함께가요 미래로!
Enabling People

# OOP - 2

# 목차

함께가요 미래로!
Enabling People

# 상속(Inheritance)

## ❖ 객체지향 언어의 특징

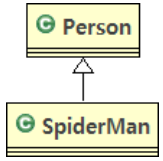
- OOP is A P.I.E

| 특성 | 내용 |
|---|---|
| Abstraction(추상화) | 현실의 객체를 추상화 해서 클래스를 구성한다. |
| Polymorphism(다형성) | 하나의 객체를 여러 가지 타입(형)으로 참조할 수 있다. |
| Inheritance(상속) Encapsulation(데 | 부모 클래스의 자산을 물려받아 자식을 정의함으로 코드의 재사용이 가능하다. |
| 이터 은닉과 보호) | 데이터를 외부에 직접 노출시키지 않고 메서드를 이용해 보호할 수 있다. |

## ❖ 상속(Inheritance: Java Is A PIE)

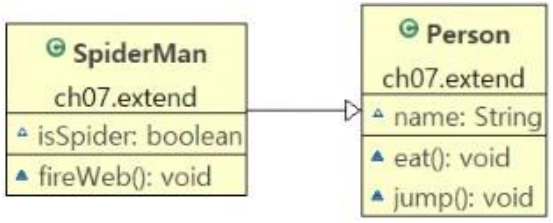
- 기존 클래스의 자산(멤버)을 자식 클래스에서 재사용하기 위한 것
  - ◆ 부모의 생성자와 초기화 블록은 상속하지 않는다.
- 기존 클래스의 멤버를 물려 받기 때문에 코드의 절감
  - ◆ 부모의 코드를 변경하면 모든 자식들에게도 적용 → 유지 보수성 향상
- 상속의 적용
  - ◆ extends 키워드 사용

```
public class Person {
  String name;

  void eat(){}
  void jump(){}
}


public class SpiderMan extends Person {
  boolean isSpider;
  void fireWeb(){}
}
```




**조상 클래스 – 부모 클래스, 상위 클래스 , 슈퍼 클래스**

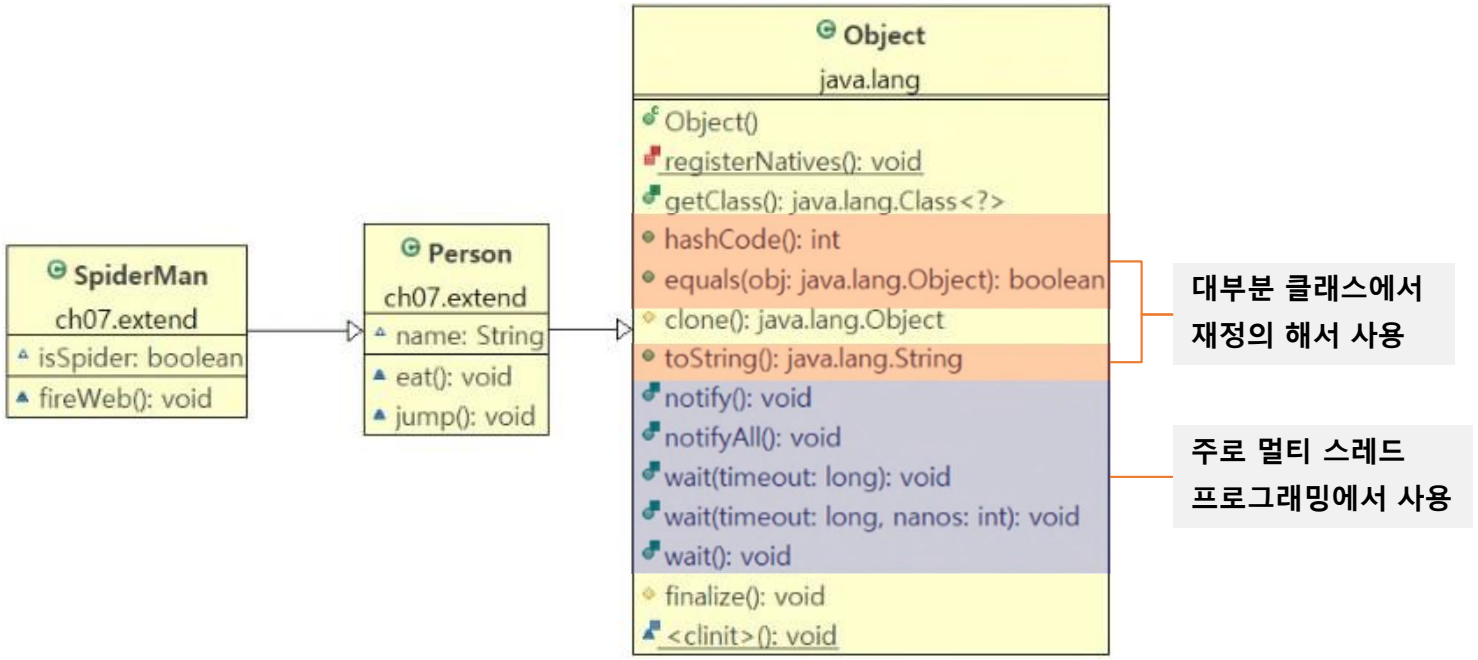
**자식 클래스 – 자손 클래스, 하위 클래스 , 서브 클래스**



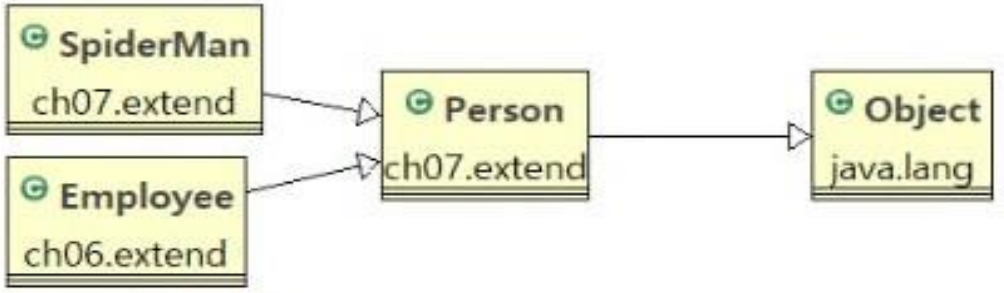

**SpiderMan의 멤버?**

## ❖ Object 클래스

- 모든 클래스의 조상 클래스
  - ◆ 별도의 extends 선언이 없는 클래스들은 extends Object 가 생략됨
  - ◆ 따라서 모든 클래스에는 Object 클래스에 정의된 메서드가 있음



❖ 다양한 상속 관계

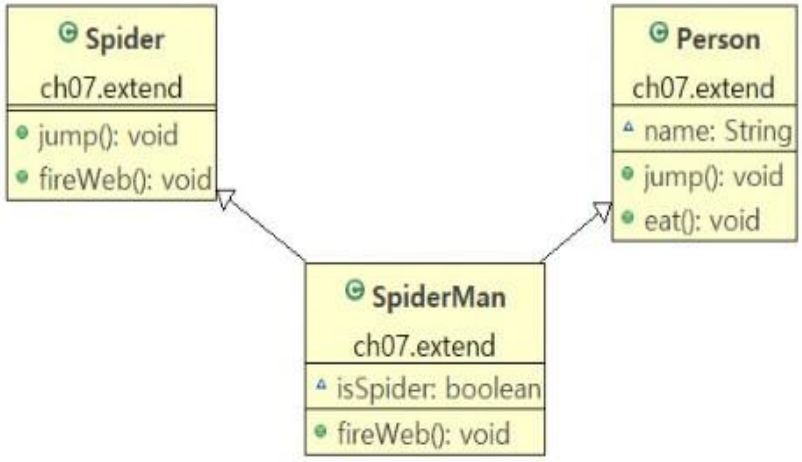



- 상속의 관계는 is a 관계라고 함
  - ◆ Person is a Object, SpiderMan is a Person

- Person과 Employee의 관계?
- Object와 Employee의 관계?
- Employee와 SpiderMan의 관계?

