06_2 Polymorphism

Object-Oriented Programming

Polymorphism에 대해 강의하겠습니다.

Automatic Type Conversion

- The automatic type conversion of a Class:
  - Descendant to the Ancestor type.
  - **Upcasting**이라 부름

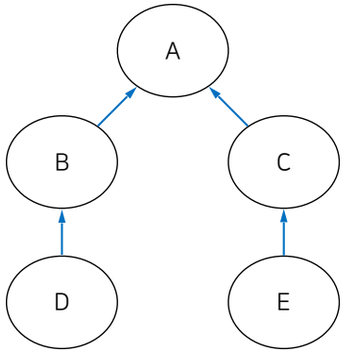
- Example)

```
class Animal { ... }
class Cat extends Animal { ... }

Cat cat = new Cat();
Animal animal1 = cat;
Animal animal2 = new Cat();
```


페이지 2

Class 간의 automatic type conversion이 가능한 경우는
Descendant class의 object를 ancestor class object에
assign하는 경우 입니다.
하위 (descendant) object를 상위 (ancestor) object에 assign하는 것이므로
이것을 upcasting이라 부릅니다.
upcasting 시에는 automatic type conversion이 되므로
explicit type conversion이 필요하지 않습니다.
예를 들어, Cat class가 Animal class의 child일 때
cat object를 animal object에 직접 assign하는 것이 가능합니다.

Inheritance Tree and Auto Type Conversion



class hierarchy
(child → parent)

```
B b = new B();
C c = new C();
D d = new D();
E e = new E();
```

```
A a1 = b; //ok
A a2 = c; //ok
A a3 = d; //ok
A a4 = e; //ok

B b1 = d; //ok
C c1 = e; //ok

B b3 = e; //error
C c2 = d; //error
```


페이지 3

이 inheritance tree는 class의 hierarchy를 나타내고 있습니다.
먼저 b, c, d, e를 각각 class B, C, D, E의 object로 생성하였습니다.
A가 B, C, D, E의 ancestor이기 때문에
A의 object에는 b, c, d, e들이
automatic type conversion되면서 assign 될 수 있습니다.
또, B는 D의 parent, C는 E의 parent이므로
d, e 는 각각 b1, c1에
automatic type conversion되면서 assign될 수 있습니다.
그러나 B는 E의 ancestor가 아니며
C는 D의 ancestor가 아니기 때문에
e를 b3에 directly assign 할 수 없고
d를 c2에 directly assign 할 수 없습니다.

Polymorphism in Method Call

```java
class Parent {
    void method1() { ... }
    void method2() { ... }
}
```

```java
class Child extends Parent {
    @override
    void method2() { ... }
    void method3() { ... }
}
```

```java
class ParentChildDemo {
    public static void main(String[] args) {
        Child child = new Child();
        Parent parent = child;

        parent.method1();

        // Child's overridden method2()
        parent.method2(); // by Polymorphism

        // ERROR: no method3() in Parent
        parent.method3();
    }
}
```


페이지 4

Parent class가 method1과 method2를 가지고 있고
Child class는 method2를 overriding하며
method3를 더 가지고 있다고 가정해 보겠습니다.
ParentChildDemo class의 main에서
child object를 하나 생성하고
이를 Parent object의 reference인 parent에 assign 하였습니다.
이 때 parent.method1() 을 call하면
이것은 당연히 Parent class의 method1을 실행합니다.
그런데 parent.method2() 를 call하면
Parent class의 method2() 와 Child class의 method2() 중에
어떤 것을 실행할까요?
parent object가 Parent class라는 점을 생각하면
Parent의 method2() 를 실행해야 할 것 같고
parent에 assign된 child가 원래 Child class의 object였던 것을 고려한다면
overriding된 Child의 method2() 를 실행해야 할 것도 같습니다.
여기서는 Child class의 method2() 를 실행해 줍니다.
parent에 assign된 child가 원래 Child class의 object였기 때문입니다.
이렇게 원래 생성된 출신 class를 기억하고 있다가
원래 생성된 class의 method를 실행해주는 이 기능을
polymorphism이라고 합니다.
그 아래에서는 parent.method3() 를 call 했습니다만
이 것은 compile error를 발생시킵니다.
방금 parent에 assign된 child가 Child class object로 생성되었다고 했으니
Child class에 있는 method3() 를 call 해 주는 것이 맞을 것 같은데
이것은 왜 error가 되는 것일까요?
그것은 parent에 assign된 object가 원래 Child class로 생성되었기는 하지만
현재 외적인 형식이 Parent class이며,
Parent class에는 method3() 가 없기 때문입니다.
즉 child object가 parent에 assign되는 순간
외형은 Parent object로 변하게 되며,
따라서 method3() 에 대한 정보는 완전히 잃어버리게 되는 것입니다.

