Chapter 06

클래스

06-1. 객체 지향 프로그래밍

혼자 공부하는 자바 (신용권 저)

❖ 목차

시작하기 전에

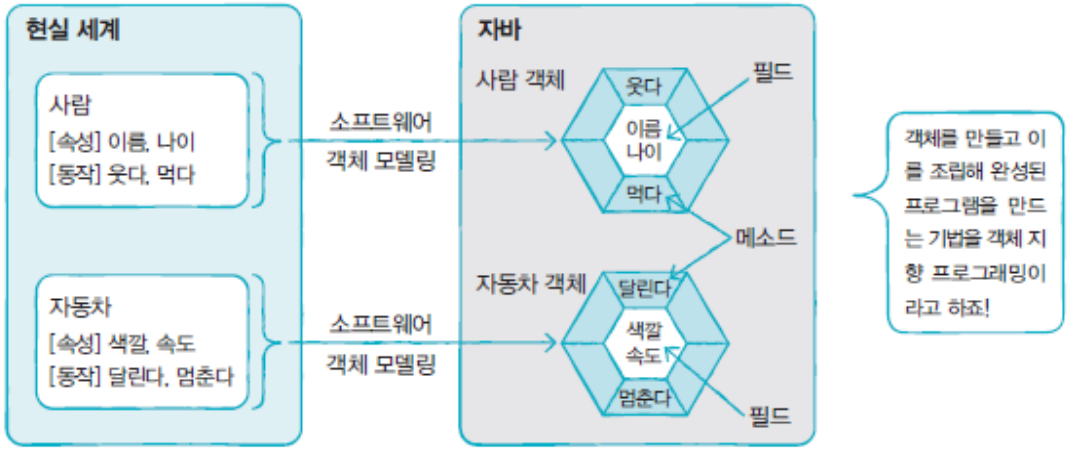
[핵심 키워드] : 클래스, 객체, new 연산자, 클래스 변수, 인스턴스, 클래스 멤버

[핵심 포인트]

객체의 개념과 객체의 상호작용에 대해 알아본다.
클래스로부터 객체를 생성하고 변수로 참조한다.

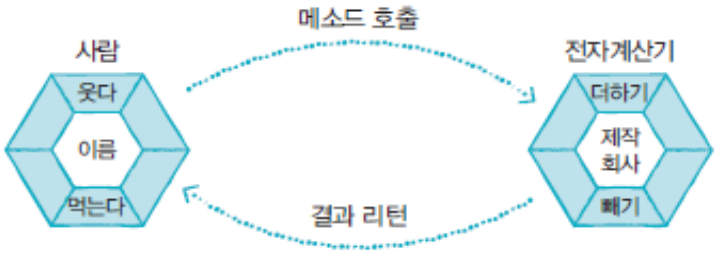
❖ 객체 (Object)

- 물리적으로 존재하거나 추상적으로 생각할 수 있는 것 중에서 자신의 속성을 가지며 식별 가능한 것
- 속성 (필드(field)) + 동작(메소드(method))로 구성



객체의 상호작용

- 객체와 객체 간의 상호작용
  - 메소드를 통해 객체들이 상호작용
  - 메소드 호출 : 객체가 다른 객체의 기능을 이용하는

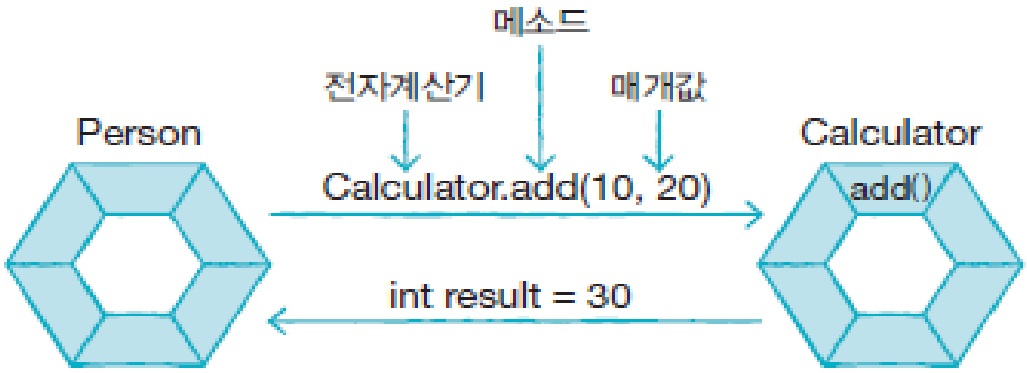



```
int result = Calculator.add(10, 20);
```

