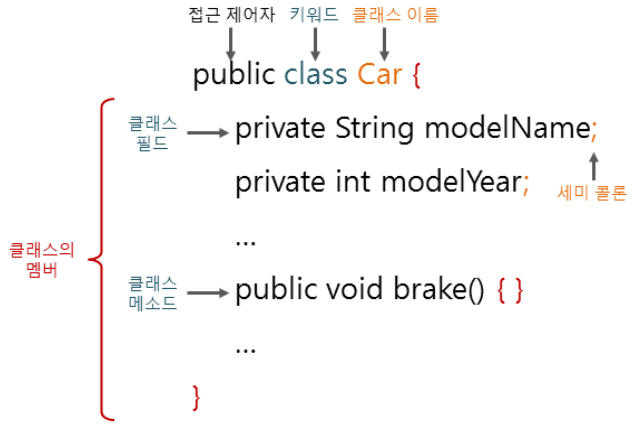
클래스

클래스란**?**
객체 지향 프로그래밍의 추상화(abstraction)라는 개념을 직접 구현한 것

추상화란**?**
복잡한 자료, 모듈, 시스템 등으로부터 핵심적인 개념 도는 기능을 간추려 내는 것
ex. 하드디스크-파일, 네트워크-포트, 메모리-주소, CPU-프로세스 등

클래스 정의



접근 제어자(Access Modifier)

private : 해당 클래스 내에서만 접근이 가능
default : 접근 제어자를 별도로 설정하지 않았을 때, 동일한 패키지 내에서만 접근 가능
protected : 동일 패키지의 클래스 또는 해당 클래스를 상속 받은 다른 패키지의 클래스에서만 접근 가능
public : 어떤 클래스에스도 접근이 가능

```
package house;  // 패키지가 동일하다.

public class JaeminBaek {
    private String job = "developer";
    String lastname = "baek";
    protected String firstname = "jaemin";
    public int age = 26;
}

public class JaeminBaekChild extends JaeminBaek{     // JaeminBaek을 상속
받음
    setFirstName(String name){ this.firstname = name; }
```

```
    getFirstName(){ return firstname; }
```

```
package house;  // 패키지가 동일하다.

public class HouseBaek {
    public static void main(String[] args) {
        JaeminBaek baek = new JaeminBaek();
        System.out.println(baek.lastname);
    }
}

package ground;  // 패키지가 다르다.
import house.BaekJaemin;

public class GroundBaek {
    public static void main(String[] args) {
        JaeminBaek baek = new JaeminBaek();
        System.out.println(baek.age);
    }
}
```

객체 생성

인스턴스의 생성

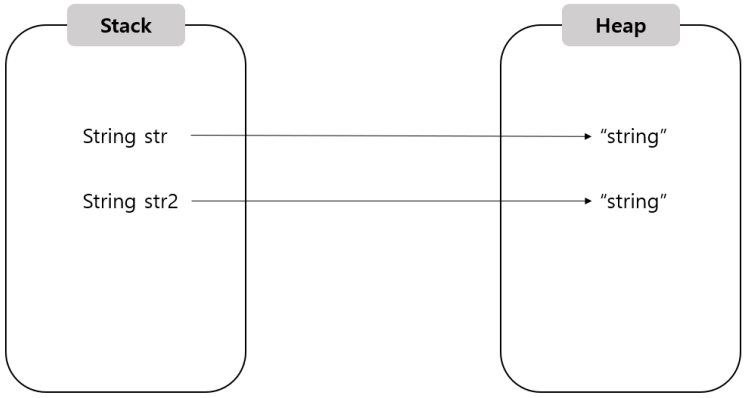
```
//클래스이름 객체참조변수이름;
Car myCar;

//객체참조변수이름 = new 클래스이름();
myCar = new Car();

//클래스이름 객체참조변수이름 = new 클래스이름();
Car myCar = new Car();
```

new 키워드를 이용하여 객체 인스턴스 생성

new 란?