## ❖ 단일 상속 (Single Inheritance)

- 다중 상속의 경우 여러 클래스의 기능을 물려받을 수 있으나 관계가 매우 복잡해짐
  - 동일한 이름의 메서드가 두 부모에게 있다면 자식은 어떤 메서드를 쓸 것인가?

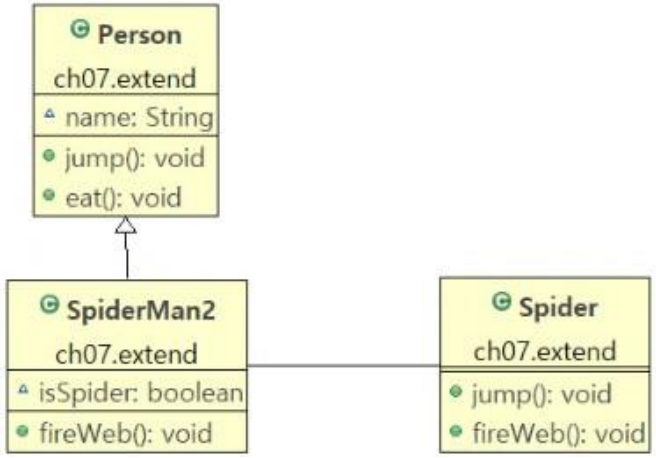



**SpiderMan의 jump 방식은?**

- 자바는 단일 상속만 지원
  - ◆ 대신 interface와 포함 관계(has a)로 단점 극복

## ❖ 포함 관계

- 상속 이외에 클래스를 재활용 하는 방법
  - ◆ 2개 이상의 클래스에서 특성을 가져올 때 하나는 상속, 나머지는 멤버 변수로 처리




```
public class SpiderMan2 extends Person {
  Spider spider;
  boolean isSpider;

  void fireWeb() {
    if (isSpider) {
      spider.fireWeb();
    }else {
      System.out.println("Person은 거미줄 발사 불가");
    }
  }
}
```

  - ◆ 포함 관계의 UML 표현: 실선

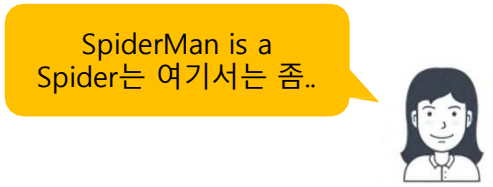
  - ◆ Spider의 코드를 수정하면 SpiderMan에도 반영되므로 유지 보수성 확보

## ❖ 포함 관계

◆ 상속이냐 포함이냐 그것이 문제로다.

◆ 어떤 클래스를 상속 받고 어떤 클래스를 포함해야 하는가?

- 문법적인 문제는 아니며 프로젝트의 관점 문제
- 상속: is a 관계가 성립하는가? → SpiderMan is a Person.
- 포함: has a 관계가 성립하는가? → SpiderMan has a Spider.



함께가요 미래로!
Enabling People

# 메서드 재정의

# 메서드 재정의



## ❖ 매서드 오버라이딩(overriding)

- 조상 클래스에 정의된 메서드를 자식 클래스에서 적합하게 수정하는 것

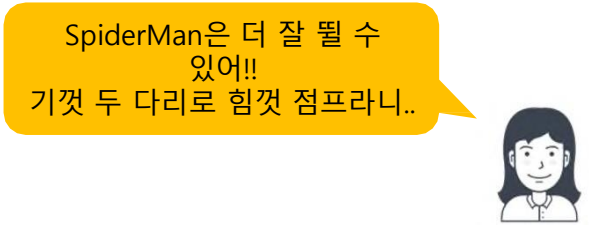
```
public class Person {
  void jump(){
    System.out.println("두 다리로 힘껏 점프");
  }
}
```

물려받은 jump(). 성능이 좋지 않다.

탐나는 jump()를 가지고 있다.

```
public class Spider {
  void jump(){
    System.out.println("키 * 1000만큼 엄청난 점프");
  }
}
```

```
public class SpiderMan2 extends Person {
  Spider spider = new Spider();
  boolean isSpider;
}
```



# 메서드 재정의

Confidential

## ❖ 매서드 오버라이딩(overriding)

```
public class Person {
  void jump(){
    System.out.println("두 다리로 힘껏 점프");
  }
}
```

물려받은 jump(). 성능이 좋지 않다.

탐나는 jump()를 가지고 있다.

```
public class Spider {
  void jump(){
    System.out.println("키 * 1000만큼 엄청난 점프");
  }
}
```

```
public class SpiderMan2 extends Person {
  Spider spider = new Spider();
  boolean isSpider;

  void fireWeb() {. . .}
  void jump() {
    if (isSpider) {
      spider.jump();
    } else {
      System.out.println("두 다리로 힘껏 점프");
    }
  }
}
```

이 시국에 코드 중복이라니..

- 오버라이딩의 조건
  - ◆ 메서드 이름이 같아야 한다.
  - ◆ 매개 변수의 개수, 타입, 순서가 같아야 한다.
  - ◆ 리턴 타입이 같아야 한다.
  - ◆ 접근 제한자는 부모 보다 범위가 넓거나 같아야 한다.
  - ◆ 조상보다 더 큰 예외를 던질 수 없다.

