - 복합 데이터 타입(complex data type)이라 부르기도 하는데, 여러 개의 값을 하나의 영역에 담기 때문에 그렇다. - 참조형(reference type)이라고 하는데, 실제 데이터가 변수가 데이터를 담는게 아니라 저장된 위치(메모리 주소)를 가리키고 있다. (궁금하면 포인터 개념 학습을…)

자바 기본형 분류

- 자바의 기본형은 논리형, 문자형, 정수형, 실수형으로 구분할 수 있다.
- 또한 데이터형은 메모리에서 차지하는 공간의 크기에 따라 분류할 수도 있다.

논리형	참(true)과 거짓(false) 등 2가지 값만 표현할 수 있다.
문자형	하나의 문자를 표현할 수 있다. 자바는 유니코드(Unicde) 체계를 따르므로 전세계에서 사용되는 다양한 문자를 표현할 수 있다.
정수형	소수점이 없는 수치를 표현할 수 있다.
실수형	소수점이 있는 수치를 표현할 수 있다.

		1 byte	2 byte	4 byte	8 byte
기본형	논리형	boolean			
	문자형		char		
	정수형	byte	short	int	long
	실수형			float	double
참조형	기본형을 제외한 모든 타입은 4 byte				

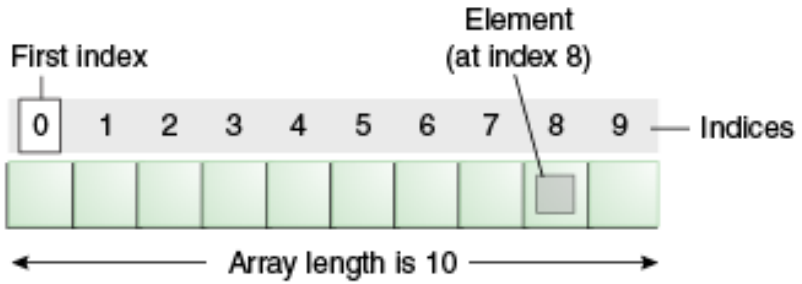
데이터 타입 별 변수의 기본 값과 범위

- 자바의 데이터 타입 별 기본 값 및 저장 가능한 값의 범위는 아래와 같다.

데이터 타입	타입 설명	기본 값	저장 가능한 값의 범위
boolean	참(true) 혹은 거짓(false)	false	false, true
char	유니코드(unicode) 한 문자	'\u0000'	'\u0000' ~ '\uffff' (0 ~ 65,535)
byte	소수점 이하 값이 없는 숫자	0	-128 ~ 127
short	-	0	-32,768 ~ 32,767
int	-	0	-2,147,483,648 ~ 2,147,483,647
long	-	0L	-9223372036854775808 ~ 9223372036854775807
float	소수점 이하 값을 포함할 수 있는 숫자	0.0f	1.4E-45 ~ 3.4028235E38
double	-	0.0d	4.9E-324 ~ 1.7976931348623157E308
object type	메모리 내에서의 객체 위치	null	0x0 ~ 0xffffffff

그리고, 배열(array)

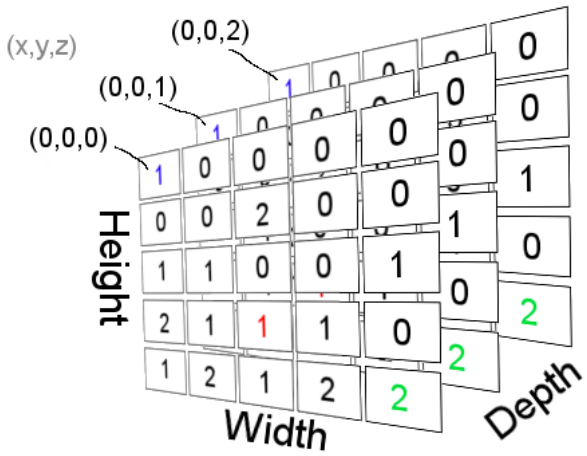
- 배열(array)은 동일한 타입의 변수 여러 개를 하나로 묶어놓은 형태를 말한다.
- 배열도 타입의 일종으로 여러 개의 값을 저장할 수 있는 공간을 정의하기 위해 사용하는 것이며,
  배열의 크기는 그것을 선언하는(만드는) 순간 고정된다. 또한, 1차원, 2차원 그리고 n차원 형태로 만들 수 있다.



