

제09장

# 상속과 다형성

구디아카데미 ▷ 민경태 강사

# 학습목표

1. 상속의 개념에 대해서 알 수 있다.
2. 상속 관계를 가지는 부모 클래스와 자식 클래스를 생성할 수 있다.
3. 메소드 오버라이드에 대해서 알 수 있다.
4. 상속 관계에 있는 클래스들의 형 변환에 대해서 알 수 있다.
5. 다형성에 대해서 알 수 있다.
6. Object 클래스에 대해서 알 수 있다.

# 목차

1. 상속의 개념
2. 클래스 상속
3. 메소드 오버라이드
4. 업캐스팅과 다운캐스팅
5. 다형성
6. Object 클래스

```

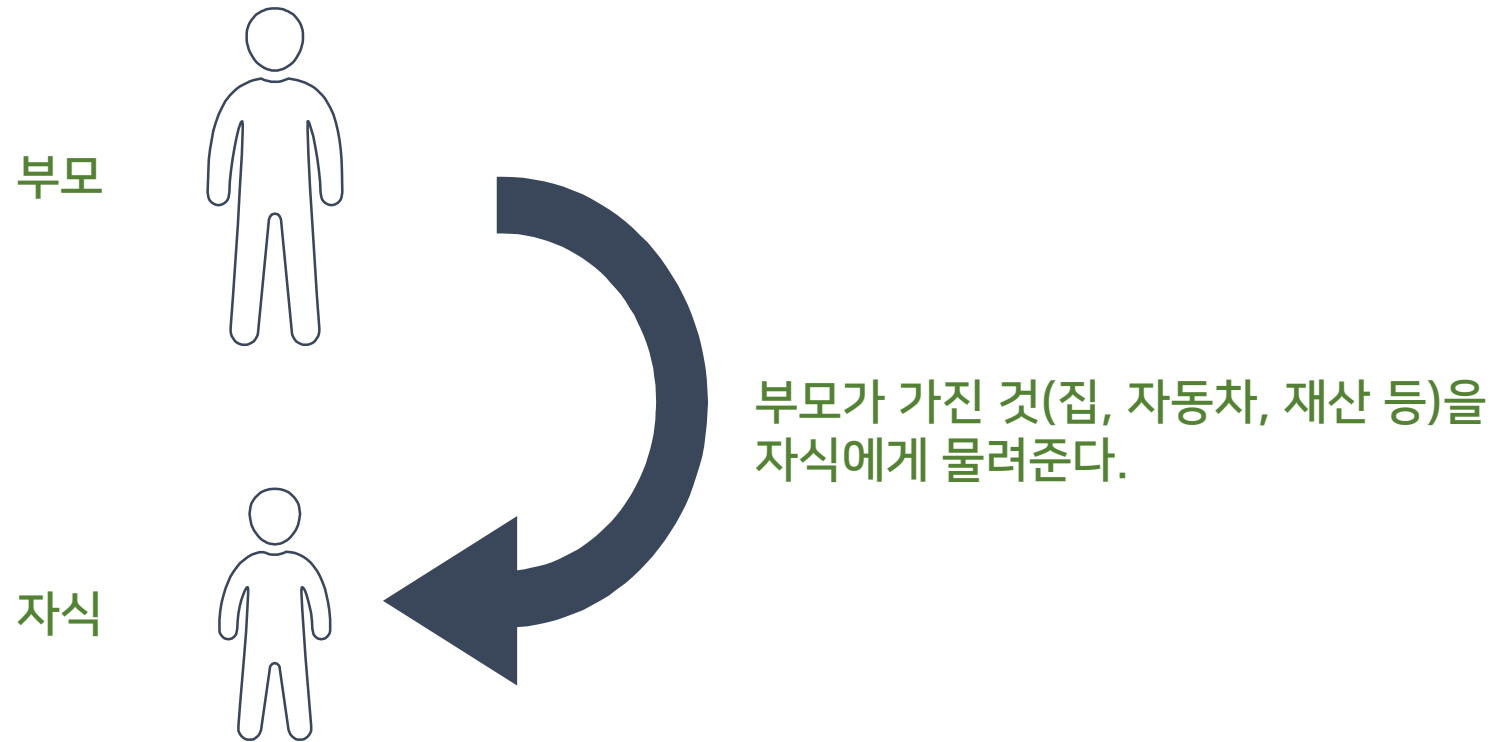
each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > r ? Math.max(0, r + n) : r; r in t && t[r] === e) return r;
    }
}

```

# 1. 상속의 개념

# 현실 세계의 상속

- 현실 세계의 상속  
부모가 가지고 있는 것을 자식에게 물려준다.



# 자바의 상속

- 자바의 상속

부모 클래스가 가지고 있는 메소드를 자식 클래스에게 물려준다.



# 상속의 개념

- Inheritance
- 어떤 클래스의 필드와 메소드를 다른 클래스가 물려 받아 사용하는 것을 의미함
- 상속을 이용하면 클래스들을 계층 구조로 관리할 수 있음
- 필드와 메소드를 제공하는 클래스를 부모 클래스라고 함
- 필드와 메소드를 제공받는 클래스를 자식 클래스라고 함

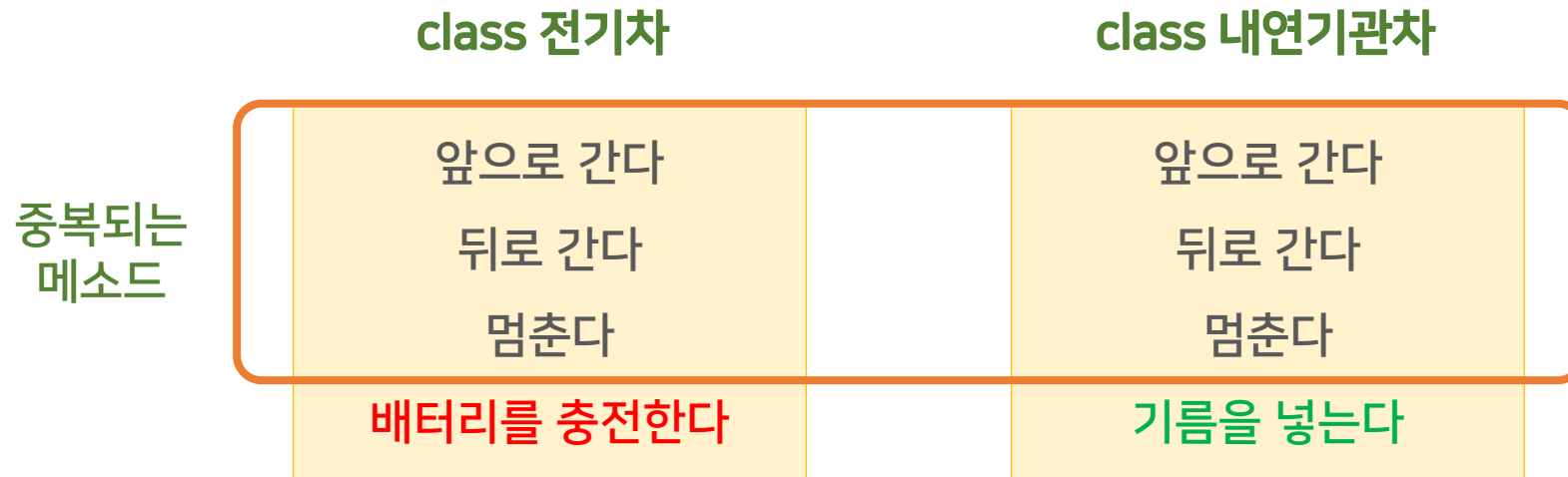
# 상속의 장점

- 코드의 중복을 줄일 수 있음
  - 이미 만들어진 부모 클래스의 코드를 재사용하기 때문임
- 새로운 클래스를 빠르게 만들 수 있어 개발 시간을 줄일 수 있음
  - 자식 클래스를 만들 때는 부모 클래스의 메소드를 다시 만들 필요가 없음
- 클래스의 유지 보수 시간을 줄일 수 있음
  - 부모 클래스를 수정하면 자식 클래스가 모두 수정되는 효과가 있음



