상속과 다형성

새로운 클래스를 만들 때 기존 클래스의 활용 방법은 포함과 상속이라는 두 가지 방법 중에서 하나를 선택할 수 있다. 상속은 'IS-A' 관계 즉, '이다' 관계가 자연스럽게 성립되는 두 클래스의 관계를 적용하는 구문으로 재사용을 높이고 코드의 중복을 제거하여 프로그램의 생산성과 유지보수에 크게 기여하며 다형성(polymorphism)이라는 OOP의 중요한 특징을 제공하여 유연성 있는 프로그램 개발이 가능하도록 한다.

상속

상속이란 기존의 클래스를 재사용하여 새로운 클래스를 작성하는 것이다. 상속을 통해서 클래스를 작성하면 보다 적은 양의 코드로 새로운 클래스를 작성할 수 있고 코드를 공통적으로 관리할 수 있기 때문에 코드의 추가 및 변경이 매우 용이하다.

■ 상속 관계를 적용한 Java 클래스 구현

새로운 클래스 구현 시 상속 관계를 적용하고자 하는 경우에는 클래스의 헤더에 extends 라는 절을 사용한다.

▶ extends 절을 생략한 클래스 정의

새로운 클래스를 만들 때 extends 절을 생략하면 java.lang 패키지의 Object 이라는 클래스를 자동 상속하게 된다. 컴파일 시 다음과 같이 extends 절이 컴파일러에 의해 추가된다.

```
class HelloApp {
}
```

```
class HelloApp extends java.lang.Object{
}
```

java.lang.Object 클래스는 Java 에서 구현되는 모든 클래스들의 최상위 클래스이다. 그러므로 생성되는 모든 클래스들은 직간접적으로 java.lang.Object 클래스를 상속하게 된다. 이 클래스 안에는 Java 프로그램이 수행하는 동안 만들어지는 모든 객체들이 공통으로 가지고 있어야 할 11개의 메서드들을 정의하고 있다.

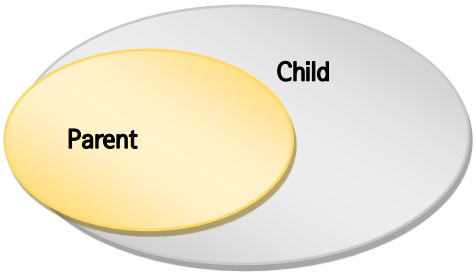
Object클래스의 메서드	설 명
protected Object clone()	객체 자신의 복사본을 반환한다.
public boolean equals(Object obj)	객체 자신과 객체 obj가 같은 객체인지 알려준다.(같으면 true)
protected void finalize()	객체가 소멸될 때 가비지 컬렉터에 의해 자동적으로 호출된다. 이 때 수행되어야하는 코드가 있는 경우에만 오버라이딩한다.
public Class getClass()	객체 자신의 클래스 정보를 담고 있는 Class인스턴스를 반환한다.
public int hashCode()	객체 자신의 해시코드를 반환한다.
public String toString()	객체 자신의 정보를 문자열로 반환한다.
public void notify()	객체 자신을 사용하려고 기다리는 쓰레드를 하나만 깨운다.
public void notifyAll()	객체 자신을 사용하려고 기다리는 모든 쓰레드를 깨운다.
public void wait()	다른 쓰레드가 notify()나 notifyAll()을 호출할 때까지 현재 쓰레드를 무한히 또는 지정된 시간(timeout, nanos)동안 기다리게 한다. (timeout은 천 분의 1초, nanos는 10^9분의 1초)
public void wait(long timeout)	
public void wait(long timeout, int nanos)	

▶ extends 절을 명시하는 클래스 정의

새로운 클래스를 만들 때 상속하고자 하는 클래스가 존재하는 경우에는 extends 절을 사용하여 상속 관계를 정의한다. 상속이라는 것은 기존의 클래스를 재사용하여 새로운 클래스를 작성하는 것이고 두 클래스를 조상과 자손이라는 관계로 맺어주는 것으로 자손은 조상의 생성자, 초기화 블록을 제외한 모든 멤버를 상속받는다.

```
class Child extends Parent{
    :
}
```

Parent 클래스를 확장(상속)하여 Child 클래스를 정의하고 있다.

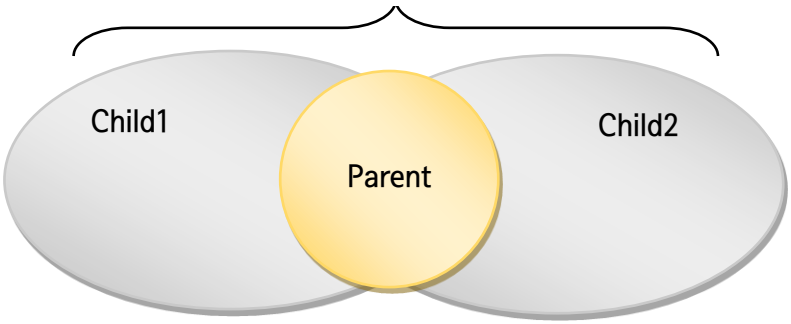


조상인 Parent 의 기능을 자손인 Child 에서 포함하고 있다. Child 는 Parent 의 기능을 대신할 수 있다.