# 메서드 재정의



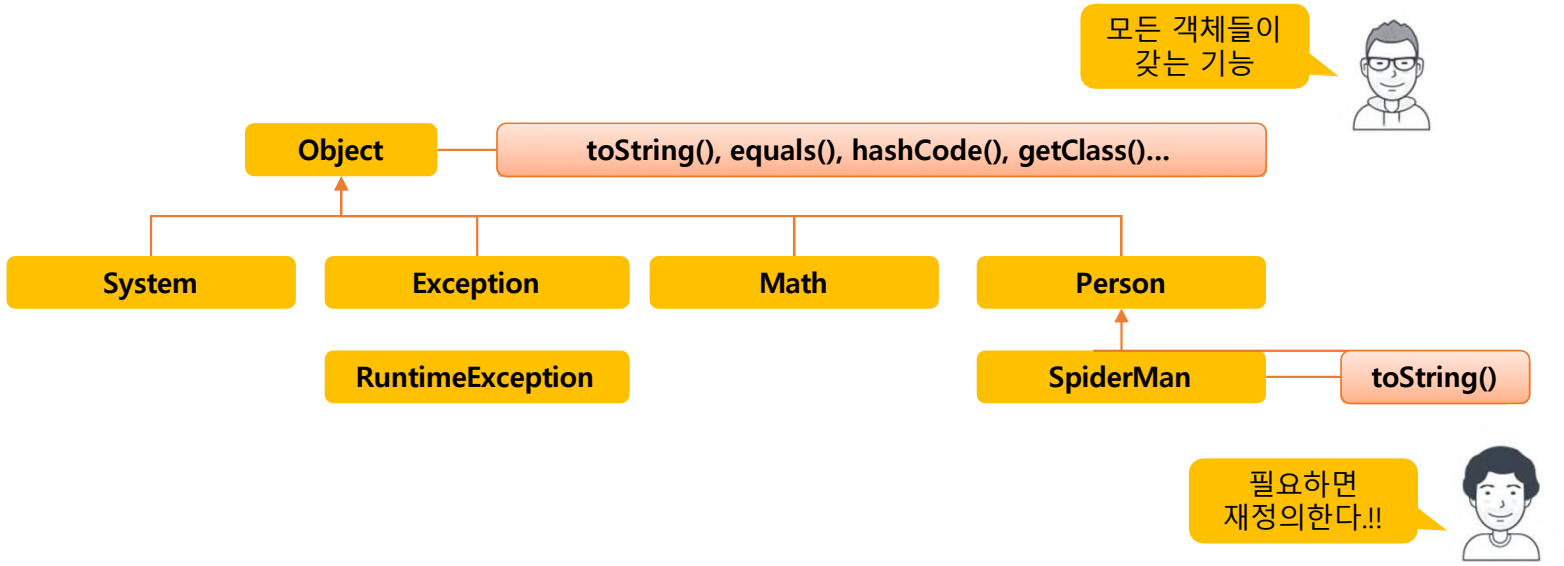
❖ Annotation

- 사전적 의미: 주석
- 컴파일러, JVM, 프레임워크 등이 보는 주석
- 소스코드에 메타 데이터를 삽입하는 형태
  - ◆ 소스 코드에 붙여 놓은 라벨
  - ◆ 코드에 대한 정보 추가 → 소스 코드의 구조 변경, 환경 설정 정보 추가 등의 작업 진행

- JDK 1.5의 기본 annotation의 예
  - ◆ @Deprecated
    - 컴파일러에게 해당 메서드가 deprecated 되었다고 알려줌
  - ◆ @Override
    - 컴파일러에게 해당 메서드는 override한 메서드 임을 알려줌
    - @Override 가 선언된 경우 반드시 super class에 선언 되어있는 메서드 여야 함
  - ◆ @SuppressWarnings
    - 컴파일러에게 사소한 warning의 경우 신경 쓰지 말라고 알려줌

## ❖ Object 클래스

- 가장 최상위 클래스로 모든 클래스의 조상
  - ◆ Object의 멤버는 모든 클래스의 멤버




```
System.out.println(person.toString());                    // Object의 toString 사용

System.out.println(spiderMan.toString());          // SpiderMan의 toString 사용
```

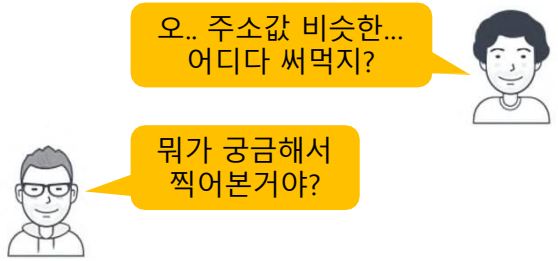
# Object - equals



## ❖ toString 메서드

- 객체를 문자열로 변경하는 메서드

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```




- 정작 궁금한 내용은 주소값 따위가 아닌 내용

```
@Override
public String toString() {
    return "SpiderMan [isSpider=" + isSpider + ", name=" + name + "]";
}
```

# Object - equals



## ❖ equals 메서드

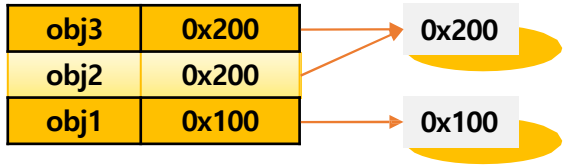
- 두 객체가 같은지를 비교하는 메서드

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

**등가비교 연산자 ==로 두 객체의 주소값 비교**

- 두 개의 레퍼런스 변수가 같은 객체를 가리키고 있는가?

```
Object obj1 = new Object();
Object obj2 = new Object();
Object obj3 = obj2;
System.out.printf("obj1 == obj2: %b%n", obj1==obj2);
System.out.printf("obj1 equals obj2: %b%n", obj1.equals(obj2));

System.out.printf("obj2 == obj3: %b%n", obj2==obj3);
System.out.printf("obj2 equals obj3: %b%n", obj2.equals(obj3));
```



# Object - equals



## ❖ equals 메서드

- 우리가 비교할 것은 정말 객체의 주소 값 인가?
  - ◆ 두 객체의 내용을 비교할 수 있도록 equals 메서드 재정의

```
private static void testString() {
  String s1 = new String("Hello");
  String s2 = new String("Hello");
  System.out.println((s1 == s2) + " : " + s1.equals(s2));
}

private static void testPhone() {
  Phone p1 = new Phone("0100000000");
  Phone p2 = new Phone("0100000000");
  System.out.println((p1 == p2) + " : " + p1.equals(p2));
}
```

```
class Phone {
  String number = "전화번호";

  public Phone(String number) {
    this.number = number;
  }
/*
  @Override
  public boolean equals(Object obj) {
    if (obj != null && obj instanceof Phone) {
      Phone casted = (Phone) obj;
      return number.equals(casted.number);
    }
    return false;
  }
*/
}
```

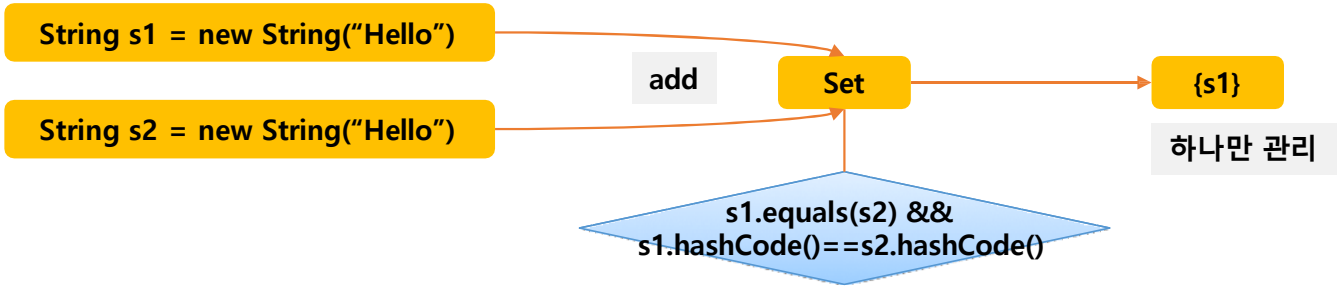
  - ◆ 객체의 주소 비교: == 활용
  - ◆ 객체의 내용 비교: equals 재정의