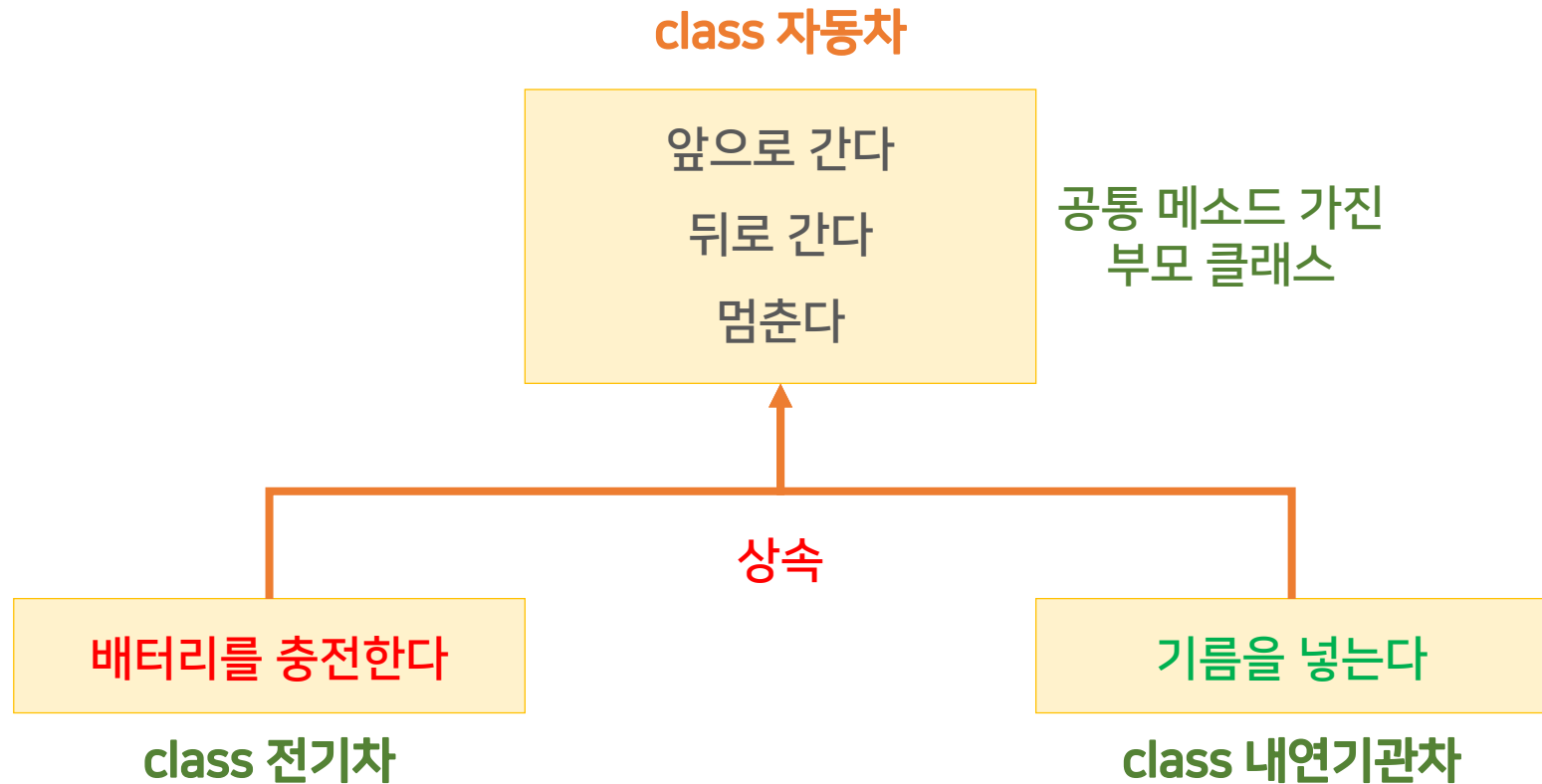
# 상속 도입 이전

- 상속이 없다면 클래스마다 동일한 메소드를 여러 번 정의해야 함



# 상속 도입 이후

- 클래스들이 공동으로 사용하는 메소드를 가진 부모 클래스를 생성함



# 슈퍼 클래스와 서브 클래스

- 슈퍼 클래스 : 부모 클래스를 의미함
- 서브 클래스 : 자식 클래스를 의미함



상속 관계를 나타내는 방법

```

each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n : 0; r in t && t[r] === e) return r;
    }
}

```

## 2. 클래스 상속

# extends

- 자식 클래스를 정의할 때 사용하는 키워드
- 자식 클래스는 오직 하나의 부모 클래스만 가질 수 있음 (다중 상속 불가)
- 형식  
class 자식클래스 extends 부모클래스 {  
  
}

# extends 예시

## ■ Car 클래스와 EV 클래스

```
public class Car {  
    public void forward() { }  
    public void reverse() { }  
    public void stop() { }  
}
```

부모

```
public class EV extends Car {  
    public void charge() { }  
}
```

자식

```
public class MainClass {  
  
    public static void main(String[] args) {  
  
        EV ev = new EV();  
        ev.forward();  
        ev.reverse();  
        ev.stop();  
        ev.charge();  
  
        자식 클래스 객체는  
        부모 클래스의 메소드를  
        호출할 수 있다.  
    }  
}
```

# super

- super
- 부모 클래스의 참조값을 의미함
- 부모 클래스의 멤버(필드, 메소드)를 호출할 때 사용할 수 있음  
단, 상속 관계라 하더라도 부모 클래스의 private 멤버에는 접근할 수 없음
- 형식
  - super.필드
  - super.메소드()

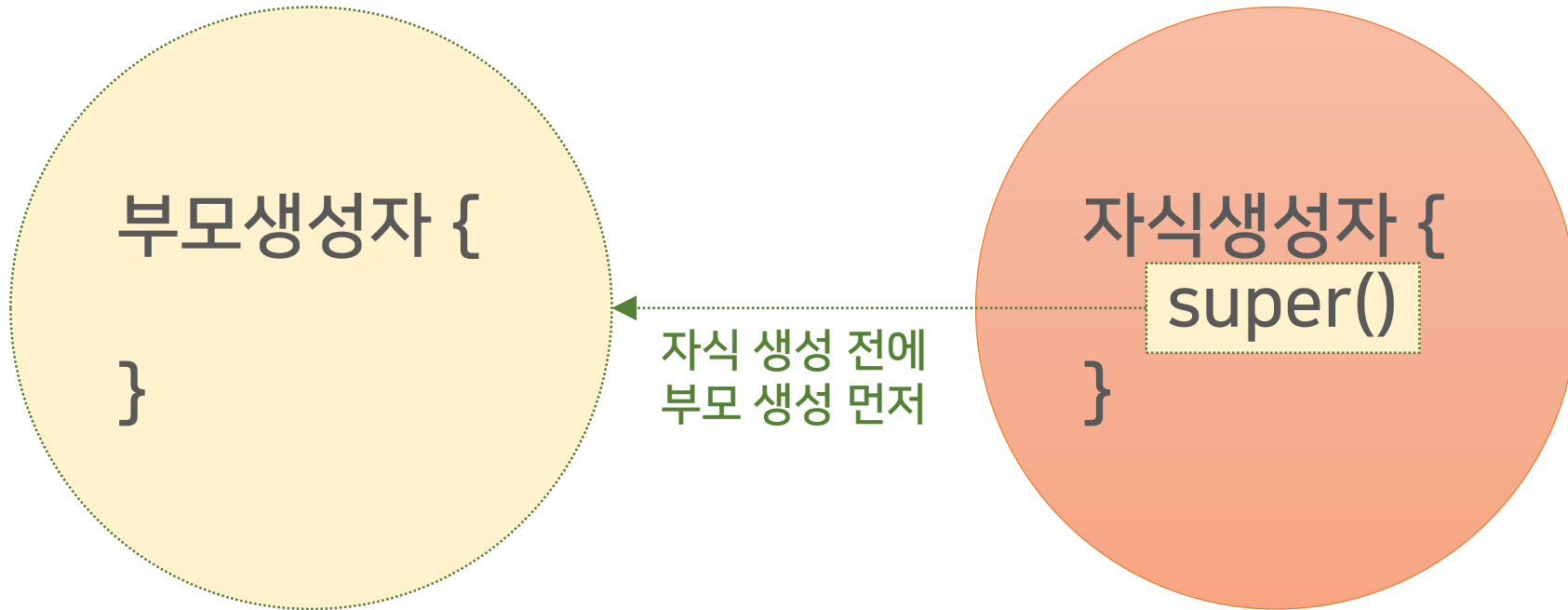
# 자식 클래스 객체 생성

- 현실세계
  - 부모가 없는데 자식이 태어날 수는 없는 노릇
- Java
  - 부모 객체가 먼저 생성된 뒤 자식 객체가 생성될 수 있음
- 부모 클래스가 생성자를 가지고 있는 경우 자식 클래스는 반드시 생성자를 만들고 그 내부에서 부모 클래스의 생성자를 호출해야 함
- 부모 클래스에 생성자가 없으면 자식 클래스도 부모 클래스의 생성자를 호출할 필요가 없음 (부모 클래스의 디폴트 생성자가 자동으로 사용됨)