heap 영역에 저장할 공간을 할당해서 참조 값을 객체에게 반환
new 연산자는 객체를 **heap**이라는 메모리 영역에 메모리 공간을 할당해주고 메모리 주소를 반환한 후 생성자를 실행
리터럴과는 달리 **new** 연산자로 생성된 객체는 똑같은 변수 값을 가진 객체가 있어도 서로 다른 메모리를 할당하여 서로 다른 객체로 분류

```
public class Main {

	public static void main(String[] args) {
		String str = new String("string");
		String str2 = new String("string");

		System.out.println(System.identityHashCode(str));
		//result:2008362258
		System.out.println(System.identityHashCode(str2));
		//result:760563749
		//메모리주소는 컴퓨터마다 다르기때문에 실행결과가 다르게 나올수도 있다.
	}

}

public class Main {

	public static void main(String[] args) {
		String str = "string";
		String str2 = "string";

		System.out.println(System.identityHashCode(str));
		//result:2008362258
		System.out.println(System.identityHashCode(str2));
		//result:2008362258
	}

}
```



클래스의 new 키워드를 이용한 메모리 참조로 만들어진 변수와 달리 리터럴 변수의 경우에는 같은 주소 값을 갖는다.
즉, new 연산자는 객체를 생성할 때 사용하고 heap이라는 메모리 영역에 각각의 새로운 메모리 공간을 할당해주는 역할을 한다.

메소드 정의

메소드란?

어떠한 문제를 처리하기 위한 방법을 소스 코드로 묶어놓고 필요(호출)에 따라 동작하는 기능
다른 언어에서는 함수(function)으로 부른다.

```
접근제어자 반환타입 메소드이름(매개변수목록) { // 선언부
    // 구현부
}
```

접근 제어자 : 해당 메소드에 접근할 수 있는 범위 명시
반환 타입(return type) : 메소드가 모든 작업을 마치고 반환하는 데이터의 타입을 명시
메소드 이름 : 메소드를 호출하기 위한 이름을 명시
매개변수 목록(parameters) : 메소드 호출 시에 전달되는 인수의 값(argument)을 저장할 변수 명시
구현부 : 메소드의 고유 기능을 수행하는 명령문의 집합
(+) 메소드 시그니처(method signature)?
  메소드의 선언부에 명시되는 매개변수의 리스트
  만약 두 메소드가 매개변수의 개수, 타입 그리고 순서가 모두 같다면, 이 두 메소드의 시그니처는 같다고
  할 수 있다.

accelerate 메소드 정의

```
class Car {
    private int currentSpeed;
    private int accelerationTime;
    ...
```

```
    public void accelerate(int speed, int second) { // 선언부
        // 구현부
        System.out.println(second + "초간 속도를 시속 " + speed + "(으)로
가속함!!");
    }
    ...
}

Car myCar = new Car();
myCar.accelerate(60, 3);
```

접근 제어자 - public : 해당 객체를 사용하는 프로그램 어디에서나 직접 접근 가능
반환 타입 - void : 어떠한 값도 반환하지 않음
메소드 이름 - accelerate
매개변수 - int speed, int second
호출 - myCar.accelerate(60, 3) : 멤버 참조 연산자(.)를 사용해 호출

생성자

생성자란

클래스를 가지고 객체를 생성하면 해당 객체는 메모리에 즉시 생성
하지만 이렇게 생성된 객체는 모든 인스턴스 변수가 아직 초기화되지 않은 상태이다.
private 변수를 갖는 클래스에서 사용자가 원하는 값으로 인스턴스 변수를 초기화하려면,
초기화를 위한 public 메소드가 필요하다.
이러한 초기화만을 위한 메소드는 객체가 생성된 후부터 사용되기 전까지 반드시 인스턴스
변수의 초기화를 위해 호출되어야 한다.
이때, 해당 메소드의 이름은 클래스의 이름과 같아야 한다.

생성자의 특징

1. 생성자는 반환값이 없지만, 반환 타입을 void형으로 선언하지 않는다.
2. 생성자는 초기화를 위한 데이터를 인수로 전달받을 수 있다.
3. 개체를 초기화하는 방법이 여러 개 존재할 경우, 하나의 클래스가 여러 개의
   생성자를 가질 수 있다.
   즉, 생성자도 하나의 메소드이므로, 메소드 오버로딩이 가능하다.

