메서드와 생성자

- 메서드 종류
  - 인스턴스 메서드
    - 인스턴스명.메서드명
    - ex) myCar.getName();
  - static 메서드
    - 클래스이름.메서드명
    - ex) Car.getName();
  - abstract 메서드

- 생성자
  - 생성자는 메서드다.
  - 필드값의 설정(초기화)
  - 클래스 이름과 같은 메서드
  - 메서드 리턴 타입 사용 X

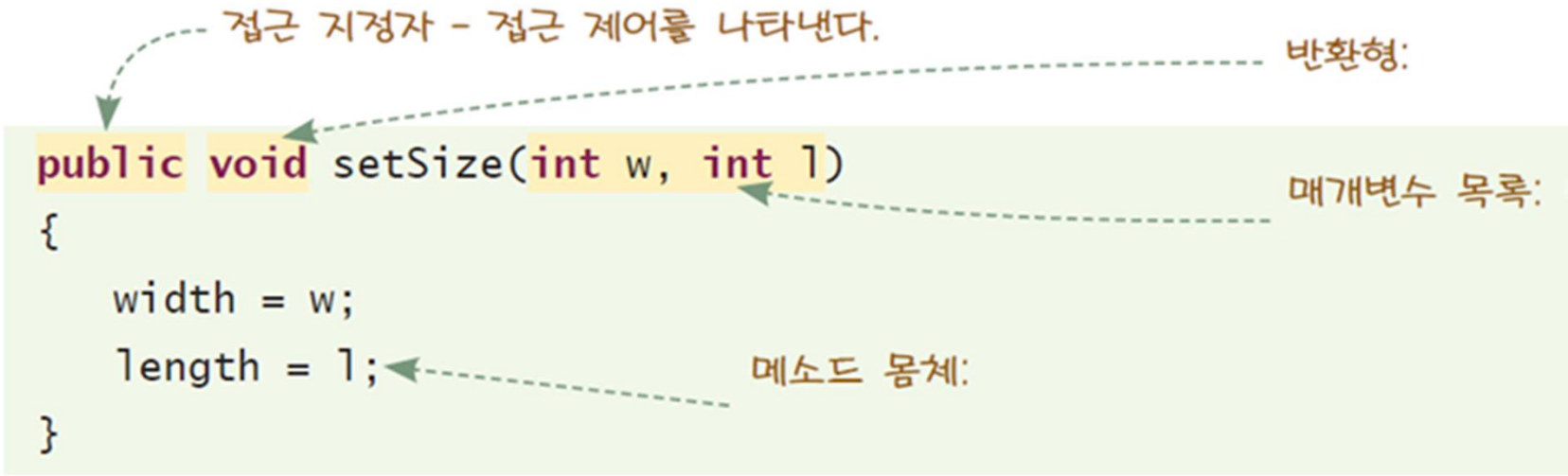
- 다형성 : 동명이메
  - overloading (수평)
  - overriding(수직)

- 메서드 구분 방법
  - **매개 변수의 개수**
  - **매개 변수의 타입**
  - **매개 변수의 순서**

- JVM 메모리 구조
  - 스택 영역 : 변수나 메서드
  - 힙 영역 : 인스턴스

메서드

- 메서드는 객체가 할 수 있는 **동작이나 행동을 나타낸다**.
- 클래스 안에 포함된 함수를 메서드라고 한다.
- 어떤 값을 입력 받아서 처리한 결과를 돌려준다.
  (입력 받는 값이 없을 수도 있고 결과를 돌려주지 않을 수도 있다.)



메서드

- 메서드의 장점과 작성지침
  - **반복적인 코드를 줄이고** 코드의 관리가 용이하다.
  - 반복적으로 수행되는 여러 문장을 메서드로 작성한다.
  - 하나의 메서드는 한 가지 기능만 수행하도록 작성하는 것이 좋다.
  - 관련된 여러 문장을 메서드로 작성한다.

메서드의 정의

▶ 메서드는 클래스 안에 메서드 밖에 정의한다.

```
메서드리턴타입 메서드명(매개변수, 매개변수, ...)
{
    // 메서드 호출시 수행될 코드. 반복되는 코드
    ...
}
```

