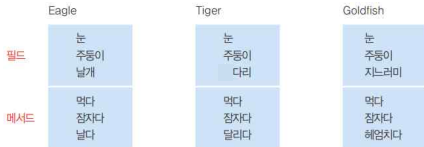




상속

# 상속이란

## ■ 필요성



(a) 상속을 적용하기 전



(b) 상속을 적용한 후

# 상속이란

## ■ 상속과 클래스 멤버

- 자식 클래스는 부모 클래스에서 물려받은 멤버를 그대로 사용하거나 변경할 수 있고, 새로운 멤버도 추가할 수 있다.
- 기존의 클래스로 새로운 클래스를 작성하는 것(코드의 재사용)
- 두 클래스의 부모와 자식으로 관계를 맺어주는 것
- 자식 클래스는 대체로 부모 클래스보다 속성이나 동작이 더 많다.
- `class Parent { }`
- `class Child extends Parent { }`



# 클래스 상속

## ■ 부모·자식 클래스의 관계 : 상속과 포함

- 상속은 is-a 관계

is-a(상속 관계)	has-a(소유 관계)
<ul style="list-style-type: none"><li>• 원은 도형이다.</li><li>• 사과는 과일이다.</li><li>• Tandem은 Bike다.</li></ul>	<ul style="list-style-type: none"><li>• 자동차는 엔진이 있다.</li><li>• 스마트폰은 카메라가 있다.</li><li>• 컴퓨터는 마우스가 있다.</li></ul>

- has-a 관계 ( 포함 관계 ) : 클래스의 멤버로 참조변수를 선언하는 것
- 작은 단위의 클래스를 만들고, 이 들을 조합해서 클래스를 만든다.

```
class Engine {  
    ...  
    ...  
}
```



```
class Car {  
    Engine engine; // Car 클래스가 Engine 객체를 포함  
    ...  
}
```

# 클래스 상속

## ■ 상속의 선언

- extends 키워드 사용

부모 클래스

```
class SuperClass {  
    // 필드  
    // 메서드  
}
```

상속

자식 클래스

```
class SubClass extends SuperClass {  
    // 필드  
    // 메서드  
}
```

- 자바는 다중 상속은 안되고 단일 상속만을 허용함.(C++은 다중 상속 허용)
- 단 비중이 높은 클래스 하나만 상속관계로 하고, 나머지는 포함관계로 하면 다중상속을 하는 것처럼 사용할 수도 있다.

```
class SubClass extends SuperClass1, SuperClass2 {  
}
```

# 클래스 상속

## ■ 현실 세계와 상속 적용

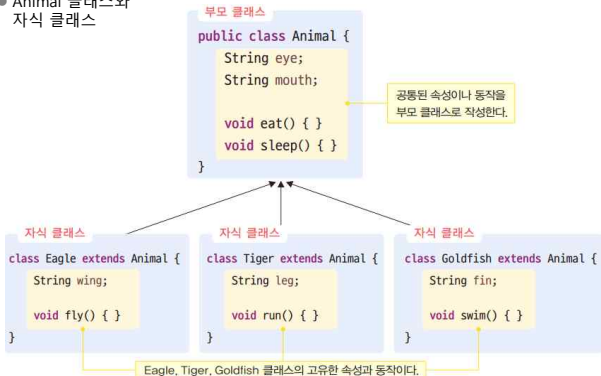
- 객체 지향의 상속을 적용할 수 있는 현실 세계의 예

부모 클래스	자식 클래스
Animal	Eagle, Tiger, Goldfish
Bike	MountainBike, RoadBike, TandemBike
Circle	Ball, Cone, Cylinder
Drinks	Beer, Coke, Juice, Wine
Employee	RegularEmployee, TemporaryEmployee, ContractEmployee

# 클래스 상속

## ■ 현실 세계와 상속 적용

- Animal 클래스와 자식 클래스



# 클래스 상속

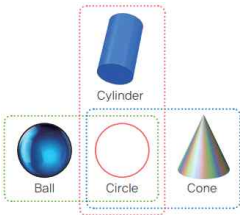
## ■ 현실 세계와 상속 적용

- 예제 : [sec02/Circle](#) [sec02/Ball](#), [sec02/InheritanceDemo](#)

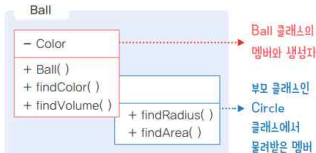
- 문제점은?