▶ 상속 관계에 있는 클래스들의 특징

```
class Parent{
    :
}
class Child1 extends Parent{
    :
}
class Child2 extends Parent{
    :
}
```

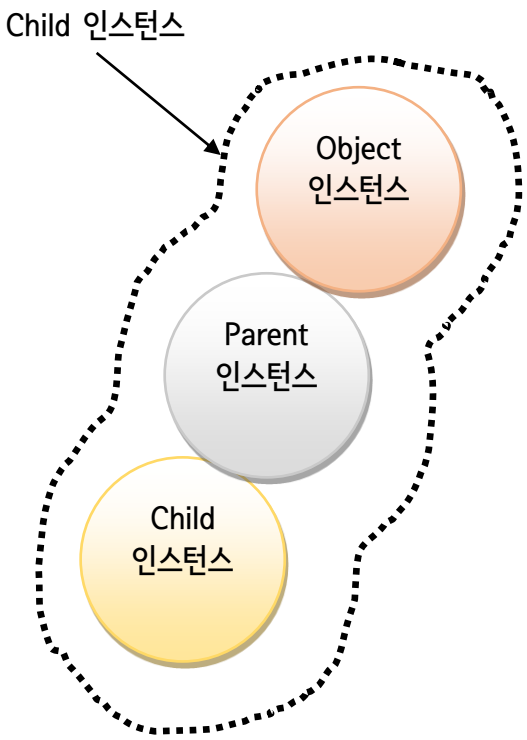
Child1 과 Child2 의 공통 기능은 Parent 에 정의하고 각 클래스에 특화된 기능은 Child1 과 Child2 에 구현한다. Child1 과 Child2 모두 Parent 의 기능을 포함한다.



다음 예와 같이 Child 인스턴스를 생성하면 Child 클래스, Parent 클래스 그리고 Object 클래스의 인스턴스가 모두 생성되며 이 세 개의 인스턴스를 모아놓은 것이 바로 Child 의 인스턴스가 된다. 상속을 한다는 것은 조상의 멤버들을 포함하여 조상의 기능을 대신할 수 있는 결과가 되는 것이다. 그러므로 obj 변수로 Child 에서 제공되는 z 변수뿐만 아니라 조상클래스인 Parent 에서 제공되는 x 와 y 변수도 사용할 수 있게 된다.

```
class  Parent{
    int x=10;
    int y=20;
}
class  Child  extends  Parent{
    int z=30;
}

class  ParentChildTest {
    public static void main(String args[]) {
        Child obj = new Child();
        System.out.println("x="+obj.x);
        System.out.println("y="+ obj.y);
        System.out.println("z="+ obj.z);
    }
}
```



■ super 변수와 super() 메서드

자손 클래스의 클래스 로딩시에는 조상 클래스의 클래스가 로딩되어 있는 상태인지를 점검하여 로딩되어 있지 않은 상태라면 조상 클래스도 로딩되며 자손 클래스의 인스턴스 생성시에는 조상 클래스의 인스턴스들도 생성된다.

그러므로 클래스의 인스턴스 생성과정에서 생성자 메서드가 호출될 때 조상들의 생성자도 호출되어 조상 클래스의 인스턴스도 초기화된다. 조상의 생성자들 중에서 어떠한 생성자가 호출되기를 원하는가에 따라 자손 클래스의 생성자 안에 super() 메서드를 사용하여 직접 호출할 수도 있다.

▶ super() 메서드

super() 메서드는 this() 메서드와 같이 생성자 메서드내에서만 사용 가능하며 첫 번째 수행 문장으로만 호출이 가능한 메서드이다. Java 에서 최상위 클래스인 Object 클래스를 제외하고 모든 클래스의 생성자 메서드에서는 super() 메서드를 호출해주어야 하는데 생략되면 파라미터를 전달하지 않는 super() 메서드 호출문장이 컴파일러에 의해서 자동적으로 추가된다.

```
class  Parent{
    int x=10;
    int y=20;
}
class  Child  extends  Parent{
    int z=30;
    Child() {
        x = 100;
        y = 200;
        z = 300;
    }
```

컴파일러에 의해서 자동으로 super(); 가 추가된다. super() 는 조상의 파라미터를 전달하지 않는 생성자를 호출하는 기능이다.