선언부

정의부

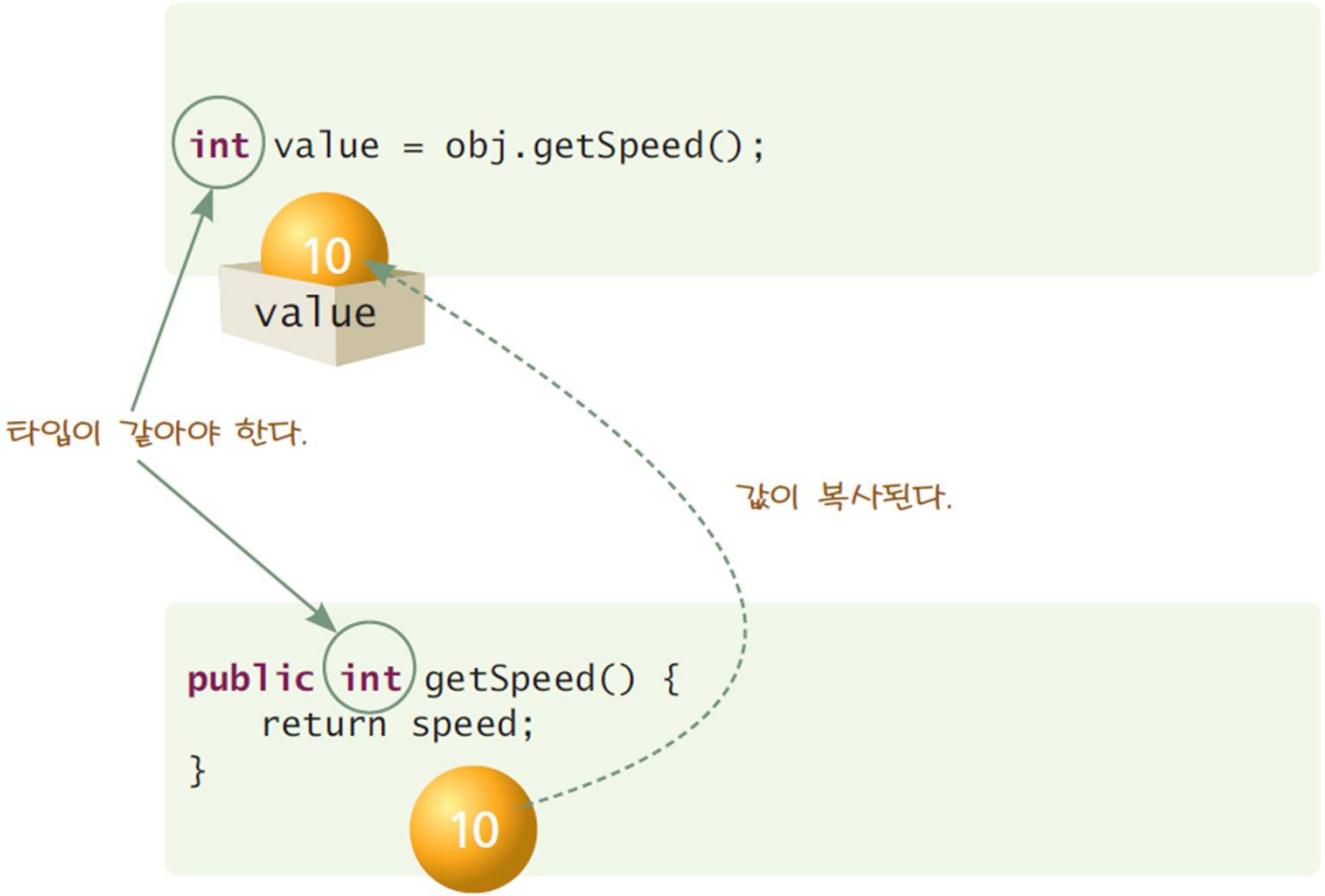
```
int add(int a, int b) // 반환값이 있는 경우 : 반환값의 타입 사용
{
    int result = a + b;
    return result; // 반환값의 타입과 메서드 리턴 타입이 동일
}
```

암기

```
void add( int x )     // 반환값이 없는 경우 : void 타입 사용
{
    System.out.println( x );
    return ;
}
```

메서드는 값을 반환할 수 있다.

- 메서드는 작업의 결과값을 반환할 수 있다.
- 반환값은 하나만 가능하다.



return문

- 메서드가 정상적으로 종료되는 경우
  - 메서드의 블럭{}의 끝에 도달했을 때
  - 메서드의 블럭{}을 수행 도중 return문을 만났을 때

- return문

  - 현재 실행 중인 메서드를 종료하고 호출한 메서드로 되돌아간다.

