J05-07. 클래스와 객체

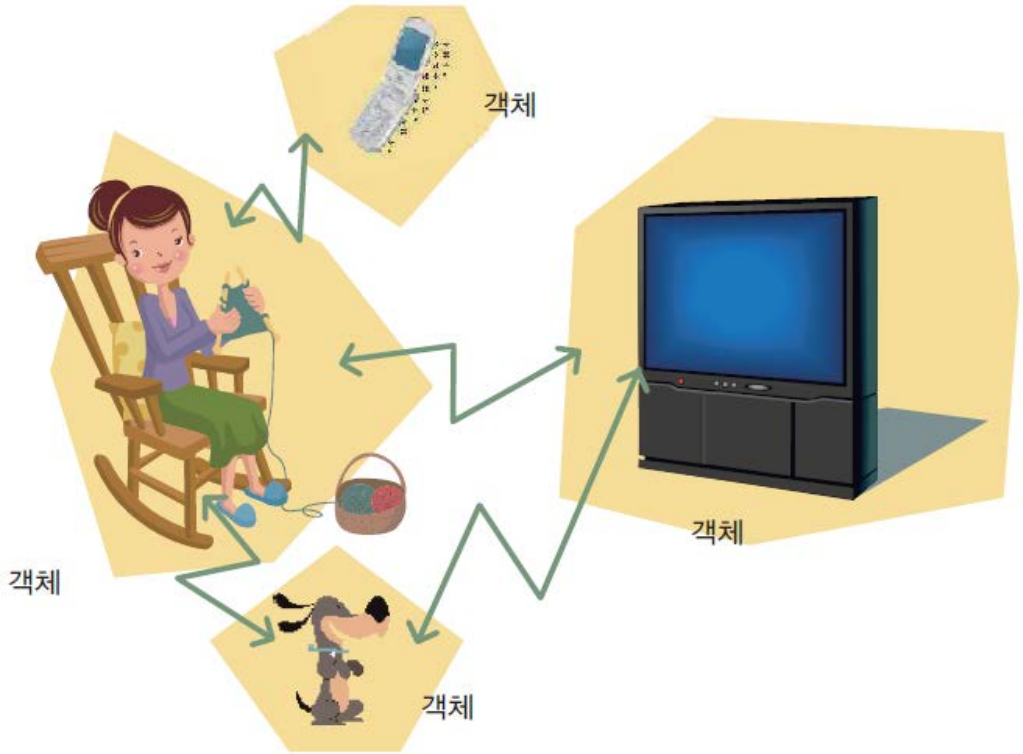
충남대학교
컴퓨터공학과

학습 내용

- 객체지향의 이해
  - 객체
  - 메시지
  - 클래스
  - 객체지향의 장점
  - String 클래스
  - 객체의 일생

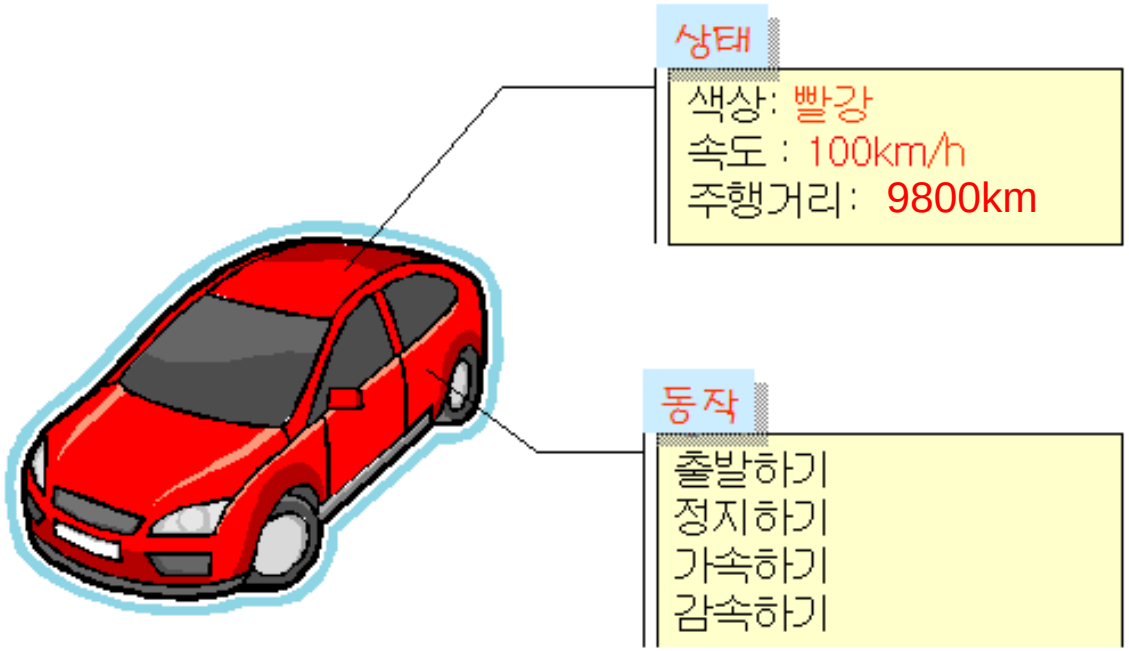
우리가 사는 세상

- 객체들로 이루어져 있다.
- 객체들 사이에 상호작용이 이루어진다.



객체란?

- 객체(Object)는 상태와 동작을 가지고 있다.
  - 객체의 상태(state): 객체의 특징값(속성)
  - 객체의 동작(behavior) 또는 행동: 객체가 취할 수 있는 동작