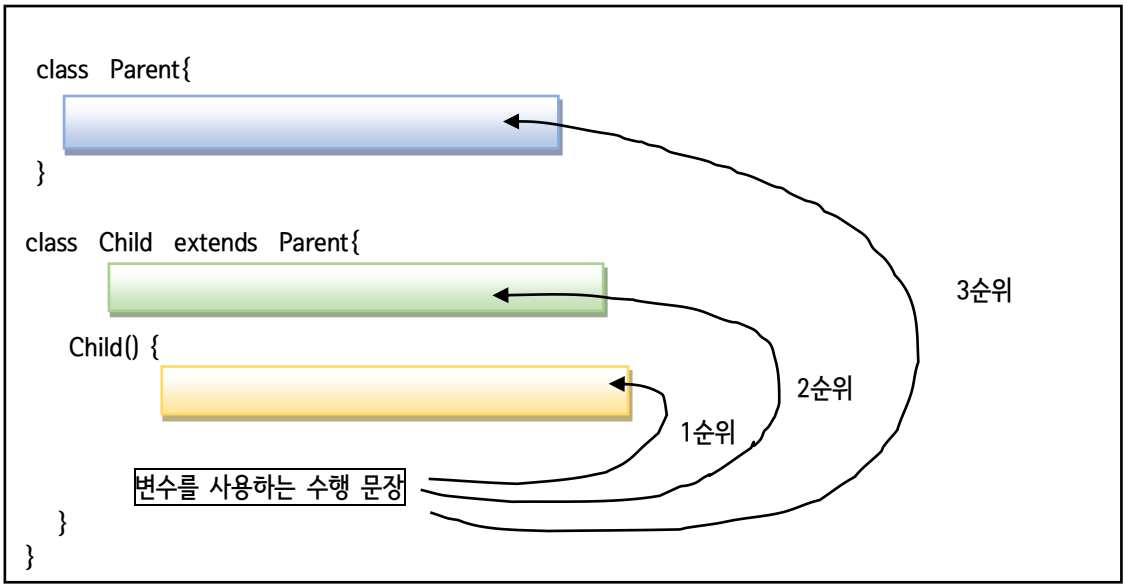
컴파일러가 자동으로 추가해주는 super() 는 괄호안에 파라미터 사양이 없으므로 조상의 파라미터를 전달받지 않는 생성자를 호출하게 된다. 상속하려는 조상 클래스에 파라미터를 전달받지 않는 생성자가 없다면 어떻게 해야 되는가? 자손 클래스의 생성자에서 조상 클래스가 가지고 있는 생성자 또는 호출하고자 하는 사양의 생성자를 호출하도록 super() 메서드를 직접 구현해야 한다.

```
class  Parent{
    int x, y;
    Parent(int  x, int  y) {
        this.x = x;
        this.y = y;
    }
}

class  Child  extends  Parent{
    int z;
    Child() {
        // 조상클래스의 생성자 중에서 int 형 데이터 2개를 파라미터로 전달받는 생성자
        // 를 호출하여 100, 200 을 각각 x 와 y 에 설정한다.
        super(100, 200);
         z = 300;
    }
}
```

▶ super 변수

자손 클래스의 생성자 메서드와 인스턴스 메서드에서 사용 가능한 변수로서 조상 인스턴스를 참조 값을 갖도록 자동 초기화되는 변수로서 동일 클래스의 멤버와 조상의 멤버를 구별하는 용도로 사용된다.
다음은 메서드에서 변수를 사용할 때 적용되는 우선 순위를 소개하는 그림이다. 메서드 내에서 변수가 사용되면 동일 메서드의 지역 변수가 우선 적용되고 다음은 동일 클래스의 멤버 변수, 그 다음은 조상 클래스의 멤버 변수가 된다.
이 때 동일 메서드의 지역 변수 또는 동일 클래스의 멤버 변수가 있음에도 불구하고 조상 클래스의 멤버 변수를 사용하려는 경우 사용되는 변수가 바로 super 이다.



다음은 super 변수가 사용된 예제 소스이다.

```
class  Parent{
   int x, y;
}

class  Child  extends  Parent{
   int  z;
   Child(int x, int y, int z) {
      super.x = x;
      super.y = y;
      this.z = z;
   }
}
```

■ 메서드 오버라이딩

조상 클래스로 부터 상속받은 메서드의 수행 코드를 자손 클래스에 맞게 재정의하는 것을 메서드 오버라이딩이라 한다. 메서드 오버라이딩을 구현하기 위해서는 반드시 메서드의 선언부가 동일해야 한다.

선언부가 동일하다는 것은 다음에 제시된 예제 소스에서와 같이 메서드명, 매개변수 사양, 리턴값의 유형이 동일하다는 것을 뜻한다.

```
class  Parent{
   int x, y;
   Parent(int  x, int  y) {
      this.x = x;
      this.y = y;
   }
   String getInfo() {
      return "x="+x+",y="+y;
   }
}
class  Child  extends  Parent{
   int z;
   Child() {
       super(100, 200);
       z = 300;
   }
   String getInfo() {        // 조상 클래스의 getInfo() 메서드를 오버라이딩 했다.
      return "x="+x+",y="+y+",z="+z;
   }
}
```

[메서드 오버라이딩시의 주의 사항]

- 리턴 값의 유형, 메서드명, 매개변수 사양이 모두 동일해야 한다.
- 매개변수 사양이 다르면 메서드 오버로딩이 된다.
- 메서드 명이 다르면 새로운 메서드를 추가한 것이 된다.
- 리턴 값의 유형이 다르면 컴파일 오류가 발생한다.
- 접근 제어자는 접근 범위가 동일하거나 더 넓어질 수는 있지만 더 좁아지게 설정할 수 없다.
- 메서드 헤더에 지정 가능한 throws 절은 조상과 동일하게 주거나 생략할 수는 있어도 다른 예외 클래스를 추가하거나 조상 예외 클래스를 설정할 수는 없다.

위의 예제에서 getInfo() 메서드를 자손 클래스를 오버라이딩 할 때 다음과 같이 super 를 사용하여 수행하려는 일부 기능을 조상의 메서드를 호출하여 대신 처리하도록 할 수 있다.

```
String getInfo() {
        return   super.getInfo()+",z="+z;
}
```

다형성(polymorphism)

다형성이란 여러 가지 형태를 가질 수 있는 능력으로서 하나의 참조형 변수로 여러 타입의 객체(클래스의 인스턴스)를 참조할 수 있는 기능을 말한다.