리턴한 값을 int 변수에 저장

```
리턴값 = 전자계산기객체.메소드(매개값1, 매개값2, …);
```

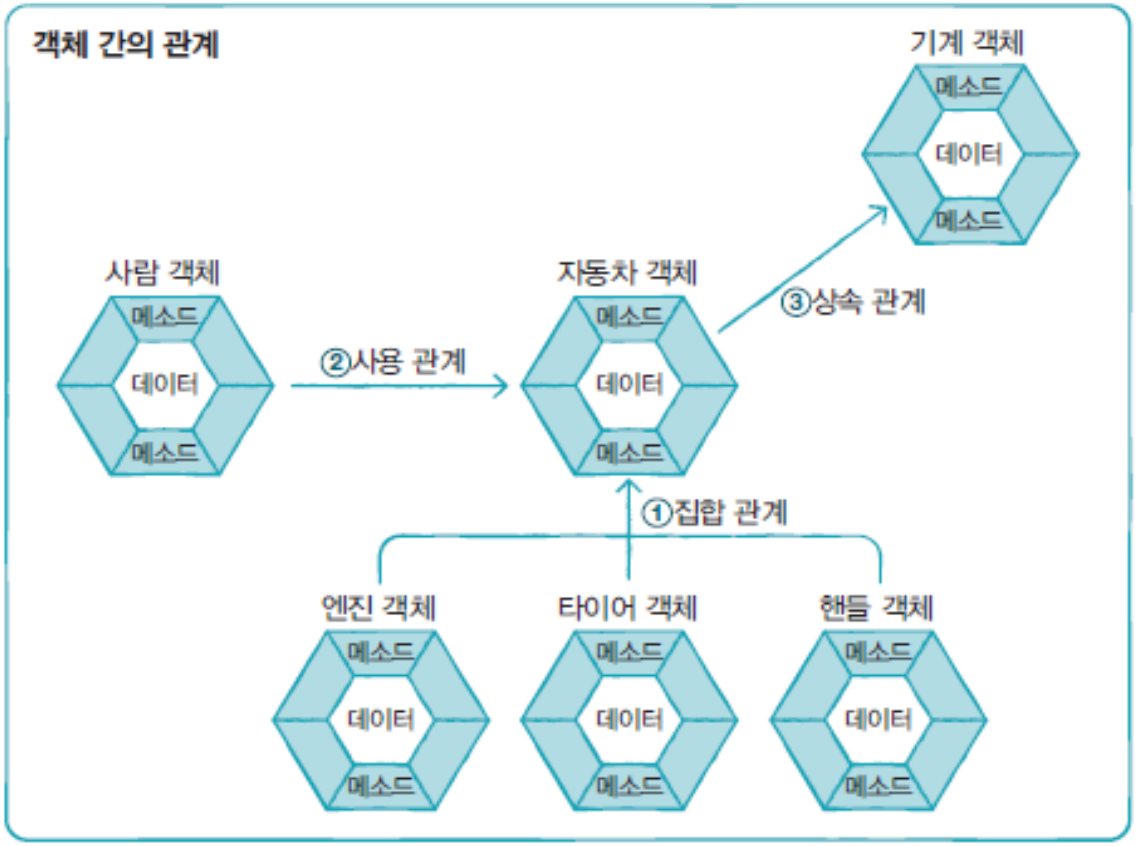
① ②



객체 간의 관계

❖ 객체 간의 관계

- 집합 관계
  - 부품과 완성품의 관계
- 사용 관계
  - 객체 간의 상호작용
- 상속 관계
  - 상위(부모) 객체를 기반으로
  - 하위(자식) 객체를 생성

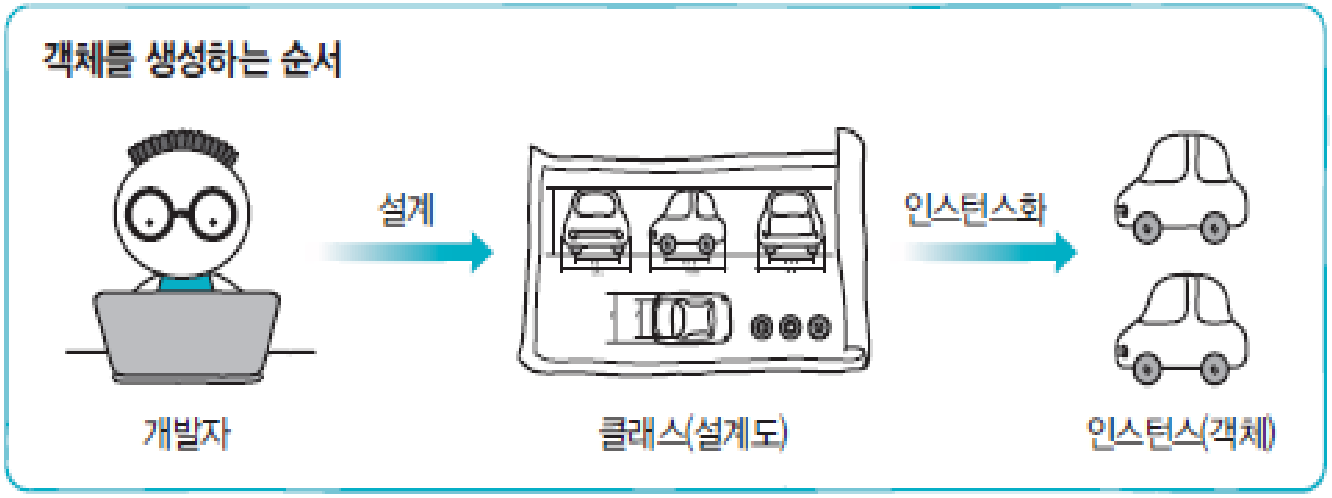


- 객체 지향 프로그래밍
  - 집합/사용 관계에 있는 객체를 하나씩 설계한 후 조립하여 프로그램 개발

객체와 클래스

❖ 클래스 (class)

- 자바의 설계도
- 인스턴스 (instance): 클래스로부터 만들어진 객체
- 객체지향 프로그래밍 단계
  - 클래스 설계 -> 설계된 클래스로 사용할 객체 생성 -> 객체 이용



클래스 선언

❖ 클래스 선언

- 객체 구상 후 클래스 이름을 결정
  - 식별자 작성 규칙에 따라야 함
    - **하나 이상의 문자로 이루어질 것**
    - **첫 글자에는 숫자 올 수 없음**
    - **$, _ 외의 특수 문자는 사용할 수 없음**
    - **자바 키워드는 사용할 수 없음**

```
Calculator, Car, Member, ChatClient, ChatServer, Web_Browser
```