1. 반환값이 없는

retu

2. 반환값이 있는

retu

타입이 일치해야한다.

```
int add(int a, int b)
{
    int result = a + b;
    return result;
}
```

return문

- 반환값이 있는 메서드는 모든 경우에 return문이 있어야 한다.

```
int max(int a, int b) {
    if(a > b)
        return a;
}
```

```
int max(int a, int b) {
    if(a > b)
        return a;
    else
        return b;
}
```

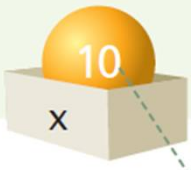
- return문의 개수는 최소화하는 것이 좋다.

```
int max(int a, int b) {
    if(a > b)
        return a;
    else
        return b;
}
```

```
int max(int a, int b) {
    int result = 0;
    if(a > b)
        result = a;
    else
        result = b;
    return result;
}
```

기본형 매개변수 vs 참조형 매개변수

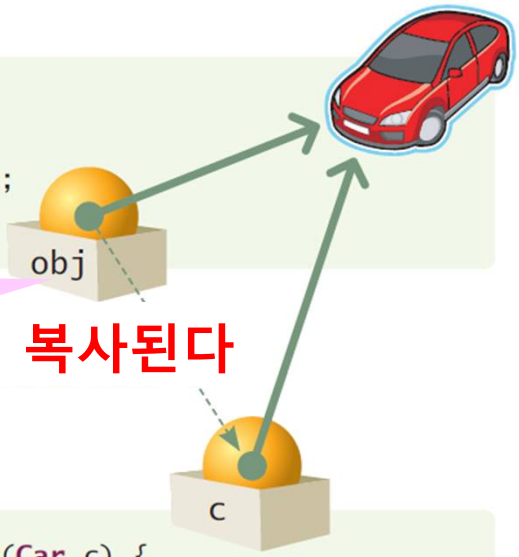

```
Car c = new Car();
x = 10;
c.speedUp(x);
```



실제값이 복사된다


```
public void speedUp(int increment) {
    speed += increment;
}
```

```
Car obj = new Car();
myCar.speedEquals(obj);
```



주소값이 복사된다

```
public int speedEquals(Car c) {
    if( speed == c.speed ) return true;
    else return false;
}
```

매개변수-가변 인수

- JDK 5부터 가변 길이 인수(variable-length arguments) 사용 가능

VarArgsTest.java

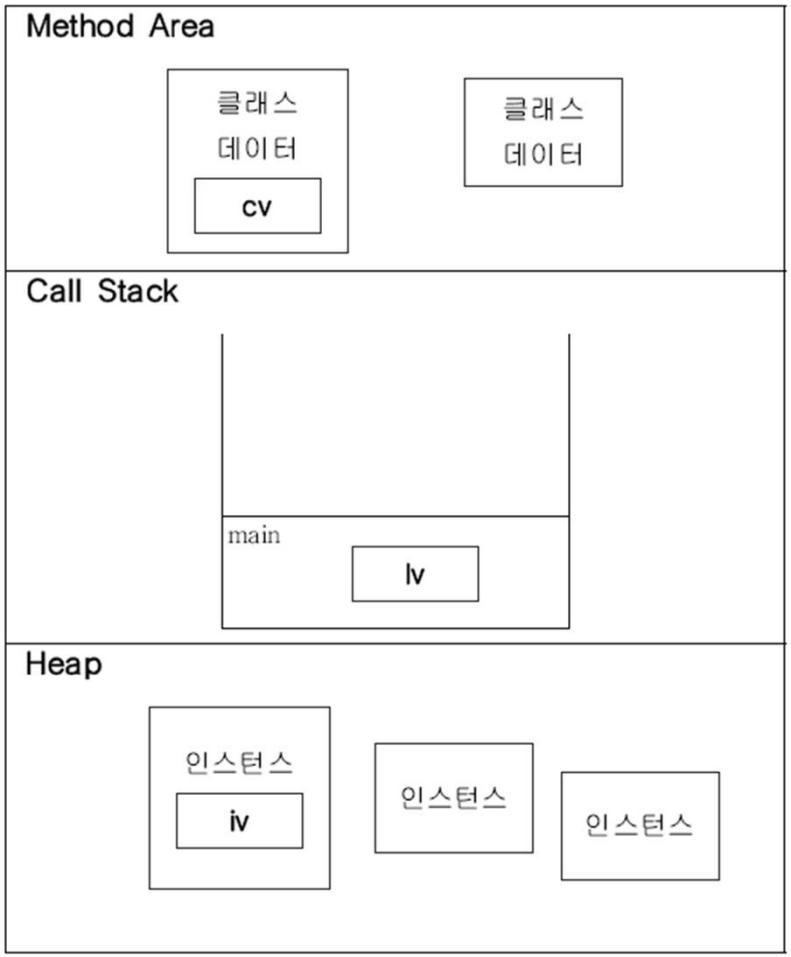
```
01  class Test {
02      void sub(int... v) {
03          System.out.println("인수의 개수 : " + v.length);
04          for (int x : v)
05              System.out.print(x + " ");
06          System.out.println();
07      }
08  }
09  public class VarArgsTest {
10      public static void main(String args[]) {
11          Test c = new Test();
12          c.sub(1);
13          c.sub(2, 3, 4, 5, 6);
14          c.sub();
15      }
16  }
```

가변 길이 인수로 몇 개의 인수라도 받을 수 있다.