Polymorphism

- In the previous example, when a parent class object behaves as if it were a child class object in some situations, this is called "polymorphism (다형성)".
- In other words, when a child class object is assigned to the parent class object, the parent object will call the overridden method in the child class.
- Polymorphism is implemented by Dynamic Binding (Late Binding).
- This means that we don't decide which 'method2' to run at compile time, but we decide which method2 to run at runtime.

- Advantages of Polymorphism
  - Flexibility in code: the same code can work for many different objects
  - Maintainability: New classes can be added without changing the code
  - Extensibility: Can extend functionality without code modifications

페이지 5

이렇게 parent class object이지만 마치 child class object처럼 실행되는 현상을
"polymorphism" (다형성) 이라 합니다.
다른 표현으로, child class object가 parent class object에 assign되었을 때
parent object는 overriden된 child의 method를 call해 준다는 것입니다.
이렇게 Polymorphism이 구현될 수 있는 이유는
method call이 어떤 method를 실행시키는지가 compile time이 아닌
runtime에 결정되는 dynamic binding mechanism을 이용하기 때문입니다.
Dynamic binding은 late binding이라 불리기도 합니다.
Polymorphism이 자칫 복잡하게 보일 수 있으나
몇가지 장점을 가지고 있으므로 잘 이용하는 것이 중요합니다.
먼저 code의 flexibility인데, 같은 code가 여러 서로 다른 object들에 대해
공통적으로 사용될 수 있다는 점 입니다.
Parent class를 inherit한 많은 descendant class의 object들은
각각이 서로 다를 수 있지만 parent class object에서
descendant들이 공통으로 가지고 있는 overriding된 method를 call해 주면
각각의 서로 다른 class의 object들이
서로 다른 일을 해 줄 수 있게 됩니다.
이러한 장점은 parent object에서의 method call 부분을 바꾸지 않고도
새로운 descendant class를 추가만 해 주면 새로운 일을 할 수 있기 때문에
유지보수를 쉽게 하면서도 코드를 바꾸지 않고 기능을 추가할 수 있게 해 줍니다.

Example: AnimalCoversionTest.java

```java
public class AnimalConversionTest {

    public static void main(String[] args) {
        Dog dog = new Dog("Jane", 8, "bulldog");
        Cat cat = new Cat("Kitti", 5, "white");

        Animal animal = new Animal("Tom", 3);
        animal.makeSound();  // original Animal's makeSound()
                             // “Some generic animal sound"
        animal = dog;
        animal.makeSound();  // dog's overridden makeSound()
                             // “Bark"
        animal = cat;
        animal.makeSound();  // cat's overridden makeSound()
                             // “Meow”
    }
}
```


페이지 6

이제 간단한 example로 polymorphism을 사용하는 예를 들어보겠습니다.
Dog와 Cat이 Animal의 child class들이고
Animal class의 makeSound() method가
Dog와 Cat에서 각각 overriding되어
다른 울음소리를 프린트한다고 가정해 봅니다.
Animal class object인 animal에서 makeSound를 call하면
“Some generic animal sound"를 print합니다.
이제 Dog object인 dog를 animal에 assign하고
animal.makeSound() 를 call하면 “Back"라고 print 합니다.
다시 Cat object인 cat을 animal에 assign하고
animal.makeSound() 를 call하면 “Meow" 라고 print 합니다.
이처럼 parent object에 assign된 child class object들이
다른 child class 소속일 때
parent object에서의 method call은
child class에서 overriding된 method를 찾아서 call 해주게 됩니다.