# super()

- 부모 클래스의 생성자를 호출하는 코드
- 자바는 자식 객체가 생성되기 전에 반드시 부모 객체를 생성해야만 함

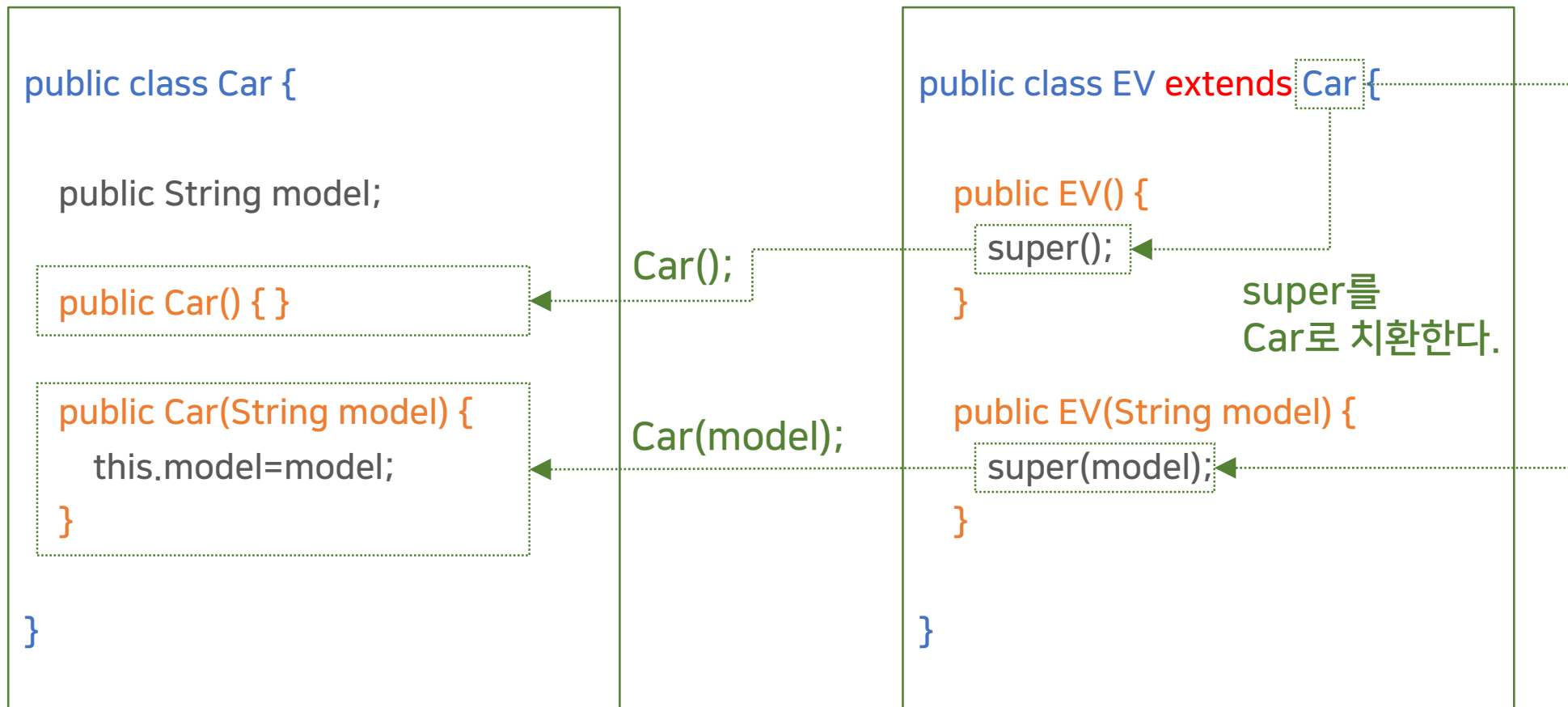


# super() 호출 규칙

- super() 호출 규칙
  - 부모 클래스가 생성자를 가지고 있는 경우에는 자식 클래스는 반드시 super()를 호출해야 함
  - 또한 자식 클래스의 생성자 내부에서는 super()를 가장 먼저 호출해야 함
- super() 호출이 생략 가능한 경우
  - 부모 클래스에 생성자가 없는 경우
  - 부모 클래스의 생성자 중에서 디폴트 형식의 생성자를 호출하는 경우

# 부모 클래스 생성자 호출

## ■ Car 클래스와 EV 클래스



# 부모 클래스 디폴트 생성자 호출

- 부모 클래스의 디폴트 생성자 호출은 생략 가능함

```
public class Car {
```

```
    public String model;
```

```
    public Car() { }
```

```
    public Car(String model) {  
        this.model=model;  
    }
```

```
}
```

Car();

```
public class EV extends Car {
```

```
    public EV() {  
        super();  
    }
```

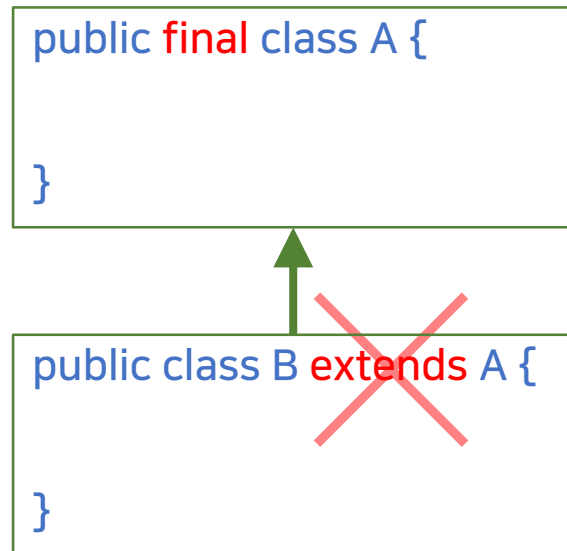
```
    public EV(String model) {  
        super(model);  
    }
```

```
}
```

디폴트 생성자 호출은 생략 가능하다.

# final 클래스

- final 키워드를 추가한 final 클래스는 상속이 불가능함



final 클래스를 상속 받는 것은 불가능하다.

```

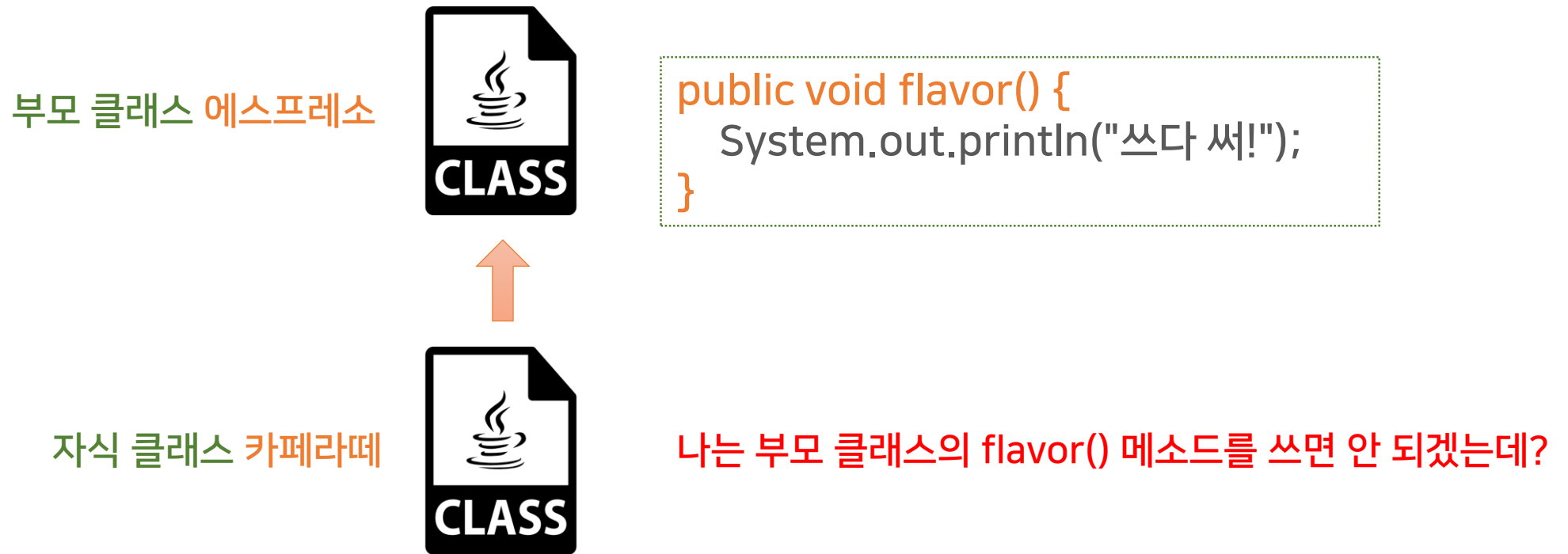
each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n; n in t && t[n] === e) return n;
    }
}