실행결과

```
인수의 개수 : 1
1
인수의 개수 : 5
2 3 4 5 6
인수의 개수 : 0
```

JVM 메모리 구조



▶ 메서드영역(Method Area)

- static 변수가 저장되는 곳

▶ 호출스택(Call Stack)

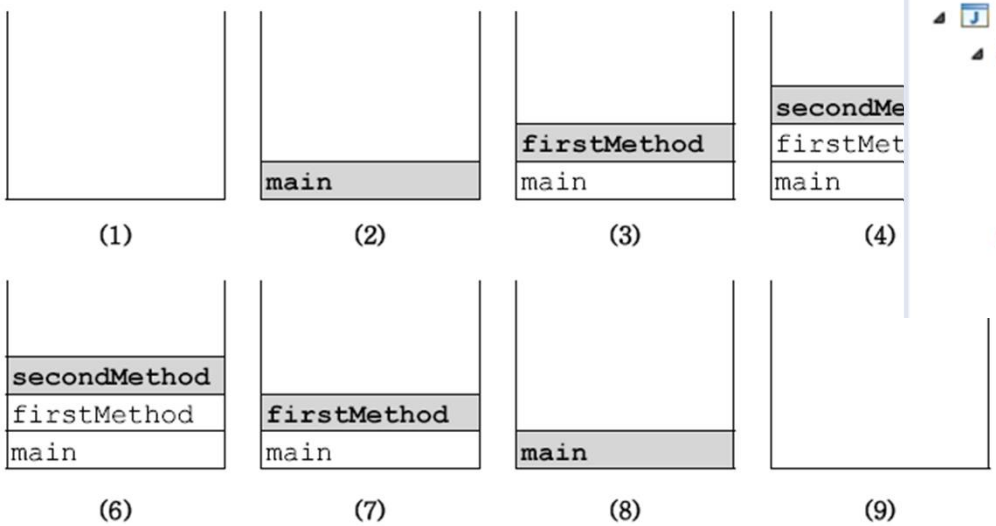
- 메서드의 작업공간.
- 메서드가 호출되면 메서드 수행에 필요한 메모리 공간이 할당
- 메서드가 종료되면 사용하던 메모리를 반환한다.

▶ 힙(Heap)

- 인스턴스가 생성되는 공간.
- new연산자에 의해서 생성되는 모든 객체
- 배열이나 참조형 타입이 여기에 생성된다.

JVM 메모리 구조 - 메서드 호출

```
class jv13_03_메서드메모리호출구조 {
    public static void main(String[] args) {
        firstMethod();
    }
    public static void firstMethod() {
        secondMethod();
    }
    public static void secondMethod() {
        System.out.println("secondMethod()");
    }
}
```



