

Java Handbook

01. 클래스와 Java 프로그램의 기본구조

```
import java.util.Scanner; // package import

/*
 * Hello Program
 */
public class Hello
{
    public static int sum(int x, int y)
    {
        return x+y;
    }

    public static void main(String[] args)
    {
        final int MAX = 100;      // 기호상수선언
        int a, c;                 // 내부변수선언

        // Scanner 클래스를 이용한 입력
        Scanner in = new Scanner(System.in);
        System.out.println("a = ");
        a = in.nextInt(); // 정수읽기

        c = sum(a, MAX);
        System.out.println(a + "+" + MAX + "=" + c);
    }
}
```

- 하나의 파일에는 반드시 하나의 public class만 존재하여야 한다.
- 소스파일의 이름은 public class의 이름과 동일하여야 한다. (예: One.java)

```
class Other
{
}

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

- static 메소드는 내부변수 및 상수 이외에는 static 멤버만 접근할 수 있다.
 - static메소드인 main() 메소드에서 sum() 메소드를 호출하려면 반드시 sum() 메소드가 static이어야 한다.
- 다른 패키지의 클래스를 참조할 때에는 다음의 두 가지 방법으로 참조할 수 있다.
 - (1) 패키지이름과 함께 사용

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

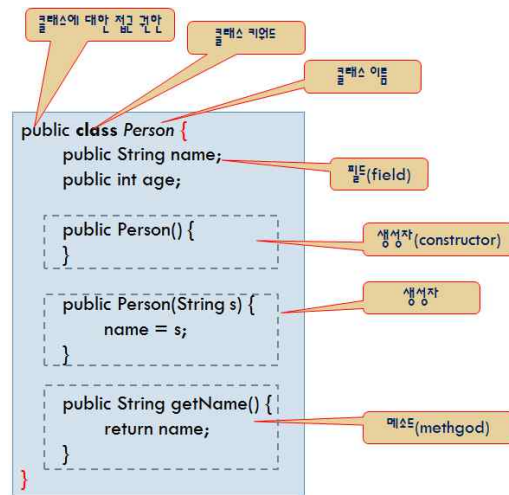
 - 이 경우에는 클래스를 참조할 때, 매번 패키지이름을 함께 사용
 - (2) 필요한 패키지 또는 클래스를 import한 후 사용

```
import java.util.Scanner;

Scanner in = new Scanner(System.in);
```

 - 이 경우에는 클래스이름을 직접 사용할 수 있음
- 클래스는 class 키워드를 사용하여 선언하며, 접근권한을 지정한다.
 - 접근지정자 : private, protected, public, 생략
 - 접근권한을 생략하면 default 접근권한을 갖는다(default는 키워드가 아님).

- 클래스는 필드(field), 생성자(constructor), 메소드(method)를 포함할 수 있다.



- 변수는 선언된 위치에 따라 다음과 같이 구분된다.
 - 필드 : 클래스내에 선언된 변수
 - 내부변수 : 메소드내에 선언된 변수
- 클래스의 모든 멤버(필드와 메소드)는 접근지정자를 사용하여 접근권한을 부여한다.
 - 접근지정자의 종류와 의미
 - private : 객체 자신 이외의 모든 클래스 접근 불가
 - default (생략시) : 같은 패키지내에 속한 클래스 이외의 모든 클래스 접근 불가
 - protected : 같은 패키지 및 상속된 서브클래스 이외의 모든 클래스 접근 불가
 - public : 모든 클래스 접근 가능
 - 접근지정자 접근범위의 크기 : private < default < protected < public
- 생성자 메소드
 - 클래스내에 선언된 메소드 중에서 주로 초기화를 수행하는 메소드
 - 객체가 생성될 때 자동으로 호출되어 실행된다.
 - 생성자 메소드는 반환형이 없으며, 클래스 이름과 동일한 이름을 갖는다.
 - 매개변수의 유무에 따라 다음과 같이 구분된다.
 - 기본생성자 : 매개변수가 없는 생성자
 - 일반생성자 : 매개변수가 있는 생성자
 - 만일 생성자가 하나도 정의되지 않으면, 컴파일러가 기본생성자를 자동으로 생성
 - 만일 생성자가 하나 이상 정의되면, 기본생성자를 자동으로 생성하지 않음

02. 객체

- 모든 클래스는 사용하기 전에 반드시 객체를 생성하여야 한다.
- 객체의 생성은 반드시 new 키워드를 이용한다.
 - Person man = new Person(); // 기본생성자를 이용한 객체생성
 ^ 기본생성자 실행
 - Person man = new Person("홍길동"); // 일반생성자를 이용한 객체생성
 ^ 일반생성자 실행

- 객체 생성의 예외 (String 클래스의 객체생성)
 - String str = new String("한국교통대학교");
 - String str = "한국교통대학교";
- new 키워드를 사용하지 않으면, 객체의 이름은 단순한 reference 변수
 - Person woman; // reference 변수선언
 - woman = new Person("영희"); // 객체 생성
- 객체가 생성되면 .연산자를 이용하여 필드와 메소드에 접근할 수 있다.
 - man.age = 50;
 - String s = man.getName();

03. 인수전달

- call-by-value : 모든 기본자료형
 - boolean, char, byte, short, int, long, float, double
- call-by-reference : 배열, 객체
 - 배열과 객체의 이름은 메모리주소를 가리키는 reference 변수

04. 메소드 오버로딩 (정적바인딩)

- 메소드의 이름이 동일한 메소드를 2개 이상 정의할 수 있는 것
 - 규칙 : 인수의 개수가 서로 다르거나, 인수의 자료형이 달라야 한다.
: 메소드의 반환형은 고려대상이 아니다.
- 메소드 오버로딩의 예와 호출

