1. 상속(inheritance)

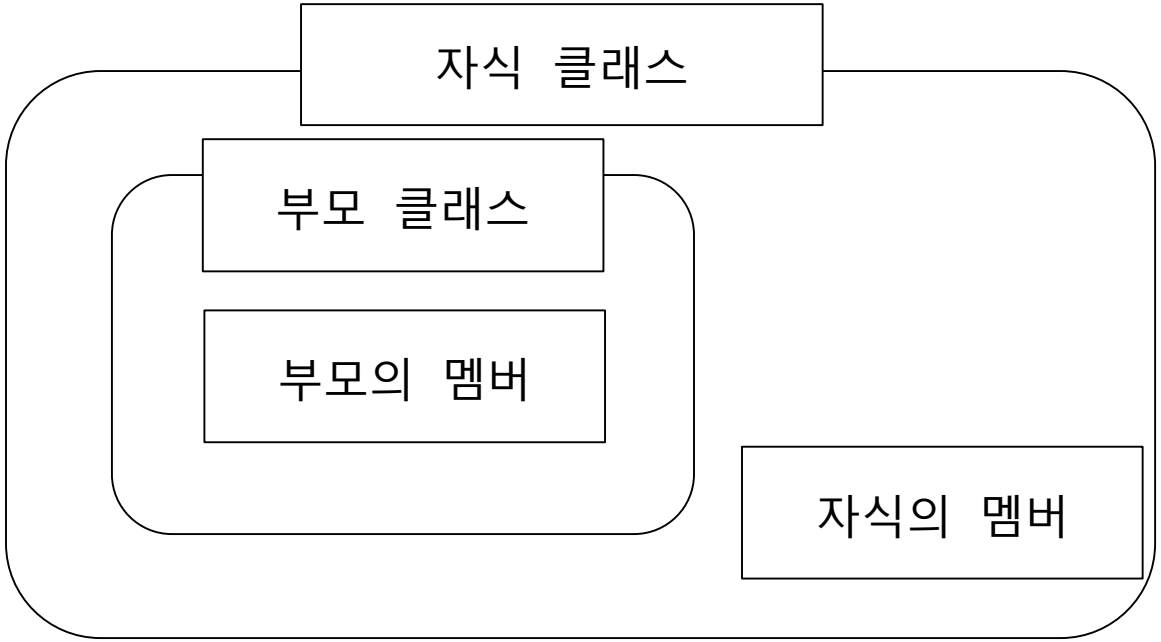
1.1 상속(inheritance)

상속

- 기존의 클래스에서 기능을 추가하거나 재정의 하는 것
- 상속을 통해 두 클래스는 조상과 자손 관계 형성
- 자손 클래스는 조상의 속성과 기능을 상속받음

1.2 클래스간의 관계 – 상속관계(inheritance)

- 공통된 부분을 조상 클래스로 작성하여, 조상 클래스만 변경하면 자식 클래스들을 따로 변경할 필요가 없다
- 자식을 변경하는 것은 조상에 영향을 주지 않음



1.2 클래스간의 관계 – 포함관계(composite)

포함관계

- 다른 클래스의 객체를 멤버 변수로 사용하는 것

```
class Car {
    int id;
    int speed;
    Wheel[] wheel;  // Wheel 클래스에서 객체를 생성하여 포함관계 형성
    void accelerate() {
        speed++;
    }
}
```

1.3 클래스간의 관계결정하기 – 상속 vs. 포함

- 상황에 맞는 관계를 선택하여 적절히 사용해야 함
- 상속은 is – a 관계, 포함은 has – a 관계라고도 함

상속

자동차는 바퀴를 가지고 있다.(has – a 관계)

포함

대형차(LargeCar)는 차(Car) 이다.(is – a 관계)

1.4 단일상속(single inheritance)

- 하나의 부모에게서만 상속 받을 수 있다.
- 여러 개의 클래스의 코드를 재사용 하고 싶다면 포함관계를 활용해야 한다.

1.5 Object클래스 – 모든 클래스의 최고조상

- 조상이 없는 클래스는 Object 클래스를 상속 받고 있는 것이 생략되어 있다.
- 모든 클래스는 Object 클래스의 멤버를 상속받고 있어서 사용할 수 있다.
- 대표적인 예시: toString(), equals()

2. 오버라이딩(overriding)

2.1 오버라이딩

- 조상 클래스에서 상속 받은 메서드를 새롭게 재정의 하는 것

```
class Car {
    int id;
    int speed;
    Wheel[] wheel;  // Wheel 클래스에서 객체를 생성하여 포함관계 형성
    void accelerate() {
        speed++;
    }
}

class LargeCar extends Car {
    void accelerate() {     // 오버라이딩
        speed += 2;
    }
}
```

2.2 오버라이딩의 조건

1. 메서드의 선언부는 기존 메서드와 같아야 한다. 즉, 이름, 매개변수, 리턴 타입이 달라지면 안된다.
2. 부모 클래스의 메서드보다 좁은 범위의 접근 제어자로 변경할 수 없다.(자식이 더 넓거나 같아야 됨)
3. 부모 클래스의 메서드보다 더 큰 범위의 예외를 선언 할 수 없다.(자식은 더 적은 예외를 선언해야 함)