기본 생성자(default constructor)

자바의 모든 클래스에는 하나 이상의 생성자가 정의되어 있어야 한다.
하지만 특별히 생성자를 정의하지 않고도 인스턴스를 생성할 수 있는데, 이는 자바
컴파일러가 기본 생성자(default constructor)를 제공해주기 때문이다.
기본 생성자는 매개변수를 하나도 가지지 않으며, 아무런 명령어도 포함하고 있지 않다.
하지만 생성자를 하나라도 정의했다면, 기본 생성자는 자동으로 추가되지 않는다.

this

this는 인스턴스 자기 자신을 의미한다.
매개변수의 이름을 멤버변수와 다른 이름으로 설정한다면 this 키워드를 쓰지 않아도 된다.
하지만 가독성을 위해 매개변수명과 멤버 변수를 동일하게 쓰는 것을 권장하고 있다.
또한, 생성자에서 다른 생성자를 호출할 경우 사용한다.
하지만 이때, this 키워드 위에 다른 구문을 작성할 경우 에러가 발생한다.
마지막으로 인스턴스 자신의 주소를 반환할 때 사용한다.

```
public class Student {

	private int studentID; //학번
	private String studentName; //학생 이름

	public Student() {
				//studentId = 100;	에러 발생
		this(150, "피망");
	}

	public Student(int studentID, String studentName) {
		this.studentID = studentID;
		this.studentName = studentName;
	}

	public Student getSelf() {
		return this;	// 주소 반환
	}

	public void setStudentID(int studentID) {
		this.studentID = studentID;
	}

	public void setStudentName(String studentName) {
		this.studentName = studentName;
	}
}
```

상속

상속이란

상속

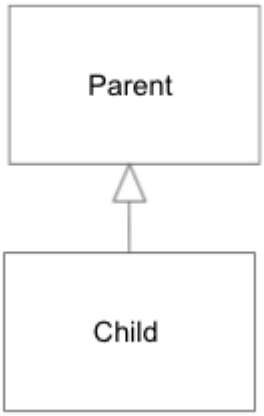
- 부모가 자식에게 물려주는 행위
- 객체 지향 프로그램에서도 부모(상위) 클래스의 멤버를 자식(하위)클래스에 물려주어 자식 클래스가 갖고 있는 것처럼 사용할 수 있다.

상속의 장점

- 코드 중복 감소(이전에 개발된 클래스를 재사용하여 새로운 클래스를 생성)
- 유지 보수 기간 감소(부모 클래스의 수정으로 모든 자식 클래스들의 수정 효과를 가져옴)

상속의 방법

extends 키워드를 이용하여 상속 받는다.



자바 상속의 제한

- 다중 상속 불가능 : extends 뒤에 클래스(부모)는 하나만이 위치해야 한다.
- 상속을 해도 부모 클래스의 모든 필드와 메소드들을 물려받는 것은 아니다. 아래 같은 경우 상속이 제한된다.
    - 부모 클래스에서 private 접근 제한을 갖는 필드와 메소드
    - 부모 클래스와 자식 클래스가 다른 패키지에 존재하며, default 접근 제한을 갖는 필드와 메소드

super 키워드

super 키워드란?

super 키워드는 부모 클래스로부터 상속 받은 필드나 메소드를 자식 클래스에서 참조하는데 사용하는 참조 변수이다.
부모 클래스의 멤버와 자식 클래스의 멤버 이름이 같을 경우 super 키워드를 사용하여 구별할 수 있다.
this와 마찬가지로 super 참조 변수를 사용할 수 있는 대상도 인스턴스 메소드 뿐이며, 클래스 메소드에서는 사용 할 수 없다.

```
class Parent { int a = 10; }

class Child extends Parent {
    void display() {
        System.out.println(a);
        System.out.println(this.a);
        System.out.println(super.a);
    }
}

public class Inheritance02 {
    public static void main(String[] args) {
        Child ch = new Child();
        ch.display();
    }
}

// 출력 결과
// 10
// 10
// 10

class Parent {
    int a = 10;
}

class Child extends Parent {
    int a = 20;

    void display() {
        System.out.println(a);
        System.out.println(this.a);
        System.out.println(super.a);
    }
}
```

```java
public class Inheritance03 {
    public static void main(String[] args) {
        Child ch = new Child();
        ch.display();
    }
}

// 출력 결과
// 20
// 20
// 10
```