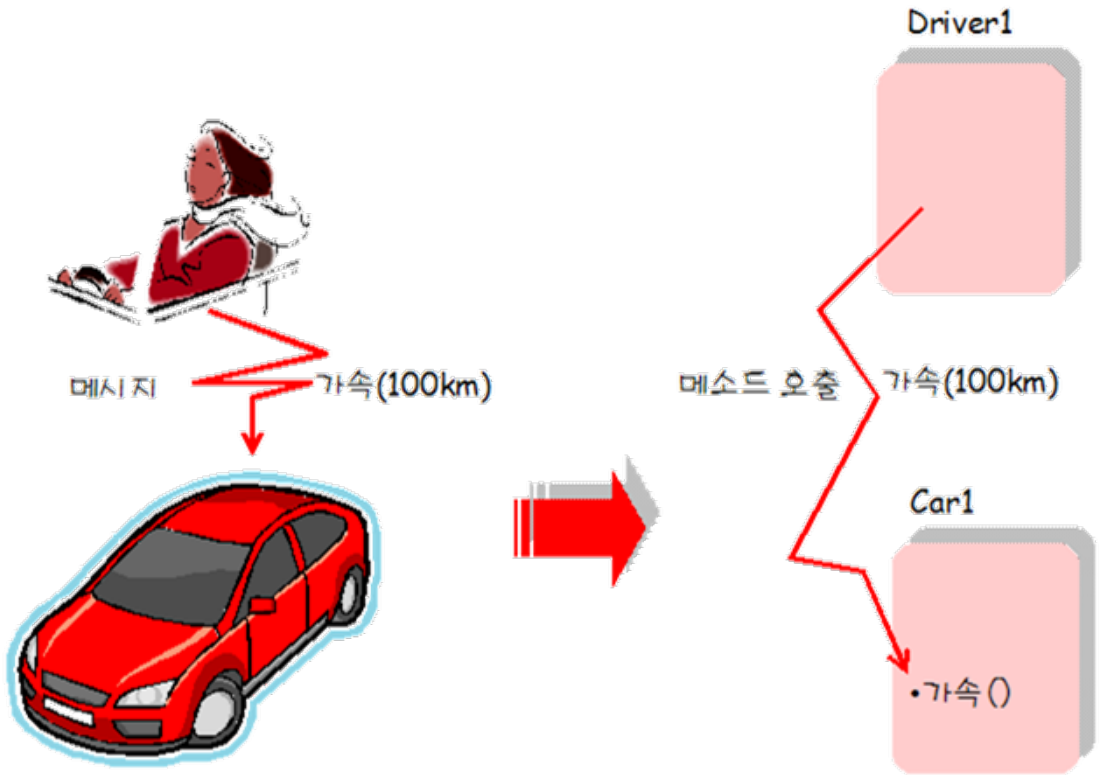


객체의 예

객체	상태	동작
전 구	종류, 색상	켜지기, 꺼지기
라디오	색상	켜지기, 꺼지기
강아지	종류	짖기, 달리기, 물기
자전거	종류, 색상	출발하기, 정지하기, 가속하기, 감속하기
사 자	성별	짖기, 달리기

메시지

- 객체는 메시지(message)를 통해 다른 객체와 통신하고 서로 상호 작용한다.



클래스와 객체

- 클래스(class): 객체를 만드는 설계도
- 클래스로부터 만들어지는 각각의 객체를 특별히 그 클래스의 **인스턴스(instance)**라고도 한다.

자동차 클래스

상태

색상: 빨강
속도 : 100km/h
주행거리: 9800km

동작

가속하기
감속하기

```
public class Car
{
        int color;          // 필드
        int speed;
        int mileage;

        void speedUp(int s) {       // 메소드
                speed += s;
        }
        void speedDown(int s) {
                speed -= s;
        }
}
```

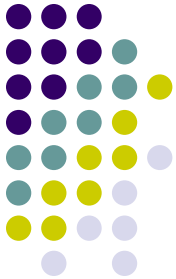
클래스 내부

Key Point

클래스 내부는 상태를 나타내기 위해 필요한 데이터(data)를 선언하는 필드와 행동을 정의하는 메소드(method) 선언들로 구성된다.

- 데이터는 상태를 표현하기 위한 상수, 변수 등을 의미하고
- 메소드는 그 객체의 행동을 표현하는 함수 혹은 프로시저라고 할 수 있다.

클래스의 구조



클래스 이름

```
class 클래스이름 {
    // 필드 정의
    int field1;
    ...

    // 메소드 정의
    void method1(parameter) { ... }
    ...
}
```

필드 정의:
객체의 속성을 나타낸다.

메소드 정의:
객체의 동작을 나타낸다.

클래스의 예

```
class Car {

    String color;    // 모델
    int speed;       // 현재 속도
    int gear;        // 현재 기어 단수

    void print() {
        System.out.println("( " + color + ", " + speed + ", " + gear + " )");
    }
}
```

필드 정의:
객체의 속성을 나타낸다.

메소드 정의:
객체의 동작을 나타낸다.

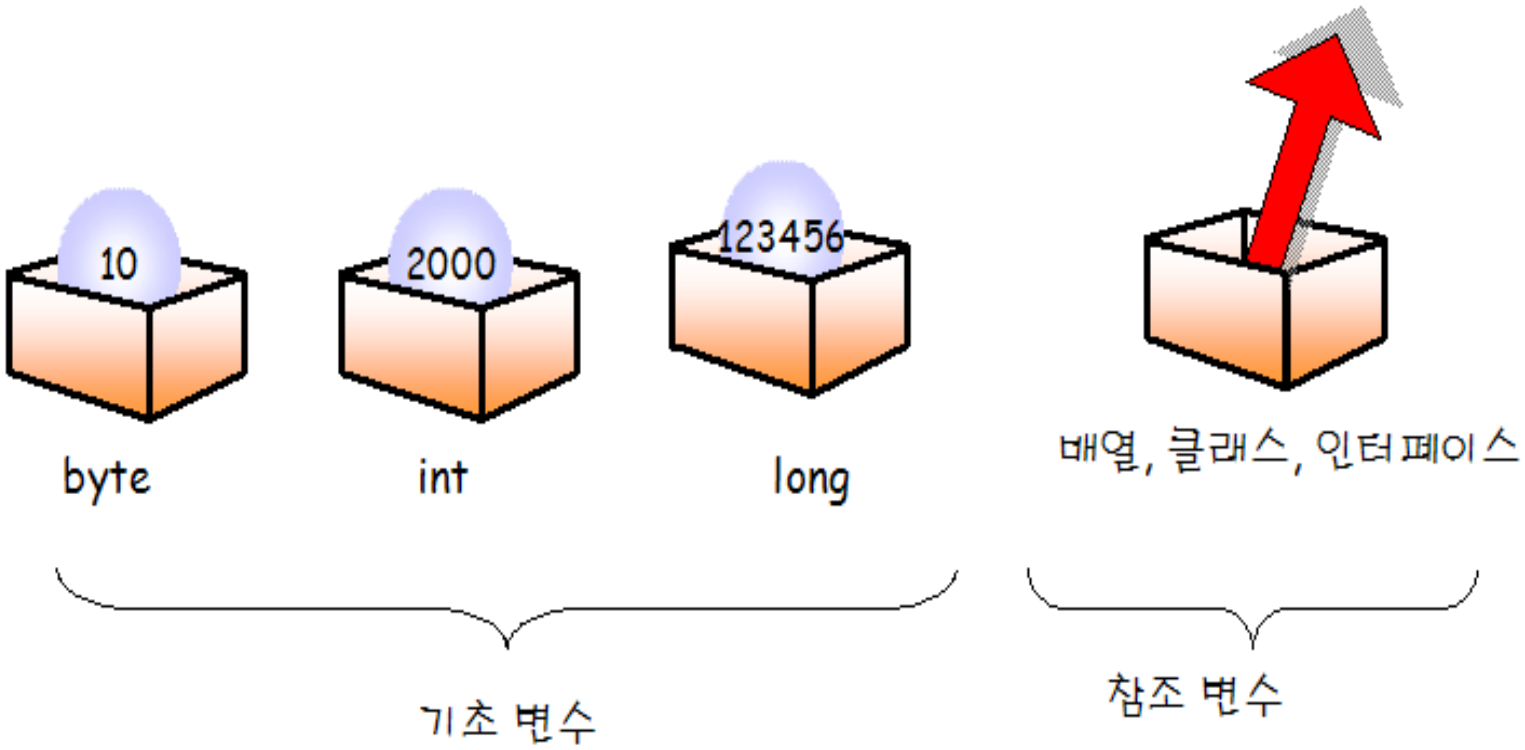
```
class Box2 {
    int width;
    int length;
    int height;

    int calVolume()
    {
        return width*length*height;
    }
}
```

필드 정의:
객체의 속성을 나타낸다.