2.3 오버로딩 vs. 오버라이딩

오버로딩

- 새로운 메서드를 정의

오버라이딩

- 부모 메서드의 동작을 변경

```
class Car {
    int id;
    int speed;
    Wheel[] wheel;  // Wheel 클래스에서 객체를 생성하여 포함관계 형성
    void accelerate() {
        speed++;
    }
}

class LargeCar extends Car {
    void accelerate() {     // 오버라이딩
        speed += 2;
    }
    void accelerate(int x) {    // 오버로딩
        speed++;
    }
}
```

2.4 super – 참조변수

this

- 인스턴스 자신을 의미하는 참조 변수.

super

- this와 같은 역할을 함. 부모와 자식의 이름이 같을때 구분하기 위해 사용(부모)

2.5 super() – 조상의 생성자

- this() 메서드가 같은 클래스의 다른 생성자를 호출하는 것처럼 super() 메서드는 부모 클래스의 생성자를 호출하는데 사용 됨
- 조상의 멤버를 상속 받기 때문에 먼저 조상이 존재 해야 하므로 자손의 생성자 첫 줄에 실행 되어야 함.
- 조상의 멤버를 초기화 하는 목적일 때 super()를 이용하면 코드 중복을 줄일 수 있음
- 따로 명시하지 않으면 모든 생성자의 첫 줄에 super()를 자동으로 호출함

3. package와 import

3.1 패키지(package)

- 클래스와 인터페이스의 집합
- 디렉토리 형태로 되어 있음
- 패키지는 다른 패키지를 포함할 수 있음. 계층 구조는 (.)으로 구분

Ex) java.lang.String – String 클래스는 java.lang 패키지에 들어 있음

3.2 import문

- 다른 파일에 속한 클래스를 사용하기 위해 사용
- import한 클래스는 패키지를 생략할 수 있음
- java.lang 패키지에 포함된 클래스들은 import 없이 사용 가능
- *은 해당 패키지의 모든 클래스를 의미

문법

import 패키지경로.클래스이름;

import 패키지경로.*;

4. 제어자(modifiers)

4.1 제어자(modifier)란?

- 클래스와 클래스 멤버의 선언 시 사용
- 부가적인 의미를 부여함
- 접근 제어자와 기타 제어자로 나뉨
- 하나의 대상에 여러 제어자를 조합하여 사용 하는 것도 가능(접근 제어자는 하나만 사용 가능)

4.2 static

- 멤버변수, 메서드 초기화 블록에 사용
- static이 붙으면 인스턴스를 생성하지 않고 사용 가능

4.3 final – 변경 불가

- 클래스, 메서드, 멤버변수, 지역변수에 사용 가능
- 변수에 사용 시 상수가 됨(변경 불가)
- 메서드에 사용 시 오버라이딩 불가
- 클래스에 사용 시 상속 불가

4.4 생성자를 이용한 final 멤버변수 초기화

- final이 붙은 상수는 선언과 동시에 초기화 해야 됨
- 그러나 생성자를 이용하면 선언 후 인스턴스 생성시 초기화 하는 것이 가능

```
class Wheel {
    final int SIZE;

    Wheel(int size) {
        SIZE = size;
    }
}
```

4.5 abstract – 추상의, 미완성의

- 클래스와 메서드에 사용
- 클래스에 사용되면 클래스 안에 추상 메서드가 포함 되어 있음을 의미
- 메서드에 사용되면 구현되지 않은 추상 메서드임을 의미

4.6 접근 제어자(access modifier)

- 클래스, 멤버변수, 메서드, 생성자에서 사용 가능
- 접근제어자를 사용하여 사용자가 굳이 알 필요 없는 정보를 숨길 수 있음
- 이를 통해 사용자는 최소한의 정보로 프로그램을 사용 가능

접근 범위

public > protected > default > private

	같은 클래스	같은 패키지	자손 클래스	그 외
public	O	O	O	O
protected	O	O	O	X
default	O	O	X	X
private	O	X	X	X

5. 다형성(polymorphism)

5.1 다형성(polymorphism)

- 여러 가지 타입을 가질 수 있는 것
- 부모 클래스 타입의 참조 변수로 자식 타입의 인스턴스를 참조 할 수 있음

```
class LargeCar extends Car {
    void accelerate() {      // 오버라이딩
        speed += 2;
    }
    void accelerate(int x) {     // 오버로딩
        speed++;
    }
}

public class Main {
    public static void main(String[] args) {
        Car car;     // 객체를 다룰 참조변수 선언. 아직은 null
        car = new Car();     // 객체 생성. 생성된 객체의 주소를 할당

        Car c = new LargeCar(); // 자식의 타입을 할당 받음
```