```
class Exam
{
    ...생략...

    public int getSum(int i, int j)           // ①
    {
        return i + j;
    }

    public int getSum(int i, int j, int k)    // ②
    {
        return i + j + k;
    }

    public double getSum(double i, double j) // ③
    {
        return i + j;
    }

    public static void main(String args[])
    {
        int i, j, k;

        Exam obj = new Exam();
        i = obj.getSum(1, 2);                // ①번 메소드 호출
        j = obj.getSum(1, 2, 3);             // ②번 메소드 호출
        k = obj.getSum(1.0, 2.0);            // ③번 메소드 호출
    }
}
```

- 메소드 호출시, 주어진 매개변수 (실인수)의 자료형과 일치하는 메소드를 호출

05. this와 this()

- this 레퍼런스 : 객체 자신을 의미
- this() : 다른 생성자 메소드 호출

```

class Complex
{
    private double real;
    private double img;

    public Complex(double real)
    {
        this.real = real;
        img = 0.0;
    }
    public Complex(double real, double img)
    {
        this(real);        // 다른 생성자 호출
        this.img = img;
    }
    public String toString()
    {
        return real + ":" + img;
    }
}

public class Hello
{
    public static void main(String[] args)
    {
        Complex n = new Complex(3.4, 2.0);
        System.out.println(n);
    }
}

```

- this 레퍼런스는 주로 필드명과 매개변수명이 동일 한 경우에 구분을 위해 사용
- this()는 반드시 첫 문장에 나와야 함

06. static 멤버

- static 필드 : 모든 객체에 의해 공유되는 메모리
- static 메소드 : static 필드를 처리하기 위한 메소드
 - static 메소드는 반드시 static 멤버(필드, 메소드)만 접근가능
 - this 키워드 사용 불가
- 일반 메소드는 static 필드 및 메소드 모두에 접근 가능
- static 멤버는 객체를 생성하지 않고, 클래스 이름으로 접근 가능

```

public class Hello
{
    public static void main(String[] args)
    {
        double n = Math.random(); //0~1사이의 난수발생
        System.out.println("난수 : " + n);
    }
}

```

07. 상속

- 클래스는 extends 키워드를 이용하여 다른 클래스를 상속받을 수 있다.
 - 상속하는 클래스 : super class, parent class
 - 상속받는 클래스 : sub class(derived class), child class
- 상속의 선언
 - extends 키워드 사용
- 상속범위
 - 부모클래스의 private 멤버와 생성자메소드를 제외한 모든 멤버는 상속된다.
- 다중상속을 지원하지 않는다.
- 모든 클래스는 자동으로 java.lang.Object 클래스를 상속받는다.

```

class Person
{
    String name;
    int age;

    public Person()
    {
        name = "";
        age = 0;
    }
    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public String getName()
    {
        return name;
    }
    public int getAge()
    {
        return age;
    }
    public void print()
    {
        System.out.println(name + ":" + age);
    }
}

class Student extends Person
{
    int grade;

    public Student(String name, int age, int grade)
    {
        super(name, age); // 부모클래스의 생성자 호출
        // super.name = name; // 부모클래스의 필드 참조
        // super.age = age; // 부모클래스의 필드 참조
        // this.name = name; // 상속받은 필드 참조
        // this.age = age; // 상속받은 필드 참조
        this.grade = grade;
    }
    public int getGrade()
    {
        return grade;
    }
    public void print() // method overriding
    {
        System.out.println(name + ":" + age + ":" + grade);
    }
}