super 메소드

this() 메소드와 달리 super() 메소드는 부모 클래스의 생성자를 호출할 때 사용된다.
자식 클래스의 인스턴스를 생성하면, 해당 인스턴스에는 자식 클래스의 고유 멤버뿐만 아니라 부모 클래스의 모든 멤버까지도 포함되어 있다.
따라서 부모 클래스의 멤버를 초기화하기 위해서는 자식 클래스의 생성자에서 부모 클래스의 생성자를 호출해야 한다.
이러한 부모 클래스의 생성자 호출은 모든 클래스의 부모 클래스인 Object 클래스의 생성자까지 계속 거슬러 올라가며 수행된다.
따라서 자바 컴파일러는 부모 클래스의 생성자를 명시적으로 호출하지 않는 모든 자식 클래스의 생성자 첫줄에 자동으로 **super();**를 추가하여 부모 클래스의 멤버를 초기화할 수 있도록 한다.
자바 컴파일러는 생성자가 하나도 정의되어 있지 않아야 자동으로 기본 생성자를 추가해주므로 매개변수를 가지는 생성자를 부모 클래스에 하나라도 선언했다면, 기본 생성자가 존재하지 않아 **super();**를 사용할 수 없다.

메소드 오버라이딩

메소드 오버라이딩이란?

상속 관계에 있는 부모 클래스에서 이미 정의된 메소드를 자식 클래스에서 같은 시그니쳐를 갖는 메소드로 재정의하는 것

오버라이딩의 조건

1. 메소드의 동작만을 재정의하는 것이므로, 메소드의 선언부는 기존 메소드와 동일해야한다. 하지만 메소드의 반환 타입은 부모 클래스의 반환 타입으로 타입 변환할 수 있는 타입이라면 변경할 수 있다.
2. 부모 클래스에서 정해진 메소드의 접근 제어자를 바꿀 수 없다.

```
class Parent {
    void display() { System.out.println("부모 클래스의 display()
메소드입니다."); }
}
class Child extends Parent {
    void display() { System.out.println("자식 클래스의 display()
메소드입니다."); }
}

public class Inheritance05 {
    public static void main(String[] args) {
        Parent pa = new Parent();
        pa.display();
        Child ch = new Child();
        ch.display();
        Parent pc = new Child();
        pc.display(); // Child cp = new Parent();
    }
}

// 출력 결과
// 부모 클래스의 display() 메소드입니다.
// 자식 클래스의 display() 메소드입니다.
// 자식 클래스의 display() 메소드입니다.
```

오버로딩과 오버라이딩

오버로딩(overloading) : 새로운 메소드를 정의 - 정적 바인딩 / 컴파일 타임 바인딩 / Early 바인딩
오버라이딩(overriding) : 상속받은 기존의 메소드를 재정의 - 동적 바인딩 / 런타임 바인딩 / 늦은 바인딩

```
class Parent {
    void display() { System.out.println("부모 클래스의 display() 메소드입니다."); }
}
class Child extends Parent {
    // 오버라이딩된 display() 메소드
    void display() { System.out.println("자식 클래스의 display() 메소드입니다."); }
    void display(String str) { System.out.println(str); }
    // 오버로딩된 display() 메소드
}

public class Inheritance06 {
    public static void main(String[] args) {
        Child ch = new Child();
        ch.display();
        ch.display("오버로딩된 display() 메소드입니다.");
    }
}

// 출력 결과
// 자식 클래스의 display() 메소드입니다.
// 오버로딩된 display() 메소드입니다.
```