[1차원 배열]

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

[2차원 배열]



[3차원 배열]

[4차원 배열]

자바 Wrapper 클래스

- 기본형 데이터를 기본형 변수가 아니라, 필요에 따라 객체 형태의 변수에 보관하고 싶을 때 사용하는 클래스
- 자바는 객체지향 언어이며, 다양한 API 들이 인자(parameter)로 객체 타입만 허용하는 경우가 있기 때문에, '기본형' 데이터를 굳이 객체로 변환해야 하는 상황이 발생하게 된다.

기본형 (primitive type)	대응하는 Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Char
boolean	Boolean
void	Void

문자열 (String)

- 복수의 문자의 집합(단어, 문장 등)을 변수에 저장하기 위한 클래스. (기본형이 아니라 객체형이다.)
- java.lang.String 클래스를 사용하며, 기본형이 아니기 때문에 문자열 조작을 위한 각종 메소드를 포함하고 있다.
- String 클래스의 자주 사용되는 메소드들은 다음과 같다.

String 클래스의 메소드	설명
equals	2개의 문자열을 비교하고 같으면, true 아니면 false 를 반환한다.
indexOf	지정한 인자(parameter) 문자열이 문자열 내에서 위치한 순서를 반환한다. 만일 없으면, -1을 반환한다.
replaceAll	인자로 지정한 정규식(regular expression)에 매칭되는 모든 부분 문자열을 교체한 후, 바뀐 문자열을 반환한다.
substring	문자열의 지정한 범위에 속한 문자열을 반환 한다.
length	문자열의 길이를 반환한다.
split	문자열을 지정한 구분자(delimiter)로 분리한 후, 결과 문자열들의 배열을 반환한다.
trim	문자열 앞/뒤 공백을 모두 제거한 후 반환한다.
charAt	문자열 내에서 지정한 문자의 위치를 반환한다.
startWith	문자열이 지정한 문자열로 시작하는지 검사하고, 맞으면 true 아니면 false 를 반환한다.

자료 구조 (data structure)

- 자료구조(data structure)란 자료의 집합입니다.
  앞서 얘기한 데이터 타입(data type)은 개별 데이터의 크기와 성질(정수만 저장한다라거나, 최대값은 얼마까지 넣을 수 있다 등등)
  만을 정의하는 겁니다. 자료 구조는 **데이터의 집합을 어떤 형태로 구성**할 것이냐를 정의하는 것입니다.
- 자료들을 조직적, 체계적으로 관리하는 기법을 말합니다.
  (돌을 어떻게 쌓아야 무너지지 않을까요? 라는 질문에 답하는 것과 같습니다.)
- 좋은 자료구조는 효율성(efficiency), 추상화(abstraction), 재사용성(reusability) 등 3가지 잇점을 제공합니다.

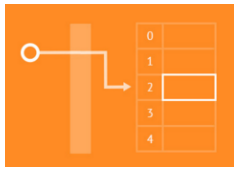
효율성(effciency)

자료를 읽고 쓰고 찾는데 걸리는 시간

같은 시간에 처리할 수 있는 작업량

추상화 (abstraction)

데이터 집합의 특징적인 형태에 따라
이름을 부여하는 것.



해시 테이블
(Hash table)

, 트리(Tree)

재사용성(reusability)

빈번하게 사용하는 '데이터 집합'의
형태를 제어하는 API를 만들어 두고,
매번 구현하지 않고 가져다 쓰는 것.