메소드 정의:
객체의 동작을 나타낸다.

기초 변수와 참조 변수

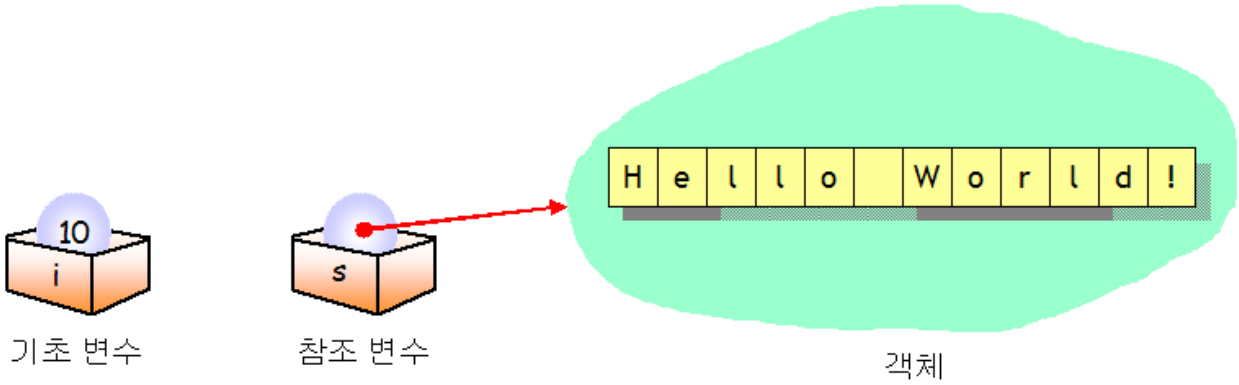


기초 변수와 참조 변수의 비교

```
int    i;      // 기초 변수
String s;      // 참조 변수
```

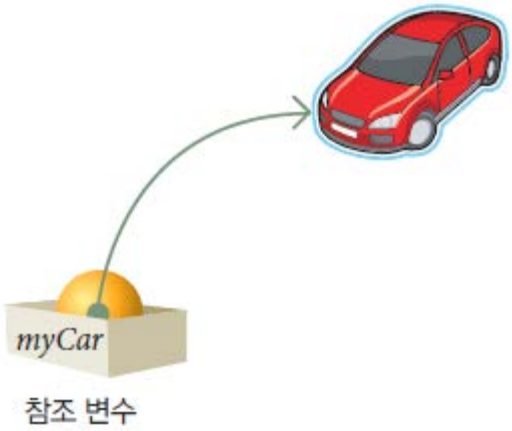
위와 같이 정의하면 두 변수 모두 처음에는 데이터를 담고 있지 않다. 즉 초기화가 되지 않은 상태이다.

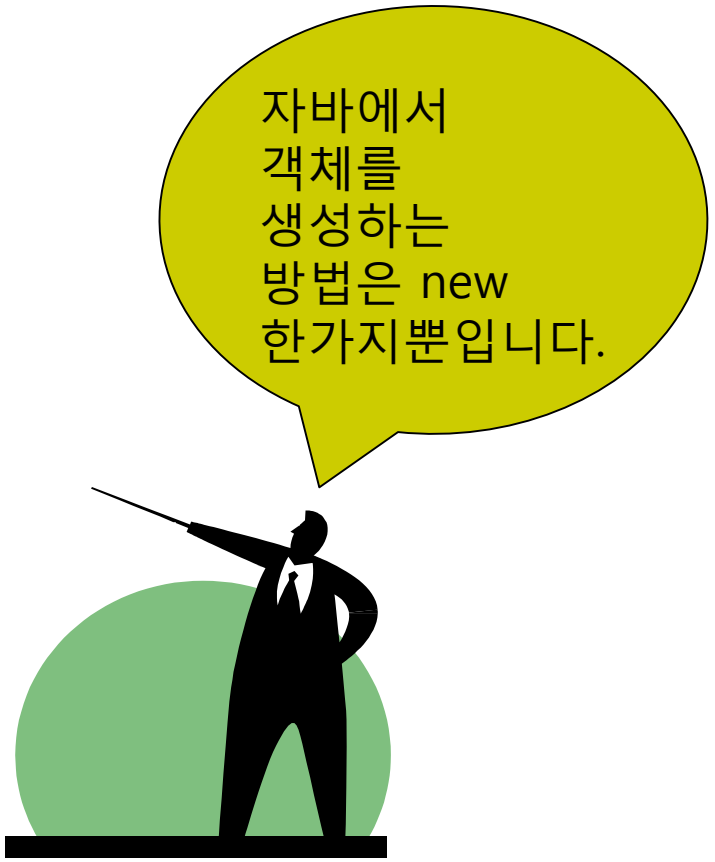

```
i = 10;                                  // 기초 변수에 값을 대입
s = new String("Hello World!"); // 객체를 생성하고 참조 변수에 객체의 주소를 대입
```



객체를 생성하려면

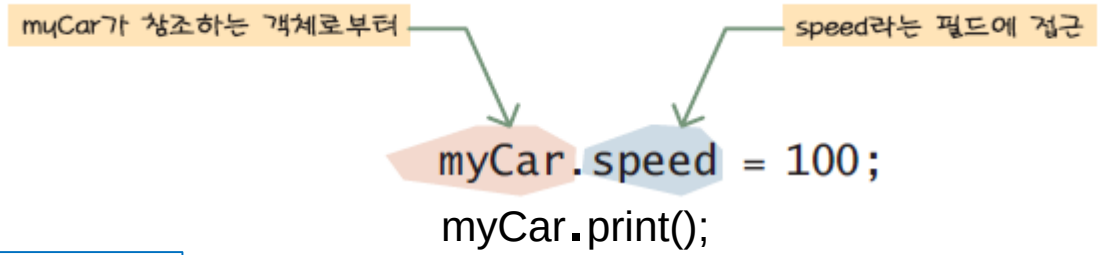
- Car myCar = **new** Car( );
- Car myCar;
  myCar = **new** Car( );