```

### 3. 메소드 오버라이드

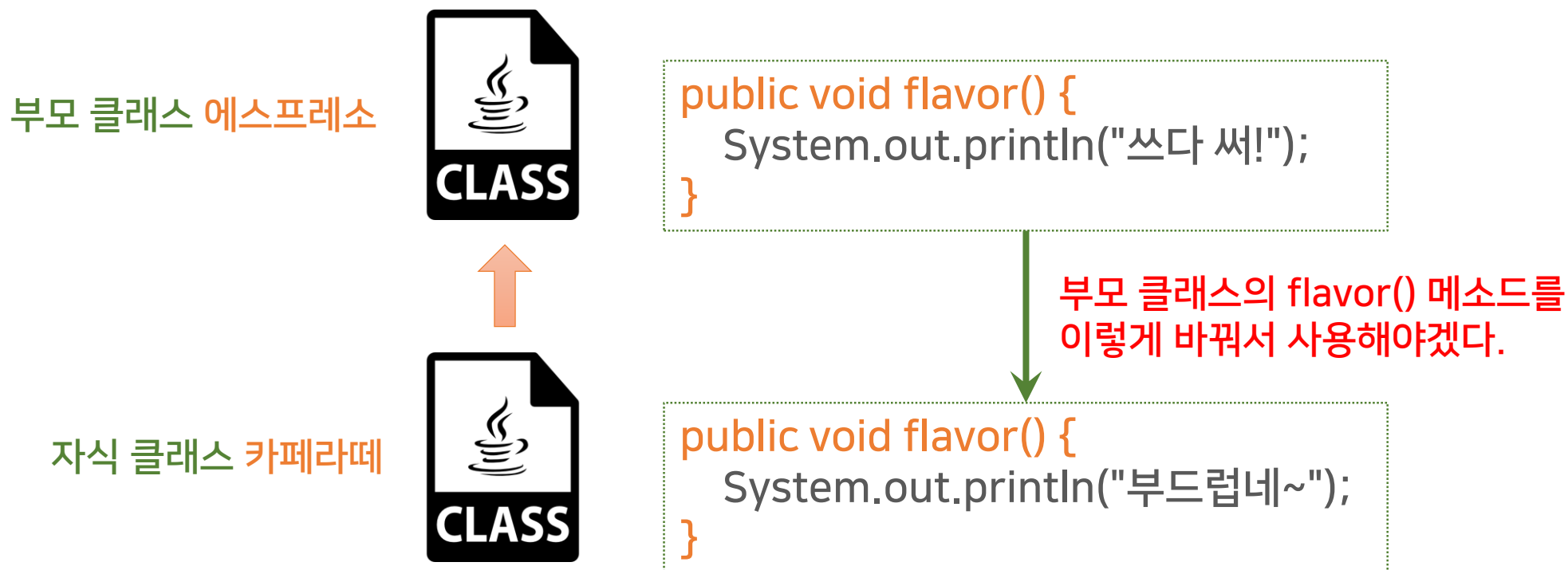
# 부모 클래스 메소드의 한계

- 자식 클래스가 항상 부모 클래스의 메소드를 사용할 수 있는 건 아님



# 부모 클래스 메소드 무시

- 부모 클래스의 메소드가 적절하지 않다면 다시 만들어도 상관 없음





# 메소드 오버라이드

- Method Override
- 자식 클래스가 부모 클래스의 메소드를 사용할 수 없어서 해당 메소드를 다시 만드는 것을 의미함
- 생성 규칙
  - 반드시 부모 클래스의 메소드와 동일한 원형(반환타입, 메소드명, 매개변수)으로 만들어야 함
  - @Override 어노테이션을 추가해서 메소드 오버라이드 규칙에 맞는지 점검하는 것을 권장함

# @Override

- 오버라이드된 메소드에 추가하는 어노테이션(Annotation)
- 메소드 오버라이드 규칙을 지켰는지 점검하는 역할을 수행함
- 만약 메소드 오버라이드 규칙을 어겼다면 컴파일 오류를 발생시킴
- 필수는 아니지만 메소드 오버라이드를 할 땐 항상 추가하기를 권장함

# 메소드 오버라이드와 접근제어자

- 메소드 오버라이드를 할 때 부모 클래스의 접근제어자와 같거나 더 넓은 범위를 가지는 접근제어자를 사용할 수 있음
- 메소드 오버라이드와 접근제어자

부모 클래스의 접근제어자	메소드 오버라이드시 사용 가능한 접근제어자
public	public
protected	public, protected
default	public, protected, default
private	public, protected, default, private

# 메소드 오버라이드 예시

- 부모 클래스 "꼬깔과자"와 맛이 다른 자식 클래스 "매콤달콤꼬깔과자", "허니버터꼬깔과자", "새콤달콤꼬깔과자"는 맛() 메소드를 다시 만듦

```
public class 꼬깔과자 {  
    public void 모양() {  
        System.out.println("꼬깔모양");  
    }  
    public void 맛() {  
        System.out.println("고소한맛");  
    }  
}
```

```
public class 매콤달콤꼬깔과자 extends 꼬깔과자 {  
    @Override  
    public void 맛() {  
        System.out.println("매콤달콤맛");  
    }  
}
```

```
public class 허니버터꼬깔과자 extends 꼬깔과자 {  
    @Override  
    public void 맛() {  
        System.out.println("허니버터맛");  
    }  
}
```

```
public class 새콤달콤꼬깔과자 extends 꼬깔과자 {  
    @Override  
    public void 맛() {  
        System.out.println("새콤달콤맛");  
    }  
}
```

# final 메소드

- final 키워드를 추가한 final 메소드는 오버라이드가 불가능함

```
public class A {  
    public final void method() {}  
}
```



```
public class B extends A {  
    @Override  
    public void method() {}  
}
```