# Object - hashCode



## ❖ hashCode

- 객체의 해시 코드: 시스템에서 객체를 구별하기 위해 사용되는 정수 값
- HashSet, HashMap 등에서 객체의 동일성을 확인하기 위해 사용



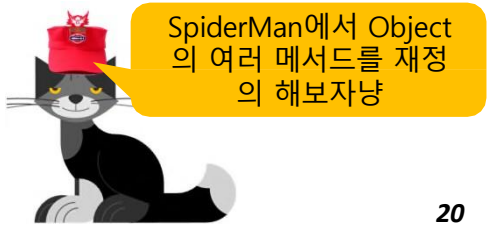

- equals 메서드를 재정의할 때는 반드시 hashCode도 재정의할 것
  - ◆ 미리 작성된 String이나 Number 등에서 재정의된 hashCode 활용 권장

# Object - hashCode



## ❖ Object의 메서드 재정의

```
class Product {
  String sn;

  public Product(String sn) {
    this.sn = sn;
  }

  @Override
  public boolean equals(Object obj) {
    if (obj != null && obj instanceof Product) {
      Product casted = (Product) obj;
      return sn.equals(casted.sn);
    }
    return false;
  }

  @Override
  public String toString() {
    return "Product [sn=" + sn + "]";
  }

  @Override
  public int hashCode() {
    return sn.hashCode();
  }
}
```

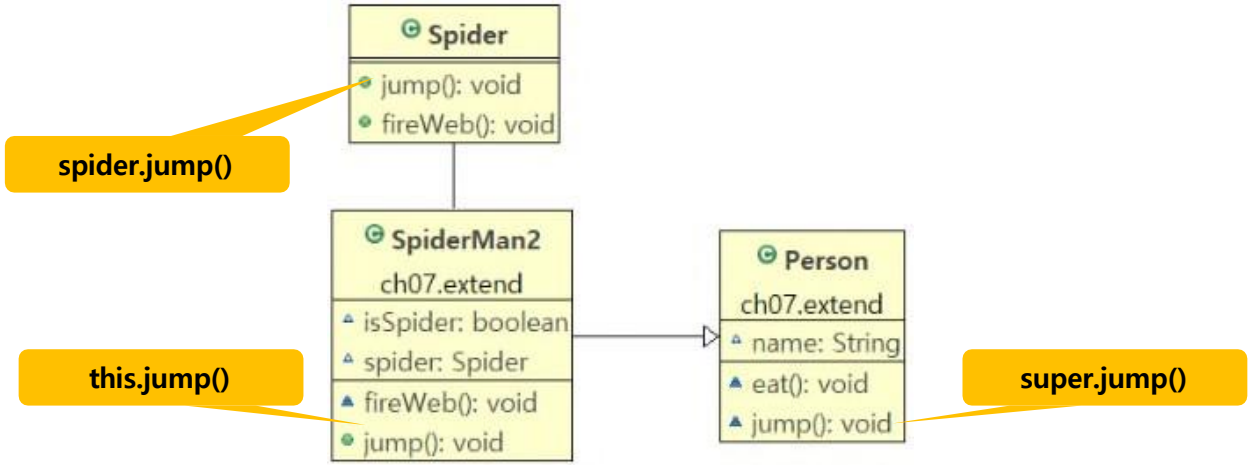


## ❖ super 키워드

- this통해 멤버에 접근했듯이 super를 통해 조상 클래스 멤버 접근
  - ◆ super.을 이용해 조상의 메서드 호출로 조상의 코드 재사용

```
void jump() {
  if (isSpider) {
    spider.jump();
  } else {
    System.out.println("뛰기");
  }
}
```

```
void jump() {
  if (isSpider) {
    spider.jump();
  } else {
    super.jump();
  }
}
```




SpiderMan이 최적의 jump를 할 수 있도록 해주자냥

## ❖ super 키워드

- 변수의 scope
  - ◆ 사용된 위치에서 점점 확장해가며 처음 만난 선언부에 연결됨
  - ◆ method 내부 → 해당 클래스 멤버 변수 → 조상 클래스 멤버 변수

```
class Parent{
  String  x = "parent";
}

class Child extends Parent{
  String  x = "child";

  void method(){
    String  x = "method";
    System.out.println("x : "+x);
    System.out.println("this.x : "+this.x);
    System.out.println("super.x : "+super.x);
  }
}

public class ScopeTest{

  public static void main(String[] args) {
    Child child = new Child();
    child.method();
  }
}
```

1. 현재 상태에서 출력 결과 예상
2. method의 x를 주석하면?
3. child의 x를 주석하면?

## ❖ super 키워드

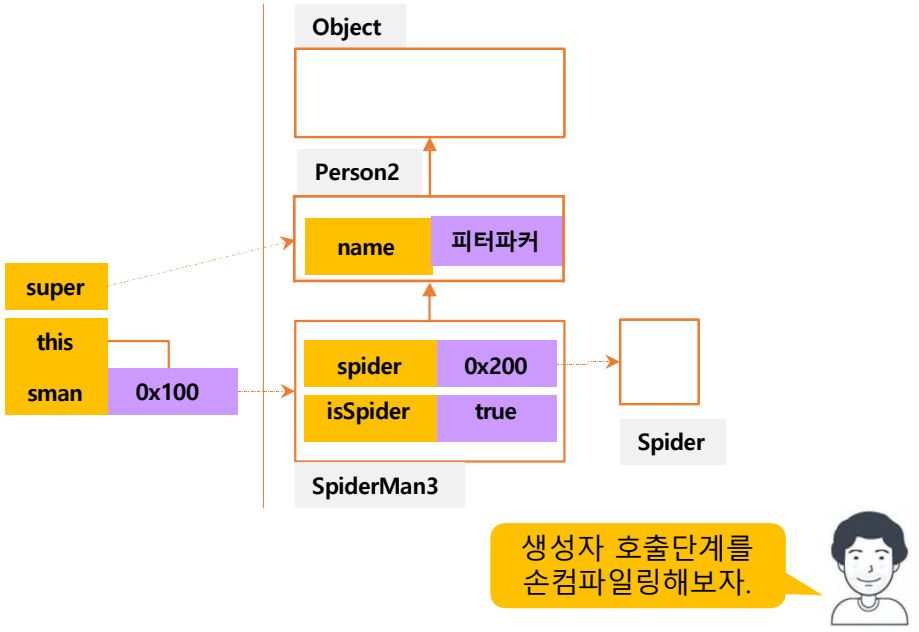
- this()가 해당 클래스의 다른 생성자를 호출하듯 super()는 조상 클래스의 생성자 호출
  - ◆ 조상 클래스에 선언된 멤버들은 조상 클래스의 생성자에서 초기화가 이뤄지므로 이를 재활용
  - ◆ 자식 클래스에 선언된 멤버들만 자식 클래스 생성자에서 초기화
- super()는 자식 클래스 생성자의 맨 첫 줄에서만 호출 가능
  - ◆ 즉 생성자의 첫 줄에만 this() 또는 super() 가 올 수 있다.
- 명시적으로 this() 또는 super()를 호출하지 않는 경우 컴파일러가 super() 삽입
  - ◆ 결론적으로 맨 상위의 Object까지 객체가 다 만들어지는 구조

```
class Person2 {
  String name;

  Person2(String name) {
    // super();// --> Object의 기본 생성자 호출
    this.name = name;
  }
}
```