5.2 참조변수의 형변환

- 상속 관계인 경우 형 변환 가능
- 자손 -> 조상 : 생략 가능(up-casting)
- 조상 -> 자식 : 생략 불가능(down-casting)

```
class LargeCar extends Car {
    void accelerate() {      // 오버라이딩
        speed += 2;
    }
    void accelerate(int x) {     // 오버로딩
        speed++;
    }
}

public class Main {
    public static void main(String[] args) {
        Car car;     // 객체를 다룰 참조변수 선언. 아직은 null
        car = new Car();     // 객체 생성. 생성된 객체의 주소를 할당

        LargeCar largeCar = new LargeCar();
        Car c = largeCar;    // 자식 -> 조상. 생략 가능
        largeCar = (LargeCar)car;    // 조상 -> 자식. 생략 불가능
```

5.3 instanceof연산자

- 참조변수가 실제 참조하고 있는 인스턴스의 타입인지 확인하는 연산자
- 연산 결과는 true와 false
- true가 나오면 형 변환 가능

문법

참조변수 instanceof 클래스명

```
public static void main(String[] args) {
    Car car;     // 객체를 다룰 참조변수 선언. 아직은 null
    car = new Car();     // 객체 생성. 생성된 객체의 주소를 할당

    LargeCar largeCar = new LargeCar();

    if(car instanceof Car) {
        System.out.println("Car 인스턴스");
    }
    if(car instanceof LargeCar) {
        System.out.println("LargeCar 인스턴스");
    }
```

5.4 참조변수와 인스턴스변수의 연결

- 멤버 변수의 이름이 같은 경우 참조 변수의 타입에 맞는 값이 출력
- 메서드의 이름이 같은 경우에는 참조 변수의 타입이 아닌 실제 인스턴스(new를 이용하여 할당한 인스턴스)의 타입에 맞는 메서드가 호출

```
class Car {
    int id = 5;
    int speed;
    void accelerate() {
        speed++;
        System.out.println("Car");
    }
}

class LargeCar extends Car {
    int id = 10;
    void accelerate() {      // 오버라이딩
        speed += 2;
        System.out.println("LargeCar");
    }
}
```

```
LargeCar largeCar = new LargeCar();

System.out.println(((Car)largeCar).id);
System.out.println(largeCar.id);

largeCar.accelerate();
((Car)largeCar).accelerate();
```

```
5
10
LargeCar
LargeCar
```

6. 추상클래스(abstract class)

6.1 추상클래스(abstract class)

- 추상 메서드를 포함하고 있는 클래스
- 반드시 자식 클래스에서 추상 메서드를 오버라이딩 해야만 사용 가능
- 추상 메서드를 가지고 있다는 것 외에는 일반 클래스랑 차이는 없음
- 공통된 기능과 분류 되는 기능이 다 있는 경우 유용함

6.2 추상메서드(abstract method)란?

- 선언부만 있고 구현은 되지 않은 메서드
- 반드시 자식 클래스에서 오버라이딩 해야만 사용 가능
- 자손마다 다르게 구현되는 경우 구현

6.3 추상클래스의 작성

- 공통으로 사용 되는 기능은 작성하고 자손에 따라 다르게 동작할 메서드는 추상 메서드로 남겨서 오버라이딩 하여 작성

```
abstract class Animal {
    abstract void cry();

    void run() {
        System.out.println("run!!!!!!!!!!");
    }
}

class Cat extends Animal {
    void cry() {
        System.out.println("야옹");
    }
}

class Dog extends Animal {
    void cry() {
        System.out.println("멍멍");
    }
}
```

7. 인터페이스(interface)

7.1 인터페이스(interface)란?

- 추상클래스와 비슷하지만 추상 메서드와 상수만을 멤버로 가짐
- 인스턴스를 생성할 수 없음
- 클래스를 작성하는데 기본 틀을 제공함
- 다른 클래스 사이의 중간 매개 역할을 담당

7.2 인터페이스의 작성

- ‘class’대신 ‘interface’를 사용한다는 것 외에는 클래스 작성과 동일하다.

```
interface 인터페이스 이름{
  public static final 타입 상수이름 = 값;
  public abstract 메소드이름(매개변수목록);
}
```

- 모든 메서드는 public abstract, 변수는 public static final 이어야 하며 생략 가능

7.3 인터페이스의 상속

- implements를 이용하여 상속 가능(다중상속도 허용)

```
interface Animal {
    public abstract void cry();
}

interface Pet {
    public abstract void play();
}

class Cat implements Animal, Pet {

    public void cry() {

        System.out.println("야옹");

    }

    public void play() {
        System.out.println("놀기");

    }

}
```

7.4 인터페이스의 장점

일관된 표준화 가능

- 프로젝트에서 사용될 기본적인 틀을 인터페이스로 작성하여 개발자들에게 가이드라인을 제공함

동시에 작업 가능

- 인터페이스가 작성 되어 있으면 여러 명의 개발자들이 각각의 기능을 개발하는 것이 가능해 짐

클래스와 클래스 간의 관계를 맺어줌

- 상속 관계에 있지 않은 관련 없는 클래스들을 하나의 인터페이스를 제공하여 관계를 맺어줄 수 있음