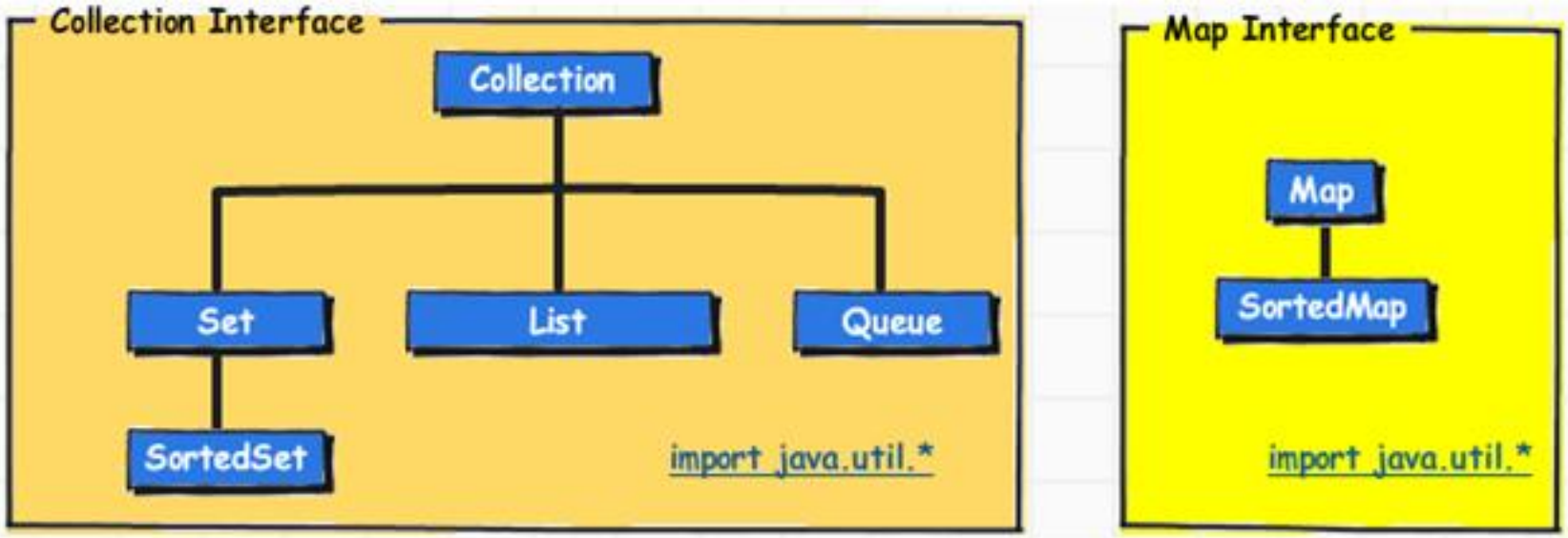
```
class Stack
```

```
class HashMap
```

```
class HashSet
```

Java Collection Framework

- 자바 컬렉션(collection) 프레임워크는 같은 타입의 객체들을 모아서 담아두는 자바 API 클래스들의 모음입니다.
- 컬렉션에 포함되어 있는 각 클래스마다 객체를 담거나 꺼내는 메소드를 제공하며 기본적으로 Set, Map, List 등 3가지 유형으로 분류할 수 있습니다. Set 은 중복된 객체를 넣을 수 없고, Map은 키(key)와 값(value)의 쌍(pair)을 담습니다. List는 객체를 순서대로 담을 수 있으면서 중복된 객체가 담겨질 수 있습니다. (반면에 Set은 저장된 객체에 순서를 부여하지 않습니다.)
- 컬렉션 프레임워크의 클래스들은 "java.util" 패키지에 담겨 있습니다. Collection 은 모든 컬렉션 클래스의 부모 인터페이스(parent interface)입니다.



이미지 출처 : http://www.jitendrazaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/

Collection, Set, List, Queue, Map

Collection: 컬렉션 계층 구조의 최상위(root) 인터페이스이며, 컬렉션은 요소(element)라고 불리우는 동일한 타입으로 만들어진 객체들의 묶음(group)입니다. 어떤 컬렉션 클래스(타입)은 묶음에 담기는 요소들의 중복을 허용하지만, 허용하지 않는 클래스도 있습니다. 또한 묶음에 담긴 요소들의 순서(order)가 지정되는 것도 있고 아닌 것도 있습니다. 자바 플랫폼은 Collection 인터페이스를 직접 구현한 클래스를 제공하지 않으며, 좀 더 구체적인 기능을 정의하는 Set 이나 List 하위 인터페이스를 제공합니다.