객체의 필드와 메소드 접근

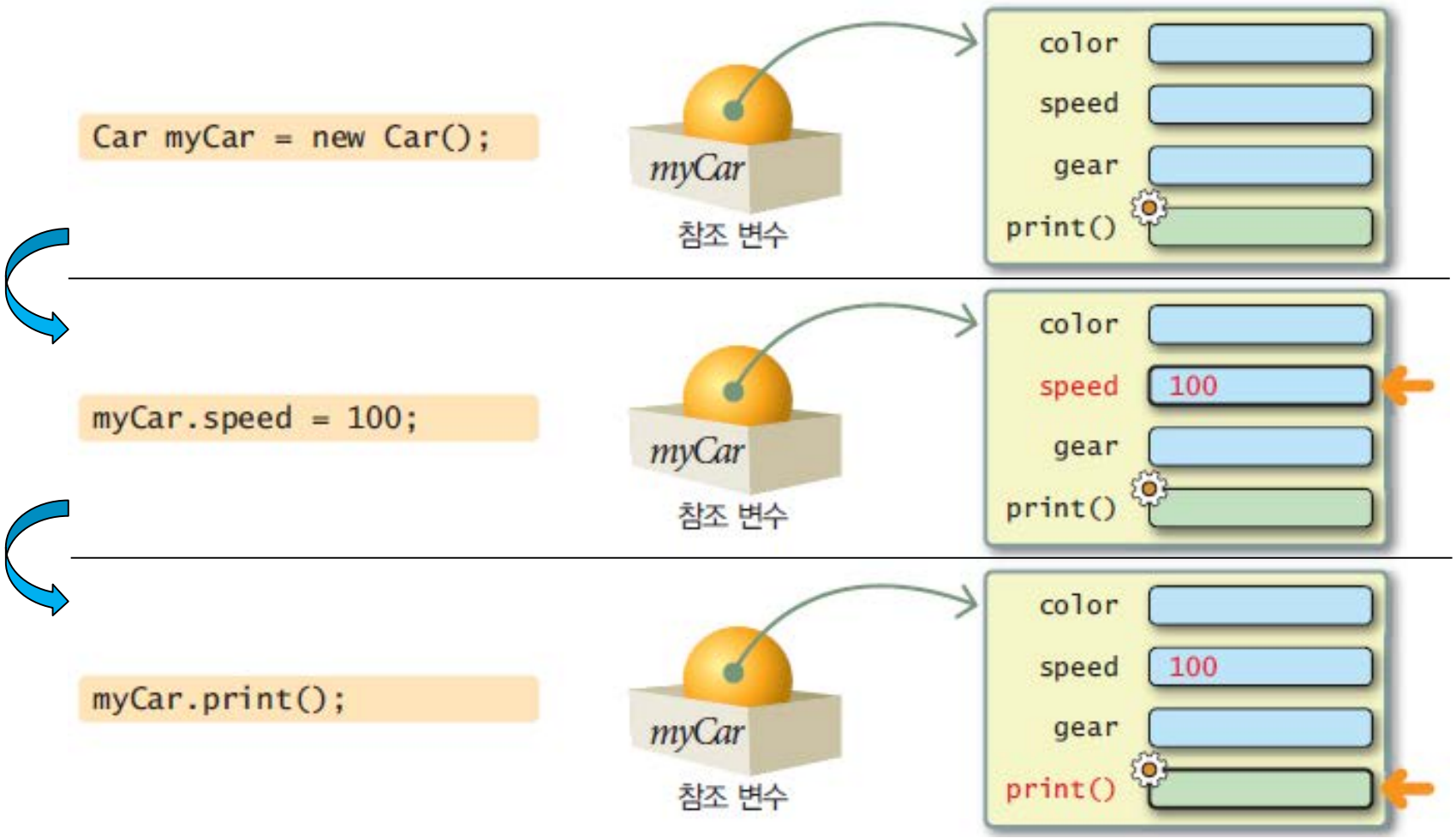
- 도트(.) 연산자




```
public class Car {
    String  color;
    int   speed;
    int   gear;
    ……
    void print() {
        System.out.println(……);
    }
}
```

객체의 멤버를 사용하려면 도트(.) 연산자 사용

필드와 메소드 접근


Car myCar = new Car();
myCar
참조 변수
color
speed
gear
print()
myCar.speed = 100;
myCar
참조 변수
color
speed
100
gear
print()
myCar.print();
myCar
참조 변수
color
speed
100
gear
print()

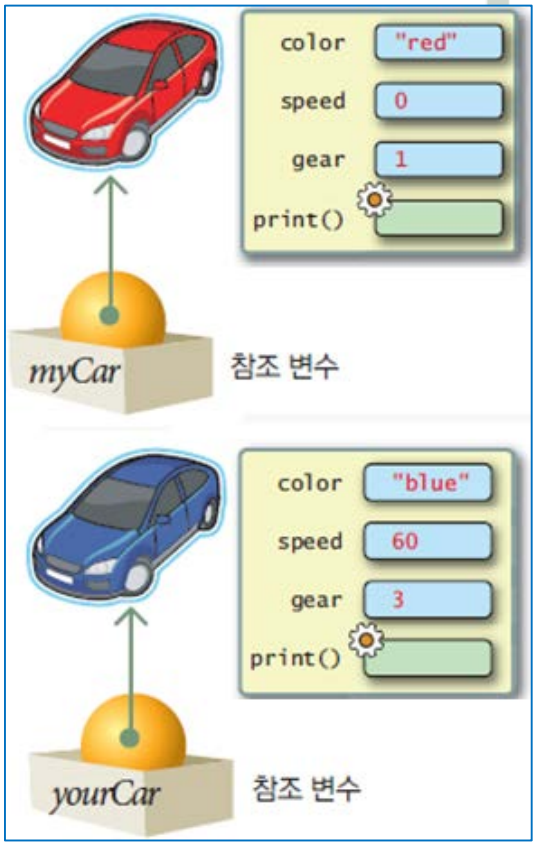
객체 생성 코드

클래스

```
01  class Car {
02      // 필드 정의
03      String color;    // 색상
04      int speed;       // 현재 속도
05      int gear;        // 현재 기어
06      void print() {
07          System.out.println("( " + color + ", " + speed + ", " + gear + " )");
08      }
09  }
10  class CarTest {
11      public static void main(String[] args) {
12
13          Car myCar = new Car();          // 객체 생성
14          myCar.color = "red";            // 객체의 필드 변경
15          myCar.speed = 0;                // 객체의 필드 변경
16          myCar.gear = 1;                 // 객체의 필드 변경
17          myCar.print();                  // 객체의 메소드 호출
18
19          Car yourCar = new Car();        // 객체 생성
20          yourCar.color = "blue";         // 객체의 필드 변경
21          yourCar.speed = 60;             // 객체의 필드 변경
22          yourCar.gear = 3;               // 객체의 필드 변경
23          yourCar.print();                // 객체의 메소드 호출
24      }
25  }
```

객체

객체



객체의 일생

객체의 생성

객체의 사용

객체의 파괴

Car c = new Car();

c.speedUp();

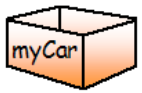
c = null;

객체의 생성

```
Car     myCar;                  // ① 참조 변수를 선언
myCar = new  Car();             // ② 객체를 생성하고 ③ 참조값을 myCar에 저장
```

① 참조 변수 선언

Car 타입의 객체를 참조할 수 있는 변수 myCar를 선언한다.