- '클래스 이름.java'로 소스 파일 생성

```
public class 클래스이름 {


}
```

객체 생성과 클래스 변수

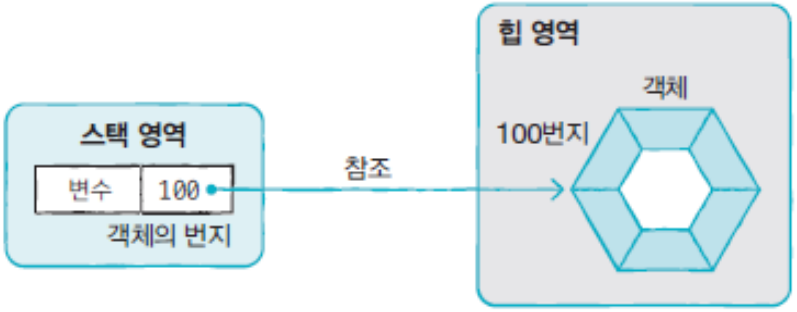
❖ 클래스로부터 객체를 생성

- new 클래스();
- new 연산자로 메모리 힙 영역에 객체 생성

- 객체 생성 후 객체 번지가 리턴
  - 클래스 변수에 저장하여 변수 통해 객체 사용 가능

```
클래스 변수;
변수 = new 클래스();
```

```
클래스 변수 = new 클래스();
```

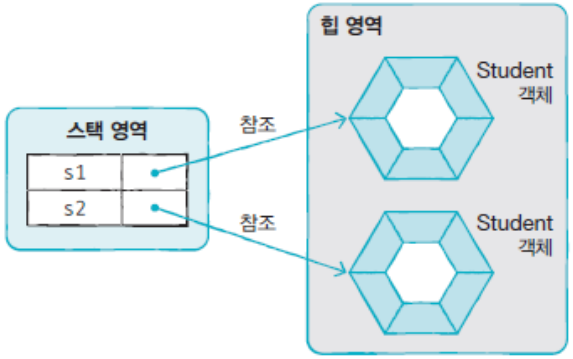


객체 생성과 클래스 변수

```
public class Student {
}
```

```
public class StudentExample {
  public static void main(String[] args) {
    Student s1 = new Student();
    System.out.println("s1 변수가 Student 객체를 참조합니다.")

    Student s2 = new Student();
    System.out.println("s2 변수가 또 다른 Student 객체를 참조합니다.");
  }
}
```



❖ 클래스의 두 용도

- 라이브러리(API : Application Program Interface) 클래스
  - 객체 생성 및 메소드 제공 역할 - Student.java
- 실행 클래스
  - main() 메소드 제공 역할 - StudentExample.java

클래스의 구성 멤버

❖ 클래스 멤버

- 필드(Field)
  객체의 데이터가 저장되는 곳
- 생성자(Constructor)
  객체 생성 시 초기화 역할 담당
- 메소드(Method)
  객체의 동작에 해당하는 실행 블록

```
public class ClassName {

  //필드
  int fieldname;

  //생성자
  ClassName() { … }

  //메소드
  void methodName() { … }

}
```

키워드로 끝내는 핵심 포인트

- 클래스: 객체를 만들기 위한 설계도
- 객체 : 클래스로부터 생성되며 'new 클래스()'로 생성
- new 연산자 : 객체 생성 연산자이며 생성자 호출하고 객체 생성 번지를 리턴
- 클래스 변수 : 클래스로 선언한 변수이며 해당 클래스의 객체 번지가 저장됨
- 인스턴스 : 객체는 클래스의 인스턴스
- 클래스 멤버 : 클래스에 선언되는 멤버로 필드, 생성자, 메소드가 있음

Chapter 06

클래스

06-2. 필드

혼자 공부하는 자바 (신용권 저)

❖ 목차

- 시작하기 전에
- 필드 선언
- 필드 사용
- 키워드로 끝내는 핵심 포인트

시작하기 전에

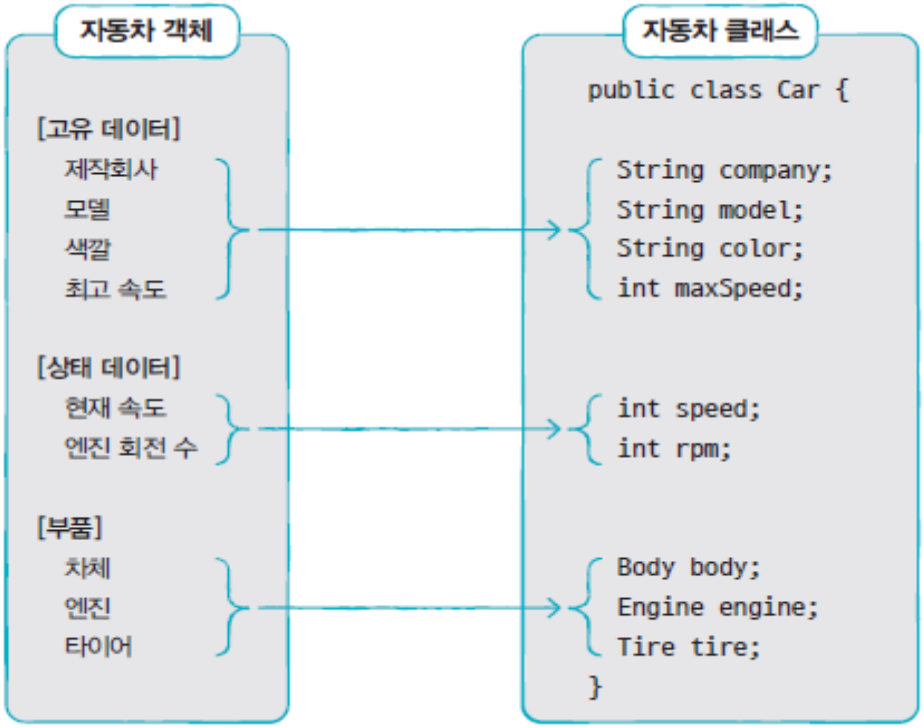
[핵심 키워드] : 필드 선언, 필드 사용

[핵심 포인트]

필드는 객체의 고유 데이터, 부품 객체, 상태 정보를 저장
필드를 선언하고 생성한 뒤 이를 읽고 변경하는 방법을 학습

❖ 필드 (field)