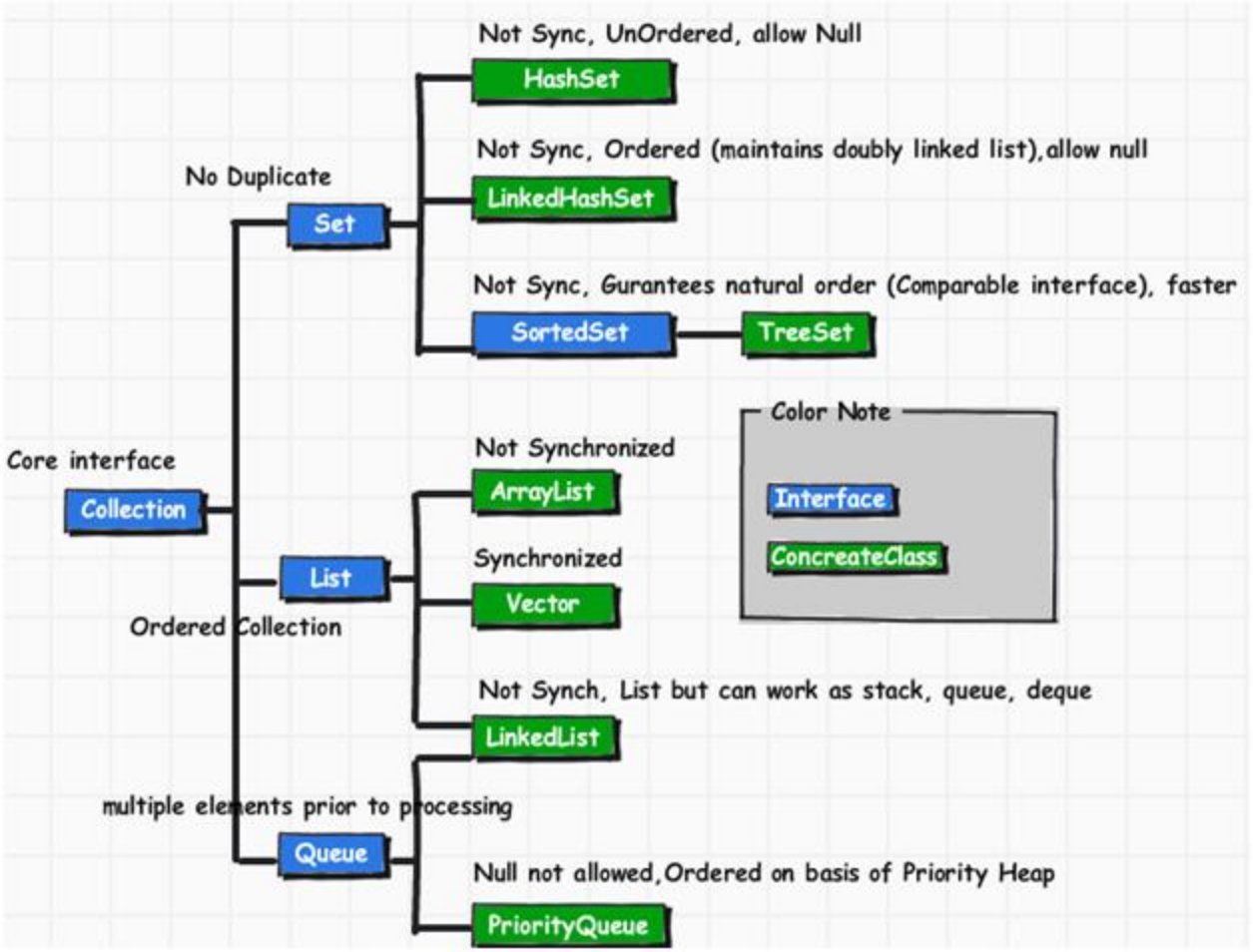
Set: 중복되지 않는 요소들을 담는 컬렉션입니다.

List: 읽고 쓰이는 요소들에 순번(sequencce)가 부여되는 컬렉션입니다. List는 중복된 객체를 담을 수 있습니다. (하나의 객체가 첫번째와 두번째 자리를 동시에 차지할 수도 있는 것입니다. 컬렉션은 객체 자체가 아니라 객체의 참조를 저장하기 때문에 중복이 가능합니다.) 요소에 순번이 부여되기 때문에 특정 위치(index or position)에 원하는 요소를 추가하거나 읽을 수 있고, 순서대로 읽고 쓰는 조작도 가능합니다.