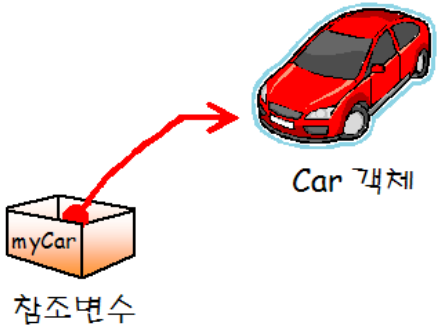
② 객체 생성

new 연산자를 이용하여 객체를 생성하고 객체 참조값을 반환한다.

Car 객체

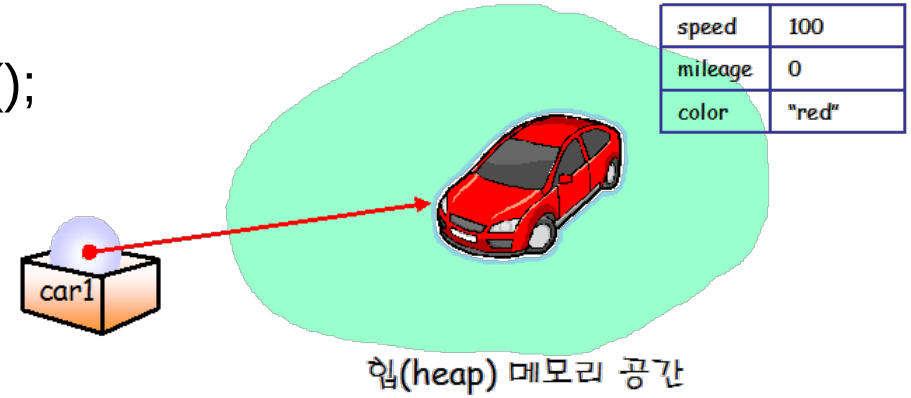
③ 참조 변수와 객체의 연결

생성된 새로운 객체의 참조값을 myCar라는 참조 변수에 대입한다.

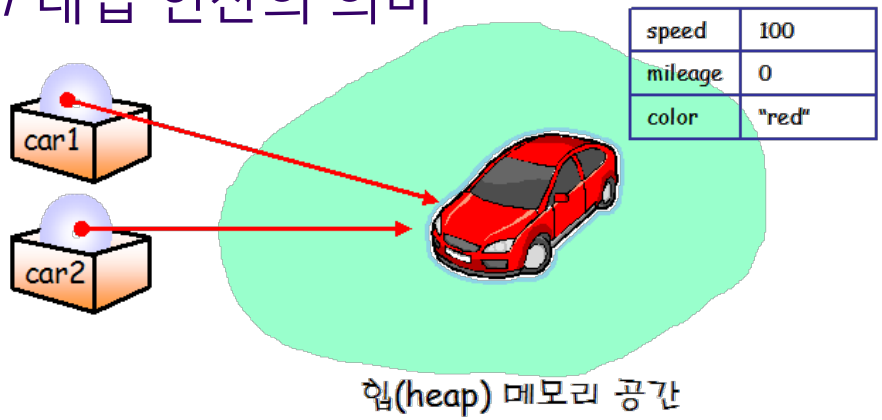


객체의 대입 연산

- Car car1 = new Car();

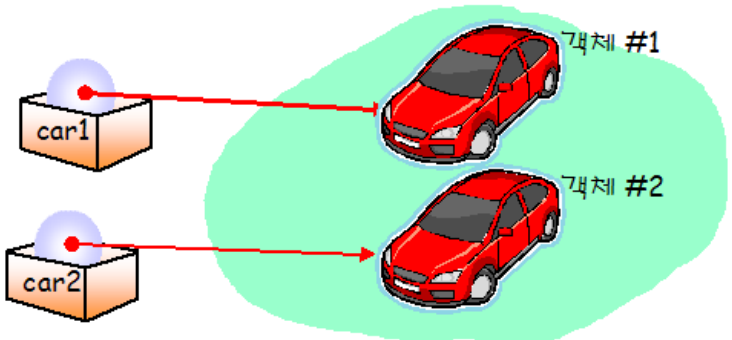


- Car car2 = car1; // 대입 연산의 의미

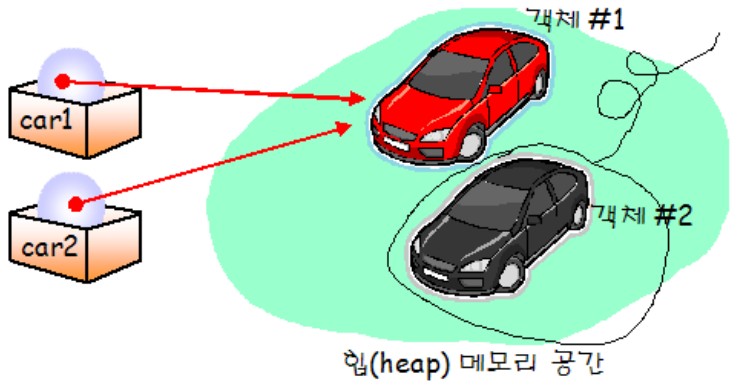


참조와 객체의 소멸

```
Car car1 = new Car();    // 첫번째 객체
Car car2 = new Car();    // 두번째 객체
```




```
car2 = car1;    // car1과 car2는 같은 객체를 가리킨다.
                // car2가 가리켰던 객체는 쓰레기 수집기에 의하여 수거된다.
```

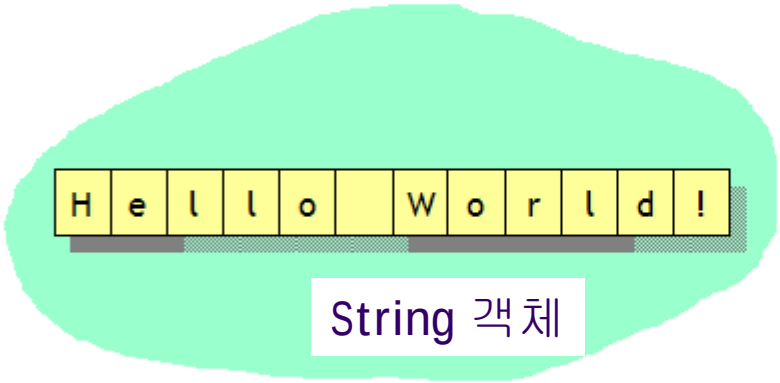


객체는 참조가 없어지면 소멸!!

null 참조

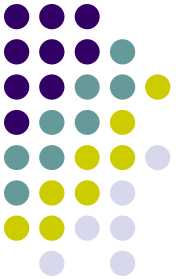
```
car1 = null;     // 객체 1은 아직도 활성화된 참조가 있기 때문에 소멸되지 않는다.
```

문자열 객체



객체

클래스에서 객체를 생성하는 방법



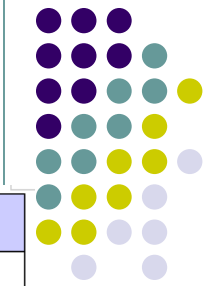
- 단 하나의 방법만이 존재한다.
- String s = new String("Hello World");

- **실체화(Instantiation)**
  - new 연산자를 이용하여 객체를 생성하는 것을 말하며
  - 객체는 클래스의 **실체(instance)**라고 한다.