Polymorphism in Parameters

- Take a parent (ancestor) class type as a parameter in a method
- Then, any child (descendant) class of the parameter type can be passed as the parameter
- Ex)

```
class Parent { }
class Child extends Parent { }
...
Child c = new Child();
Parent p = new Parent();
...
void someMethod(Parent p) { }
...
someMethod(p);
someMethod(c);  // OK! because of the polymorphism
```


페이지 7

Polymorphism의 개념은 method의 parameter passing에서도
유용하게 사용될 수 있습니다.
어떤 method에서 ancestor class type의 parameter를
formal parameter p로 받는다고 가정할 때
p와 같은 class의 object는 물론이고
p의 모든 descendant class 의 object들은
p에 대응하는 actual parameter로 pass될 수 있습니다.
예를 들어보면
Parent class가 있고
Parent를 inherit한 Child class가 있을 때
c와 p는 각각 Child와 Parent의 object들 입니다.
이 때 someMethod가 Parent p를 parameter로 받는다면
someMethod(p); 도 가능하지만
someMethod(c); 도 가능합니다.
만일 someMethod안에서 p.method() 라는 call이 있었을때
p가 Parent class출신이면 Parent의 method()가 실행될 것이고
p가 Child class출신이면 overriding된 Child의 method()가
실행될 것입니다.

Example) ShapeDemo (1/2)

```java
class Shape {
    void draw() {
        System.out.println("Drawing Something");
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
```


페이지 8

좀 더 구체적인 예를 들어보도록 하겠습니다.
Shape class는 draw() 라는 method를 가지고 있습니다.
실제로 무엇인가를 그려내는 method여야 하겠지만
여기서는 그냥 "Draw Something" 이라는
message를 print하는 것으로 하겠습니다.
Shape의 child인 Circle class에서는
draw()를 override하여
"Drawing a Circle" 을 print하고
역시 Shape의 child인 Rectangle에서는
"Drawing a Rectangle"을 print합니다.

Example) ShapeDemo (2/2)

```java
class ShapeDrawer {
    // Use polymorphism to take a Shape (Parent, Ancestor) type as a parameter
    public void drawShape(Shape shape) {
        shape.draw(); // At runtime, appropriate class' draw method is called
    }
}

public class ShapeDemo {
    public static void main(String[] args) {
        ShapeDrawer shapeDrawer = new ShapeDrawer();

        Shape myShape = new Shape();
        Shape myCircle = new Circle();
        Shape myRectangle = new Rectangle();

        // Using polymorphism for method parameter
        shapeDrawer.drawShape(myShape);  // Output: Drawing Something
        shapeDrawer.drawShape(myCircle); // Output: Drawing a Circle
        shapeDrawer.drawShape(myRectangle); // Output: Drawing a Rectangle
    }
}
```


페이지 9

ShapeDrawer class에는 drawShape이라는 method가 있는데
이 method는 Shape class type인 shape을 parameter로 받아서
shape.draw() 를 call하여 주고 있습니다.
이제 ShapeDemo class의 main method에서
먼저 ShapeDrawer class object인 shapeDrawer를 생성하고
Shape, Circle, Rectangle class object들인
myShape, myCircle, myRectangle을 생성하였습니다.
이제 shapeDrawer.drawShap() 을 세번 call 하는데
그 parameter를 각각 myShape, myCircle, myRectangle로 하면
각각의 parameter의 class type별로 다른 draw() method가 실행되어
적절한 message들이 프린트되게 됩니다.

Upcasting vs Downcasting

- Upcasting
  - Automatic type conversion from descendant class to ancestor class
  - No explicit casting
  - ex) Child c = new Child();
    Parent p = c; // upcasting
- Downcasting
  - Type conversion from ancestor class to descendant class
  - Explicit casting needed
  - ex) Parent p1 = new Child(); // upcasting
    Child c = (Child) p1; // downcasting OK
    Parent p2 = new Parent();
    Child c = (Child) p2; // ERROR!! Runtime error