원 :  
반지름이 10.0센티이다.  
넓이는  $(\pi * \text{반지름} * \text{반지름})$ 이다.

공 :  
반지름이 10.0센티이다.  
빨간색 공이다.  
넓이는  $(\pi * \text{반지름} * \text{반지름})$ 이다.  
부피는  $4/3 * (\pi * \text{반지름} * \text{반지름} * \text{반지름})$ 이다.



(a) Circle 클래스와 자식 클래스의 크기



(b) 자식 클래스인 Ball의 멤버

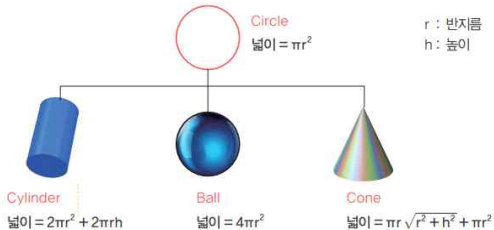


# 메서드 오버라이딩

## ■ 의미

- 메서드 오버라이딩(Method Overriding)은 물려받은 메서드를 자식 클래스에게 맞도록 수정하는 것

## ■ 예: 넓이를 구하는 findArea() 메서드



# 메서드 오버라이딩

## ■ 규칙

- 부모 클래스의 메서드와 동일한 시그니처를 사용한다. 심지어 반환 타입까지 동일해야 한다.
- 부모 클래스의 메서드보다 접근 범위를 더 좁게 수정할 수 없다.
- 추가적인 예외(Exception)가 발생할 수 있음을 나타낼 수 없다.

## ■ 오버라이딩 불가

- private 메서드 : 부모 클래스 전용이므로 자식 클래스에 상속되지 않는다.
- 정적 메서드 : 클래스 소속이므로 자식 클래스에 상속되지 않는다.
- final 메서드 : final 메서드는 더 이상 수정할 수 없으므로 자식 클래스가 오버라이딩할 수 없다.

# 메서드 오버라이딩

## ■ 예제

- [sec03/Circle](#), [sec03/Ball](#), [sec03/InheritanceDemo](#)

원 :

반지름이 10.0센티이다.

넓이는  $(\pi * \text{반지름} * \text{반지름})$ 이다.

공 :

반지름이 10.0센티이다.

빨간색 공이다.

넓이는  $4 * (\pi * \text{반지름} * \text{반지름})$ 이다. — 오버라이딩한 메서드의 결과로 올바른 값이다.

부피는  $4/3 * (\pi * \text{반지름} * \text{반지름} * \text{반지름})$ 이다.

- 어노테이션

@Override

어노테이션이다.

```
void findArea() {
```

```
    // 부모 클래스에서 상속받은 메서드를 수정한 코드
```

```
}
```

오버라이딩한 메서드이다.

# 메서드 오버라이딩

## ■ 부모 클래스의 멤버 접근

- 자식 클래스가 메서드를 오버라이딩하면 자식 객체는 부모 클래스의 오버라이딩된 메서드를 숨긴다.
- 그 숨겨진 메서드를 호출하려면 `super` 키워드를 사용한다.
- `super`는 현재 객체에서 부모 클래스의 참조를 의미

## ■ 예제

- [sec03/spr/Circle](#),  
[sec03/spr/Ball](#),  
[sec03/spr/InheritanceDemo](#)

원 :

반지름이 10.0센티이다.

넓이는  $(\pi * \text{반지름} * \text{반지름})$ 이다.

공 :

반지름이 10.0센티이다.

빨간색 공이다.