Queue: 요소들을 가공(처리)하기 위한 목적으로 잠시 담아두는 컬렉션입니다. 컬렉션의 기본 조작(operation)에 더해서 Queue는 추가적인 삽입, 조회(추출), 검사 기능을 제공합니다. Queue 는 꼭 필요하지는 않지만 일반적으로, 선입선출(FIFO: First In First Out) 방식으로 요소를 쓰고 꺼냅니다.

Map: 객체를 키(key)와 값(value)으로 매핑(mapping) 합니다. Map 은 중복된 키(key)를 저장할 수 없으며, 각각의 키는 기껏해야 하나의 값을 가리킵니다.

컬렉션 구현 클래스 (Concrete Classes)



http://www.jitendrazaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/

컬렉션 구현 클래스 (Concrete Classes)

HashSet :

HashSet 클래스는 Set 인터페이스를 구현한 해시 테이블(hash table) 인스턴스입니다. (실질적으로는 HashMap 클래스의 인스턴스) HashSet 은 포함된 객체들에 대한 조회순서(interation)를 보장하지 않습니다. 포함된 객체 전부를 하나씩 읽어내는 작업을 할 때 마다, 읽어내는 순서가 바뀔 수도 그렇지 않을 수도 있다는 것입니다. 아울러 null 요소를 추가할 수도 있습니다.

LinkedHashSet :

Set 인터페이스의 구현체이면서 해시 테이블과 링크드 리스트(linked list)를 구현한 클래스이며, 조회순서(iteration)가 항상 일정하게끔 유지됩니다. HashSet 클래스와 다른 점은 포함된 객체들을 더블 링크드 리스트(double-linked list) 방식으로 관리한다는 점입니다. 링크드 리스트는 요소가 추가된 순서(insertion-orer)대로 조회순서가 정해집니다. 아울러 이미 추가한 객체를 다시 추가한다고 해서 객체의 조회 순서는 바뀌지 않습니다. Set의 특성 상 중복을 허용하지 않기 때문에 이미 들어 있는 객체를 add() 메소드를 이용해 추가하더라도 내부에서는 아무것도 바뀌지 않습니다.

interface SortedSet:

SortedSet에 저장되는 객체들을 저절로 올림차순(ascending order)으로 정렬된 후, 그 순서에 따라 읽혀집니다. 올림차순으로 정렬할 때, 각 객체의 순서를 판단하는 기준은 Comparable 혹은 Comparator 인터페이스의 구현체에 의해 정해집니다. 추가적으로 정렬(ordering)의 잇점을 활용할 수 있는 다양한 조작(operation)을 제공합니다.

컬렉션 구현 클래스 (Concrete Classes)

TreeSet:

TreeSet 클래스는 Set 인터페이스를 구현한 TreeMap 인스턴스입니다. 포함된 객체들을 올림차순으로 정렬한 상태로 저장합니다.

ArrayList:

List 인터페이스를 구현한 가변 길이 배열(resizable-array)입니다. 리스트(list) 관련 부가적인 조작을 제공하며, null 요소 추가를 허용합니다. List 인터페이스를 구현하는 것 뿐만 아니라, 내부에서 리스트를 저장하는데 사용되는 배열의 크기를 조정할 수 있는 메소드를 제공합니다.