조상 타입의 변수로 자손 타입의 인스턴스를 다룰 수 있도록 하는 것

■ 참조형 변수의 인스턴스 참조

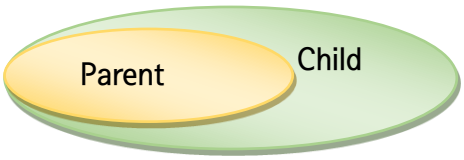
사용하려는 인스턴스의 타입이 Date 형이면 Date 형 변수 GregorianCalendar 형이면 GregorianCalendar 타입의 변수에 대입하여 사용하게 된다. 그래야만 해당 인스턴스를 참조하는 참조형 변수를 가지고 인스턴스의 필요한 모든 멤버를 접근하고 사용할 수 있기 때문이다.

```
Date obj1 = new Date();
GregorianCalendar obj2 = new GregorianCalendar();
StringBuffer obj3 = new StringBuffer(300);
```

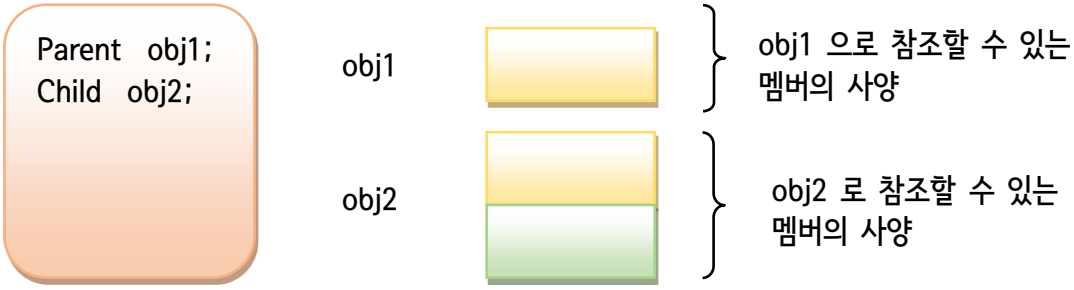
참조형 변수는 객체(클래스의 인스턴스)를 참조하는 기능을 지원하며 어떠한 타입인가에 따라서 멤버 연산자로 참조할 수 있는 멤버들의 사양이 정해진다.

Parent 타입의 참조형 변수이면 Parent 에 정의된 그리고 상속받은 멤버들을 접근할 수 있는 성격의 변수가 되는 것이고 Child 타입의 참조형 변수이면 Child 에 정의된 그리고 상속받은 멤버들을 접근할 수 있는 성격의 변수가 되는 것이다. Parent 클래스와 Child 클래스의 멤버 사양이 다음과 같은 경우

```
class Parent {
}
class Child extends Parent {
}
```

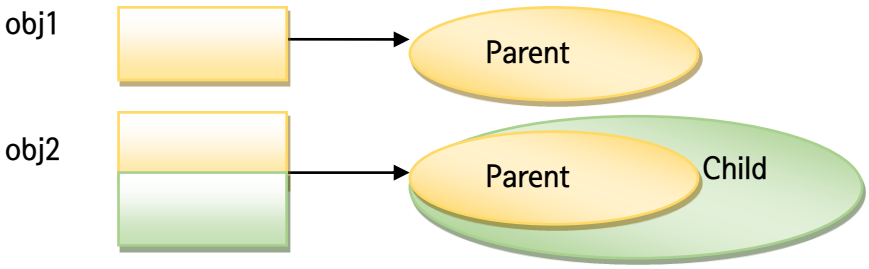


obj1 과 obj2 가 참조할 수 있는 멤버 사양은 다음 그림과 같다.



obj1 에 Parent 인스턴스를 그리고 obj2 에 Child 인스턴스를 대입하여 멤버들의 접근 범위를 소개하는 그림이다.

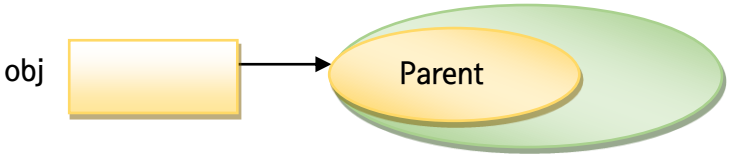
```
Parent  obj1 = new Parent();
Child  obj2 = new Child();
```



위의 그림을 보면 참조형 변수가 참조하려는 멤버들의 사양과 대입된 인스턴스가 가지고 있는
멤버들의 사양이 동일한 것을 볼 수 있다. 그러므로 일반적인 구현에서는 인스턴스를 생성하면 동일 유형의 참조형 변수에 담아서 사용하게 되는 것이다.

다형성이라는 것은 조상의 참조형 변수로 자손의 인스턴스를 참조할 수 있는 기능으로서 인스턴스를 생성하면 조상 타입의 변수에 대입하여 사용할 수도 있다는 것이다. 다음 그림과 같이 자손의 인스턴스는 조상 타입의 변수에 대입하여 사용하는 것이 가능하다.

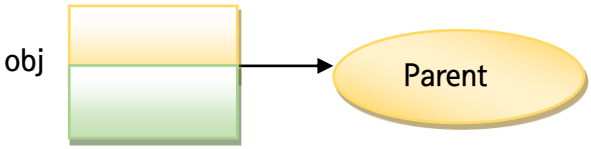
```
Parent  obj = new Child();
```



Child 에는 Parent 의 기능이 포함되어 있으므로 obj1 으로 참조 가능한 멤버의 사양을 Child 인스턴스를 통해 접근할 수 있다.

그러나 다음 그림과 같이 조상의 인스턴스는 자손 타입의 변수에 대입하여 사용하는 것은 불가능하다.

```
Child  obj = new Parent();//오류
```



Child 에서 추가된 멤버를 Parent 인스턴스를 통해 접근할 수 없으므로 수 있다. 조상의 인스턴스를 자손 타입의 변수에 대입하는 것은 불가능하다.