- 객체의 고유 데이터,
  객체가 가져야 할 부품,
  객체의 현재 상태 데이터 등을 저장



필드 선언

❖ 필드 선언

- 클래스 중괄호 블록 어디서든 존재 가능
- 생성자와 메소드 중괄호 블록 내부에는 선언 불

```
타입 필드 [ = 초기값] ;
```

- 변수와 선언 형태 유사하나 변수 아님에 주의

- class XXX {

```
String company = "현대자동차";
String model = "그랜저";
int maxSpeed = 300;
int productionYear;
int currentSpeed;
boolean engineStart;
```

- }

필드 선언

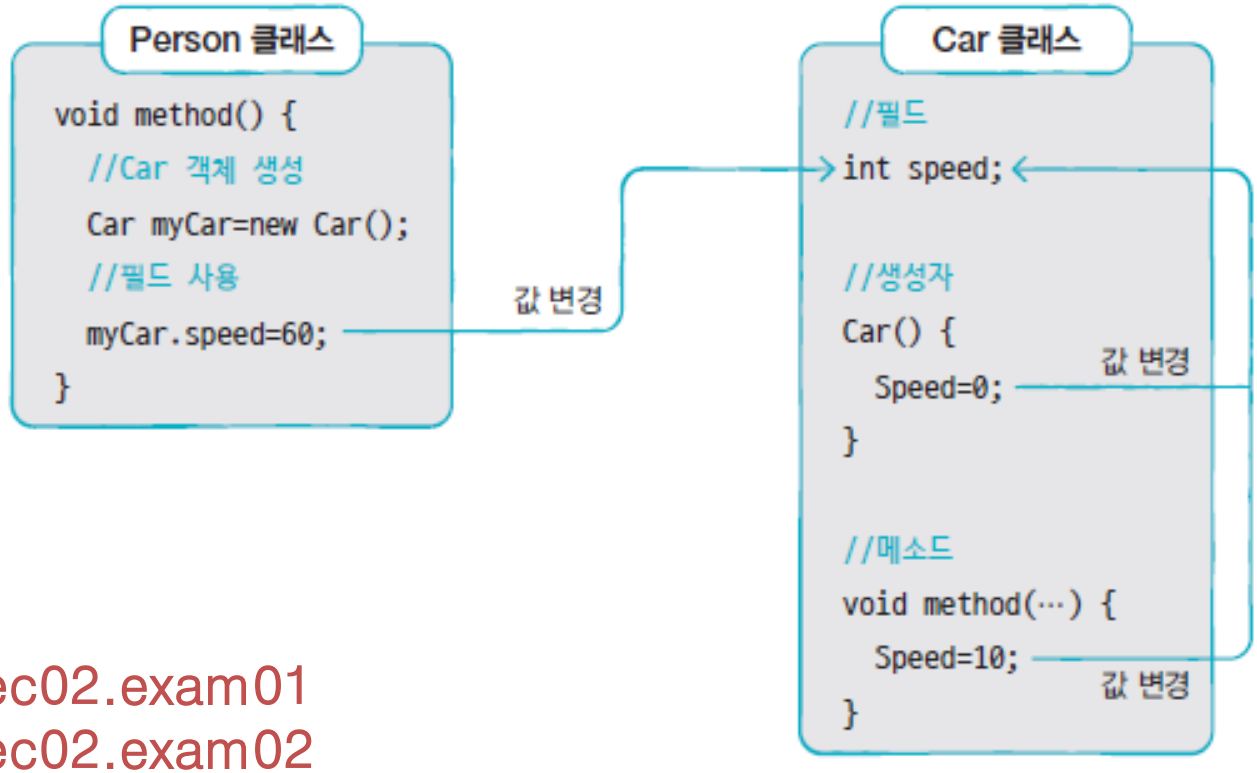
- 초기값은 주어질 수도, 생략할 수도 있음
  - 초기값 지정되지 않은 필드는 객체 생성 시 자동으로 기본 초기값 설정

분류		타입	초기값
기본 타입	정수 타입	byte char short int long	0 \u0000 (빈 공백) 0 0 0L
	실수 타입	float double	0.0F 0.0
	논리 타입	boolean	false
참조 타입		배열 클래스(String 포함) 인터페이스	null null null

필드 사용

❖ 필드 사용

- 필드값 읽고 변경하는 작업
- 클래스 내부 생성자 및 메소드에서 사용하는 경우 : 필드 이름으로 읽고 변경
- 클래스 외부에서 사용하는 경우 : 클래스로부터 객체 생성한 뒤 필드 사용



sec02.exam01
sec02.exam02

키워드로 끝내는 핵심 포인트

- 필드 선언 : 클래스 중괄호 블록 어디서든 선언하나 생성자나 메소드 내부에서는 사용 불가

- 필드 사용 :
  - 클래스 내부의 생성자와 메소드에서 바로 사용 가능
  - 클래스 외부에서 사용할 경우 반드시 객체 생성하고 참조 변수 통해 사용

Chapter 06

클래스

06-3. 생성자

혼자 공부하는 자바 (신용권 저)

목차

- 시작하기 전에
- 기본 생성자
- 생성자 선언
- 필드 초기화
- 생성자 오버로딩
- 다른 생성자 호출 : this()
- 키워드로 끝내는 핵심 포인트

시작하기 전에

[핵심 키워드] : 기본 생성자, 생성자 선언, 매개 변수, 객체 초기화, 오버로딩, this()

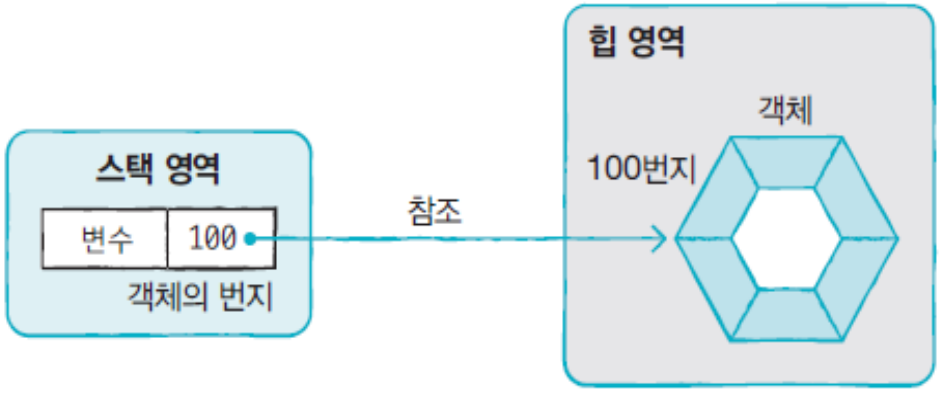
[핵심 포인트]

생성자는 new 연산자로 호출되는 중괄호{} 블록이다.
객체 생성 시 초기화를 담당한다.