다이나믹 메소드 디스패치(Dynamic Method Dispatch)

메소드 디스패치(Method Dispatch)란

어떤 메소드를 호출할 지 결정하여 실제로 실행시키는 과정
자바는 런타임 시 객체를 생성하고, 컴파일 시에는 생성할 객체 타입에 대한 정보만 보유한다.
이 과정은 static과 dynamic으로 나눈다.
여기서 dynamic method dispatch란 컴파일러가 어떤 메소드를 호출하는 지 모르는 경우이다. 이는 런타임 시점에서 결정된다.
즉, 인터페이스나 추상 클래스에 정의된 추상 메소드를 호출하는 경우로 인터페이스 또는 추상 클래스로 선언하고 구현/상속 받은 하위 클래스의 인스턴스를 생성,
컴파일러가 알고 있는 타임에 대한 정보를 토대로 런타임 시 해당 타입의 객체를 생성하고 메소드를 호출한다.

```
런타임 전에는 객체 생성이 되지 않기 때문에 Car car = new A();를 해도
컴파일러는 A가 생성됨을 알 수 없다. 그렇기에 Car가 정의한 print()메소드만
접근이 가능하다.

추상 클래스
추상 클래스란?
하나 이상의 추상 메소드(abstract method)를 포함하는 클래스이다.
추상 메소드는 선언만 있고 구현부는 없는 함수이며 선언부에 'abstract' 키워드를
사용하여 선언한다. 추상 메소드가 포함되었다면 클래스 또한 추상 클래스이므로
클래스명 앞에도 'abstract' 키워드를 붙인다.

추상 클래스는 추상 메소드를 포함하고 객체화할 수 없다는 점만 제외하면 일반
클래스와 다르지 않으며 생성자, 멤버변수와 일반 메소드도 가질 수 있다. 추상
클래스 자체로는 클래스로
```

런타임 전에는 객체 생성이 되지 않기 때문에 **Car car = new A();**를 해도 컴파일러는 A가 생성됨을 알 수 없다. 그렇기에 Car가 정의한 print()메소드만 접근이 가능하다.

추상 클래스

추상 클래스란?

하나 이상의 추상 메소드(abstract method)를 포함하는 클래스이다.
추상 메소드는 선언만 있고 구현부는 없는 함수이며 선언부에 'abstract' 키워드를 사용하여 선언한다. 추상 메소드가 포함되었다면 클래스 또한 추상 클래스이므로 클래스명 앞에도 'abstract' 키워드를 붙인다.

추상 클래스는 추상 메소드를 포함하고 객체화할 수 없다는 점만 제외하면 일반 클래스와 다르지 않으며 생성자, 멤버변수와 일반 메소드도 가질 수 있다. 추상 클래스 자체로는 클래스로의 역할을 하지 못하며 객체를 생성할 수 없지만 새로운 클래스를 작성하는데 있어서 부모 클래스로서 중요한 역할을 갖는다.

```
abstract class Animal {
    public String sName;
    public void move() {
        System.out.println("어슬렁 어슬렁");
    }
    abstract void howl();
}

class Dog extends Animal {
    public void move() {
        System.out.println("팔짝 팔짝");
    }

    void howl() {
            System.out.println("멍멍");
    }
}

class Cat extends Animal {
    void howl() {
            System.out.println("냐옹");
    }
}

public class Tut02 {

    public static void main(String[] args) {
        Dog happy = new Dog();
        Cat julia = new Cat();
        happy.move(); // 오버라이드된 멤버함수 호출
        happy.howl(); // 구현된 멤버함수 호출
        jular.move(); // 일반 멤버함수 호출
```

```
        julia.howl(); // 구현된 멤버함수 호출
    }
}
// 출력 결과
// 팔짝 팔짝
// 멍멍
// 어슬렁 어슬렁
// 냐옹
```

추상 메소드의 접근 지정자로 private는 사용할 수 없는데, 이는 자식 클래스에서 받아서 구현되어야 하기 때문이다. 다른 접근 지정자(public, protected)는 사용할 수 있고 생략되면 default로 일반 클래스의 선언과 동일하다.
추상 클래스는 다른 클래스들에게서 공통으로 가져야하는 메소드들의 원형을 정의하고 그것을 상속받아서 구현하도록 하는데 사용된다. 메소드 오버라이드(override)와 유사해 혼동하기 쉬우나 추상 메소드는 자식 클래스에서 구현이 강제된다는 점이 차이다.

final 키워드

final 키워드란?