## ❖ super 키워드

- 생성자 호출과 객체 생성의 단계

```
class Person2 {
  String name;
  Person2(String name) {
    this.name = name;
  }
}

public class SpiderMan3 extends Person2 {
  Spider spider = new Spider();
  boolean isSpider;

  SpiderMan3(String name, Spider spider, boolean isSpider) {
    super(name);
    this.spider = spider;
    this.isSpider = isSpider;
  }

  SpiderMan3(String name) {
    this(name, new Spider(), true);
  }

  public static void main(String[] args) {
    SpiderMan3 sman = new SpiderMan3("피터 파커");
  }
}
```



함께가요 미래로!
Enabling People

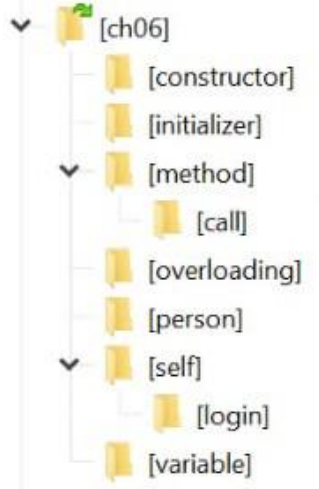
# package & import

# package와 import



## ❖ Package

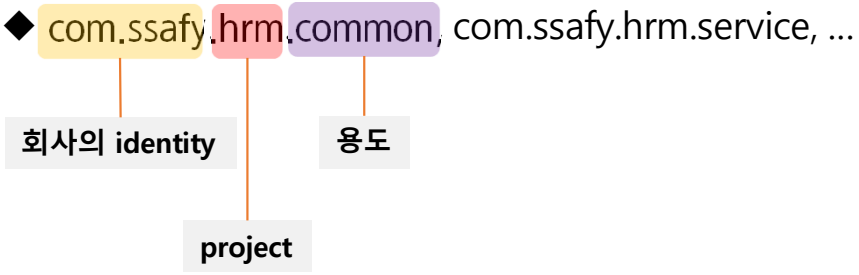
- PC의 많은 파일 관리 → 폴더 이용
  - ◆ 유사한 목적의 파일을 기준으로 작성
  - ◆ 이름은 의미 있는 이름으로, 계층적 접근
    - c:₩Program Files₩Internet Explorer₩images₩bing.ico
- 프로그램의 많은 클래스 → 패키지 이용
  - ◆ 패키지의 이름은 의미 있는 이름으로 만들고 . 를 통해 계층적 접근
  - ◆ 물리적으로 패키지는 클래스 파일을 담고 있는 디렉터리
- package의 선언
  - ◆ package package_name;
  - ◆ 주석, 공백을 제외한 첫 번째 문장에 하나의 패키지만 선언
  - ◆ 모든 클래스는 반드시 하나의 패키지에 속한다.
    - 생략 시 default package
    - default package는 사용하지 않는다.




```
package ch06.method.call;

public class First {
  . . .
}
```

## ❖ Package

- 일반적인 package naming 룰
  - ◆ 소속.프로젝트.용도
  - ◆ com.ssafy.hrm.common, com.ssafy.hrm.service, …



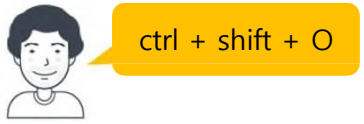
## ❖ import

- 다른 패키지에 선언된 클래스를 사용하기 위해 키워드
  - ◆ 패키지와 클래스 선언 사이에 위치
  - ◆ 패키지와 달리 여러 번 선언 가능
- 선언 방법
  - ◆ import 패키지명.클래스명;
  - ◆ import 패키지명.*;
    - 하위 패키지까지 import 하지는 않는다.

```
import java.io.InputStream;
import java.util.*;

public class ImportTest {
  Date date;
  List list;

  InputStream input;

  java.awt.List list2;
}
```

- import 한 package의 클래스 이름이 동일하여 명확히 구분해야 할 때
  - ◆ 클래스 이름 앞에 전체 패키지 명을 입력

```
java.util.List list  = new java.util.ArrayList();
```




- default import package
  - ◆ java.lang.*;

## ❖ 일반적인 클래스 레이아웃

**패키지 선언부**

**외부 패키지 import**

**class 선언부**

**멤버 변수**

**초기화 블록**

**생성자**

**멤버 메서드**

```
package structure;

import java.io.*;

public class ClassStructure {

  String name;
  int age;

  {name="andy";}

  public ClassStructure(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public void setName(String name) {
    this.name = name;
  }

  public static void main(String [] args) {
    ClassStructure cs = new ClassStructure("hong", 10);
  }
}
```

함께가요 미래로!
Enabling People

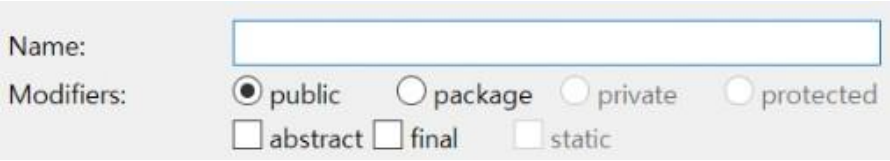
# 접근제한자와 데이터 은닉과 보호

# 접근 제한자 및 데이터 은닉과 보



❖ 제한자(modifier)

- 클래스, 변수, 메서드 선언부에 함께 사용되어 부가적인 의미 부여
- 종류
  - ◆ 접근 제한자 : public, protected, (default = package), private
  - ◆ 그 외 제한자
    - static: 클래스 레벨의 요소 설정
    - final: 요소를 더 이상 수정할 수 없게 함
    - abstract: 추상 메서드 및 추상 클래스 작성
    - synchronized: 멀티스레드에서의 동기화 처리
    - . . .
- 하나의 대상에 여러 제한자를 조합 가능하나 접근 제한자는 하나만 사용 가능
- 순서는 무관
  - ◆ 일반적으로 접근 제한자를 맨 앞으로



# 접근 제한자 및 데이터 은닉과 보



## ❖ final

- 마지막, 더 이상 바뀔 수 없음
- 용도
  - ◆ final class-더 이상 확장할 수 없음:상속 금지→ 오버라이드 방지

```
final class PerfectClass{. . .}
//The type FinalClassTest cannot subclass the final class PerfectClass
public class FinalClassTest extends PerfectClass{. . .}
```