객체의 메소드의 호출

- String s = "Hello World!";
- **int** size = s.length(); // size는 12가 된다.

String 클래스의 메소드



메소드 요약	
char	**charAt**(int index) 지정된 인덱스에 있는 문자를 반환한다.
int	**compareTo**(String anotherString) 사전적 순서로 문자열을 비교한다.앞에 있으면 -1, 같으면 0, 뒤에 있으면 1이 반환된다.
String	**concat**(String str) 주어진 문자열을 현재의 문자열 뒤에 붙인다.
boolean	**equals**(Object anObject) 주어진 객체와 현재의 문자열을 비교한다.
boolean	**equalsIgnoreCase**(String anotherString) 대소문자를 무시하고 비교한다.
boolean	**isEmpty**() length()가 0이면 true를 반환한다.
int	**length**() 현재 문자열의 길이를 반환한다.
String	**replace**(char oldChar, char newChar) 주어진 문자열에서 oldChar를 newChar로 변경한, 새로운 문자열을 생성하여 반환한다.
String	**substring**(int beginIndex, int endIndex) 현재 문자열의 일부를 반환한다.
String	**toLowerCase**() 문자열의 문자들을 모두 소문자로 변경한다.
String	**toUpperCase**() 문자열의 문자들을 모두 대문자로 변경한다.

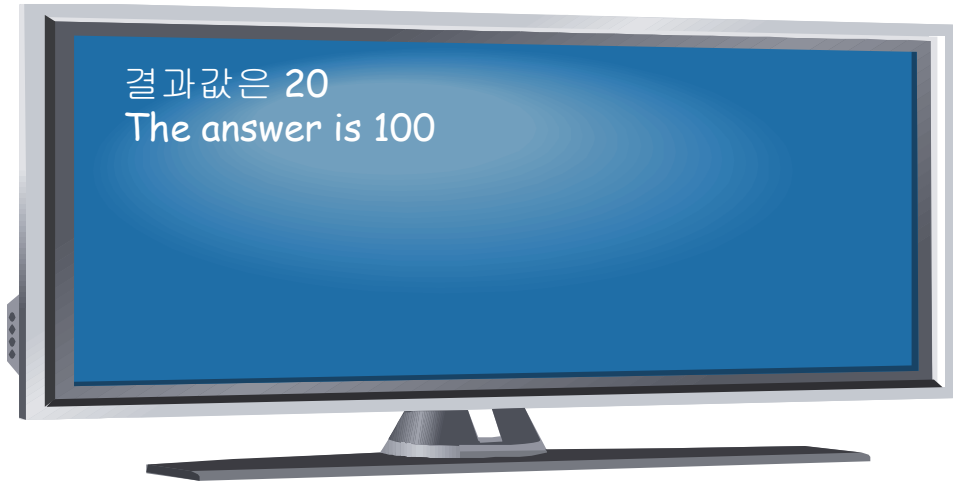
문자열의 결합

- 두 개의 문자열은 + 연산자를 이용하여 결합될 수 있다.

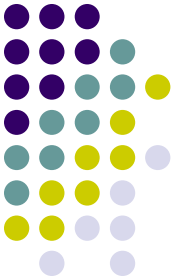
- String subject = "Money";
- String other = " has no value if it is not used";
- String sentence = subject + other;

숫자를 문자열로 변환

- **int** x = 20;
- System.out.println("결과값은 " + x);
  // "결과값은 20" 이 출력된다.
- String answer = "The answer is " + 100;
  // "The answer is 100"



문자열 관련 메소드 사용의 예



StringTest.java

```
public class StringTest
{
   public static void main (String[] args)
   {
      String proverb = "A barking dog";          // new 연산자 생략
      String s1, s2, s3, s4;      // 참조 변수로서 메소드에서 반환된 참조값을 받는다.

      System.out.println ("문자열의 길이 =" + proverb.length());

      s1 = proverb.concat (" never Bites!");       // 문자열 결합
      s2 = proverb.replace ('B', 'b');            // 문자 교환
      s3 = proverb.substring (2, 5);              // 부분 문자열 추출
      s4 = proverb.toUpperCase();                 // 대문자로 변환

      System.out.println(s1);
      System.out.println(s2);
      System.out.println(s3);
      System.out.println(s4);
   }
}
```

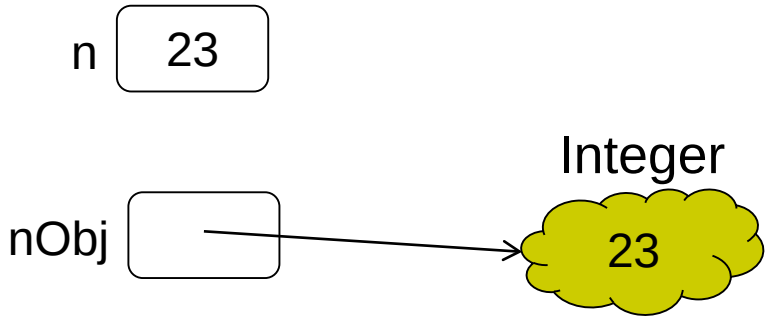
출력결과

```
문자열의 길이 =13
A barking dog never Bites!
A barking dog
bar
A BARKING DOG
```

문자열을 숫자로 변환하기

- 문자열 “123”을 숫자 123으로 변환
- 랩퍼 클래스(wrapper class)를 사용한다.
  - (예) Integer obj = new Integer(10);
- Wrapper Class
  - 기본 타입의 값을 객체의 값으로 가질 수 있는 class 타입으로 포장해 준다.

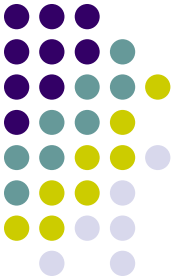
```
int  n = 23;
Integer  nObj = new Integer(23);
```



기본 타입의 Wrapper Class

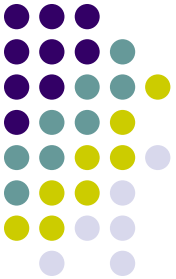
기초 자료형	랩퍼 클래스
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

Integer 클래스의 유용한 메소드



반환값	메소드 이름	설명
byte	byteValue()	int형을 byte형으로 변환한다.
double	doubleValue()	int형을 double형으로 변환한다.
float	floatValue()	int형을 float형으로 변환한다.
int	parseInt(String s)	문자열을 int형으로 변환한다.
String	toBinaryString(int i)	int형의 정수를 2진수 형태의 문자열로 변환한다.
String	toHexString(int i)	int형의 정수를 16진수 형태의 문자열로 변환한다.
String	toOctalString(int i)	int형의 정수를 8진수 형태의 문자열로 변환한다.