final 메소드를 오버라이드 하는 것은 불가능하다.

```

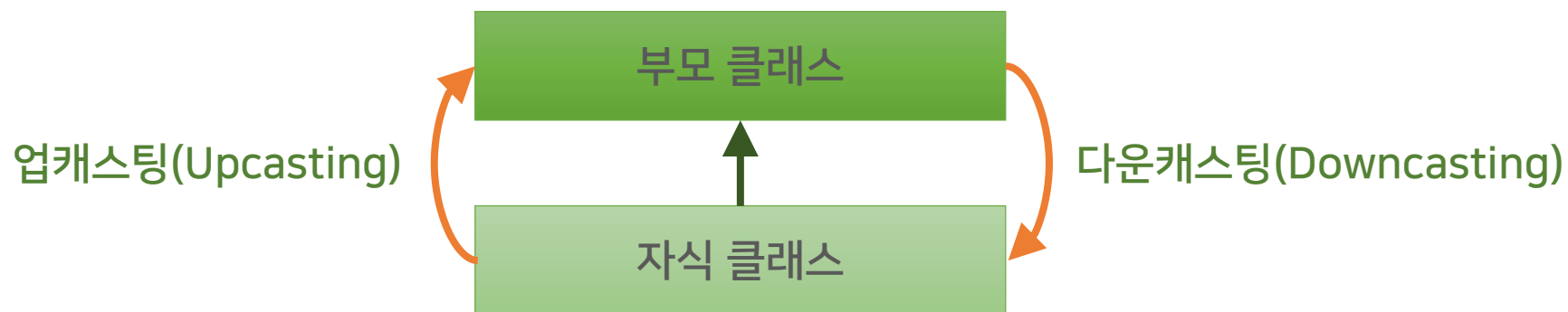
each: function(e, t, n) {
  var r, i = 0,
      o = e.length,
      a = M(e);
  if (n) {
    if (a) {
      for (; o > i; i++)
        if (r = t.apply(e[i], n), r === !1) break;
    } else
      for (i in e)
        if (r = t.apply(e[i], n), r === !1) break;
  } else if (a) {
    for (; o > i; i++)
      if (r = t.call(e[i], e[i]), r === !1) break;
  } else
    for (i in e)
      if (r = t.call(e[i], e[i]), r === !1) break;
  return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
  return null == e ? "" : b.call(e);
} : function(e) {
  return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
  var n = t || [];
  return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
  var r;
  if (t) {
    if (n) return m.call(t, e, n);
    for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n : 0; r < n; r++)
      if (n in t && t[r] === e) return r;
  }
}

```

## 4. 업캐스팅과 다운캐스팅

# 업캐스팅과 다운캐스팅

- 상속 관계의 부모 클래스와 자식 클래스 간 타입 변환을 의미함



# 업캐스팅

- Upcasting
- 자식 클래스 객체는 부모 클래스 타입을 형 변환을 할 수 있음
- 자식 객체를 생성하고 부모 타입의 참조 변수에 전달하면 자동으로 업캐스팅이 됨
- 형식  
부모클래스 참조변수 = new 자식클래스()



# 업캐스팅 예시

## ■ Car 클래스와 Bus 클래스

```
public class Car {  
  
}
```

부모

```
public class Bus extends Car {  
  
}
```

자식

```
public class MainClass {
```

```
    public static void main(String[] args) {
```

```
        Car car = new Bus();
```

부모타입    자식객체

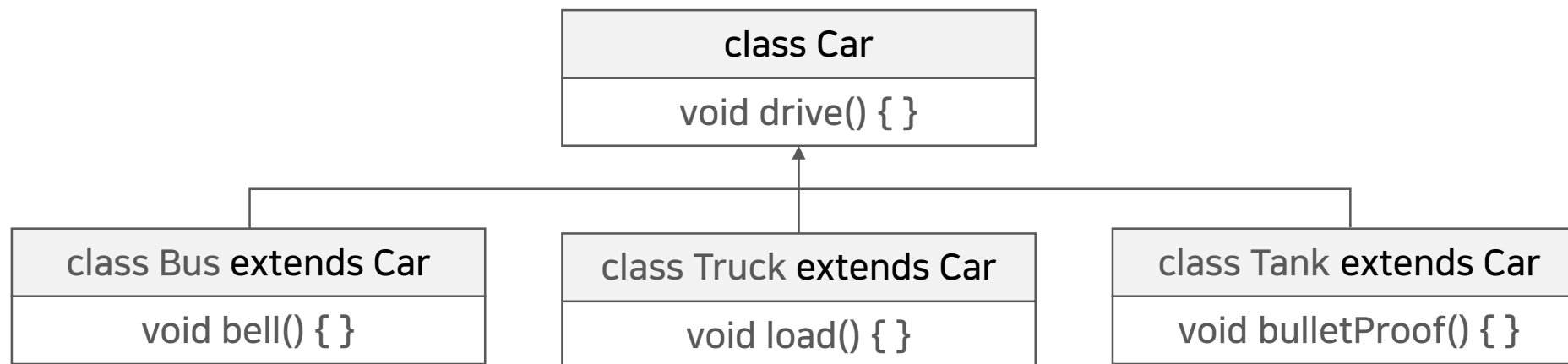
업캐스팅

```
    }
```

```
}
```

# 업캐스팅의 특징

- 하나의 타입으로 여러 객체를 참조할 수 있음 (다형성을 위한 중요한 특징)



```
Car car;
```

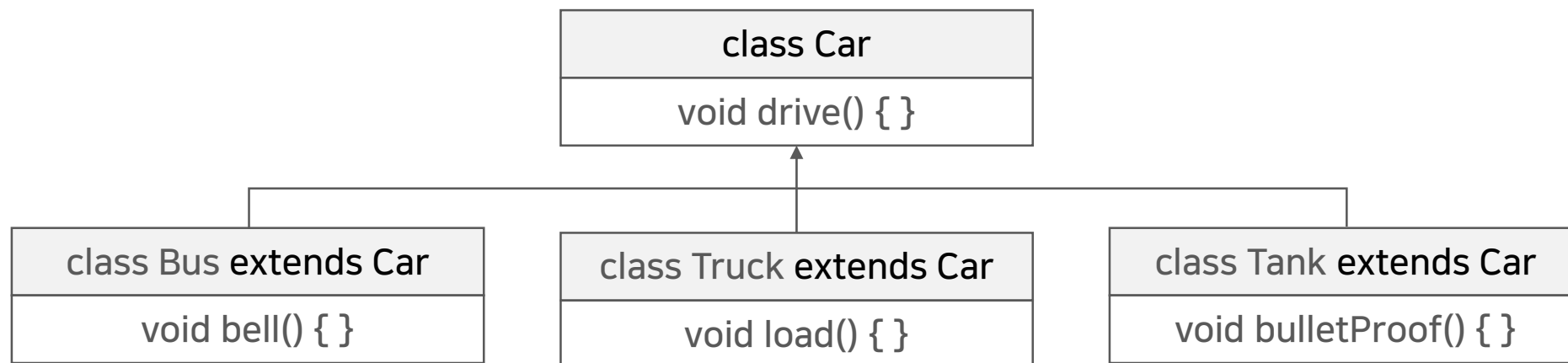
Car 클래스 타입의 참조 변수 car

```
car = new Bus();
car = new Truck();
car = new Tank();
```

Car 클래스의 모든 자식 객체는 Car 클래스 타입으로 자동 변환할 수 있다.

# 업캐스팅의 특징

- 주의! 부모 클래스에 존재하는 메소드만 호출할 수 있음\*



```
Car car = new Bus();
```

객체 `car`는 `Car` 클래스 타입이다.

```
car.drive();
car.bell();
```

객체 `car`는 `Car` 클래스의 메소드만 참조할 수 있어서 `bell()` 메소드를 호출할 수 없다.

\* 향후 다운캐스팅이나 메소드 오버라이드를 이용해서 이 문제를 해결할 수 있다.

# 다운캐스팅

- Downcasting
- 부모 클래스 타입의 객체를 자식 클래스 타입으로 변환할 수 있음
- 강제로 변환해야 하므로 형 변환(Casting) 코드를 작성해야 함

# 다운캐스팅 예시

## ■ Car 클래스와 Bus 클래스

```
public class Car {  
  
}
```

부모

```
public class Bus extends Car {  
  
}
```

자식

```
public class MainClass {
```

```
    public static void main(String[] args) {
```

```
        Car car = new Bus(); ; Car 타입의 Bus 객체
```

```
        Bus bus = (Bus) car; ; Bus 타입으로 다운캐스팅
```

```
    }
```

```
}
```

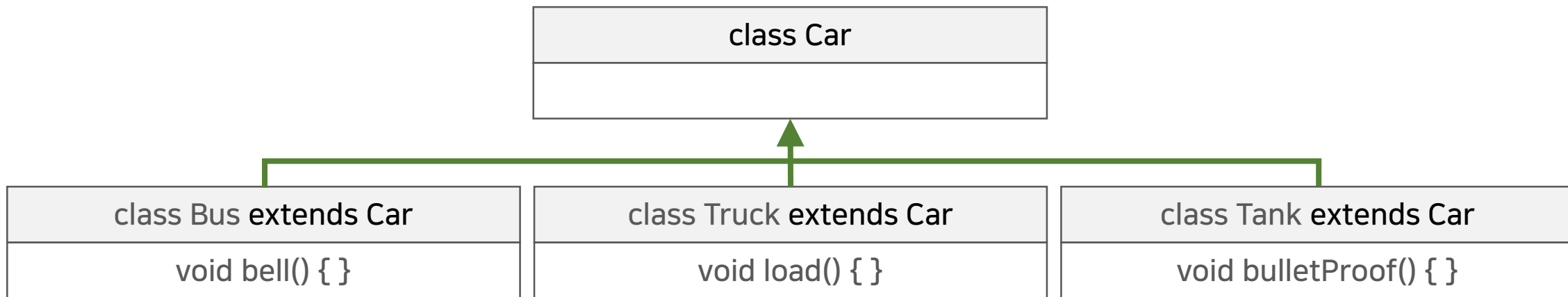
# 다운캐스팅의 특징

- 자식 클래스 타입으로 형 변환할 때 타입 체크를 안 함
- 실수로 잘못된 타입으로 변환하더라도 실행하기 전에는 틀렸다는 것을 알 수 없음
- 잘못된 타입으로 변환한 뒤 실행하면 `ClassCastException`\* 예외가 발생하므로 주의해야 함

\* 예외는 [11.예외처리]에서 다룬다.