❖ 생성자 (constructor)
- 클래스로부터 new 연산자로 객체를 생성할 때 호출되어 객체의 초기화를 담당

❖ 객체 초기화
- 필드를 초기화하거나 메소드를 호출해서
  객체를 사용할 준비를 하는 것

❖ 생성자가 성공적으로 실행
- 힙 영역에 객체 생성되고 객체 번지가 리



기본 생성자

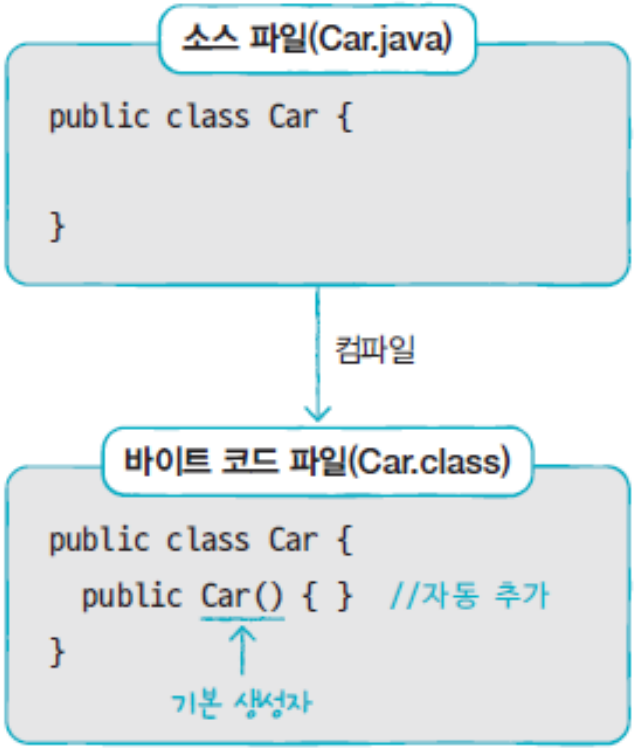
❖ 기본 생성자 (default constructor)

- 클래스 내부에 생성자 선언 생략할 경우 바이트 코드에 자동 추가

```
[public] 클래스() { }
```

- 클래스에 생성자 선언하지 않아도 new 생성자()로 객체 생성 가능

```
Car myCar = new Car();
            ↑
        기본 생성자
```



생성자 선언

❖ 생성자 선언

```
클래스( 매개변수선언, … ) {
  //객체의 초기화 코드
}
```
생성자 블록

- 매개 변수 선언은 생략할 수도 있고 여러 개 선언할 수도 있음

```
public class Car {
  //생성자
  Car(String model, String color, int maxSpeed) { … }
}
```

- 클래스에 생성자가 명시적으로 선언되었을 경우 반드시 선언된 생성자 호출하여 객체 생성

```
Car myCar = new Car("그랜저", "검정", 300);
```

필드 초기화

❖ 생성자의 필드 초기화

```
public class Korean {
  //필드
  String nation = "대한민국";
  String name;
  String ssn;

  //생성자
  public Korean(String n, String s) {
    name = n;
    ssn = s;
  }
}
```

```
Korean k1 = new Korean("박자바", "011225-1234567");
Korean k2 = new Korean("김자바", "930525-0654321");
```

필드 초기화

- 매개 변수 이름 은 필드 이름과 유사하거나 동일한 것 사용 권장
- 필드와 매개 변수 이름 완전히 동일할 경우 this.필드로 표현

```
public Korean(String name, String ssn) {
  this.name = name;
       ↑      ↑
      필드  매개 변수
  this.ssn = ssn;
       ↑     ↑
      필드 매개 변수
}
```

생성자 오버로딩

❖ 생성자 오버로딩 (overloading)

- 매개 변수를 달리하는 생성자 여러 개 선언
- 외부에서 제공되는 다양한 데이터를 사용하여 객체 화하기 위해

```
public class 클래스 {
  클래스 ( [타입 매개변수, …] ) {
    …
  }

  클래스 ( [타입 매개변수, …] ) {
    …
  }
}
```

[생성자의 오버로딩]
매개 변수의 타입, 개수, 순서가 다르게 선언

생성자 오버로딩

```
public class Car {
  Car() { … }
  Car(String model) { … }
  Car(String model, String color) { … }
  Car(String model, String color, int maxSpeed) { …
}
```

```
Car car1 = new Car();
Car car2 = new Car("그랜저");
Car car3 = new Car("그랜저", "흰색");
Car car4 = new Car("그랜저", "흰색", 300);
```

- 매개 변수의 타입, 개수, 선언된 순서 같은 경우, 매개 변수 이름만 바꾸는 것은 생성자 오버로딩 아님

```
Car(String model, String color) { … }
Car(String color, String model) { … }  //오버로딩이 아님
```

다른 생성자 호출: this()

❖ this() 코드

- 생성자에서 다른 생성자 호출
- 필드 초기화 내용을 한 생성자에만 집중 작성하고 나머지 생성자는 초기화 내용 가진 생성자로 호출
  - 생성자 오버로딩 증가 시 중복 코드 발생 문제 해결

```
클래스( [매개변수, …] ) {
  this( 매개변수, …, 값, … );  <-- 클래스의 다른 생성자 호출
  실행문;
}
```