자바 언어에서 final은 오직 한번만 할당할 수 있는 entity를 정의할 때 사용된다. final로 선언된 변수가 할당되면 항상 같은 값을 가진다. 만약 final 변수가 객체를 참조하고 있다면, 그 객체의 상태가 바뀌어도 final 변수는 매번 동일한 내용을 참조한다.

1. final 클래스 : final로 선언한 클래스는 상속할 수 없다. 가령 java.lang.System이나 java.lang.String처럼 자바에서 기본적으로 제공하는 라이브러리 클래스는 final을 사용한다.
2. final 메소드 : final로 선언된 메소드의 경우 오버라이딩으로 수정할 수 없다.
3. final 변수 : final 변수는 한번 값을 할당하면 수정할 수 없으므로 초기화는 한번만 가능하다. 이는 초기화를 반드시 해야한다는 의미가 아닌 값을 할당한 후에 변경할 수 없음으로 상수와 다르다.

static and final

static : 메모리에 한번만 할당되어 이후 메모리를 효율적으로 사용할 수 있게 해줌
final : 한번의 할당 이후 값이 변하지 않음
=> 효율적인 메모리 운영을 위해 자주 같이 사용된다.

Object 클래스

java.lang.Object 클래스

java.lang 패키지 중에서도 가장 많이 사용되는 클래스이다.
Object 클래스는 모든 자바 클래스의 최상위 조상 클래스가 된다.
그렇기에 자바의 모든 클래스는 Object 클래스의 모든 메소드를 바로 사용할 수 있다.
Object 클래스는 필드를 가지지 않으며, 총 11개의 메소드만으로 구성되어 있다.

Object 클래스의 메소드

메소드	설명
protected Object clone()	해당 객체의 복제본을 생성하여 반환
boolean equals(Object obj)	해당 객체와 전달 받은 객체가 같은지 여부 반환
protected void finalize()	해당 객체를 더는 아무도 참조하지 않아 가비지 컬렉터가 객체의 리소스를 정리하기 위해 호출
Class<T> getClass()	해당 객체의 클래스 타입을 반환
int hasCode()	해당 객체의 해쉬 코드값을 반환
void notify()	해당 객체의 대기(wait)하고 있는 하나의 스레드를 다시 실행할 때 호출
void notifyAll()	해당 객체의 대기(wait)하고 잇는 모든 스레드를 다시 실행할 때 호출
String toString()	해당 객체의 정보를 문자열로 반환
void wait()	해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행할 때까지 현재 스레드를 일시적으로 대기(wait) 시킴
void wait(long timeout)	해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행하거나 전달 받은 시간(timeout)이 지날 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출
void wait(long timeout, int nanos)	해당 객체의 다른 스레드가 notify()나 notifyAll() 메소드를 실행하거나 현재 스레드를 인터럽트(interrupt)할 때까지 현재 스레드를 일시적으로 대기(wait)시킬 때 호출

인터페이스

인터페이스란?

극단적으로 동일한 목적 하에 동일한 기능을 수행하게끔 강제하는 것
즉, 자바의 다형성을 극대화하여 개발 코드 수정을 줄이고 프로그램 유지 보수성을 높이기 위해 사용된다.
interface 키워드를 통해 선언할 수 있으며, Implements 키워드를 통해 일반 클래스에 해당 인터페이스를 적용시킬 수 있다.