# 다운캐스팅의 특징

- 잘못된 타입으로 변환할 가능성이 존재함



```
Car car = new Bus();  
((Truck) car).load();
```

`car` 객체는 `Bus` 객체이다.  
Truck 타입으로 잘못된 다운캐스팅을 해도 컴파일 오류는 발생하지 않는다.

# instanceof 연산자

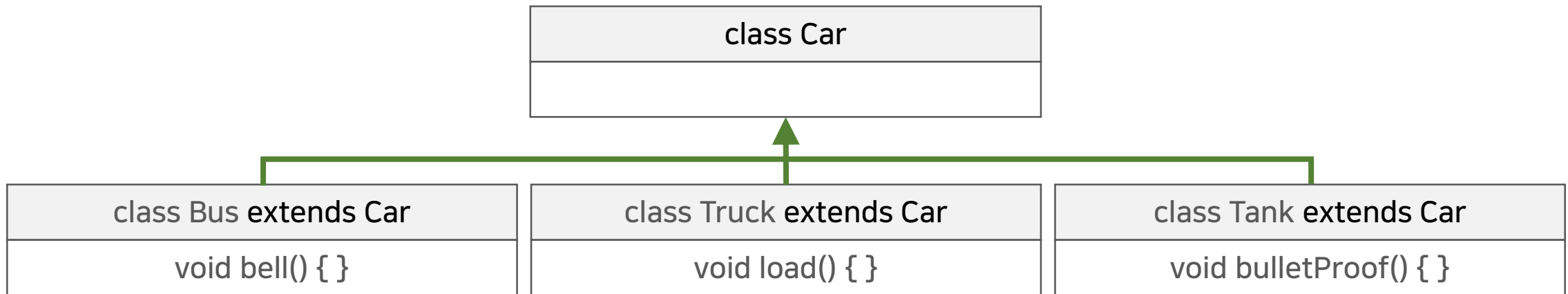
- instanceof
- 자식 클래스 타입으로 형 변환할 때 타입 체크를 위한 연산자
- 변환하려는 타입의 객체가 맞으면 true, 아니면 false를 반환함
- 일반적인 instanceof 연산자의 사용방법

```
if(객체 instanceof 타입) {  
    타입 변환  
}
```



# instanceof 연산자 활용

- instanceof 연산자를 활용한 타입 체크



```
Car car = new Bus();
if(car instanceof Truck) {
    ((Truck)car).load();
}
```

car 객체는 Bus 객체이다.  
car 객체가 실제로 Truck 타입이 맞는지 체크한 뒤 다운캐스팅을 할 수 있다.  
car 객체는 Truck 타입이 아니므로 Truck 타입으로 캐스팅되지 않는다.

```

each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n; r in t && t[r] === e) return r;
    }
}

```

## 5. 다형성

# 상속의 목적

- 상속을 하는 가장 큰 이유는 무엇일까?

1.

동일한 코드를 여러 번 작성할  
필요가 없다.

2.

하나의 클래스 타입으로  
여러 객체를 저장할 수 있다.

# 상속의 목적

- 상속을 하는 가장 큰 이유는 무엇일까?

1.

동일한 코드를 여러 번 작성할  
필요가 없다.

2.

상속을 하는 가장 큰 이유

하나의 클래스 타입으로  
여러 객체를 저장할 수 있다.

# 상속의 중요한 특징들

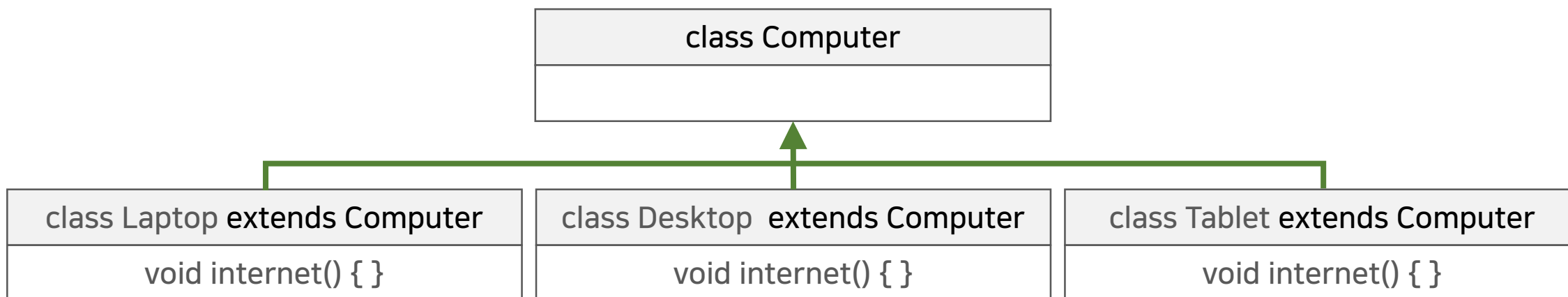
- 여러 객체들이 동일한 메소드를 가질 수 있음
  - 상속의 기본적인 특징임
  - 자식 클래스는 부모 클래스의 멤버를 사용할 수 있음
- 하나의 참조 타입으로 서로 다른 여러 객체를 저장할 수 있음
  - 업캐스팅을 의미함
  - 자식 클래스 객체는 부모 클래스 타입을 사용할 수 있음
- 부모 클래스의 메소드를 다시 만들어 사용할 수 있음
  - 메소드 오버라이드를 의미함
  - 부모 클래스의 메소드를 자식 클래스들이 다시 만들어 사용할 수 있음

# 다형성

- Polymorphism
- 하나의 코드가 여러 가지 동작을 가진다는 의미임
- 다형성의 예시
  - + 연산자는 Number에서는 산술 연산, String에서는 연결 연산을 수행함
- 다형성 구현을 위한 상속 개념
  - 업캐스팅 : 하나의 참조 타입에 여러 개의 객체를 저장할 수 있다.
  - 메소드 오버라이드 : 모든 객체는 기능이 다르지만 동일하게 호출할 수 있는 메소드를 가질 수 있다.

# 다형성을 위한 상속 구조 - 업캐스팅

- 공통 기능을 가진 클래스들의 부모 클래스를 만들고 자식 객체들의 타입을 부모 클래스 타입으로 통일함



```
Computer[] computers = new Computer[3];
```

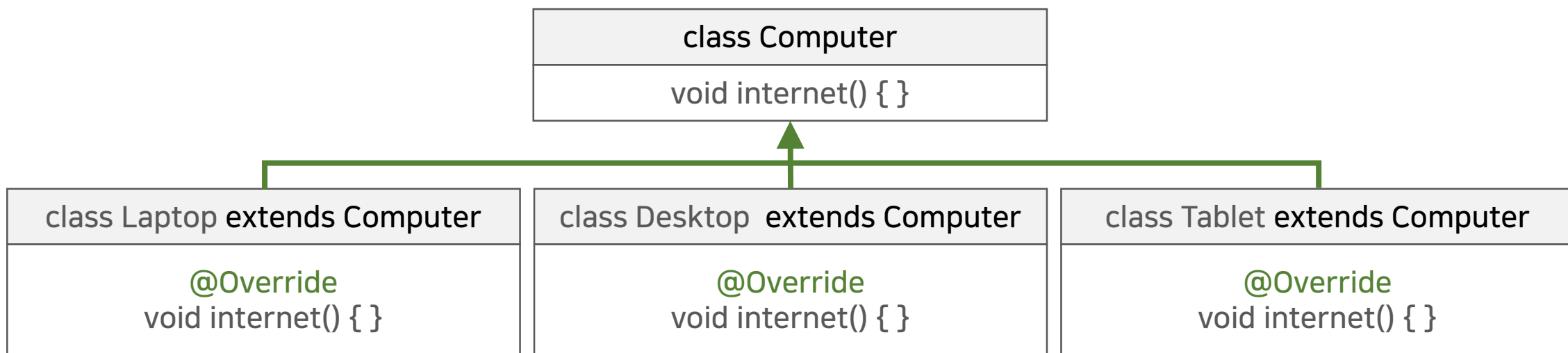
```
computers[0] = new Laptop();
computers[1] = new Desktop();
computers[2] = new Tablet();
```

Computer 클래스 타입의 computers 배열

부모 클래스 타입을 이용하면  
하나의 배열에 3개의 타입을 가진 객체를 저장할 수 있다.