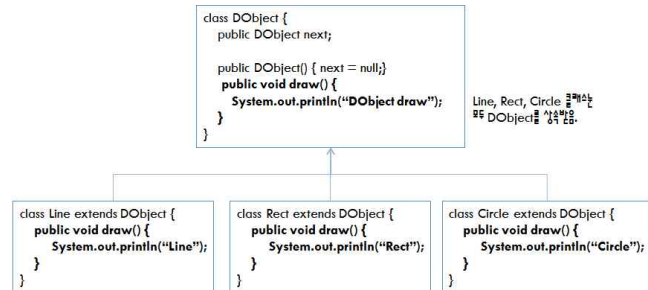
public class Hello
{
    public static void main(String[] args)
    {
        Student s = new Student("유관순", 23, 4);
        s.print();
    }
}

```

- 생성자 메소드 호출순서
 - 자식클래스의 객체가 생성될 때, 생성자는 자식클래스부터 부모클래스의 순으로 차례대로 호출된다.
- 생성자 메소드 실행순서
 - 자식클래스의 객체가 생성될 때, 생성자는 부모클래스부터 자식클래스의 순으로 차례대로 실행된다.
 - 기본적으로 부모클래스의 기본생성자를 찾아 실행한다.
 - 만일 기본생성자 메소드가 없다면, 반드시 자식클래스의 생성자 메소드 첫 줄에 `super()` 키워드를 사용하여 원하는 생성자를 호출하여야 한다.
 - 기본생성자가 있더라도, 원하는 생성자를 호출하기 위해 `super()` 키워드 사용

08. 메소드 오버라이딩 (동적바인딩)

- 부모클래스의 메소드를 무시하고, 재정의 하는 행위



- 반드시 부모클래스의 메소드와 같은 반환형, 이름, 인수를 가져야 한다.
- 접근지정자는 부모클래스의 접근지정자 보다 접근범위가 좁아질 수 없다.
 - 접근지정자의 범위 : private < default < protected < public
- static, final 메소드는 오버라이딩 될 수 없다.
- 무조건 자식클래스의 메소드가 호출되어 사용된다 (동적바인딩).
 - 부모클래스의 메소드를 호출하려면, super.메소드(); 와 같이 사용해야만 한다.

09. super와 super()

- super 레퍼런스 : 부모 클래스를 의미
 - 부모클래스의 필드 또는 메소드 참조
- super() : 부모 클래스의 생성자 메소드 호출

10. final

- final 필드 : 기호상수 정의
 - public static **final** double PI = 3.141592;
 - 반드시 초기값을 지정하여야 한다.
- final 클래스 : 상속해줄 수 없는 클래스
- final 메소드 : 메소드 오버라이딩이 불가능한 메소드

11. 업캐스팅과 다운캐스팅

- 업캐스팅 : 자식이 부모로 치환되는 것

```
Person p = new Student("유관순", 23, 4); // up-casting
```

```
Person p;  
Student s = new Student("유관순", 23, 4);  
p = s; // up-casting
```

- 자동으로 형변환되며, 부모클래스의 멤버만 접근 가능하다.
- 단, 오버라이딩 된 경우에는 자식클래스의 멤버에 접근 가능
- 다운캐스팅 : 업캐스팅된 부모가 자식으로 다시 치환되는 것
 - 업캐스팅 된 것을 다시 원래대로 되돌리는 것

```

Person p = new Student("유관순", 23, 4); // up-casting
Student s;
s = (Student)p; // down-casting

```

- 반드시 cast연산자를 이용하여 강제로 형변환 해주어야 한다.
- 다운캐스팅 되면, 자식클래스의 모든 멤버에 접근 가능하다.

12. instanceof 연산자

- 업캐스팅된 객체의 원래 클래스를 식별하기 위해 사용한다.

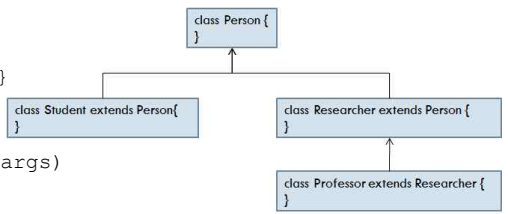
```

class Person {}
class Student extends Person {}
class Researcher extends Person {}
class Professor extends Researcher {}

public class Hello
{
    public static void main(String[] args)
    {
        Person park = new Student();
        Person kim = new Professor();
        Person lee = new Researcher();

        if (park instanceof Student) // park는 Student 타입이므로 true
            System.out.println("jee는 Student 타입");
        if (park instanceof Researcher) // park는 Researcher 타입이 아니므로 false
            System.out.println("jee는 Researcher 타입");
        if (kim instanceof Student) // kim은 Student 타입이 아니므로 false
            System.out.println("kim은 Student 타입");
        if (kim instanceof Professor) // kim은 Professor 타입이므로 true
            System.out.println("kim은 Professor 타입");
        if (kim instanceof Researcher) // kim은 Researcher 타입이기도 하므로 true
            System.out.println("kim은 Researcher 타입");
        if (kim instanceof Person) // kim은 Person 타입이기도 하므로 true
            System.out.println("kim은 Person 타입");
        if (lee instanceof Professor) // lee는 Professor 타입이 아니므로 false
            System.out.println("lee는 Professor 타입");
        if ("java" instanceof String) // "java"는 String 타입의 인스턴스이므로 true
            System.out.println("\"java\"는 String 타입");
    }
}

```