- 생성자 첫 줄에서만 허용

다른 생성자 호출: this()

```
Car(String model) {
  this.model = model;
  this.color = "은색";
  this.maxSpeed = 250;
}

Car(String model, String color) {
  this.model = model;
  this.color = color;
  this.maxSpeed = 250;
}

Car(String model, String color, int maxSpeed) {
  this.model = model;
  this.color = color;
  this.maxSpeed = maxSpeed;
}
```

중복 코드

```
Car(String model) {
  this(model, "은색", 250);
}

Car(String model, String color) {
  this(model, color, 250);
}

Car(String model, String color, int maxSpeed) {
  this.model = model;
  this.color = color;
  this.maxSpeed = maxSpeed;
}
```

호출

호출

공통 실행 코드

키워드로 끝내는 핵심 포인트

- 기본 생성자 : 클래스 선언 시 컴파일러에 의해 자동으로 추가되는 생성자
- 생성자 선언 : 생성자를 명시적으로 선언 가능. 생성자를 선언하면 기본 생성자는 생성되지 않음
- 매개 변수 : 생성자 호출 시 값을 전달받기 위해 선언되는 변수
- 객체 초기화 : 생성자 내부에서 필드값 초기화하거나 메소드 호출해서 사용 준비를 하는 것
- 오버로딩 : 매개 변수 달리하는 생성자를 여러 개 선언
- this(): 객체 자신의 또다른 생성자를 호출할 때 사용

Chapter 06

클래스

06-4. 메소드

혼자 공부하는 자바 (신용권 저)

❖ 목차

- 시작하기 전에
- 메소드 선언
- 리턴 문
- 메소드 호출
- 메소드 오버로딩
- 키워드로 끝내는 핵심 포인트

시작하기 전에

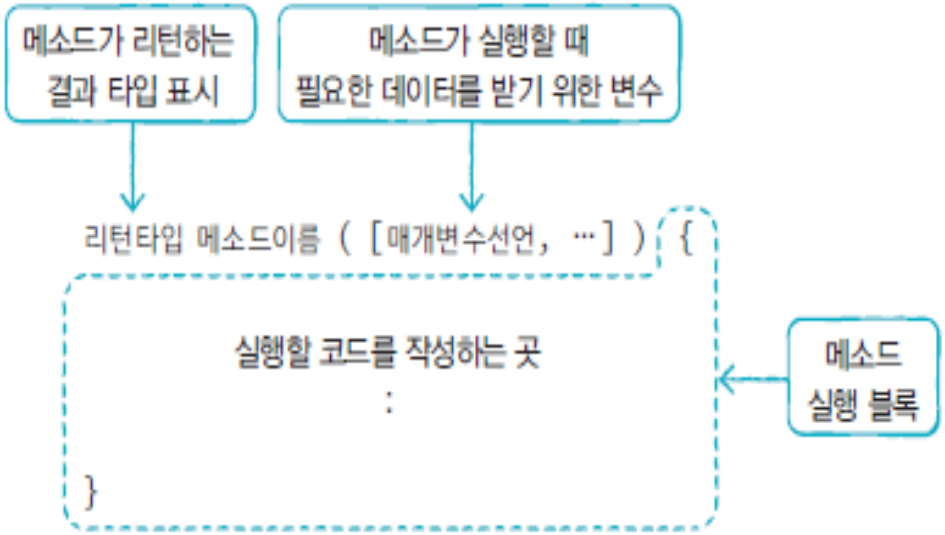
[핵심 키워드] : 선언부, void, 매개 변수, 리턴문, 호출, 오버로딩

[핵심 포인트]

메소드를 선언하고 호출하는 방법에 대해 알아본다.

❖ 메소드 선언부 (signature)

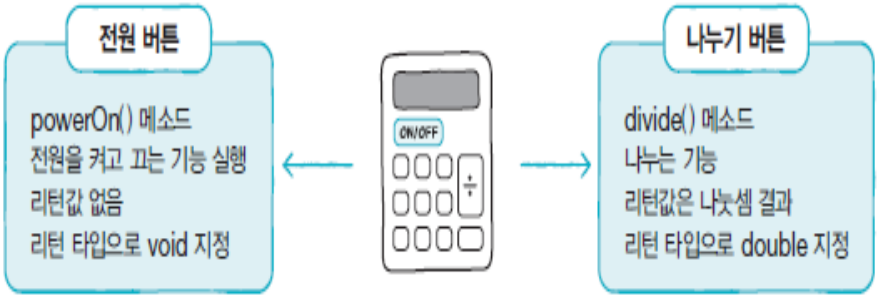
- 리턴 타입 : 메소드가 리턴하는 결과의 타입 표시
- 메소드 이름 : 메소드의 기능 드러나도록
- 식별자 규칙에 맞게 이름
- 매개 변수 선언 : 메소드 실행할 때 필요한
- 데이터 받기 위한 변수
- 메소드 실행 블록 : 실행할 코드 작성



메소드 선언

❖ 리턴 타입

- 메소드를 실행한 후의 결과값의 타입
- 리턴값 없을 수도 있음
- 리턴값 있는 경우 리턴 타입이 선언부에 명시




```
void powerOn() { … }
double divide( int x, int y ) { … }
```

- 리턴값 존재 여부에 따라 메소드 호출 방법 다름

```
powerOn();
double result = divide( 10, 20 );
```

```
int result = divide( 10, 20 );  //컴파일 에러
```

메소드 선언

❖ 메소드 이름

- 숫자로 시작하면 안 되고, $와 _ 제외한 특수문자 사용 불가
- 메소드 이름은 관례적으로 소문자로 작성
- 서로 다른 단어가 혼합된 이름일 경우 뒤이어 오는 단어의 첫 글자를 대문자로 작성

```
void run()  { … }
void startEngine() { … }
String getName() { … }
int[] getScores() { … }
```