# 다형성을 위한 상속 구조 - 메소드 오버라이드

- 부모 클래스에 호출할 메소드를 넣고 메소드 오버라이드를 하면 자식 클래스의 메소드를 호출할 수 있음



```
for(int i = 0; i < computers.length; i++) {
    computers[i].internet();
}
```

동일한 코드를 이용해 Laptop, Desktop, Tablet 클래스의 internet() 메소드를 모두 호출할 수 있다.



# TV와 라디오

- TV와 라디오는 서로 다른 객체이지만 모두 전자제품이고 play 기능이 있음



동일한 참조 타입(전자제품) ←

전자제품

전자제품

동일한 메소드(play) ←

play 기능 있음

play 기능 있음

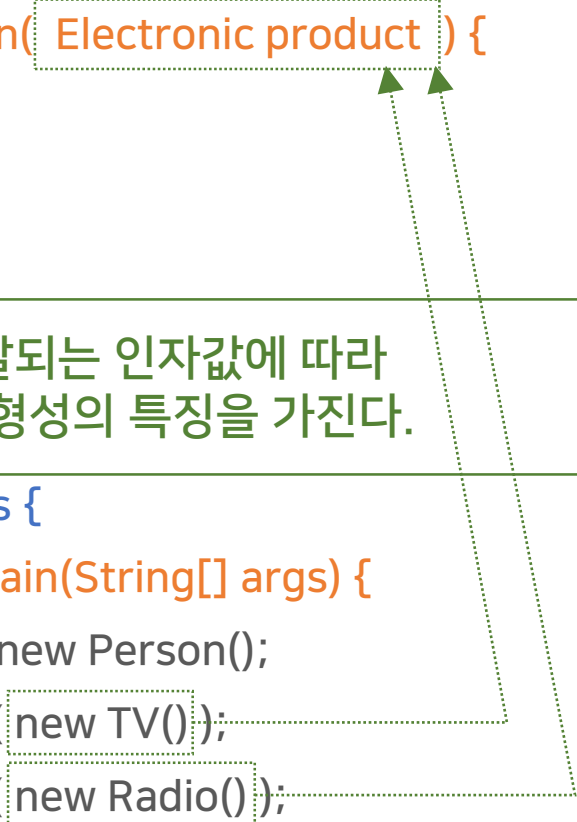
# 다형성 구현 예시

```
public class Electronic {  
    public void play() { }  
}
```

```
public class TV extends Electronic {  
    @Override  
    public void play() {  
        System.out.println("TV 재생");  
    }  
}
```

```
public class Radio extends Electronic {  
    @Override  
    public void play() {  
        System.out.println("Radio 재생");  
    }  
}
```

```
public class Person {  
    public void powerOn( Electronic product ) {  
        product.play();  
    }  
}
```



하나의 코드가 전달되는 인자값에 따라  
다르게 동작하는 다형성의 특징을 가진다.

```
public class MainClass {  
    public static void main(String[] args) {  
        Person person = new Person();  
        person.powerOn( new TV() );  
        person.powerOn( new Radio() );  
    }  
}
```

```

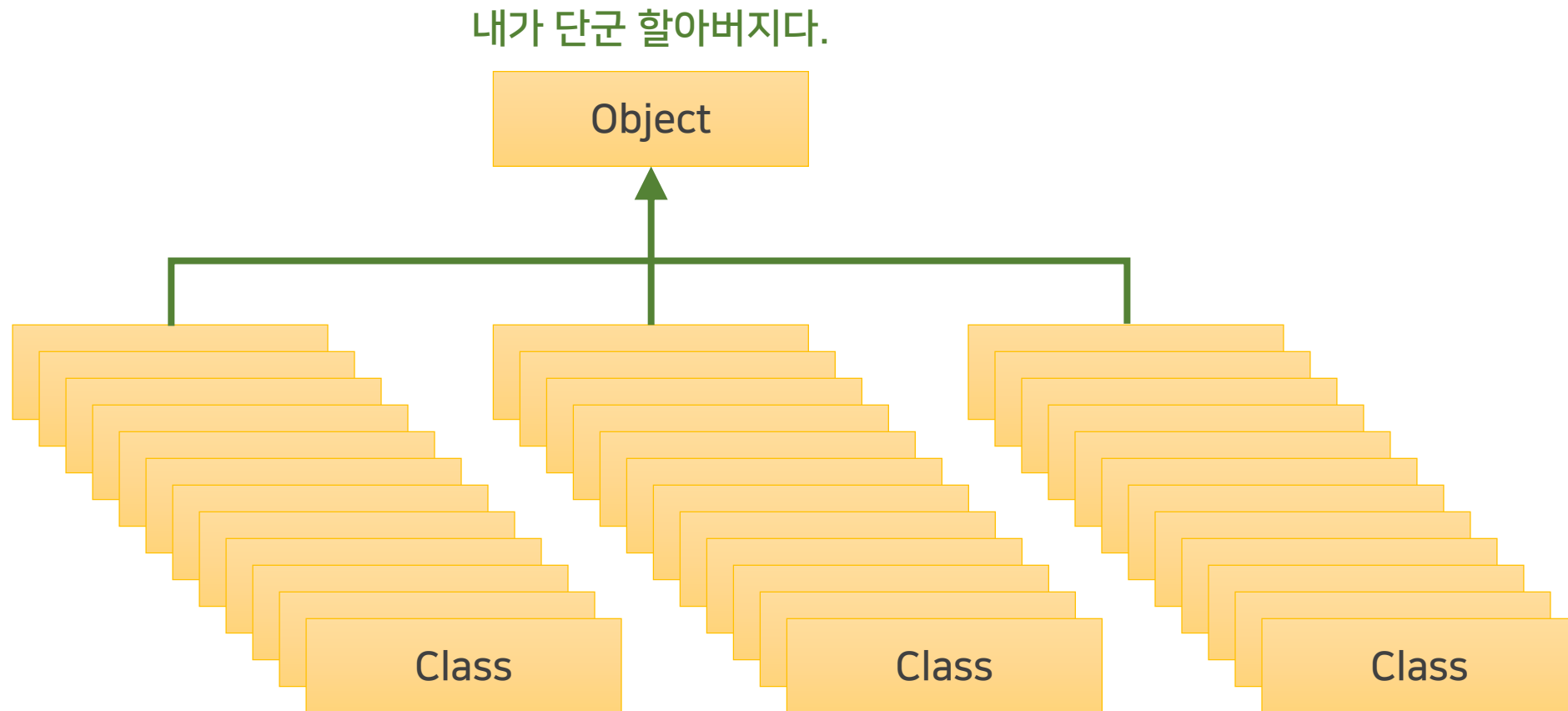
each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
    var r;
    if (t) {
        if (n) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n; n in t && t[n] === e) return n;
    }
}

```

## 6. Object 클래스

# Object 클래스

- 모든 클래스의 최상위 부모 클래스



# Object 클래스

- 패키지 : java.lang
- extends를 명시하지 않은 모든 클래스는 자동으로 Object 클래스를 상속 받음
- 모든 클래스의 부모 클래스
  - 모든 클래스 타입의 객체를 참조할 수 있는 만능 타입으로 사용 가능(업캐스팅)
  - Object 타입으로 저장한 객체는 실제 해당 클래스 타입으로 캐스팅해야 실제 클래스의 메소드를 호출할 수 있음(다운캐스팅)
- Object 클래스가 가지고 있는 메소드는 모든 클래스가 그대로 사용하거나 오버라이드 해서 사용할 수 있음