반지름이 10.0센티이다.

findRadius()의 결과이다.

넓이는  $(\pi * \text{반지름} * \text{반지름})$ 이다.

super.findArea()의 결과이다.

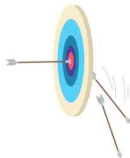
넓이는  $4 * (\pi * \text{반지름} * \text{반지름})$ 이다.

Ball 클래스의 19행 결과이다.

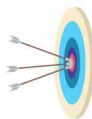
부피는  $4/3 * (\pi * \text{반지름} * \text{반지름} * \text{반지름})$ 이다.

# 메서드 오버라이딩

## ■ 메서드 오버라이딩과 메서드 오버로딩



(a) 메서드 오버라이딩



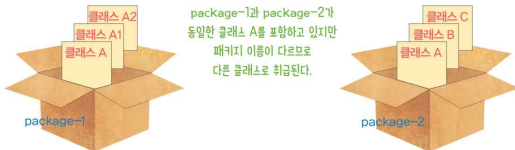
(b) 메서드 오버로딩

비교 요소	메서드 오버라이딩	메서드 오버로딩
메서드 이름	동일하다.	동일하다.
매개변수	동일하다.	다르다.
반환 타입	동일하다.	관계없다.
상속 관계	필요하다.	필요 없다.
예외와 접근 범위	제약이 있다.	제약이 없다.
바인딩	호출할 메서드를 실행 중 결정하는 동적 바인딩이다.	호출할 메서드를 컴파일할 때 결정하는 정적 바인딩이다.

# 패키지

## ■ 의미

- 클래스 파일을 묶어서 관리하기 위한 수단으로 파일 시스템의 폴더를 이용
- 패키지에 의한 장점
  - 패키지마다 별도의 이름 공간(Namespace)이 생기기 때문에 클래스 이름의 유일성을 보장.
  - 클래스를 패키지 단위로도 제어할 수 있기 때문에 좀 더 세밀하게 접근 제어



## ■ 대표적인 패키지

- java.lang 패키지는 import 문을 선언하지 않아도 자동으로 임포트되는 자바의 기본 클래스를 모아 둔 것
- java.awt 패키지는 그래픽 프로그래밍에 관련된 클래스를 모아둔 것
- java.io 패키지는 입출력과 관련된 클래스를 모아 둔 것

# 패키지

## ■ 패키지 선언

- 주석문을 제외하고 반드시 첫 라인에 위치
- 패키지 이름은 모두 소문자로 명명하는 것이 관례. 일반적으로 패키지 이름이 중복되지 않도록 회사의 도메인 이름을 역순으로 사용

```
package com.hankuk.people;
```

패키지 이름이다. 주로 도메인 이름을 역순으로 사용한다.

패키지를 선언하는 키워드이다.

# 패키지

## ■ 패키지 선언

- 예 : 소스 코드와 컴파일(명령창)

[예제 6-6] 패키지와 클래스

C:\Temp\Yona.java

```
01 package com.hankuk.people;
02
03 public class Yona {
04     public static void main(String[] args) {
05         System.out.println("안녕, 연아!");
06     }
07 }
```

패키지 이름은 반드시 첫 행에 위치해야 한다.

C:\Temp> javac -d . Yona.java

현재 폴더를 의미한다.

컴파일 명령어이다.

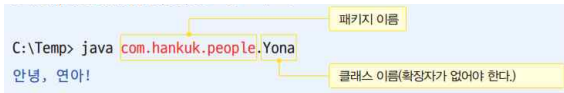
패키지 폴더를 생성하는 옵션이다.