인터페이스의 구현

```
public interface 인터페이스명 {

    //상수
    타입 상수명 = 값;
    // 인터페이스에서 값을 정해주고, 해당 인터페이스를 이용하는 클래스에서는 값을 바꿀 수 없다.

    //추상 메소드
    타입 메소드명(매개변수, ... );
    // 해당 추상 메소드의 가이드에 따라 오버라이딩해 재구현한다.

    //디폴트 메소드
    default 타입 메소드명(매개변수, ... ){
      //구현부
    }
    // 인터페이스에서 해당 메소드를 기본적으로 제공해주나, 용도에 따라 재구현하여 사용할 수 있다.

    //정적 메소드
    static 타입 메소드명(매개변수) {
      //구현부
    }
    // 인터페이스에서 구현한 대로 사용된다.

}
```

디폴트 메소드의 경우 오버라이딩하여 간편하게 사용할 수 있으나, 용도가 달라지거나 구현부의 차이가 생길 시 오버라이딩하여 재구현할 수 있다.
이미 운영되고 있는 시스템에서 추가 요건으로 인해 불가피하게 반영을 해야할 때 사용하면 효과적이다.

```
public class studyClass implements study{
```

```
    //구현 클래스 메소드
    public void say(){
        System.out.println("this studyClass");
    }

    @Override
    public String interfaceMethod() {
        return null;
    }

    @Override
    public String name() {
        return null;
    }
}


public static void main(String[] args) {
        study study = new studyClass();
        studyClass studyClass = new studyClass();

        study.say(); // 불가능
        studyClass.say();
    }
```

인터페이스 타입으로도 객체를 생성할 수 있으며, 해당 객체에 구현 클래스로 인스턴스화할 수 있다.
하지만 인터페이스 타입으로 선언한 객체는 구현 클래스 내에서 생성한 메소드, 필드를 사용할 수 없다.

인터페이스의 상속

```
// 구현부가 비었지만 상속 했으므로 멤버가 2개
interface Fightable extends Movable, Attackable { }

interface Movable{ void move(int x, int y); }
interface Attackable { void attack(Unit u); }
```

인터페이스의 조상은 인터페이스만이 가능하며 클래스와 달리 Object가 최고 조상이 아니다.
추상 메소드는 충돌해도 문제가 생기지 않기 때문에 다중 상속이 가능하다.

interface implements vs class extends

implements를 이용하여 인터페이스를 상속 받을 경우에는 다수의 인터페이스를 상속 받을 수 있다.
extends를 이용하여 클래스를 상속 받을 경우에는 하나의 클래스만 상속 받을 수 있다.

다형성 비교

상속 다형성의 기본 형태는 부모클래스 p = new childClass() 이다. 이때, 부모 클래스 p가 참조할 클래스는 자신의 클래스보다 범위가 넓거나 같아야한다.

```
class Car {/*구현*/}

class FireEngine extends Car {/*구현*/}

public static void main(String[] args) {

    /*자기 자신을 참조하여 객체 생성*/
    Car c = new Car();

    /*자기 자신보다 범위가 넓은 자식클래스를 참조하여 객체 생성*/
    Car c1 = new FireEngine();
    // FireEngine()을 참조하고 있으나 Car의 인스턴스이기에 Car 클래스에
정의된 멤버만 사용 가능

    /*형변환 하여 객체 생성*/
    FireEngin f = (FireEngine)new Car();
      // Car()를 참조하고 있지만 FireEngine()으로 형변호나 하여 FireEngine
클래스의 멤버를 사용할 수 있다.
    // 형변환하지 않을 시, 에러가 발생한다.
}
```

```
public class InterfaceTestClass implements InterfaceTest1 ,
InterfaceTest2, InterfaceTest3{/*구현*/}
```

```
public static void main(String[] args) {
    InterfaceTest1 i1 = new InterfaceTestClass();
    InterfaceTest2 i2 = new InterfaceTestClass();
    InterfaceTest3 i3 = new InterfaceTestClass();
}
// InterfaceTestClass에는 InterfaceTest1, InterfaceTest2,
InterfaceTest3의 멤버가 모두 구현되어 있으나
// 각 i1, i2, i3는 InterfaceTest1, InterfaceTest2, InterfaceTest3의
멤버만을 사용할 수 있다.
```