인스턴스메서드 vs static메서드

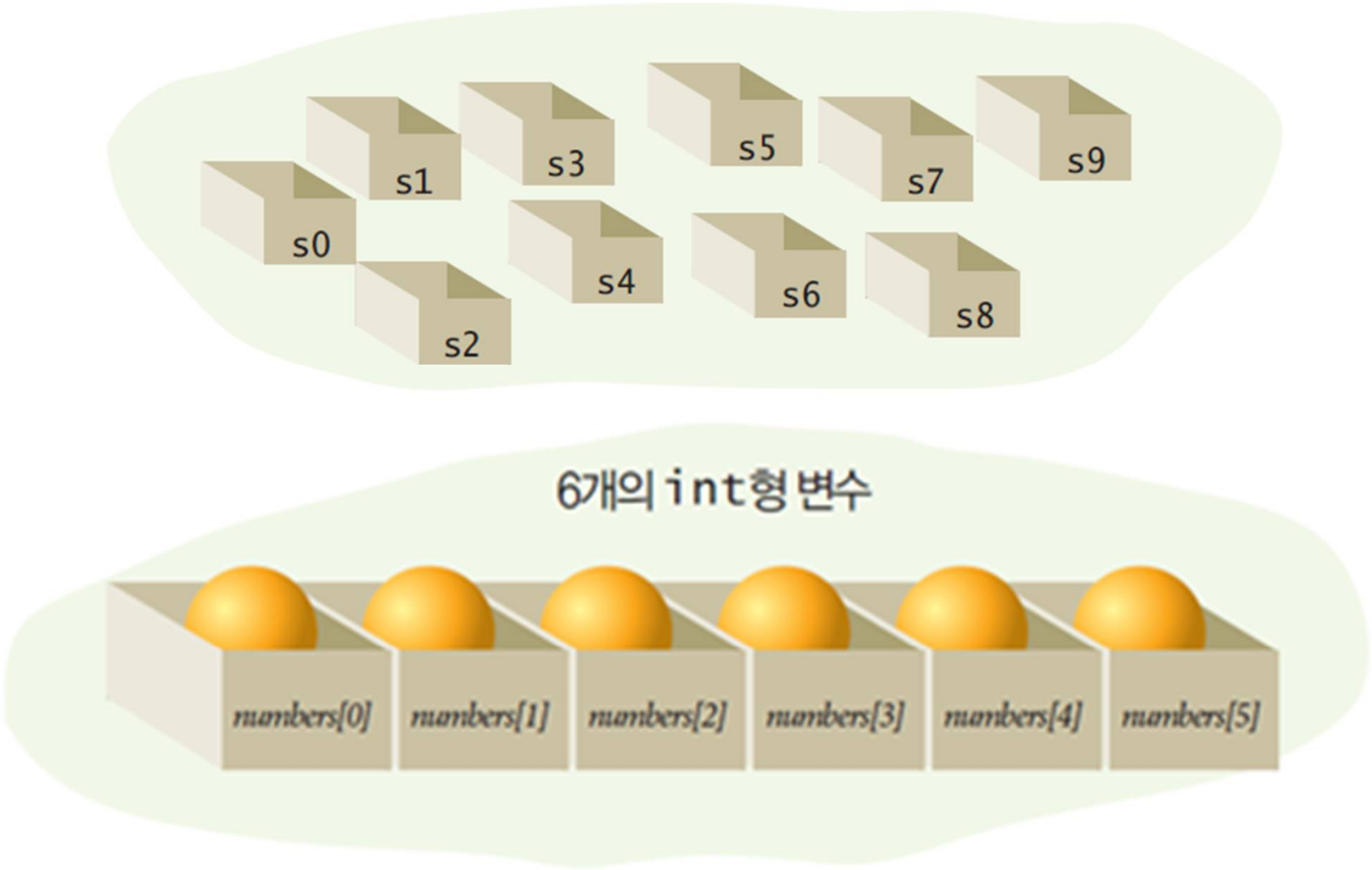
static 메서드는 인스턴스 메서드나 인스턴스 필드를 참조할 수 없다.

```
public class MethodType {
    private static int iCount = 0;
    private int nums = 0;

    public void instanceMethod() {
         iCount  = 100;
         nums = 100;
    }
    public void instanceMethod2() {
        staticMethod();
    }
    public static void staticMethod() {
        instanceMethod();  // 에러
    }
    public static void staticMethod2() {
        nums = 10 ;  // 에러
        staticMethod();
    }
}
```

```
public class MethodTypeTest {

    public static void main(String[] args) {

        MethodType instance = new MethodType();
        instance.instanceMethod();

        MethodType.staticMethod();

    }
}
```