페이지 10

이제 Upcasting과 Downcasting의 차이에 대해 알아보도록 하겠습니다.
Upcasting은 지금까지 우리가 계속 봐 왔던것과 같이
descendant object를 ancestor object에 assign하는 경우를 말합니다.
이 경우는 automatic type conversion이 지원되기 때문에
explicit type conversion을 할 필요가 없습니다.
Downcasting은 반대로 parent object를 child object에 assign하는 경우를 말합니다.
이 경우는 explicit type conversion이 필요합니다.
그런데 ancestor를 descendant에 assign하는 모든 경우가
다 downcastring이 가능한 것은 아닙니다.
example과 같이 Parent p1에 일단 Child object가 upcasting되었다가
이 p1을 Child c = (Child)p1; 과 같이 downcasting 하는 것은 가능합니다.
그러나 Parent p2 = new Parent(); 의 p2처럼
Original class가 Child가 아닌 Parent 였다면
p2를 c에 downcasting 하려는 순간 runtime error가 나게 됩니다.
즉, downcasting을 하기 전에
descendant object에 assign하려는 ancestor object가
원래 그 descendant object 출신인지를 먼저 test해 보고
downcasting을 해야 하는 것입니다.

Example) EmployeeDemo (1/3)

```java
class Employee {
    void work() {
        System.out.println("Employee is working");
    }
}

class Manager extends Employee {
    @Override
    void work() {
        System.out.println("Manager is managing");
    }

    void plan() {
        System.out.println("Manager is planning");
    }
}
```


페이지 11

Upcasting과 Downcasting의 좀 더 구체적인 이용 사례를 보도록 하겠습니다.
class Employee 의 work() method는
“Employee is working” 이라는 message를 print합니다.
Employee의 child인 Manager class의 work() method는
“Manager is managing” 을 print합니다.
Manager class에 새로 add된 plan() method는
“Manager is planning” 을 print합니다.

Example) EmployeeDemo (2/3)

```java
class Engineer extends Employee {
    @Override
    void work() {
        System.out.println("Engineer is engineering");
    }
    void design() {
        System.out.println("Engineer is designing");
    }
}

public class EmployeeDemo {
    public static void main(String[] args) {
        Employee employee1 = new Manager(); // Upcasting
        employee1.work(); // Output: Manager is managing

        if (employee1 instanceof Manager) { // original class = Manager?
            Manager manager = (Manager) employee1; // Downcasting
            manager.plan(); // Output: Manager is planning
        }
```


페이지 12

역시 Employee의 child class인 Engineer class에서는
work() method가 “Engineer is engineering” 을 print합니다.
Engineer에 새로 추가된 design() method는
“Engineer is designing” 을 print합니다.
EmployeeDemo의 main method에서
Employee object인 employee1 에 Manager object가 assign 되었습니다.
Manager가 Employee의 child이므로
이 assignment는 Upcasting입니다.
즉, explicit type conversion이 필요하지 않습니다.
이제 employee1.work() 을 call하면
Manager에서 override된 work() method가 실행되어
“Manager is managing” 이라는 message가 print됩니다.
그 아래 if (employee1 instanceof Manager) 라는 조건이 test되는데
이것은 employee1에 현재 assign되어 있는 object의
original class가 Manager인지를 test해 보는 condition입니다.
만일 그렇다고 하면 이 instanceof operator는 true를 return합니다.
우리의 example의 경우, employee1에 현재 assign되어 있는 object는
original class가 Manger이기 때문에 이 조건이 true가 됩니다.
employee1의 original class가 Manger이기 때문에
employee1은 Manger object인 manager로
downcasting이 가능합니다.
물론 explicit type conversion이 필요합니다.
그리고나서 manger.plan() 을 call하면
“Manger is planning” 이라는 message가 print되게 됩니다.

Example) EmployeeDemo (3/3)

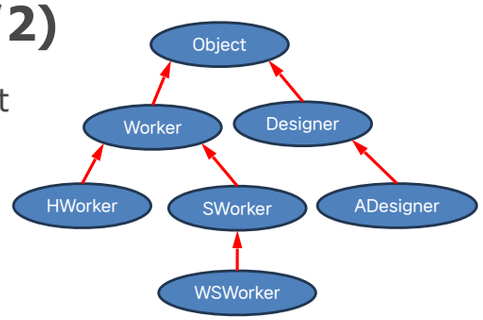
```
        Employee employee2 = new Engineer(); // Upcasting
        employee2.work(); // Output: Engineer is engineering


        if (employee2 instanceof Engineer) { // original class = Engineer?
            Engineer engineer = (Engineer) employee2; // Downcasting
            engineer.design(); // Output: Engineer is designing
        }
    }
}
```