인터페이스의 메소드

기본 메소드(Default Method), >= Java 8

인터페이스는 기능에 대한 선언만 가능했던 것에 반해, default method를 선언할 경우, 해당 메소드의 로직을 구현할 수 있다.
이는 하위 호환성을 위해 추가되었다.
기존에 존재하던 인터페이스를 이용하여서 구현된 클래스를 만들고 사용하고 있는데,
인터페이스를 보완하는 과정에서 추가적으로 구현해야할 혹은 필수적으로 존재해야할 메소드가 있다면,
이전에 해당 인터페이스를 구현한 클래스와의 호환성이 떨어진다.
이를 방지하기 위해 default method로 해당 메소드를 선언한다.

```
interface MyInterface {
    default void printHello() {
      System.out.println("Hello World");
        }
}

class MyClass implements MyInterface {}

public class DefaultMethod {
    public static void main(String[] args) {
      MyClass myClass = new MyClass();
        myClass.printHello(); //실행결과 Hello World 출력
    }
}
```

정적 메소드(Static Method), >= Java 8

default method와 동일하게 인터페이스 내에서 메소드 로직을 구현할 수 있다.
반드시 클래스 명을 통하여 메소드를 호출해야 하며, default method와 다르게 재정의(overriding)이 불가능하다.

```java
public interface Calculator {
      public int plus(int pre, int post);
    public int multi(int pre, int post);

    default int execPlus(int pre, int post){
      return pre + post;
    }

    static int execMulti(int pre, int post){ return pre * post; }
}

public class CalculatorImpl implements Calculator {
    @Override
    public int plus(int pre, int post) {
        return pre + post;
    }
    @Override
    public int multi(int pre, int post) {
        return pre * post;
    }
}

public class InterfaceEx {
      public static void main(String[] args) {
      Calculator cal = new CalculatorImpl();

        int resultPlus = cal.plus(3, 9);
        int resultMulti = cal.multi(3, 9);

        System.out.println("resultPlus 값은 : " + resultPlus);
        System.out.println("resultMulti 값은 : " + resultMulti);

        int resultExecPlus = cal.execPlus(3, 9);
        System.out.println("dafault method 호출 결과 : "+resultExecPlus);

        int resultExecMulti = Calculator.execMulti(3, 9);
        System.out.println("static method 호출 결과 : "+resultExecMulti);
      }
}
```

default, static method를 사용할 때 주의할 점

1. 클래스가 항상 이긴다.
   클래스나 슈퍼 클래스에서 정의한 메소드가 디폴트 메서드 보다 우선권을 가진다.
2. 1번 규칙 이외의 상황에서는 서브 인터페이스가 이긴다.
   상속 관계를 갖는 인터페이스에서 같은 시그니처를 갖는 메소드를 정의할 때는 서브 인터페이스가 이긴다.

즉, B가 A를 상속 받는다면, B가 A를 이긴다.

3. 끝까지 디폴트 메소드의 우선순위가 결정되지 않았다면 여러 인터페이스를 상속 받는 클래스가 명시적으로 디폴트 메소드를 재정의(override)하고 호출하자.

private Method, >= Java 9

default, static 메소드의 한계로 인해 추가되었다.

1. interface 내부 메소드에서 사용할 부분인데도 public method로 선언하는 것이 필수가 된다.
2. 인터페이스를 구현할 인터페이스 또는 클래스가 특정 method에 접근하는 것을 막을 수 없다.

위 2가지 사항을 **private**을 사용해 해결할 수 있다.

```
public interface Boo {
    /**
     * @implSpec 이 메소드는 오버라이드 하거나, 외부에서 접근할 수 없습니다.
     * @return "Bye"
     */
    private String printBye(){
        return "Bye";
    }
    default void knockDoor(){
        System.out.println("Ok.. " + printBye());
    }
}

public class Main implements Boo{
    @Override //Error -> method does not override or implement a method from a supertype
    public String printBye(){
        return "Bye";
    }
}
```

private String printBye()를 작성함으로 외부에서 접근이 불가능하게 설정할 수 있다.