■ **참조형간의 형 변환과 instanceof 연산자**

다형성은 메서드의 매개변수 선언 시에도 적용된다. 매서드에 정의된 매개변수의 타입이 참조형인 경우에는 다음과 같은 전달 규칙이 적용된다.

> Parent 형이면 Parent 와 Parent 의 자손 인스턴스가 전달될 수 있다.
> Child 형이면 Child 와 Child 의 자손 인스턴스가 전달될 수 있다.
> Object 형이면 Object 와 Object 의 자손 인스턴스가 전달될 수 있다.

특히 Object 의 경우에는 Java 에서 만들어지는 모든 객체들이 Object 의 자손이므로 “어떠한 타입에서 만들어지는 인스턴스든 전달될 수 있다”, “객체라면 무엇이든 전달 가능하다”라는 의미가 된다.

그런데 이 때 자손타입의 인스턴스를 조상 타입의 변수에 대입한 경우 조상 타입의 변수로는 자손에서 추가된 멤버를 접근할 수 없다. 조상 타입의 변수로 자손에서 추가된 멤버를 접근하려면 조상타입의 변수를 자손 형으로 형 변환을 해주어야 하는데 이 때에는 형 변환 연산자를 사용하여 명시해 주어야 한다.

▶ 참조형 간의 형 변환

서로 다른 참조형 간의 형 변환은 상속 관계에 있는 경우에만 가능하다. 자손에서 조상타입으로의 변환은 형 변환 연산자를 생략할 수 있으며 조상에서 자손타입으로의 형 변환의 경우에는 형 변환 연산자를 생략할 수 없다.

어떠한 클래스든 조상 클래스는 하나이므로 조상 타입으로 변환할 때는 형 변환 연산자를 생략할 수 있지만 자손 타입으로 변환할 때는 어떠한 자손 타입으로의 변환인지 명시해 주어야 하기 때문이다.

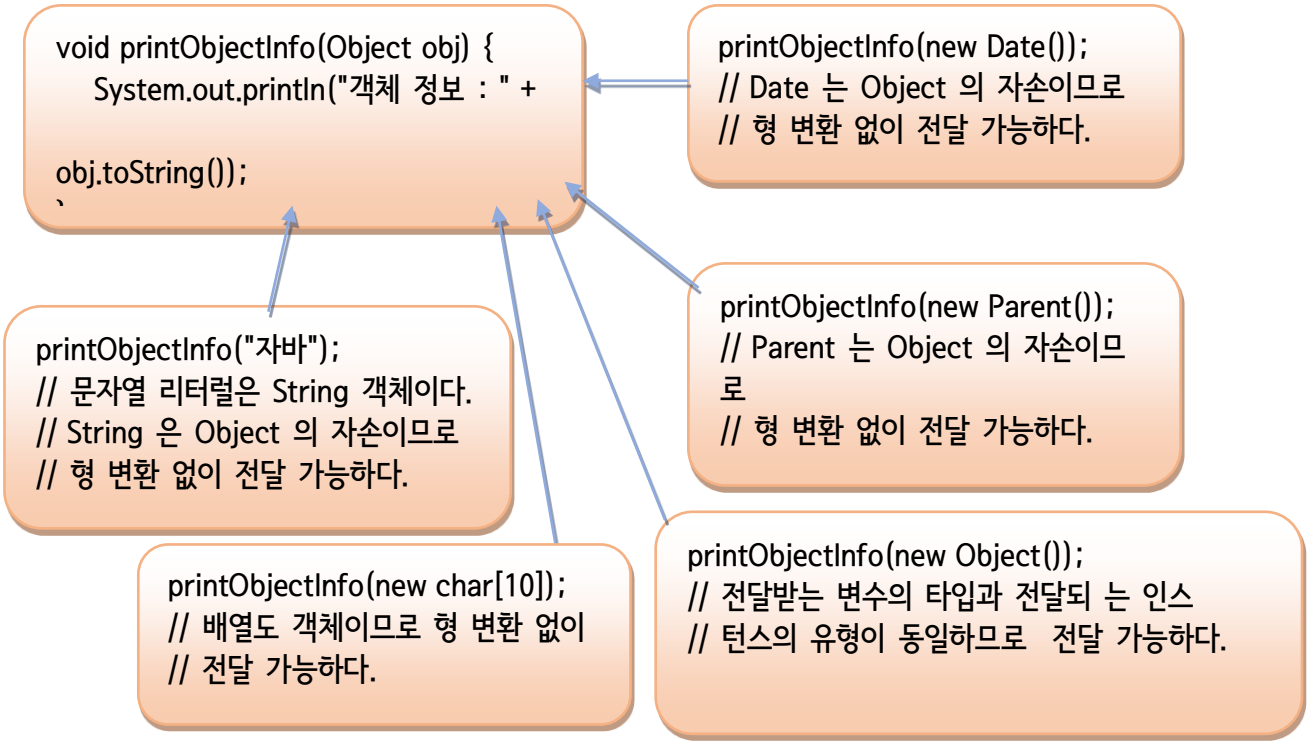
```
조상 타입 = 자손 타입;
자손 타입 = (자손 타입)조상 타입;
```