# 패키지

## ■ 패키지 선언

- 예 : 실행 결과(명령창)

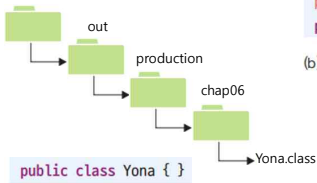


# 패키지

## ■ 패키지 선언

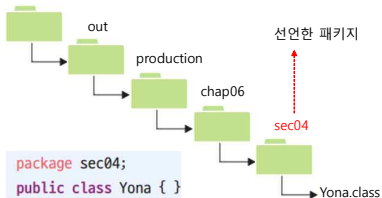
- 예 : 실행 결과(인텔리J 아이디어)

D:\workspace



(a) 패키지를 선언하지 않을 때

D:\workspace



(b) 패키지를 선언할 때

# 패키지

## ■ 패키지의 사용

- 다른 패키지에 있는 공개된 클래스를 사용하려면 패키지 경로를 컴파일러에게 알려줘야 한다.

com.hankuk.people 패키지

```
public class ShowWorldPeople {  
    public static void main(String[] args) {  
        com.usa.people.Lincoln greatman  
        = new com.usa.people.Lincoln();  
    }  
}
```

com.usa.people 패키지

```
public class Lincoln { }
```

패키지 이름을 접두어로 사용해  
다른 패키지에 있는 클래스를 이용한다.

# 패키지

## ■ import 문

- 패키지의 경로를 미리 컴파일러에게 알려주는 문장

```
import 패키지이름.클래스;
```

또는

```
import 패키지이름.*;
```

- import 문은 소스 파일에서 package 문과 첫 번째 클래스 선언부 사이에 위치
- 주의 사항

```
import com.hankuk.*;           // com.hankuk 패키지에 포함된 모든 클래스이다.  
import com.hankuk.people.*;    // com.hankuk.people 패키지에 포함된 모든 클래스이다.
```

# 패키지

## ■ import 문

### ● 예제

```
package com.hankuk.people;
```

```
import com.usa.people.Lincoln;
```

컴파일러에 Lincoln 클래스의 경로를 알려 준다.

```
public class ShowWorldPeople {  
    public static void main(String[] args) {  
        Lincoln greatman = new Lincoln();  
    }  
}
```

import 문으로 경로를 알려 주었으므로 com.usa.people  
이라는 경로 정보는 필요 없다.

# 패키지

## ■ 정적 import 문

- 패키지 단위로 임포트하지 않고 패키지 경로와 정적 메서드를 함께 임포트
- 예제 : [sec04/StaticImportDemo](#)

# 자식 클래스와 부모 생성자

- 자식 생성자를 호출하면 부모 생성자도 자동으로 호출

```
class Box {
```

```
→ public Box() {
```

```
    ...
```

```
    }
```

```
}
```

② 부모 클래스의 생성자를 호출한다.

```
class ColoredBox extends Box {
```

```
→ public ColoredBox() {
```

```
    ...
```

```
    }
```

```
}
```

① 자식 클래스의 생성자를 호출한다.

```
public class BoxDemo {
```

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

```
        ColoredBox b = new ColoredBox();
```

```
    }
```

```
}
```

③ 부모 클래스의 생성자를 마치고,  
자식 클래스의 생성자로 돌아온다.

④ 자식 클래스의 생성자를 마친다.

# 자식 클래스와 부모 생성자

- 자식 생성자는 첫 행에 부모 생성자 호출 코드가 있음

```
class Box {
```

```
    public Box() {
```

```
        ...
```

```
    }
```

```
}
```

```
class ColoredBox extends Box {
```

```
    public ColoredBox() {
```

```
    }
```

```
}
```



super( )에 의해 부모 생성자  
Box( )를 호출한다.

없다면 컴파일러가 super(): 코드를 추가한다.



# 자식 클래스와 부모 생성자

- 자식 생성자는 첫 행에 부모 생성자 호출 코드가 있음

```
class Box {  
    public Box(String s) {  
        ...  
    }  
}
```

생성자가 있으므로 컴파일러는 디폴트 생성자 Box()를 추가하지 않는다.

```
class ColoredBox extends Box {  
    // 생성자가 없음  
}
```

생성자가 없으므로 컴파일러가 디폴트 생성자 ColoredBox()를 추가한다.  
ColoredBox()는 먼저 부모 생성자 super()를 호출한다. 그런데 부모 클래스에는 Box(String)은 있지만 Box() 생성자는 없어 오류가 발생한다.