▶ static은 static 끼리 놀아야 한다

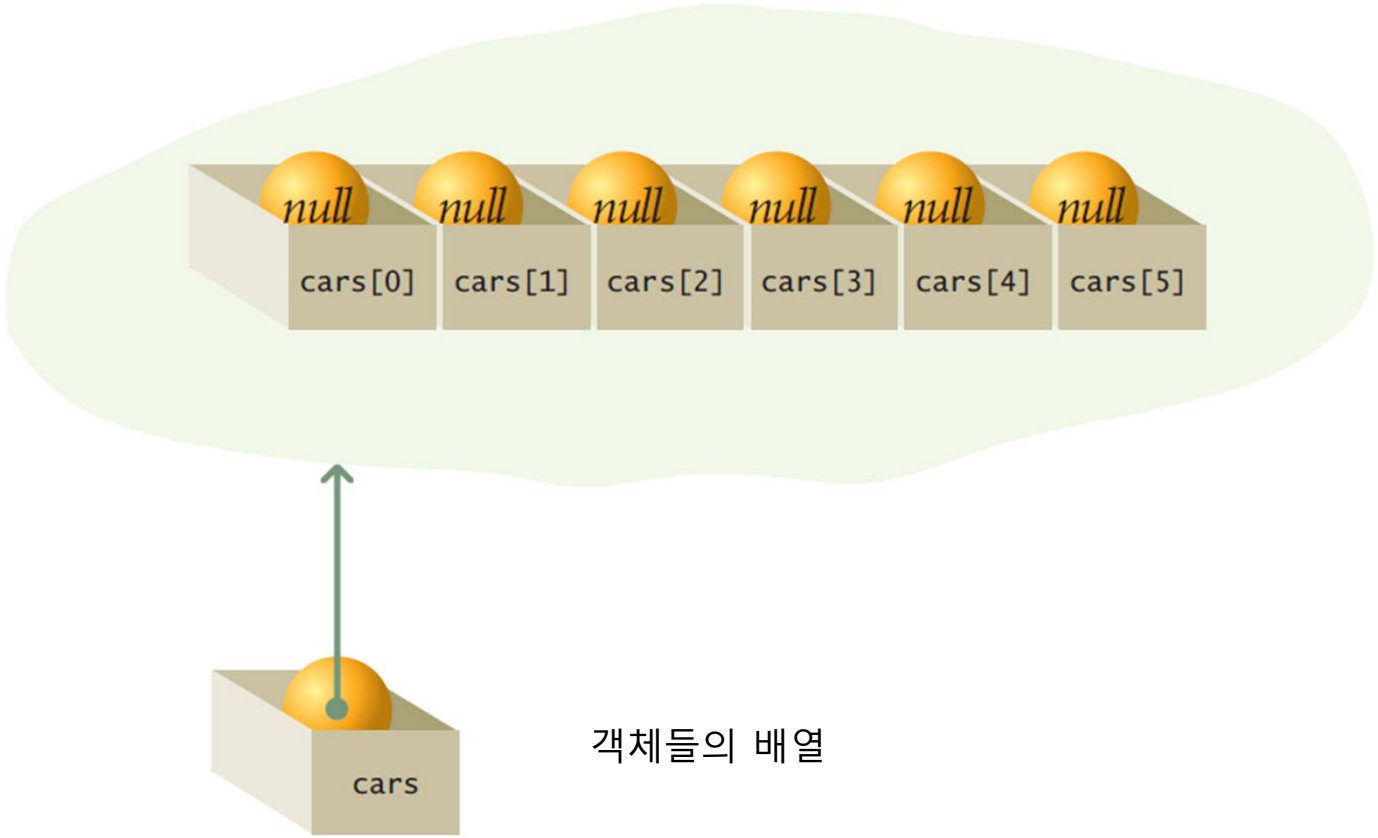
배열


s0
s1
s2
s3
s4
s5
s6
s7
s8
s9
6개의 int형 변수
numbers[0]
numbers[1]
numbers[2]
numbers[3]
numbers[4]
numbers[5]

객체 배열

- 객체 배열: 객체들이 저장된 배열
- 객체 배열에서는 객체에 대한 참조값만을 저장

Car[] cars = **new** Car[6];

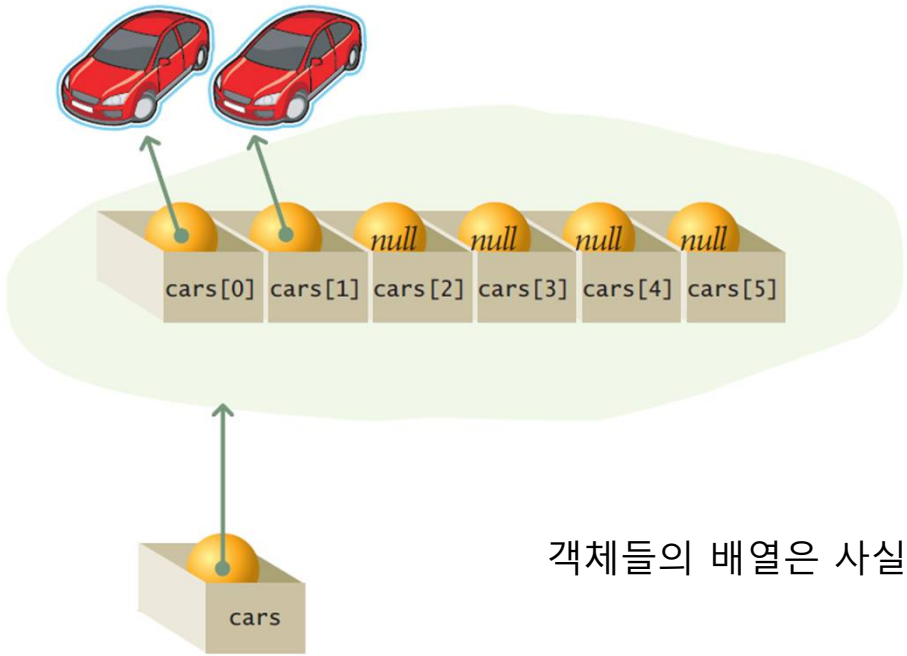


객체들의 배열

객체들의 배열

- 각 원소에 들어가는 객체는 따로 생성하여야 한다.

```
Car[] cars = new Car[6]
cars[0] = new Car();
cars[1] = new Car();
```



객체들의 배열은 사실 참조값만을 저장한다.