조상 타입을 자손 타입으로 강제 형 변환 연산을 사용한 경우 컴파일은 성공적으로 처리되며 또한 실행 시 참조되는 인스턴스를 자손형으로 변환할 수 있는지 체크하여 변환이 적절하지 않은 경우 ClassCastException 오류를 발생한다.

```
Parent   obj1 = new Parent();
Child   obj2 = new Child();
Parent obj3 = new Child();
//컴파일 오류 : 조상 인스턴스를 자손형 으로 변환할 수 없다.
Child obj4 = new Parent();
//실행 오류 : 조상형 변수를 자손형으로 변환할 때는 강제 형 변환 연산자를 사용해야
//하며 연산자를 사용하고 있어서 컴파일은 되지만  obj1 이 참조하는 인스턴스가 Child
//형으로 변환될 수 없는 경우이기 때문에 실행예외(ClassCastException)가 발생한다.
obj2 = (Child)obj1;
obj1 = obj2;        // 자손형 변수를 조상형 변수에 대입하므로 자동 형 변환이 일어난다.
//컴파일 오류 : 조상형 변수를 자손형으로 변환할 때는 강제 형 변환 연산자를 사용해야 한다.
obj2 = obj3;
obj2 = (Child)obj3;// obj3가 참조하는 인스턴스가 Child 형이므로 형 변환이 일어난다.
```

참조형 변수에 강제 형 변환 연산자를 사용했다 하더라도 참조되는 인스턴스가 원하는 타입으로 변환 가능한지의 여부는 실행시 채크가 이루어지며 변환할 수 없는 경우에는 ClassCastException 오류가 발생한다.

다음은 Object 형으로 매개변수를 선언하고 있는 메서드를 호출할 때 전달되는 인스턴스의 종류를 소개하는 예이다.



그런데 여기에서 obj 변수에 전달되는 인스턴스의 타입이 String 형인 경우에 length() 라는 메서드를 호출하고자 한다면 어떻게 구현할 수 있을까?

obj 변수에는 어떤 타입의 인스턴스든 전달될 수 있다. 전달된 인스턴스의 타입이 무엇인지를 채크하여 처리하고자 하는 경우에 사용되는 연산자가 바로 instanceof 연산자이다.

▶ instanceof 연산자

참조 형 변수가 참조하는 인스턴스의 실제 타입을 체크하는데 사용되는 연산자이다. 이항 연산자로서 왼쪽 항에는 참조형 변수를 그리고 오른쪽 항에는 채크하고자 하는 인스턴스의 타입을 지정하며 연산결과는 boolean 형이다. instanceof 의 연산결과가 true 이면 해당 타입으로 형 변환이 가능하다는 것을 의미한다.

다음은 위의 예제에서 printObjectInfo() 수정한 소스내용이다.

```
void printObjectInfo(Object obj) {
    if (obj instanceof String) {
        String s = (String)obj;       // String 에서 추가된 메서드를 호출하기 위해
                                      // String 형으로 강제 형 변환을 하고 있다.
        System.out.println("길이 : " + obj.length());
    }
    System.out.println("객체 정보 : " + obj.toString());
}
```

조상 타입의 변수인 obj 로 참조되는 자손 인스턴스에서 자손에서 추가된 멤버를 접근하려는 경우 다음과 같이

```
System.out.println("길이 : " + ((String)obj).length());
```

형 변환 연산자를 멤버 연산자와 함께 사용할 수도 있다. 중요한 것은 멤버 연산의 우선순위가 높으므로 형 변환 연산이 먼저 수행되도록 괄호를 사용해야 한다.

자손은 조상의 기능을 포함하고 있고 조상의 역할을 대신할 수 있으므로 자손의 인스턴스를 가지고 조상 타입인지를 점검하게 되면 true 이다. 다음과 같은 계층 구조를 가지고 있는 클래스들의 경우

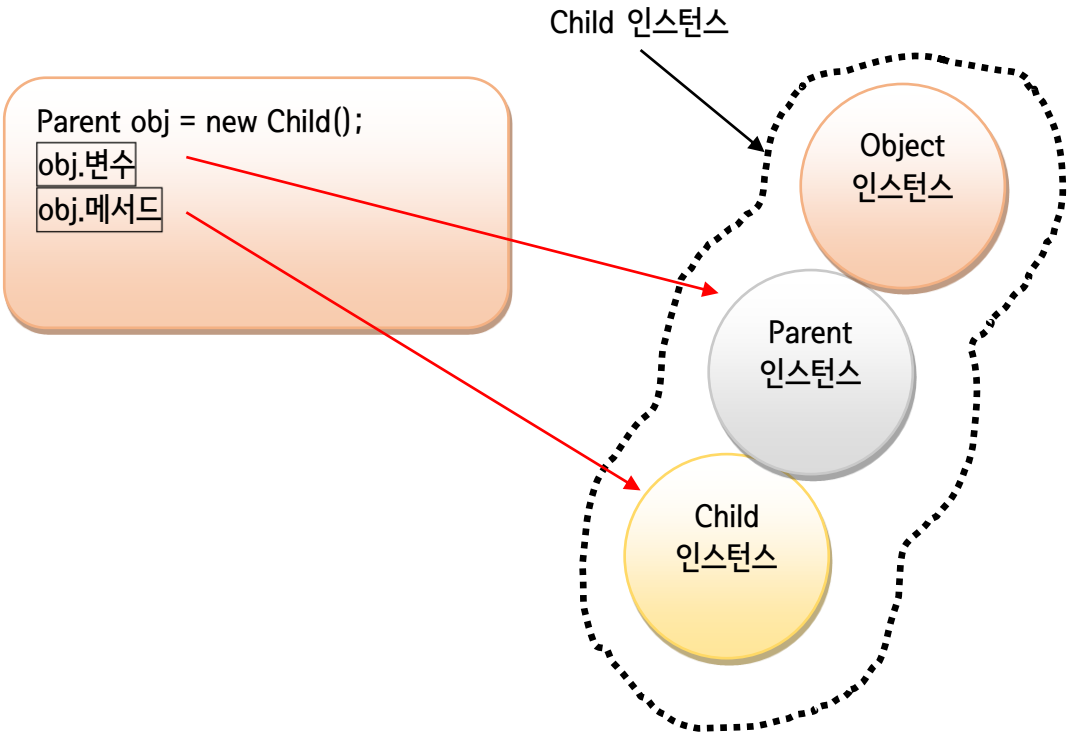
```
class Parent {
}
class Child extends Parent {
}
```

```
Child obj = new Child();
obj  instanceof  Child     // true
obj  instanceof  Parent    // true
obj  instanceof  Object    // true
```