# 자식 클래스와 부모 생성자

## ■ 자식 생성자의 첫 행에 명시적 부모 생성자 호출

```
class Box {
```

```
// 생성자가 없음
```

생성자가 없으므로 컴파일러가 디폴트 생성자 Box()를 추가한다.

```
}
```

```
class ColoredBox extends Box {
```

```
    public ColoredBox(String s) {
```

```
        super(s);
```

부모 클래스에 Box(String) 생성자가 없어 오류가 발생한다.

```
    }
```

```
}
```



```
class Ball extends Circle {
```

```
    public Ball(String s) {
```

```
        System.out.println("어이쿠!");
```

```
        super(s);
```

super(s) 생성자 호출이 맨 먼저 나타나야 한다.

```
    }
```

```
}
```

# 자식 클래스와 부모 생성자

## ■ 자식 생성자의 첫 행에 명시적 부모 생성자 호출

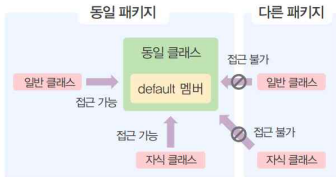
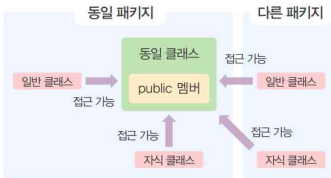
- 예제 : [sec05/AnimalDemo](#)



```
동물 : 원숭이  
포유류 : 원숭이  
동물 : 사자  
포유류 : 사자
```

# 상속과 접근 제어

## ■ 접근 지정자의 접근 범위



# 상속과 접근 제어

## ■ 접근 지정자의 접근 범위

접근 지정자	동일 클래스	동일 패키지	자식 클래스	다른 패키지
public	○	○	○	○
protected	○	○	○	×
없음	○	○	×	×
private	○	×	×	×

## ■ 접근 지정자 사용 시 주의 사항

- private 멤버는 자식 클래스에 상속되지 않는다.
- 클래스 멤버는 어떤 접근 지정자로도 지정할 수 있지만, 클래스는 protected와 private으로 지정할 수 없다.
- 메서드를 오버라이딩할 때 부모 클래스의 메서드보다 가시성을 더 좁게 할 수는 없다.

# 상속과 접근 제어

## ■ 접근 지정자의 접근 범위 예



## ■ 예제 :

[sec06/One](#),

[sec06/One1](#),

[sec06/Two](#),

[sec06/other/One2](#),

[sec06/other/Three](#)

# final 클래스와 메서드

## ■ final 클래스

- 더 이상 상속될 수 없는 클래스
- 대표적인 final 클래스로는 String 클래스
- 예제 : [sec07/FinalClassDemo](#)

## ■ final 메서드

- final 클래스는 클래스 내부의 모든 메서드를 오버라이딩할 수 없다. 특정 메서드만 오버라이딩하지 않도록 하려면 final 메서드로 선언
- 예제 : [sec07/FinalMethodDemo](#)

# 타입 변환과 다형성

## ■ 객체의 타입 변환

- 참조 타입 데이터도 기초 타입 데이터처럼 타입 변환 가능.
- 그러나 상속 관계일 경우만 타입 변환 가능
- 기초 타입처럼 자동 타입 변환과 강제 타입 변환이 있다.

## ■ 타입 변환 예시를 위한 샘플 클래스

[예제 6-16] 객체 타입 변환을 하는 Person 클래스

```
01 public class Person {  
02     String name = "사람";  
03  
04     void whoami() {  
05         System.out.println("나는 사람이다.");  
06     }  
07 }
```

[예제 6-17] 객체 타입 변환을 하는 Student 클래스

```
01 public class Student extends Person {  
02     int number = 7;  
03  
04     void work() {  
05         System.out.println("나는 공부한다.");  
06     }  
07 }
```



# 타입 변환과 다형성

## ■ 자동 타입 변환 : 조상-자손 관계의 참조변수는 서로 형변환이 가능함

```
Student s = new Student();
```

```
Person p = s;           // 자동으로 타입 변환을 한다.
```

```
Person p;
```

```
Student s = new Student();
```

`p = s;` → 자동으로 타입 변환되며 `p = (Person)s`와 동일하다.

```
// p.number = 1;
```

```
// p.work();
```

`number`와 `work()`는 부모 타입에 없는 멤버이므로  
부모 타입 변수에서 볼 수 없다.