# Object 클래스 - 메소드

## ■ Object 클래스 주요 메소드

메소드	역할
<code>boolean equals(Object obj)</code>	매개변수 obj 객체와 현재 객체가 동일하면 true 반환
<code>Class&lt;T&gt; getClass()</code>	현재 객체의 클래스타입을 반환
<code>int hashCode()</code>	현재 객체의 해시코드를 반환
<code>String toString()</code>	현재 객체의 정보를 문자열 형태로 반환
<code>void notify()</code>	대기(wait)중인 하나의 스레드를 깨움
<code>void notifyAll()</code>	대기(wait)중인 모든 스레드를 깨움
<code>void wait()</code>	현재 스레드를 일시적으로 대기(wait)시킴

# Object 클래스 - toString() 메소드

- 객체의 문자열 정보를 반환함
- 클래스명과 객체의 해시코드를 @로 연결한 문자열 형식을 반환함
- 객체의 해시코드는 일반적으로 객체의 참조값으로 구성됨

- 원형

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

# Object 클래스 - toString() 메소드 호출

- 출력 메소드(print, println 등)에 객체를 전달하면 객체의 toString() 메소드가 자동으로 호출됨
- 객체와 문자열을 + 연산하는 경우 객체의 toString() 메소드가 자동으로 호출됨

```
public class Person {  
  
}
```

실행결과

Person@7637f22

```
public class MainClass {  
    public static void main(String[] args) {  
        Person p = new Person();  
        System.out.println(p);  
    }  
}
```

메모리 7637f22번지에 저장된 Person 객체 (실행 결과는 다를 수 있다.)



# Object 클래스 - toString() 메소드 오버라이드

- Object 클래스의 toString() 메소드를 오버라이드하면 원하는 형식으로 객체의 문자열 정보를 만들 수 있음

```
public class Person {  
  
    private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public String toString() {  
        return "이름이 " + name + "이다.";  
    }  
}
```

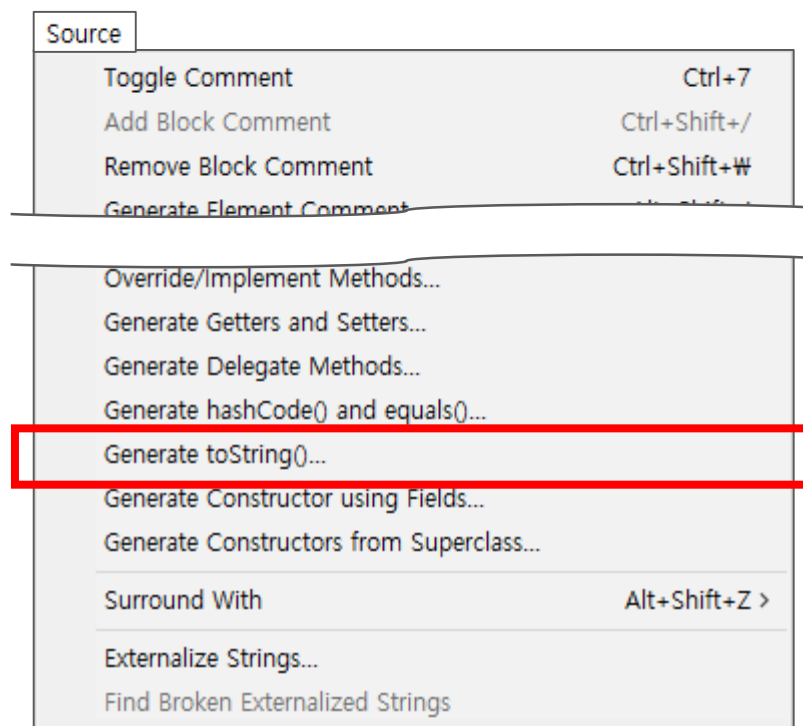
```
public class MainClass {  
    public static void main(String[] args) {  
        Person p = new Person("tom");  
        System.out.println(p);  
    }  
}
```

실행결과

이름이 tom이다.

# Generate toString()

- 이클립스는 자동으로 toString() 메소드를 오버라이드 해 줌
- [Source] - [Generate toString()...]



# Object 클래스 - equals() 메소드

- 현재 객체와 전달 받은 객체의 참조값을 비교하여 같으면 true 다르면 false 반환함
- 객체의 필드값을 비교하는 것이 아니라 객체의 참조값을 비교하는 한계를 가짐
- 원형

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

# Object 클래스 - equals() 메소드 호출

- 현재 객체와 비교할 객체를 인자값으로 사용

```
public class User {  
  
    private String id;  
  
    public User(String id) {  
        this.id = id;  
    }  
  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        User user1 = new User("tom");  
        User user2 = new User("tom");  
        System.out.println(user1.equals(user2));  
    }  
}
```

실행결과

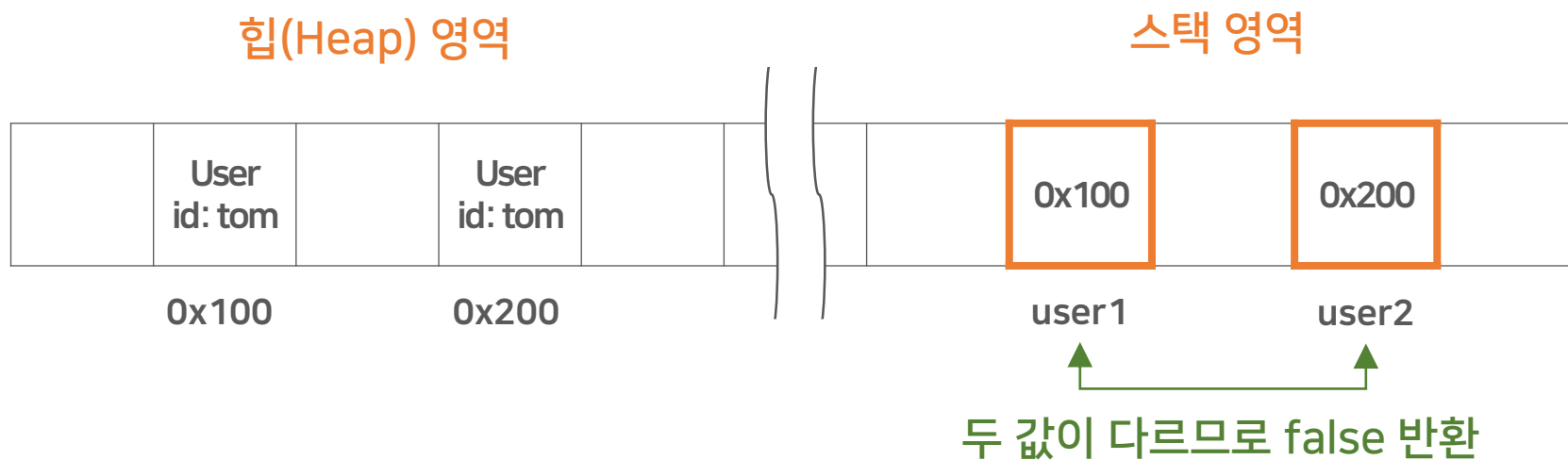
false

id가 모두 "tom"인데 왜 다르다고 할까?

# Object 클래스 - equals() 메소드 동작

- equals() 메소드는 객체의 참조값 자체를 비교함

Object 클래스의 equals() 메소드는 스택 영역의 참조값 자체를 비교한다.



# Object 클래스 - equals() 메소드 오버라이드

- 객체 동등 비교 방법을 바꾸려면 equals() 메소드를 오버라이드 해야 함

```
public class User {  
  
    private String id;  
  
    public User(String id) {  
        this.id = id;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        return id.equals(((User)obj).id);  
    }  
}
```

id가 같으면 true 반환

```
public class MainClass {  
    public static void main(String[] args) {  
        User user1 = new User("tom");  
        User user2 = new User("tom");  
        System.out.println(user1.equals(user2));  
    }  
}
```

실행결과

true

# Generate hashCode() and equals()

- 이클립스는 자동으로 equals() 메소드를 오버라이드 해 줌
- [Source] - [Generate hashCode() and equals()...]