    - 이미 완벽한 클래스들: String, Math, ...
  - ◆ final method-더 이상 재정의할 수 없음:overriding 금지

```
class ParentClass{
  public final void finalMethod() {}
}

public class FinalClassTest extends ParentClass{
  //Cannot override the final method from ParentClass
  public void finalMethod() {}
}
```

  - ◆ final variable-더 이상 값을 바꿀 수 없음:상수화

```
public void finalParameterTest(final String name) {
  System.out.println(name);
  //The final local variable name cannot be assigned.
  name="hong";
}
```

# 접근 제한자 및 데이터 은닉과 보

Confidential

❖ 접근 제한자(Access modifier)

● 멤버 등에 사용되며 해당 요소를 외부에서 사용할 수 있는지 설정

| 제한자 | 용도 | | | 접근 가능 범위 | | | |
|---|---|---|---|---|---|---|---|
| | 클래스 | 생성자 | 멤버 | 같은 클래스 | 같은 패키지 | 다른 패키지의 자손 클래스 | 전체 |
| public | O | O | O | O | O | O | O |
| protected | | O | O | O | O | O | |
| package(default) | O | O | O | O | O | | |
| private | | O | O | O | | | |

# 접근 제한자 및 데이터 은닉과 보

Confidential

## ❖ 접근 제한자(Access modifier)

| | 같은 패키지(modifier.p1) | 다른 패키지(modifier.p2) |
|---|---|---|
| 일반 클래스 | `public void method(){`<br>`  publicVariable = 10;`<br>`  protectedVariable = 10;`<br>`  defaultVariable = 20;`<br>`  //privateVariable = 10;`<br>`}` | `public void method(){`<br>`  Parent p = new Parent();`<br>`  p.publicVariable = 10;`<br>`  //p.protectedVariable = 10;`<br>`  //p.defaultVariable = 20;`<br>`  //privateVariable = 10;`<br>`}` |
| 자식 클래스 | `public void useMember() {`<br>`  this.publicVar = 10;`<br>`  this.protectVar = 10;`<br>`  this.defaultVar = 10;`<br>`  //The field Parent.privVar is not visible`<br>`  //this.privVar = 10;`<br>`}` | `public void useMember() {`<br>`  this.publicVar = 10;`<br>`  this.protectVar = 10;`<br>`  // The field Parent.privVar is not visible`<br>`  //this.defaultVar = 10;`<br>`  // this.privVar = 10;`<br>`}` |

```
package modifier.p1;

public class Parent {
  public int publicVariable;
  protected int protectedVariable;
  int defaultVariable ;
  private int privateVariable;
}
```

# 접근 제한자 및 데이터 은닉과 보

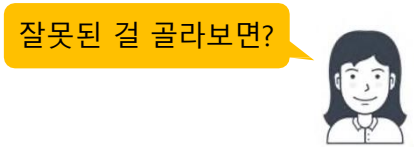
Confidential

## ❖ 접근 제한자(Access modifier)

- method override 조건의 확인
  - ◆ 부모의 제한자 범위와 같거나 넓은 범위로만 사용 가능

```
class Parent{
    protected void method() {}
}

public class OverrideRule extends Parent{
    @Override
    void method() {}

    protected void method() {}

    public void method() {}
}
```

Cannot reduce the visibility of the inherited method from Parent



# 접근 제한자 및 데이터 은닉과 보

Confidential

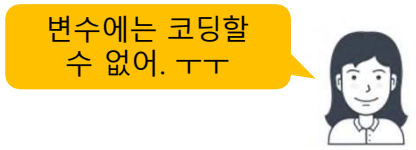
## ❖ 데이터 은닉과 보호(Encapsulation: OOP Is A PIE)

- 누군가 당신의 정보를 마음대로 바꾼다면?

```
class UnbelievableUserInfo {
  public String name = "홍길동"; // 이름은 not null
  public int account = 10000;    // 계좌는 0보다 커야 함.
}

public class UnbelievableTest {
  public static void main(String[] args) {
    UnbelievableUserInfo info = new UnbelievableUserInfo();
    System.out.printf("사용자 정보:%s, %d%n", info.name,    info.account);
    info.name = null;
    info.account = -1000;
    System.out.printf("사용자 정보:%s, %d%n", info.name, info.account);
  }
}
```

```
사용자 정보:홍길동, 10000
사용자 정보:null, 0
```

- 소중한 정보가 보호되지 못하는 이유는?
  - ◆ 외부에서 변수에 직접 접근하기 때문
- 정보를 보호하기 위한 대책은?
  - ◆ 변수는 private 접근으로 막기
  - ◆ 공개되는 메서드를 통한 접근 통로 마련: setter / getter
    - 메서드에 정보 보호 로직 작성



# 접근 제한자 및 데이터 은닉과 보

Confidential

## ❖ 데이터 은닉과 보호(Encapsulation: OOP Is A PIE)

```
class BelievableUserInfo {
  private String name = "홍길동";
  private int account = 10000;
  public String getName() {
    return this.name;
  }
  public void setName(String name) {
    if(name!=null) {
      this.name = name;
    }else {
      System.out.println("부적절한 name 할당 시도 무시: "+name);
    }
  }
  // account에 대한 getter / setter 생략
}

public class BelievableTest {
  public static void main(String[] args) {
    BelievableUserInfo info = new BelievableUserInfo();
    //System.out.printf( " 사용자 정보:%s, %d%n " , info.name, info.account);
    System.out.printf( " 사용자 정보:%s, %d%n " , info.getName(), info.getAccount());
    //info.name = null;
    info.setName(null);
    //info.account = -1000;
    info.setAccount(-10000);
    System.out.printf("사용자 정보:%s, %d%n", info.getName(), info.getAccount());
  }
}
```

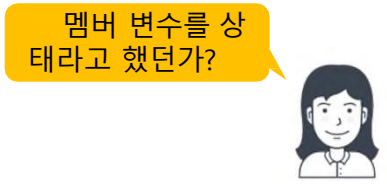
```
사용자 정보:홍길동, 10000  부적절
한 name 할당 시도 무시: null
부적절한 account 할당 시도 무시: -10000
사용자 정보:홍길동, 10000
```

# 접근 제한자 및 데이터 은닉과 보



## ❖ 객체의 생성 제어와 Singleton 디자인 패턴

- 객체의 생성을 제한해야 한다면?
  - ◆ 여러 개의 객체가 필요 없는 경우
    - 객체를 구별할 필요가 없는 경우 = 수정 가능한 멤버 변수가 없고 기능만 있는 경우
    - 이런 객체를 stateless 한 객체라고 한다.
  - ◆ 객체를 계속 생성/삭제 하는데 많은 비용이 들어서 재사용이 유리한 경우




- Singleton 디자인 패턴
  - ◆ 외부에서 생성자에 접근 금지 → 생성자의 접근 제한자를 private으로 설정
  - ◆ 내부에서는 private에 접근 가능하므로 직접 객체 생성 → 멤버 변수이므로 private 설정
  - ◆ 외부에서 private member에 접근 가능한 getter 생성 → setter는 불필요
  - ◆ 객체 없이 외부에서 접근할 수 있도록 getter와 변수에 static 추가
  - ◆ 외부에서는 언제나 getter를 통해서 객체를 참조하므로 하나의 객체 재사용