■ **다형성과 참조형 변수의 멤버 사용**

자손 클래스의 인스턴스를 조상 클래스의 변수에 대입한 경우 멤버 연산자를 사용하여 메서드를 호출할 때와 변수를 접근할 때 적용되는 규칙이 다르다.

멤버 변수를 접근할 때는 변수의 타입에 따라서 사용될 인스턴스를 선택하고 메서드를 호출할 때는 참조하는 인스턴스의 타입에 따라서 인스턴스를 선택한다. 그러므로 메서드의 경우에는 오버라이딩된 자손 클래스의 메서드가 우선적으로 호출되며 오버라이딩이 되어 있지 않은 경우에는 조상에 정의된 메서드가 호출된다.



다음은 다형성이 적용된 참조형 변수의 멤버 접근을 점검해 볼 수 있는 예제이다.

```
class   Parent{
    int x =10;
    int y = 20;
    String getInfo() {
        return "x="+x+",y="+y;
    }
    String getClassName() {
        return "Parent";
    }
}
class   Child   extends   Parent{
    int x =100;
    int y = 200;
    int z = 300;
    String getInfo() {
        return "x="+x+",y="+y+",z="+z;
    }
}
```

```
Parent obj = new Child();
// 10 이 출력됨
System.out.println(obj.x);
// 20 이 출력됨
System.out.println(obj.y);
// 오류발생
System.out.println(obj.z);
// 자손 타입으로 강제 형 변환하여
// 자손에서 추가된 멤버 사용 가능
System.out.println(((Child)obj).z);
// Child 의 getInfo()가 호출됨
System.out.println(obj.getInfo());
// Parent 의 getClassName ()이 호출됨
System.out.println(obj.getClassName());
```

제네릭스

제네릭스는 Java SE 5.0부터 추가된 구문으로서 객체의 타입 안정성을 높이고 형 변환의 번거러움을 줄여주는 구문이다. 제네릭스 구문이 사용되기 전에는 다양한 종류의 객체 타입을 다루는 메서드의 매개변수나 리턴 값의 타입 그리고 배열을 생성할 때도 Object이라는 클래스를 많이 사용해 왔지만 제네릭스 구문을 적용하게 되면 프로그램의 타입 안정성도 높이고 형 변환의 불편함을 해결할 수 있다.

■ 제네릭스 구문의 적용과 사용

제네릭스를 적용하여 구현된 클래스는 제네릭스 타입을 사용하여 객체를 생성하고 활용할 수 있다. 먼저 제네릭스를 적용하여 클래스를 구현하는 방법부터 학습한다.

▶ 제네릭스를 적용한 클래스의 구현

여러 다양한 참조 형 타입(객체 타입)의 데이터들을 처리하는 기능의 클래스라면 구현할 때는 제네릭스라는 구문을 적용하는 것을 고려할 수 있다. 제네릭스 구문을 적용하여 클래스를 만들 때는 다음과 같이 클래스명에 꺾쇠괄호(<>)와 함께 전달 받고자 하는 참조형 타입의 개수만큼 타입을 나타내는 기호를 콤마 연산자와 함께 나열한다.

```
class   클래스명<기호> {    }
class   클래스명<기호, 기호, …> {    }
```

여기에서 사용되는 기호는 **타입 파라미터**라고 불리며 Java의 식별자 규칙을 준수한다면 어떠한 문자든 사용 가능하다. 그러나 많이 사용되는 기호로는 Type을 의미하는 T, Element를 의미하는 E, Key를 의미하는 K 그리고 Value를 의미하는 V가 많이 사용되고 있다. 기호로 어떠한 명칭을 사용하든 **'임의의 참조형 타입'**이라는 의미는 모두 같다.

```
class   ObjectInfo<T> {    }
class   Box<K, V, M> {    }
class   Car<D, G> {    }
```

클래스명 뒤에 타입 파라미터가 정의되어 있다는 것은 이 클래스를 객체 생성할 때 다음과 같이 이 클래스에서 처리하게 될 기능과 관련하여 참조 형 타입에 대한 정보 즉, 제네릭스 타입을 같이 전달하라는 의미이다.

```
ObjectInfo<String> obj = new ObjectInfo<String>();
ObjectInfo<Integer> obj = new ObjectInfo<Integer>();
Box<String, String, Date> b = new Box<String, String, Date>();
```

참조형 변수를 선언할 때, 객체를 생성할 때 모두 클래스명 뒤에 꺾쇠괄호(<>) 안에 이 클래스를 통해서 처리하고자 하는 참조형 타입을 지정한다.