메소드 선언

❖ 매개 변수 선언

- 메소드 실행에 필요한 데이터를 외부에서 받아 저장할 목적

```
double divide( int x, int y ) { … }
```

```
double result = divide( 10, 20 );
```

```
byte b1 = 10;
byte b2 = 20;
double result = divide( b1, b2 );
```

- 잘못된 매개값 사용하여 컴파일 에러 발생하는 경우

```
double result = divide( 10.5, 20.0 );
```

메소드 선언

❖ 매개 변수의 개수를 모를 경우

- 매개 변수를 배열 타입으로 선언

```
int sum1(int[] values) {  }
```

```
int[] values = { 1, 2, 3 };
int result = sum1(values);
int result = sum1(new int[] { 1, 2, 3, 4, 5 });
```

- 배열 생성하지 않고 값의 목록만 넘겨주는 방식

```
int sum2(int ··· values) {  }
```

```
int result = sum2(1, 2, 3);
int result = sum2(1, 2, 3, 4, 5);
```

```
int[] values = { 1, 2, 3 };
int result = sum2(values);
int result = sum2(new int[] { 1, 2, 3, 4, 5 });
```

리턴(return)문

- 리턴값이 있는 메소드
  - 메소드 선언에 리턴 타입 있는 메소드는 리턴문 사용하여 리턴값 지정

```
return 리턴값;
```

  - return문의 리턴값은 리턴타입이거나 리턴타입으로 변환될 수 있어야 함

```
int plus(int x, int y) {
  int result = x + y;
  return result;
}
```

```
int plus(int x, int y) {
  byte result = (byte) (x + y);
  return result;
}
```

리턴(return)문

- 리턴값이 없는 메소드 : void
  - void 선언된 메소드에서 return문 사용하여 메소드 실행 강제

```
return;
```

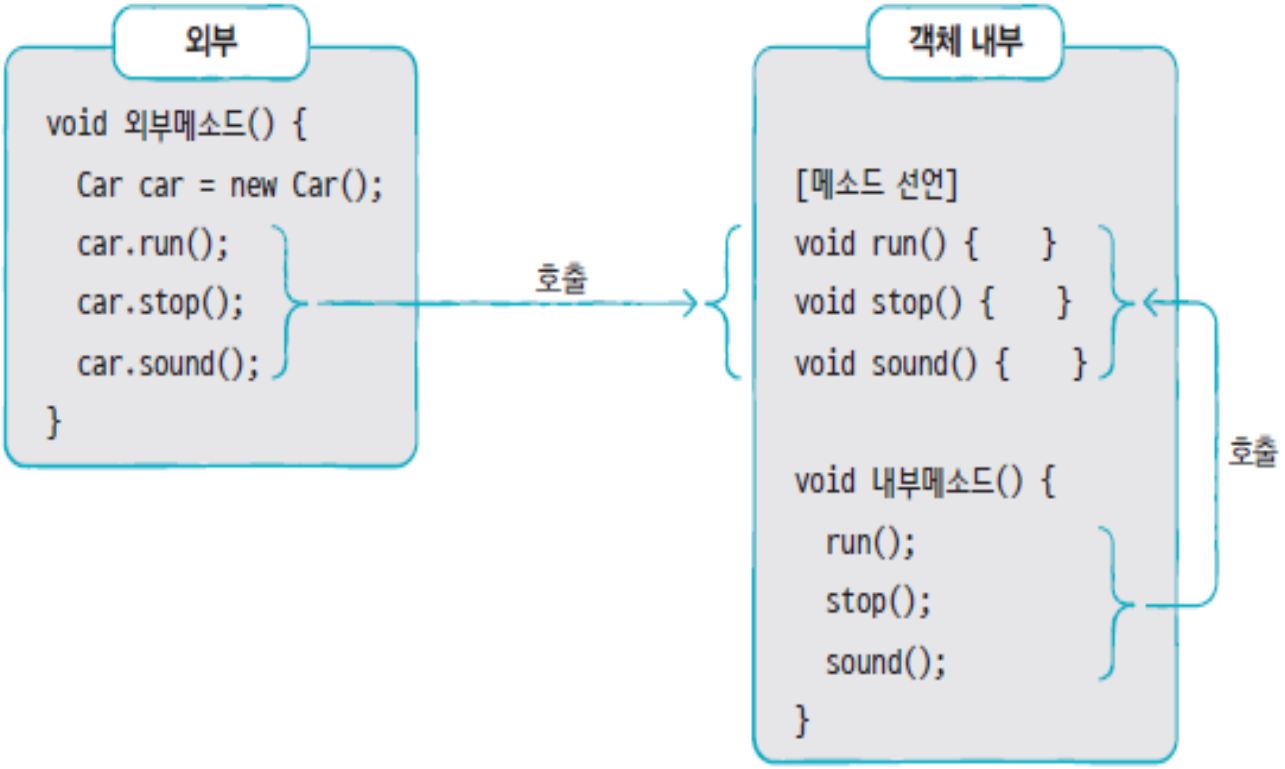
```
void run() {
  while(true) {
    if(gas > 0) {
      System.out.println("달립니다.(gas잔량:" + gas + ")");
      gas -= 1;
    } else {
      System.out.println("멈춥니다.(gas잔량:" + gas + ")");
      return;
    }
  }
}
```

run() 메소드 실행 종료

메소드 호출

❖ 메소드 호출

- 클래스 내외부의 호출에 의해 메소드 실행
  - 내부의 경우 단순히 메소드 이름으로 호출
  - 외부의 경우 클래스로부터 객체 생성한 뒤 참조 변수 사용하여 메소드 호출