직원과 매니저 클래스 작성

1. 직원(Employee)의 예를 가지고 프로그램을 작성하여 보자. 직원을 나타내는 클래스 Employee는 **이름, 주소, 주민번호, 월급** 등의 정보를 가지고 있다.
2. Employee 클래스를 테스트하기 위하여 EmployeeTest 클래스를 작성한다. 아래에 기술된 대로, 리스트를 생성하고 입력 받은 데이터를 Employee의 인스턴스로 만들고 리스트에 추가하고 리스트에 저장된 모든 데이터를 출력하여 본다.

- 패키지명: java13.emp , 클래스명: Employee, EmplyeeTest

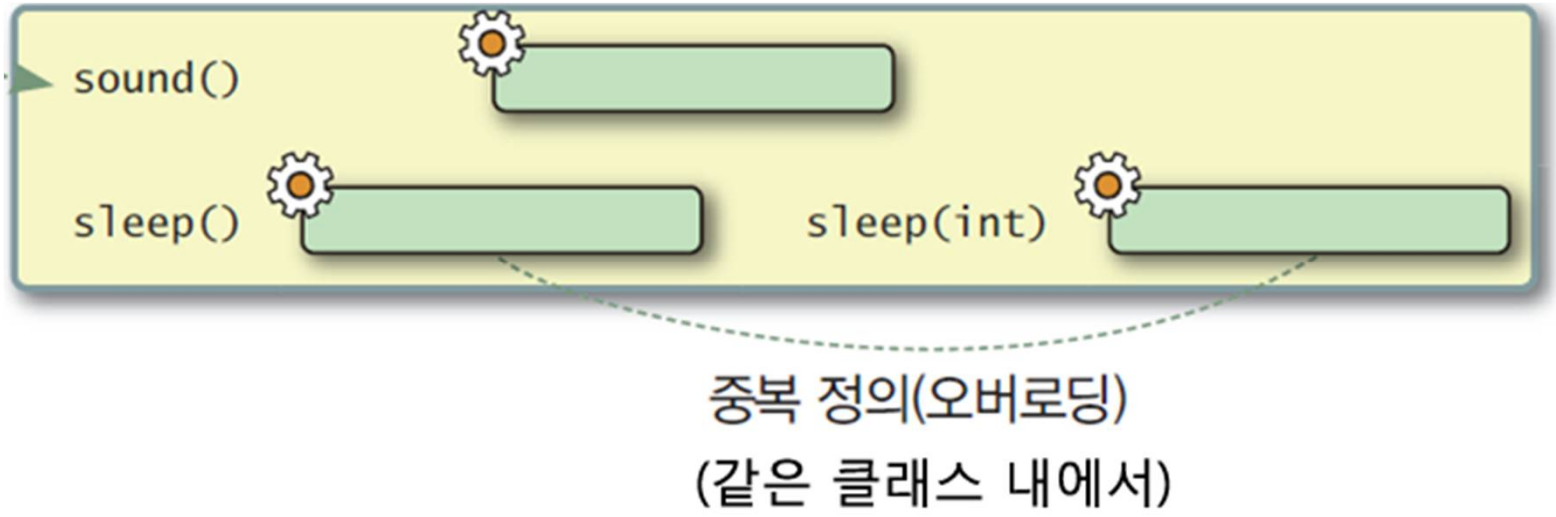
Employee
name: String address: String salary: int rrn: String
toString(): String

```
class EmployeeTest {
    public static void main(String args[])
    {
        Scanner scan = new Scanner(System.in);
        // 크기가 3인 Employee의 배열 employees을 생성한다.
        ____________________________;
        // 3명의 사원 정보를 받아서 각각 Employee 객체를 생성한 후에 배열에 추가하여 본다.
           반복 루프를 사용한다.
        ...
        // employees 배열에 저장된 모든 데이터를 출력한다. 반복 루프를 사용한다.
        ...
    }
}
```

다형성 : 동명이메

눈

다형성: 동명이메



컴파일러는

- **매개 변수의 개수**
- **매개 변수의 타입**
- **매개 변수의 순서**

로 메서드를 구분한다

동일 클래스안에서 같은 메서드명을 사용할 수 있다(메서드 동명이메)

- overloading(중복 메서드)

```
public int squre(int i ) {
     return i*i;
}

public int squre(int i, double j ) {
     return i*j;
}

public double squre(int x, double y ) {
     return x*y;
}

public int squre(double y, int x ) {
     return x*y;
}
```

컴파일러는

- **매개 변수의 개수**
- **매개 변수의 타입**
- **매개 변수의 순서**

로 메서드를 구분한다

중복 메서드 예제

CarTest2.java

```
01  class Car {
02      // 필드 선언
03      private int speed;          // 속도
04      // 중복 메소드: 정수 버전
05      public void setSpeed(int s) {
06          speed = s;
07          System.out.println("정수 버전 호출");
08      }
09
10      // 중복 메소드: 실수 버전
11      public void setSpeed(double s) {
12          speed = (int)s;
13          System.out.println("실수 버전 호출");
14      }
15  }
16
17  public class CarTest2 {
18      public static void main(String[] args) {
19          Car myCar = new Car();          // 첫 번째 객체 생성
20          myCar.setSpeed(100);            // 정수 버전 메소드 호출
21          myCar.setSpeed(79.2);           // 실수 버전 메소드 호출
22      }
23  }
```