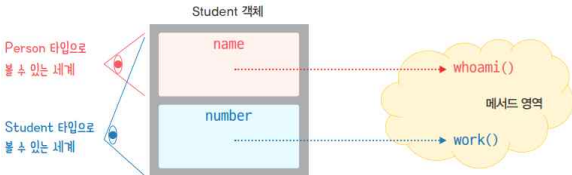
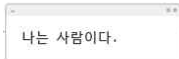
```
p.whoami();
```

`Person` 타입 멤버이므로 호출할 수 있다.

# 타입 변환과 다형성

## ■ 자동 타입 변환

- 예제 : [sec08/UpcastDemo](#)



# 타입 변환과 다형성

## ■ 강제 타입 변환

```
Person p = new Person();
```

```
Student s = (Student) p;    // 강제로 타입 변환을 하면 오류가 발생한다.
```



```
Student s1 = new Student();
```

```
Person p = s1;
```

```
Student s2 = (Student) p;    // 강제로 타입 변환을 할 수 있다.
```



부모 타입 변수이지만 자식 객체를 가리킨다.

# 타입 변환과 다형성



## ■ instanceof 연산자

- 타입 변환된 객체의 구별



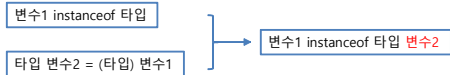
- 예제 : [sec08/InstanceofDemo](#)

```
true
true
false
오케이, 하향 타입 변환
```



## ■ instanceof 연산자 기능 개선

- 자바 16부터
- instanceof 연산자에 타입 변환 기능까지 추가



† 변수2는 특수한 형태의 지역 변수로  
null 체크가 필요 없음

- 예제 : [sec08/InstanceOf1Demo](#), [sec08/InstanceOf2Demo](#)

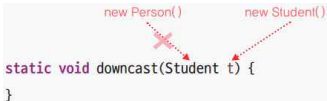
# 타입 변환과 다형성

## ■ 타입 변환을 이용한 다형성



The diagram illustrates a successful downcast. At the top, two red labels 'new Person()' and 'new Student()' have dashed red arrows pointing down to a code snippet. The code snippet is: `static void downcast(Person p) {  
}`. The arrow from 'new Student()' points directly to the parameter 'p' in the function signature, indicating that a Student object can be passed where a Person is expected.

```
static void downcast(Person p) {  
}
```



The diagram illustrates a failed downcast. At the top, two red labels 'new Person()' and 'new Student()' have dashed red arrows pointing down to a code snippet. The code snippet is: `static void downcast(Student t) {  
}`. The arrow from 'new Person()' points to the parameter 't' in the function signature, but it is crossed out with a large red 'X', indicating that a Person object cannot be passed where a Student is expected.

```
static void downcast(Student t) {  
}
```

# 타입 변환과 다형성

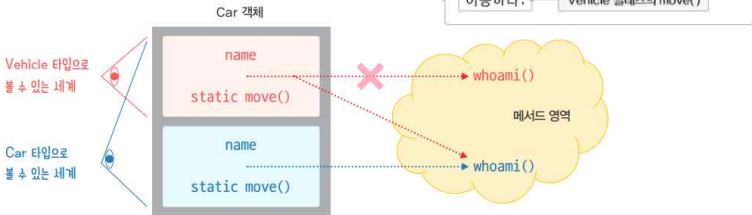
## ■ 타입 변환을 이용한 다형성

- 예제 : [sec08/OverTypeDemo](#)

```
27 Vehicle v = new Car();  
28 System.out.println(v.name);  
29 v.whoami();  
30 v.move();
```

Vehicle 타입의 name 필드에 접근한다.

v의 실제 객체는 Car 타입이므로 Car 타입의 whoami() 메서드를 호출한다.