이렇게 전달되는 참조 형 타입은 클래스에 정의된 기호를 통해서 클래스 내부의 메서드나 멤버 변수에서 사용될 수 있다. 다음 소스를 점검해 보자.

```
class Value3<T> {
    T obj;                  // 타입 파라미터로 전달된 타입으로 변수를 선언
    void put(T   obj){      // 타입 파라미터로 전달된 타입으로 매개변수를 선언
        this.obj = obj;
    }
    T get() {               // 타입 파라미터로 전달된 타입으로 리턴 타입을 선언
        return obj;
    }
}
```

Value3 를 객체 생성할 때 다음과 같이 객체를 생성하면

```
Value3<String> obj = new Value3<String>()
```

Value3 클래스에서 다음과 같이 전달된 제네릭스 타입이 타입 파라미터가 사용된 위치에서 대체되어 사용된다.

```
class Value3<String> {
    String   obj;
    void put(String   obj) {
        this.obj = obj;
    }
    String   get() {
      return obj;
    }
}
```

Value3 를 객체 생성할 때 다음과 같이 객체를 생성하면

```
Value3<Date> obj = new Value3<Date>()
```

Value3 클래스에서 다음과 같이 전달된 제네릭 타입이 타입 파라미터가 사용된 위치에서 대체되어 사용된다.

```
class Value3<Date> {
    Date   obj;
    void put(Date   obj) {
        this.obj = obj;
    }
    Date   get() {
      return obj;
    }
}
```

이렇게 클래스를 정의할 때 타입 파라미터를 지정하면 클래스를 객체 생성하는 시점에서 이 클래스에서 처리하게 될

객체의 타입을 가변적으로 설정하는 결과가 된다. 즉, 클래스를 구현하는 시점이 아니라 사용하는 시점에서 클래스에서 처리하게될 데이터의 타입을 동적으로 결정하게 되는 결과가 된다. 다음은 Value3 클래스를 활용하여 구현된 소스이다.

```
Value3<String> obj1 = new Value3<String>();
obj1.put("자바");
// 타입 파라미터에 의해 get()메서드의 리턴 타입이 String이므로 강제 형 변환을
// 할 필요가 없다.
String s = obj1.get();
System.out.println(s);

Value3<Date> obj2= new Value3<Date>();
obj2.put("자바");
// 타입 파라미터에 의해 get()메서드의 리턴 타입이 Date이므로 강제 형 변환을
// 할 필요가 없다.
Date d = obj2.get();
System.out.println(d);
```

▶ 제네릭스와 다형성

자손의 객체를 조상 타입의 변수에 대입하여 조상 타입의 변수를 통해서 객체를 사용할 수 있도록 하는 것을 다형성이라고 한다. 제네릭스에 적용되는 다형성 규칙은 다음과 같다.

```
동일한 제네릭스 타입을 적용한 조상 타입의 변수 =
                        동일한 제네릭스 타입을 적용한 자손의 객체;
```

적용 예

```
List<Integer> numlist = new ArrayList<Integer>();
List<String> strlist = new LinkedList<String>();
```

주의할 것은 제네릭스 타입에 대한 다형성은 지원하지 않는다는 것이다. 다음과 같은 대입식은 허용되지 않는다.

```
ArrayList<Number> numlist1 = new ArrayList<Integer>();
List<Number> numlist2 = new ArrayList<Double>();
List<Object> strlist = new LinkedList<String>();
```

제네릭스 타입에 대해서는 다형성을 적용할 수 없다.

제네릭스에서는 단 하나의 타입을 지정하지만 와일드 카드를 이용하여 하나 이상의 타입을 지정하는 것을 가능하게 해준다.

```
List<?> numlist1 = new ArrayList<Integer>();
ArrayList<? extends Number> numlist2 = new ArrayList<Double>();
List<? extends Object> strlist = new LinkedList<String>();
```

'?' 기호를 사용할 때는 와일드 카드를 적용하고자 하는 범위(boundary)를 지정해 줄 수도 있으며 다음과 같이 두 가지로 나뉜다.

```
<? extends T> : T 또는 T 의 자손 타입

<? super T>   : T 또는 T 의 조상 타입
```

적용 예

```
List<? extends Number> numlist = new ArrayList<Integer>();
List<? extends Number> strlist = new LinkedList<Double>();
```

▶ 제네릭스와 메서드

제네릭스 타입은 클래스뿐만 아니라 메서드의 매개변수에도 적용할 수 있다. 메서드의 매개변수에 제네릭스 타입을 사용하려면 다음과 같이 리턴값의 타입과 제어자 사이에 제네릭스 타입을 선언한 다면 매개변수의 타입으로 선언된 제네릭스 타입을 사용한다.

```
[제어자] <T> 리턴 값의 타입 메서드명(<T> param)
[제어자] <T> 리턴 값의 타입 메서드명(<T> param1, <T> param2)
[제어자] <T> 리턴 값의 타입 메서드명(<T> param1, int param2)
[제어자] <T super 클래스명> 리턴 값의 타입 메서드명(<T> param)
[제어자] <T extends 클래스명> 리턴 값의 타입 메서드명(<T> param)
[제어자] <T extends 인터페이스명> 리턴 값의 타입 메서드명(<T> param)
```

적용 예

```
<T> void   testGenericMethod(T param){
    T obj = param;
    System.out.println("전달된 객체의 클래스명 : "
                              +obj.getClass().getName());
}

testGenericMethod(new Date());
testGenericMethod("JAVA");
```