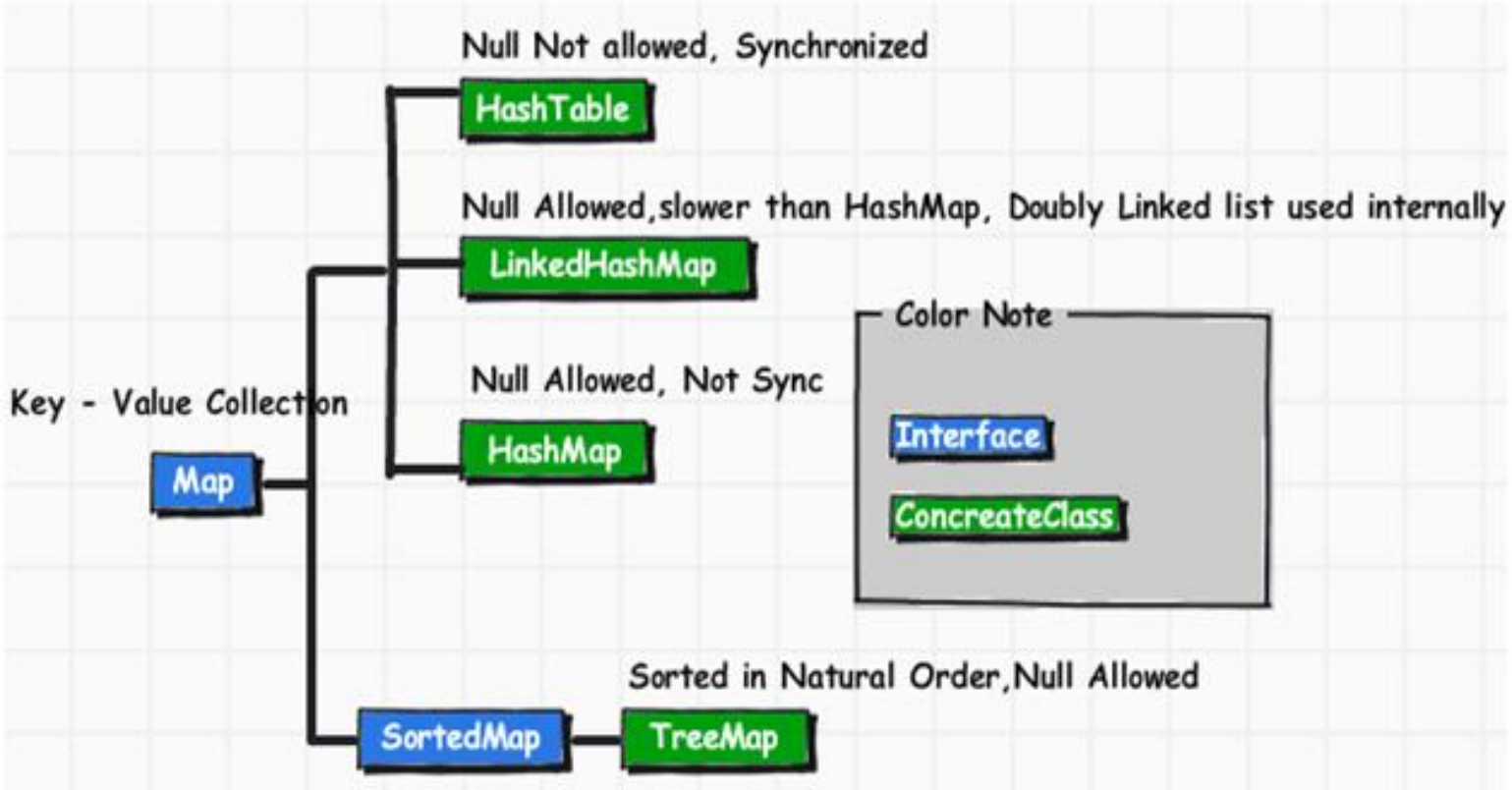
Vector:

동기화(synchronized) 처리를 제공한다는 점을 빼면 ArrayList와 거의 동일한 클래스입니다.

LinkedList:

LinkedList는 List와 Queue 인터페이스를 구현한 것입니다. 부가적인 조작을 제공하고, null 요소 입력을 허용합니다.

컬렉션 구현 클래스 (Concrete Classes)



http://www.jitendrazaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/

컬렉션 구현 클래스 (Concrete Classes)

HashMap:

The HashMap class is roughly equivalent toHashtable, except that it is unsynchronized and permits nulls.

HashTable:

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

LinkedHashMap:

Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order). Note that insertion order is not affected if a key is re-inserted into the map.

TreeMap:

This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key's class (seeComparable), or by the comparator provided at creation time, depending on which constructor is used.

Collection Summy Chart

	HashSet	LinkedHashSet	TreeSet	ArrayList	Vector	LinkedList	HashTable	LinkedHashMap	HashMap	TreeMap
Null 허용	O	O	X	O	O	O	X	O	O	O
중복 허용 (duplicate)	X	X	X	O	O	O	X	X	X	X
정렬 (sorted result)	X	X	O	X	X	X	X	X	X	O
입력 순서 대로 조회 보장	X	O	X	O	O	O	X	O	X	X