# 접근 제한자 및 데이터 은닉과 보

Confidential

## ❖ Singleton 디자인 패턴

```
class SingletonClass{
  private static SingletonClass instance = new SingletonClass();
  private SingletonClass() {}

  public static SingletonClass getInstance() {
    return instance;
  }

  public void sayHello() {
    System.out.println("Hello");
  }
}

public class SingletonTest {
  public static void main(String[] args) {
    SingletonClass sc1 = SingletonClass.getInstance();
    SingletonClass sc2 = SingletonClass.getInstance();

    System.out.printf("두 객체는 같은가? %b%n", sc1==sc2);
    sc1.sayHello();
  }
}
```

함께가요 미래로!
Enabling People

# 다형성

# 다형

Confidential

❖ 다형성(Polymorphism: Java Is A PIE)

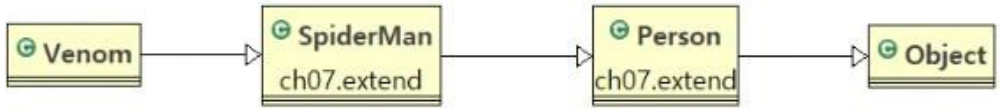
- 多形性-하나의 객체가 많은 형(타입)을 가질 수 있는 성질
  - ◆ 황금잉어빵 is a 붕어빵!!


- 다형성의 정의
  - ◆ **상속 관계**에 있을 때 조상 클래스의 타입으로 자식 클래스 객체를 레퍼런스 할 수 있다

.

## ❖ 다형성(Polymorphism: Java Is A PIE)

- 상속 관계에서 조상 클래스의 타입으로 자식 클래스 객체를 레퍼런스 할 수 있다.




```
SpiderMan onlyOne = new SpiderMan();
```

- onlyOne은 SpiderMan 타입인가?
  - ◆ SpiderMan 타입으로 onlyOne을 참조할 수 있는가?

```
SpiderMan sman = onlyOne;
```

- onlyOne은 Person 타입인가?
  - ◆ Person 타입으로 onlyOne을 참조할 수 있는가?

```
Person person = onlyOne;;
```

- onlyOne은 Object 타입인가?
  - ◆ Object 타입으로 onlyOne을 참조할 수 있는가?

```
Object obj = onlyOne;
```

- onlyOne은 Venom 타입인가?
  - ◆ Venom 타입으로 onlyOne을 참조할 수 있는가?

```
Venom venom = onlyOne;
```

## ❖ 다형성의 활용 예 1- 다른 타입의 객체를 다루는 배열

- 배열의 특징 같은 타입의 데이터를 묶음으로 다룬다.

Person [ ]에는 Person만
SpiderMan [ ]에는 SpiderMan만..

타입이 다양하면
배열 만들다 지치겠어.

- 다형성으로 다른 타입의 데이터 (Person, SpiderMan)를 하나의 배열로 관리

```
void beforePoly() {
  Person [] persons = new Person[10];
  persons[0] = new Person();
  SpiderMan [] spiderMans = new SpiderMan[10];
  spiderMans[0] = new SpiderMan();
}
```

```
void afterPoly() {
  Person [] persons = new Person[10];
  persons[0] = new Person();
  persons[1] = new SpiderMan();
}
```

## ❖ 다형성의 활용 예 1- 다른 타입의 객체를 다루는 배열

- Object는 모든 클래스의 조상!!
  - ◆ Object의 배열은 어떤 타입의 객체라도 다 저장할 수 있음
- 자바의 자료 구조를 간단하게 처리할 수 있음
  - ◆ 이와 같은 특성을 이용하여 Collection API가 등장하게 됨

```
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) {
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
    }
}
```

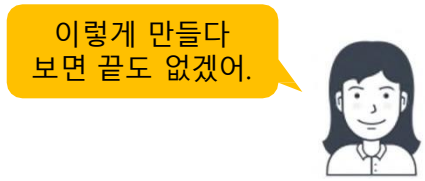
  - ◆ 기본형은 담을 수 있을까?

## ❖ 다형성의 활용 예 2- 매개변수의 다형성

- 무언가를 출력하고 싶다!!
  - ◆ 메서드가 호출되기 위해서는 메서드 이름과 파라미터가 맞아야 하는데..
  - ◆ public void println(Phone p)
  - ◆ public void println(SmartPhone sp)
  - ◆ ...
- 사실 println은..

```
public void println(Object x) {
    String s = String.valueOf(x);
    synchronized (this) {
        print(s);
        newLine();
    }
}
```

  - ◆ 조상을 파라미터로 처리한다면 객체의 타입에 따라 메서드를 만들 필요가 없어진다.



## ❖ 다형성의 활용 예 2- 매개변수의 다형성

- API에서 파라미터로 Object를 받는다는 것은 모든 객체를 처리한다는 말이다.

**println**

```
public void println(Object x)
```

Prints an Object and then terminate the line. This method calls at first String.valueOf(x) to get the printed object's string value, then behaves as though it invokes `print(String)` and then `println()`.

**Parameters:**

x - The `Object` to be printed.

**equals**

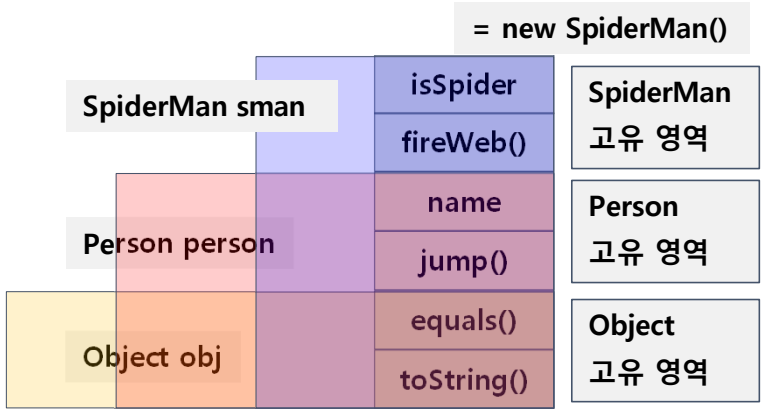
```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

- 물론 필요하다면 하위 클래스에서 오버라이딩 필요

## ❖ 다형성과 참조형 객체의 형 변환

- 메모리에 있는 것과 사용할 수 있는 것의 차이




```
Person person = new SpiderMan();
person.
```

- eat() : void - Person
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- jump() : void - Person
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

◆ 메모리에 있더라도 참조하는 변수의 타입에 따라 접근할 수 있는 내용이 제한됨

## ❖ 참조형 객체의 형 변환

- 작은 집(child)에서 큰 집 (super) 으로 → 묵시적 케스팅

```
byte b = 10;
int i = b;
```

```
Phone phone = new Phone();
Object obj = phone;
```

  - ◆ 자손 타입의 객체를 조상 타입으로 참조 : 형변환 생략 가능
    - 왜냐면 조상의 모든 내용이 자식에 있기 때문에 걱정할 필요가 없다.
- 큰집(super)에서 작은 집 (child) 으로 → 명시적 케스팅