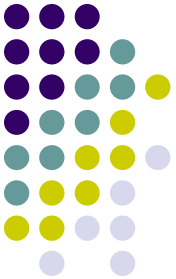
문자열을 숫자로 변환하기 – 예



- **int** i = Integer.parseInt("123");
  - 변수 i에 정수 123이 저장된다.

- **double** d = Double.parseDouble("3.141592");
  - 변수 d에 실수 3.141592가 저장된다.

객체 지향의 3대 특징



- 캡슐화
- 상속
- 다형성

캡슐화

- **캡슐화(encapsulation)**: 관련된 데이터와 알고리즘(코드)이 하나의 묶음으로 정리되어 있는 것

그림 7-4 . 객체 지향을 사용하면 코드를 재사용할 수 있다.

상속

- 상속(inheritance): 이미 작성된 클래스(부모 클래스)를 이어받아서 새로운 클래스(자식 클래스)를 생성하는 기법
- 기존의 코드를 재활용하기 위한 기법

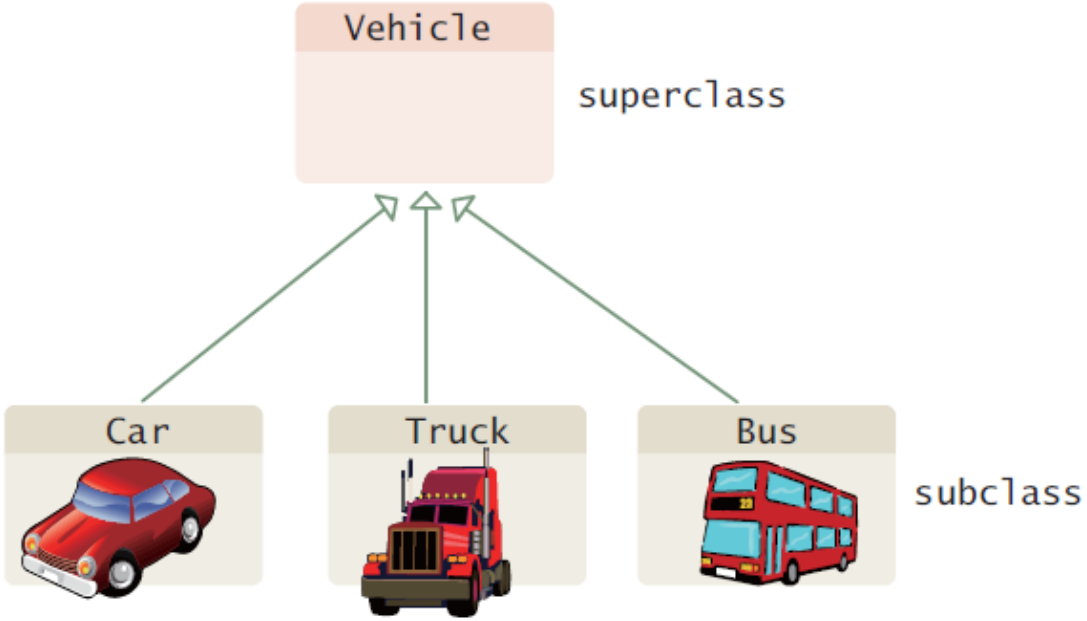
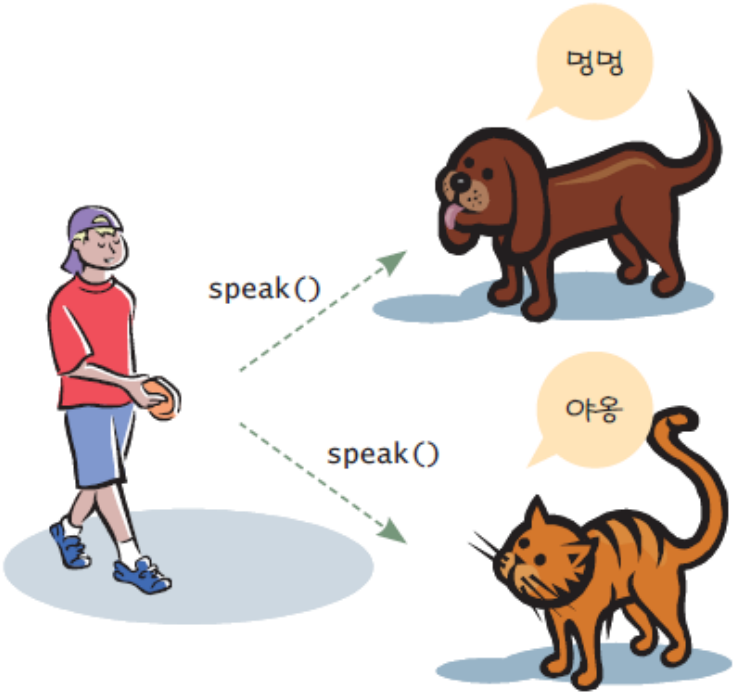


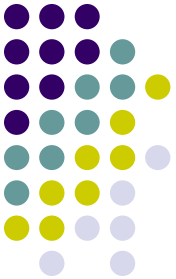
그림 7-6 . 상속의 개념

다형성

- 하나의 이름(방법)으로 많은 상황에 대처하는 기법
- 개념적으로 동일한 작업을 하는 멤버 함수들에 똑 같은 이름을 부여할 수 있으므로 코드가 더 간단해진다



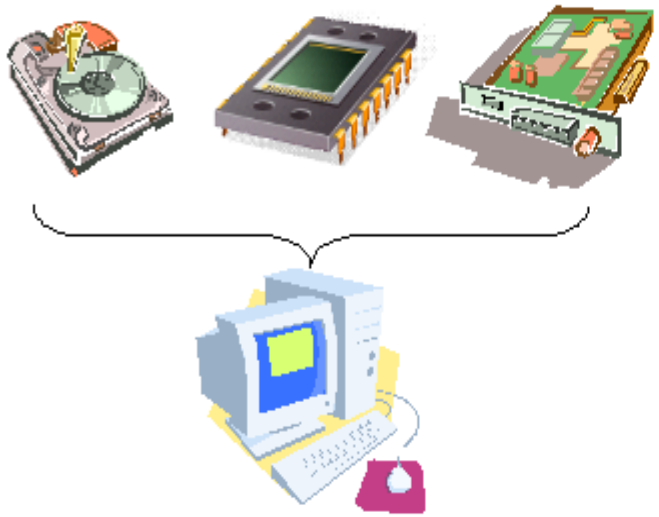
객체 지향의 장점

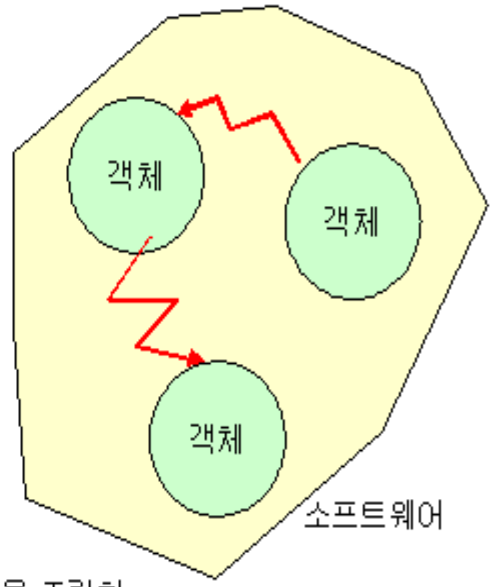


- 신뢰성있는 소프트웨어를 쉽게 작성할 수 있다.
- 코드를 재사용하기 쉽다.
- 업그레이드가 쉽다.
- 디버깅이 쉽다.