페이지 13

같은 원리로 Employee object인 employee2에
Engineer class object를 assign하였습니다.
이 assignment는 upcasting으로
explicit type conversion이 필요하지 않습니다.
employee2.work()을 call하면
original class인 Engineer에서 override되어 정의된
work()이 실행됩니다.
따라서 "Engineer is engineering" 이라는 message가
프린트 됩니다.
그 아래 if 문에서는 employee2의 original class가
Engineer인지를 test하고 있습니다.
우리의 example에서는 이 test가 true이기 때문에
employee2는 explicit type conversion을 통한
downcasting으로 engineer에 assign됩니다.

Instanceof and getClass() (1/2)

- Both can be used to check the class of an object
- instanceof
  - anObject instanceof SomeClass
    - returns true if anObject is of type SomeClass (or SomeClass's descendant)
  - ex) (other instanceof Worker) is true
    - when other is Worker, HWorker, SWorker, WSWorker

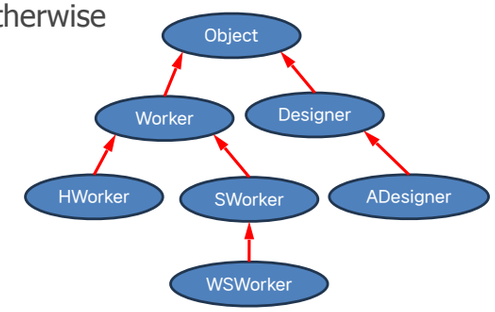


페이지 14

Instanceof operator와 getClass() method는
object의 original class가 무엇인지를 확인할 수 있는 방법입니다.
anObject instanceof SomeClass 는
anObject의 original class가 SomeClass이거나
SomeClass의 descendant일 경우 true를 return합니다.
예를 들면 그림의 class hirerarchy에서
other instanceof Worker는
other가 Worker, HWorker, SWorker, WSWorker일 경우 true가 됩니다.

Instanceof and getClass() (2/2)

- getClass()
  - true when both classes are the same, false otherwise
  - ex) Designer d = new ADesigner();
    ADesigner ad = new ADesigner();
    if (d.getClass() == ad.getClass()) { }

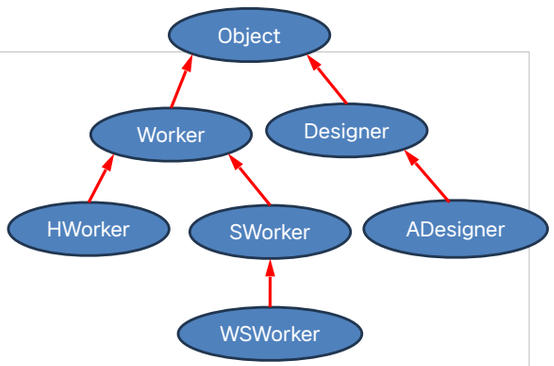


페이지 15

한편 getClass()는 toString() 처럼 어떤 class에나 존재하는 method로서
두 object a와 b에 대해 (a.getClass() == b.getClass()) 는
a와 b의 original class가 같을 경우에만 true가 됩니다.
예를 들면 그림의 class hierarchy에서
Designer d의 original class가 ADesigner이고
ad의 original class도 ADesigner이므로
d.getClass() == ad.getClass() 는 true가 됩니다.

Example (1/2)

```
class Worker { }
class Designer { }
class HWorker extends Worker { }
class SWorker extends Worker { }
class WSWorker extends SWorker { }
class ADesigner extends Designer { }


public class TestGetClass {
    public static void main(String[] args) {
        Worker worker = new Worker();
        HWorker hworker = new HWorker();
        SWorker sworker1 = new SWorker();
        SWorker sworker2 = new SWorker();
        WSWorker wsworker = new WSWorker();
        Designer designer = new Designer();
        ADesigner adesigner = new ADesigner();
```



페이지 16

이제 instanceof와 getClass() 에 대해
좀 더 많은 example을 보도록 하겠습니다.
그림의 class hierarchy를 고려하도록 하겠습니다.