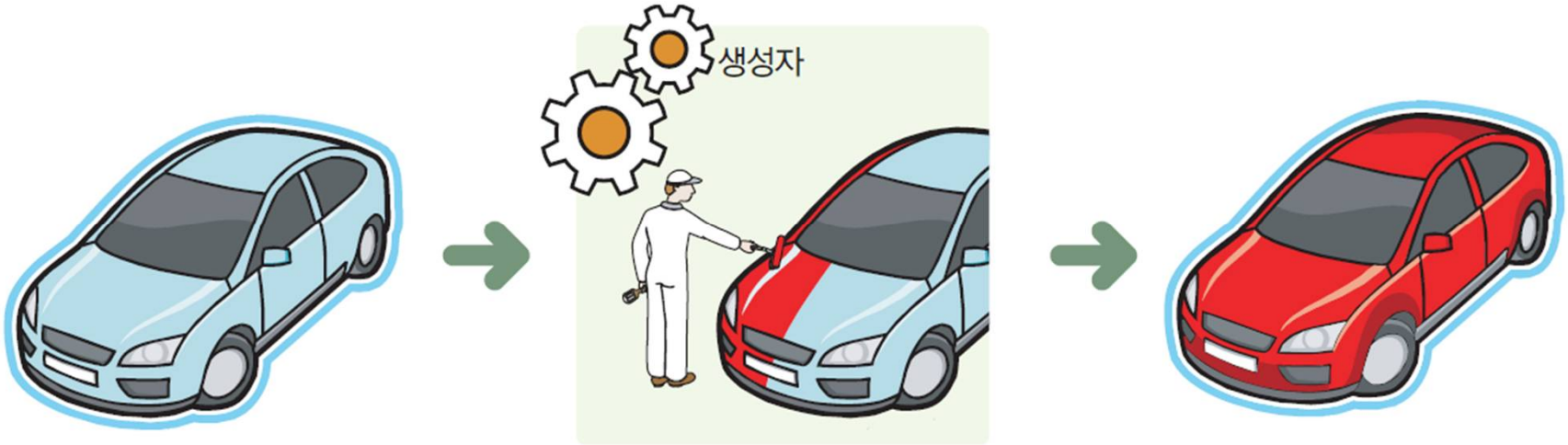
실행결과

```
정수 버전 호출
실수 버전 호출
```

생성자의 사용 목적

- 생성자(contructor):
  - 인스턴스가 생성될 때 필드에 값을 설정하는 **메소드**

  - **생성자는 클래스명과 같아야 한다.**
  - **생성자에는 메서드 리턴 타입을 사용하지 않는다.**



디폴트 생성자

- 만약 클래스 작성시에 생성자를 하나도 만들지 않는 경우에는 자동적으로 메소드의 몸체 부분이 비어있는 생성자가 만들어진다.

CarTest1.java

```
01  class Car {
02      private String color;      // 색상
03      private int speed;         // 속도
04      private int gear;          // 기어
05  }
06  public class CarTest1 {
07      public static void main(String args[]) {
08          Car c1 = new Car();    // 디폴트 생성자 호출
09      }
10  }
```

컴파일러가 디폴트 생성자를 자동으로 만든다.

주의할 점

- 생성자가 하나라도 정의되어 있으면 디폴트 생성자는 만들어지지 않는다.

CarTest2.java

```
01  class Car {
02      private String color;      // 색상
03      private int speed;         // 속도
04      private int gear;          // 기어
05      public Car(String c, int s, int g) {
06          color = c;
07          speed = s;
08          gear = g;
09      }
10  }
11  public class CarTest2 {
12      public static void main(String args[]) {
13          Car c1 = new Car();    // 오류!
14      }
15  }
```

생성자가 하나라도 선언되면 디폴트 생성자는 만들지 않는다.

this vs this()

- this 는 클래스를 가르키는 키워드
- this()는 생성자를 호출한다.

```
public class Car {
    private int speed;          // 속도
    private int gear;           // 기어
    private String color;       // 색상

    // 첫 번째 생성자
    public Car(String c, int s, int g) {
        color = c;
        speed = s;
        gear = g;
    }
    // 색상만 주어진 생성자
    public Car(String c) {
        this(c, 0, 1);          // 첫 번째 생성자를 호출한다.
    }
}
```