```
int i = 10;
byte b = (byte)i;
```

```
Phone phone = new SmartPhone();
SmartPhone sPhone = (SmartPhone)phone;
```

  - ◆ 조상 타입을 자손 타입으로 참조 : 형변환 생략 불가

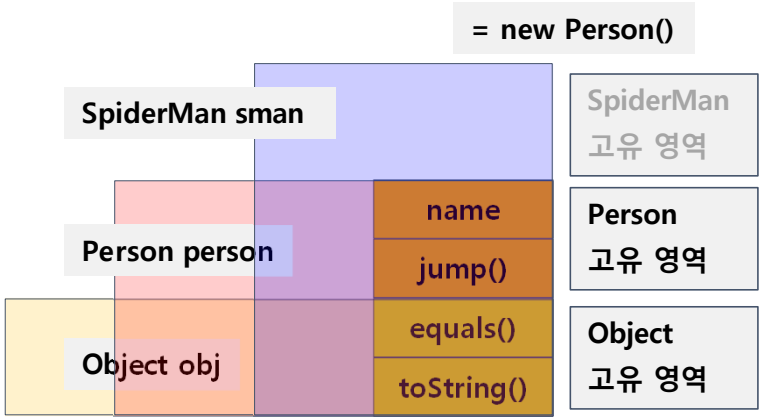
## ❖ 참조형 객체의 형 변환

- 무늬만 SpiderMan인 Person

```
Person person = new Person();
SpiderMan sman = (SpiderMan) person;
sman.fireWeb();
```

◆ 메모리의 객체는 fireWeb이 없음

```
Exception in thread "main"
java.lang.ClassCastException:
```




- 조상을 무작정 자손으로 바꿀 수는 없다.

◆ instanceof 연산자

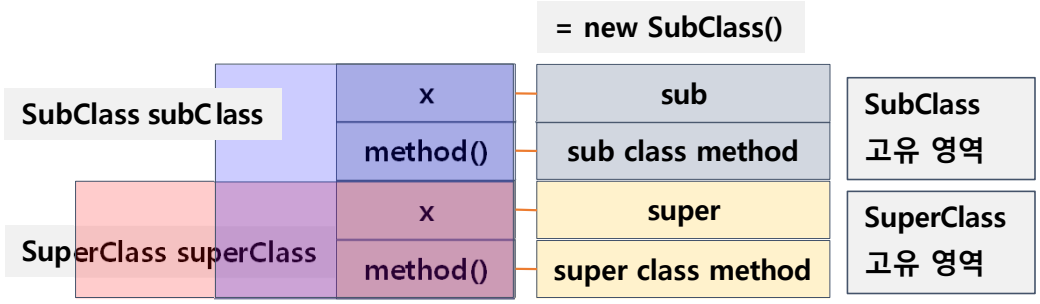
- 실제 메모리에 있는 객체가 특정 클래스 타입인지 boolean으로 리턴

```
Person person = new Person();

if (person instanceof SpiderMan) {
  SpiderMan sman = (SpiderMan) person;
}
```

## ❖ 참조 변수의 레벨에 따른 객체의 멤버 연결

```
class SuperClass {
  String x = "super";

  public void method() {
    System.out.println("super class method");
  }
}

class SubClass extends SuperClass {
  String x = "sub";

  @Override
  public void method() {
    System.out.println("sub class method");
  }
}
```

```
public class MemberBindingTest {

  public static void main(String[] args) {
    SubClass subClass = new SubClass();
    System.out.println(subClass.x);
    subClass.method();

    SuperClass superClass = subClass;
    System.out.println(superClass.x);
    superClass.method();
  }
}
```



# 다형



❖ 참조 변수의 레벨에 따른 객체의 멤버 연결

- 상속 관계에서 객체의 멤버 변수가 중복될 때
  - 참조 변수의 타입에 따라 연결이 달라짐

- 상속 관계에서 객체의 메서드가 중복될 때(메서드가 override 되었을 때)
  - 무조건 자식 클래스의 메서드가 호출됨 → virtual method invocation
  - 최대한 메모리에 생성된 실제 객체에 최적화 된 메서드가 동작한다.

## ❖ 참조 변수의 레벨에 따른 객체의 멤버 연결

- 객체가 출력되는 과정
  - ◆ System.out.print(Object obj)를 이용한 객체 출력

```
SuperClass superClass = new SubClass();
System.out.print(superClass);
```

  - ◆ PrintStream의 print 메서드

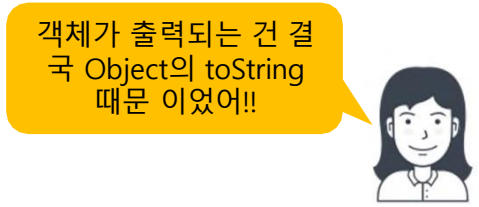
```
public void print(Object obj) {
  write(String.valueOf(obj));
}
```

  - ◆ String의 valueOf

```
public static String valueOf(Object obj) {
  return (obj == null) ? "null" : obj.toString();
}
```

  - ◆ Object의 toString

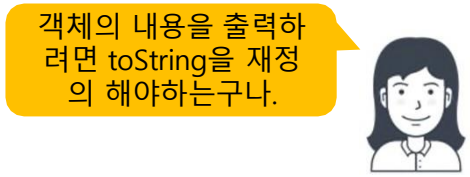
```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```



## ❖ 참조 변수의 레벨에 따른 객체의 멤버 연결

- 객체가 출력되는 과정

```
class UserInfo {
  String name = "홍길동";

  @Override
  public String toString() {
    return "이름: " + this.name;
  }
}

class MemberInfo extends UserInfo {
  String grade = "정회원";

  @Override
  public String toString() {
    return super.toString() + ", 등급: " + grade;
  }
}

public class PrintObject {

  public static void main(String[] args) {
    Object member = new MemberInfo();
    System.out.print("객체 정보: " + member);
  }
}
```



## ❖ 참조 변수의 레벨에 따른 객체의 멤버 연결

- 용도에 따른 가장 적합한 메서드 구성은?

```java
public void useJump1(Object obj) {
  if (obj instanceof Person) {
    Person casted = (Person) obj;
    casted.jump();
  }
}

public void useJump2(Person person) {
  person.jump();
}

public void useJump3(SpiderMan spiderMan) {
  spiderMan.jump();
}
```

```java
Object obj = new Object();
Person person = new Person();
SpiderMan sman = new SpiderMan();

AppropriateParameter ap = new AppropriateParameter();
// 호출은 가능하지만 실제로 jump할 수는 없다.
ap.useJump1(obj);
ap.useJump1(person);
ap.useJump1(sman);

//  ap.useJump2(obj);
ap.useJump2(person);
ap.useJump2(sman);

// ap.useJump3(obj);
// ap.useJump3(person);
ap.useJump3(sman);
```

- 상위로 올라갈 수록 활용도도 높아짐
  - ◆ 하지만 코드의 복잡성도 함께 증가
- Java API처럼 공통 기능인 경우 Object를 파라미터로 쓰겠지만
  - ◆ 많은 경우 비즈니스 로직 상 최상위 객체 사용 권장