Example (2/2)

```
        System.out.println(worker.getClass() == hworker.getClass()); // false
        System.out.println(sworker1.getClass() == sworker2.getClass()); // true
        System.out.println(sworker1.getClass() == wsworker.getClass()); // false
        //The following is ERROR! No ancestor-descendant relationship
        //System.out.println(hworker.getClass() == designer.getClass());


        System.out.println(wsworker instanceof Worker); // true
        System.out.println(sworker1 instanceof WSWorker); // false
        System.out.println(designer instanceof Object); // true
        // The following is ERROR: No ancestor-descendant relationship
        // System.out.println(worker instanceof Designer);
    }
}
```


페이지 17

worker의 original class는 Worker
hworker의 original class는 HWorker 이므로
worker.getClass() == hworker.getClass() 는 false가 됩니다.
그러나 sworker1과 sworker2의 original class는 둘 다 SWorker이므로
sowker1.getClass() == sworker2.getClass() 는 true가 됩니다.
wsworker의 original class는 WSWorker이므로
sworker1.getClass() == wsworker.getClass() 는 false 입니다.
한편, hworker와 designer의 original class들은 각각
HWorker와 Designer이고
이들은 class hierarchy에서 ancestor-descendant의 관계가 아니므로
hworker.getClass() == designer.getClass() 의 비교가 이루어질수 없으며
compile error를 일으키게 됩니다.
instanceof operator를 고려해 보면
wsworker의 original class는 WSWorker로서
Worker의 descendant이므로
wsworker instanceof Worker는 true이며
sworker1의 original class는 SWorker인데
WSWorker는 SWorker의 child이므로
sworker1 instanceof WSWorker는 false가 됩니다.
또 designer의 original class는 Designer인데
Designer는 Object의 descendant이므로
designer instanceof Object는 true가 됩니다.
한편 worker instanceof Designer는 compile error를 내는데
worker의 original class인 Worker와
Designer class간에는 ancestor-descendant의 관계가 없기 때문입니다.

Summary: Automatic Type Conversion (1/2)

- Automatic Type Conversion이 가능한 경우
  - Numbers 에 속하는 type들 중에 범위가 작은 쪽에서 큰 쪽으로
    - ex) int x = 5;
      double y = x;
  - Descendant class object를 Ancestor class object로 (Upcasting)
    - ex) class Animal { }
      class Cat extends Animal { }
      Cat c = new Cat();
      Animal a = c;

페이지 18

이제 Automatic Type Conversion에 대해 요약을 해 보도록 하겠습니다.
Automatic Type Conversion이 가능한 경우는
먼저, Numbers에 속하는 primitive type들 중에
범위가 작은 쪽에서 큰쪽으로는 auto type conversion이 가능합니다.
예를 들면 int에서 double로 assignment는
explicit type conversion이 없어도 가능합니다.
두번째로는 descendant class object를 ancestor class object로
direct assignment가 가능하며, 이를 upcasting이라 합니다.

Summary: Automatic Type Conversion (2/2)

- Automatic Type Conversion이 불가능한 경우
  - Numbers type들 중 큰 범위의 type을 작은 범위로 assign
    - ex) int x; double y = 123.51;
      x = y; // x = (int)y; 는 가능, explicit type conversion
  - Ancestor class type을 Descendant class type으로 (Down-casting)
    - ex) class Animal { } class Cat extends Animal { }
      Animal a = new Animal(); Cat c = a;
      Animal b = new Cat(); Cat c = (Cat)b; // explicit down-casting
  - 아무 관련 없는 두 type들 간의 conversion
    - ex) boolean b = false; int x; x = b; // x = (int)b; 도 불가

페이지 19

한편 Auto Type Conversion이 불가능한 경우를 보면
Numbers에 속하는 primitive type들 중
큰 범위의 type을 작은 범위의 type으로 assign하려면
explicit type conversion을 해야 합니다.
예를 들면 int x에 double y를 assign하는 경우라면
int x = (int) y; 와 같은 형태가 되어야 합니다.
두번째로 ancestor class type을 descendant class type으로
assign하는 downcasting의 경우 입니다.
세번째로는 아무 관련 없는 두 type들간의 conversion은 불가능한데
예를 들면 int x에 boolean b를 assign하는 경우는
explicit type conversion x = (int)b; 를 하더라도
compile error가 나게 됩니다.