```

classDiagram
    class Person {
    }
    class Student {
    }
    class Researcher {
    }
    class Professor {
    }
    Person <|-- Student
    Person <|-- Researcher
    Researcher <|-- Professor

```

13. 추상메소드 및 클래스

- 추상메소드
 - 선언만 되고, 구현되지 않은 메소드
 - abstract 키워드를 사용한다.
 - **abstract** public int getValue();
- 추상클래스
 - 추상 메소드를 하나이상 가지고 있는 클래스
 - 반드시 abstract class로 선언해야 한다
 - abstract class로 선언된 클래스
 - 객체를 생성하지 못한다.
- 추상클래스를 상속받는 자식클래스
 - 반드시 추상메소드를 모두 구현하여야 한다.
 - 추상메소드를 모두 구현하지 않으면, 상속받은 클래스도 추상클래스가 되므로 abstract class로 선언하여야 한다.
 - 추상메소드를 모두 구현하면, 추상클래스가 되지 않는다.

14. interface

- 모든 메소드가 추상메소드인 클래스 => 이것을 인터페이스라고 부른다.
- 인터페이스는 기호상수와 추상메소드만 갖는다.
- 메소드 선언시 `abstract`, `public` 생략 가능 (모든 메소드는 `public`으로 간주)
- 객체를 생성하지 못한다.
- 인터페이스간에 상속이 가능하며, 다중상속도 허용된다. (`extends` 키워드 사용)

```
interface MobilePhone
{
    boolean sendCall();
    boolean receiveCall();
    boolean sendSMS();
    boolean receiveSMS();
}

interface MP3
{
    void play();
    void stop();
}

interface MusicPhone extends MobilePhone, MP3
{
    void playMP3RingTone();
}
```

- 클래스에서 인터페이스 상속

```
interface USBMouseInterface
{
    void mouseMove();
    void mouseClicked();
}

interface RollMouseInterface
{
    void roll();
}

public class MouseDriver implements RollMouseInterface, USBMouseInterface {
    void mouseMove() { .... }
    void mouseClicked() { ... }
    void roll() { ... }
    // 추가적으로 다른 메소드를 작성할 수 있다.
}
```

- 인터페이스를 상속하면, 반드시 모든 추상메소드를 구현하여야 한다.
- 인터페이스 상속 (구현한다고 말한다)
- 다중인터페이스 상속이 가능하다.
- 일반상속과 인터페이스 구현을 함께 할 수 있다.

```
class MobilePhone
{
    ...
}

interface MP3
{
    public void play();
    public void stop();
}

public class Hello extends MobilePhone implements MP3
{
    ...
    public void play() {}
    public void stop() {}
}
```


15. inner class, 익명 inner class

- 클래스 내부에 클래스를 중복해서 정의할 수 있다.
- inner class

```
class Outer
{
    ...

    private class Inner
    {
        ...
    }
}
```

- 익명 inner class
 - 좌측과 같이 클래스 선언과 객체생성을 하는 경우, 오른쪽과 같이 클래스선언과 객체생성을 함께 할 수 있다.
 - 클래스명이 필요없는 inner class를 정의하고 객체를 생성하는 경우에 사용한다.
 - 형식 :

```
new 부모클래스명 (매개변수) // 생성자호출
{
    필드;
    메소드;
}
```

```
class MyOuter implements SuperCls
{
    ...
    ...
    ...
}

new MyOuter()
```

```
class Hello
{
    ...

    new SuperCls(...)
    {
        ...
    }
}
```

16. package

- 서로 관련된 클래스와 인터페이스를 하나의 디렉토리에 묶어놓은 것을 말한다.
- 패키지 만들기
 - 패키지로 사용할 디렉토리 생성
 - 소스파일내 첫줄에 package 선언

```
package MyPackage;

class A
{
    ...
}

class B
{
    ...
}

public class Hello
{
    ...
}
```

- 컴파일한 class 파일을 패키지 디렉토리에 저장
- 패키지 선언을 생략하면 default 패키지가 된다.