# 타입 변환과 다형성

## ■ 타입 변환을 이용한 다형성

- 예제 : [sec08/PolymorDemo](#)





# 모듈화

## ■ 필요성

- 클래스와 패키지로 접근을 제어하는 현재의 자바로는 다음과 같은 이유로 소프트웨어를 효과적으로 개발하기 어렵다.
- 패키지의 캡슐화 기능 부족
- 누락된 클래스의 탐지 어려움
- 단일 구성 런타임 플랫폼의 비효율성
- 예 : 패키지의 캡슐화 기능 부족

```
package programmer;
import utils.*;
import developer.*;

public class App {
    public static void main( ... ) {
        new Open().yahoo();
        new Secret().hush();
    }
}
```

library A

```
package utils;
import developer.*;

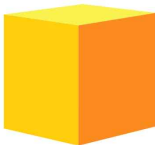
public class Open {
    public void yahoo( ) {
        new Secret().hush();
    }
}
```

```
package developer;

public class Secret {
    public void hush( ) {
    }
}
```

# 모듈화

## ■ 모노리딕 구조와 모듈 구조



(a) 모노리딕



(b) 모듈

## ■ 자바 9에서 도입한 모듈

- 모듈은 밀접한 관계가 있는 패키지, 리소스, 모듈 기술자 파일을 함께 묶어 놓은 것
- 모듈은 JAR 파일과 달리 다른 모듈과의 의존성 정보와 패키지의 공개 여부에 대한 정보를 포함
- 특별한 경우를 제외하곤 고유한 모듈 이름을 가진다
- 모듈을 통해 패키지의 캡슐화 기능을 개선하고 누락 클래스를 실행 전에 탐지할 수 있고, 배포할 런타임의 크기를 줄일 수 있다.

# 모듈화

## ■ 자바 런타임의 기본 모듈

- 모듈을 도입한 자바 9부터 자바 플랫폼 자체도 Jigsaw 프로젝트라는 이름으로 기능에 따라 재구성하여 모듈화되어 있으며, 다음과 같은 모듈을 포함
  - java.base 모듈
  - java.desktop 모듈
  - java.compiler 모듈
  - java.se 모듈
- 자바 플랫폼에 포함된 모듈 확인

```
java --list-modules
```

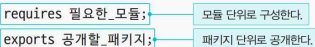
```
java --describe-modules(혹은 줄여서 -d) 모듈_이름
```

# 모듈화

## ■ 모듈 작성과 응용

- 모듈 기술자는 module-info.java라는 파일 이름을 사용하며 일반적으로 패키지 최상단 수준의 폴더에 위치
- 모듈 기술자의 구성

```
module 모듈_이름 {  
    requires 필요한_모듈;  
    exports 공개할_패키지;  
}
```



# 모듈화

## ■ 예 : 모듈의 캡슐화 기능

module1

```
package programmer;
import utils.*;
// import developer.*;

public class App {
    public static void main( ... ) {
        new Open().yahoo();
        // new Secret().hush();
    }
}
```

module-info.java

```
requires module2;
```

module2

```
package utils;
import developer.*;

public class Open {
    public void yahoo( ) {
        new Secret().hush();
    }
}
```

```
package developer;

public class Secret {
    public void hush( ) {
    }
}
```

module-info.java

```
export utils;
```

# 모듈화

## ■ 모듈 생성

- [New] → [Module] 을 선택한다. 새로운 창이 나타나면 [Next] 를 클릭한 후 모듈 이름을 입력하여 모듈을 생성
- src 폴더에 마우스 오른쪽 버튼을 눌러 [New] → [module-info.java] 를 선택하여 모듈 기술자를 생성

## ■ 모듈 경로 추가