소프트웨어 작성이 쉽다

- 부품을 구입하여 컴퓨터를 조립하듯이 소프트웨어를 작성할 수 있다.

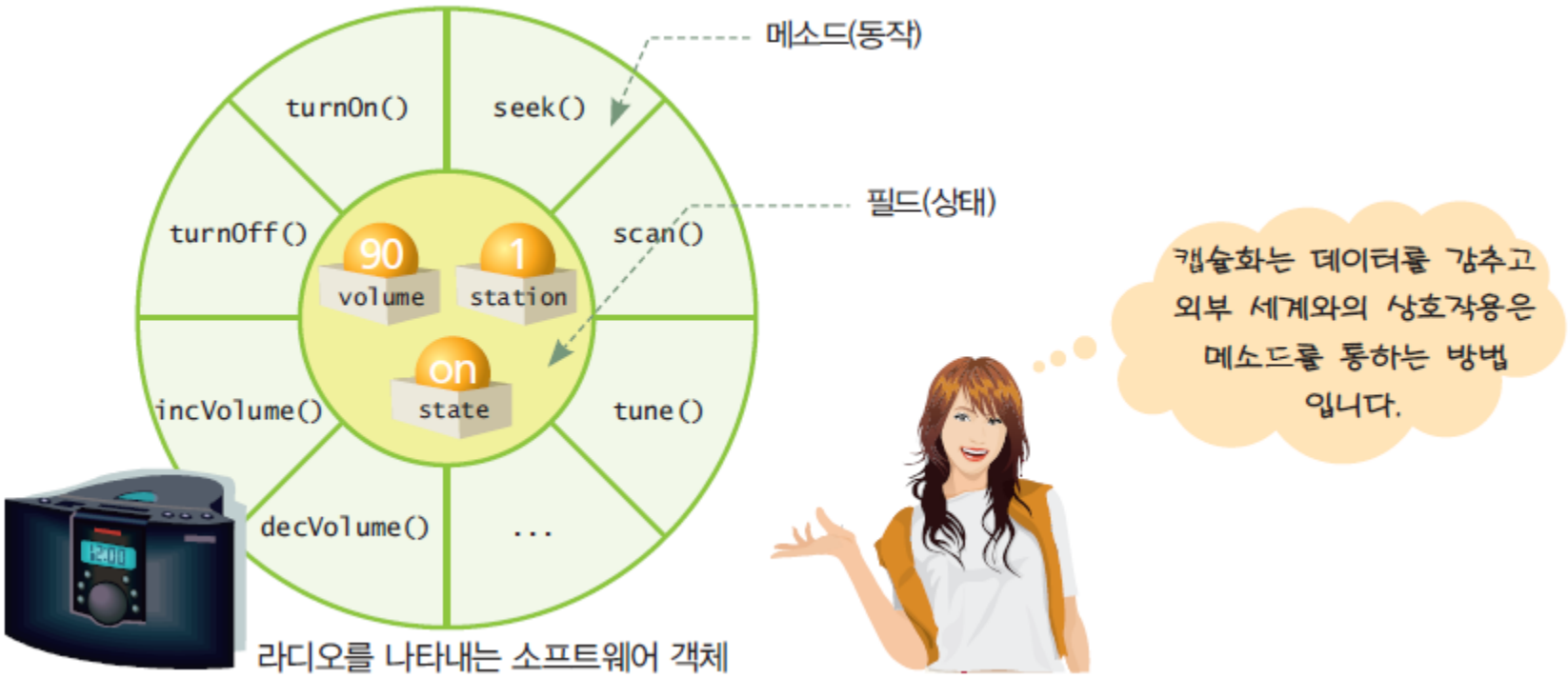




부품을 조립하여 제품을 만들듯이 객체들을 조립하여 소프트웨어를 만든다.

코드의 재사용

캡슐화와 정보 은닉

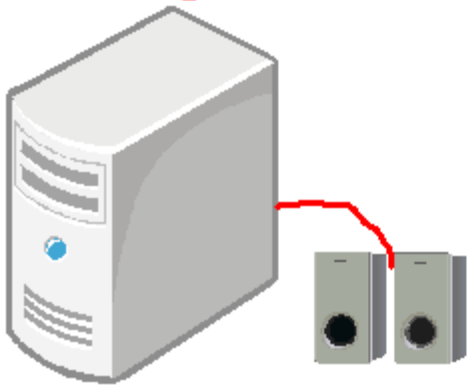


라디오를 나타내는 소프트웨어 객체

업그레이드가 쉽다.

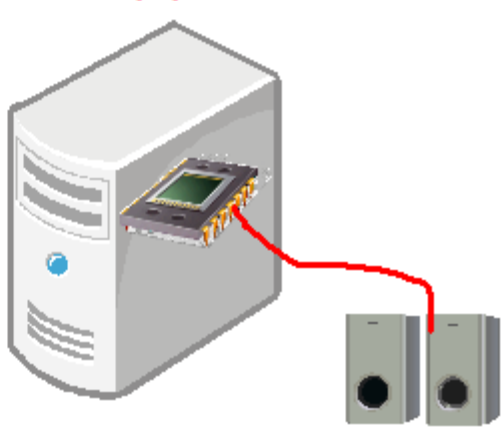
- 라이브러리가 업그레이드되면 쉽게 바꿀 수 있다.
- 정보 은닉이 가능하기 때문에 업그레이드 가능

O



만약 외부의 표준 오디오 단자를 이용하였으면 내부의 사운드 카드를 변경할 수 있다.

X



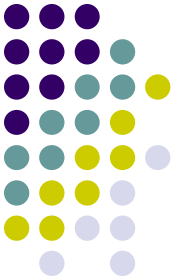
만약 내부의 오디오 제어칩의 단자에 연결하였으면 내부의 사운드 카드를 변경할 수 없다.

쉬운 디버깅

- 예를 들어서 절차 지향 프로그램에서 하나의 변수를 1000개의 함수가 사용하고 있다고 가정해보자. -> 하나의 변수를 1000개의 함수에서 변경할 수 있다.

- 객체 지향 프로그램에서 100개의 클래스가 있고 클래스당 10개의 메소드를 가정해보자. -> 하나의 변수를 10개의 메소드에서 변경할 수 있다.

- 어떤 방법이 디버깅이 쉬울까?



수고했습니다!!!