메소드 호출

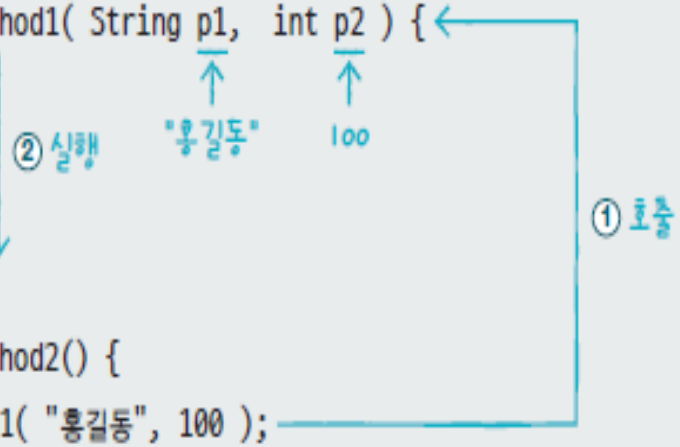
❖ 객체 내부에서 호출

- 메소드가 리턴값 없거나(void) 있어도 받고 싶지 않은 경우

```
메소드( 매개값, … );
```

```
public class ClassName {
  void method1( String p1,  int p2 ) {
  }

  void method2() {
    method1( "홍길동", 100 );
  }
}
```



메소드 호출

- 리턴값 있는 메소드 호출하고 리턴값 받고 싶은 경우

```
타입 변수 = 메소드(매개값, … );
```

리턴값

```
public class ClassName {
  int method1(int x, int y) {
    int result = x + y;
    return result;
  }

  void method2() {
    int result1 = method1(10, 20);      //result1에는 30이 저장
    double result2 = method1(10, 20);   //result2에는 30.0이 저장
  }
}
```

메소드 호출

❖ 객체 외부에서 호출

- 우선 클래스로부터 객체 생성

```
클래스 참조변수 = new 클래스( 매개값, … );
```

- 참조 변수와 도트 연산자 사용하여 메소드 호출

```
참조변수.메소드( 매개값, … );  //리턴값이 없거나, 있어도 리턴값을 받지 않을 경우
타입 변수 = 참조변수.메소드( 매개값, … );  //리턴값이 있고, 리턴값을 받고 싶을 경우
```

```
Car myCar = new Car();
myCar.keyTurnOn();
myCar.run();
int speed = myCar.getSpeed();
```

메소드 오버로딩

❖ 메소드 오버로딩 (overloading)

- 같은 이름의 메소드를 여러 개 선언
- 매개값을 다양하게 받아 처리할 수 있도록 하기 위함
- 매개 변수의 타입, 개수, 순서 중 하나가 달라야

```
class 클래스 {
  리턴 타입  메소드이름 ( 타입 변수, … ) { … }
    ↑          ↑          ↑
   동일       동일    매개 변수의 타입, 개수, 순서가 달라야 함
    ↓          ↓          ↓
  리턴 타입  메소드이름 ( 타입 변수, … ) { … }
}
```

```
int plus(int x, int y) {
  int result = x + y;
  return result;
}
double plus(double x, double y) {
  double result = x + y;
  return result;
}
```

메소드 오버로딩

- 오버로딩된 메소드 호출하는 경우 JVM은 매개값 타입 보고 메소드를 선택

```
plus(10, 20);
```

- plus(int x, int y)가 실행

```
plus(10.5, 20.3);
```

- plus(double x, double y)가 실행

```
int x = 10;
double y = 20.3;
plus(x, y);
```

- plus(double x, double y)가 실행

```
int plus(int x, int y) {
  int result = x + y;
  return result;
}
double plus(double x, double y) {
  double result = x + y;
  return result;
}
```

메소드 오버로딩

- 매개 변수의 타입, 개수, 순서 같은 경우 매개변수 이름 달라도 메소드 오버로딩 아님에 주의

```
int divide(int x, int y) { … }
double divide(int boonja, int boonmo) { … }
```

- System.out.println() 메소드

```
void println() { … }
void println(boolean x) { … }
void println(char x) { … }
void println(char[] x) { … }
void println(double x) { … }
void println(float x) { … }
void println(int x) { … }
void println(long x) { … }
void println(Object x) { … }
void println(String x) { … }
```

키워드로 끝내는 핵심 포인트

- 선언부 : 리턴 타입, 메소드 이름, 매개 변수 선언
- void : 리턴값이 없는 메소드는 리턴 타입으로 void를 기술해야 함
- 매개 변수 : 메소드 호출 시 제공되는 매개값이 대입되어 메소드 블록 실행 시 이용됨
- 리턴문 : 메소드의 리턴값을 지정하거나 메소드 실행 종료를 위해 사용할 수 있음.
- 호출 : 메소드를 실행하려면 '메소드 이름(매개값. ...)' 형태로 호출
- 오버로딩 : 클래스 내에 같은 이름의 메소드 여러 개 선언하는 것을 말함

Chapter 06

클래스

06-5. 인스턴스 멤버와 정적 멤버

혼자 공부하는 자바 (신용권 저)

목차

- 시작하기 전에
- 인스턴스 멤버와 this
- 정적 멤버와 static
- 싱글톤
- final 필드와 상수
- 키워드로 끝내는 핵심 포인트

시작하기 전에

[핵심 키워드] : 인스턴스 멤버, this, 정적 멤버, static, final 필드, 싱글톤, 상수

[핵심 포인트]

클래스에 선언된 필드와 메소드가 모두 객체 내부에 포함되는 것은 아니다.
객체가 있어야 사용 가능한 멤버가 있고, 그렇지 않는 멤버도 있다.

❖ 인스턴스 멤버

- 객체 마다 가지고 있는 멤버
  - - 인스턴스 필드: 힙 영역의 객체 마다 가지고 있는 멤버, 객체마다 다른 데이터를 저장
  - - 인스턴스 메소드: 객체가 있어야 호출 가능한 메소드,
  - 클래스 코드(메소드 영역)에 위치하지만, 이해하기 쉽도록 객체 마다 가지고 있는
  - 메소드라고 생각해도 됨

❖ 정적 멤버

- 객체와 상관없는 멤버, 클래스 코드(메소드 영역)에 위치
  - 정적 필드 및 상수: 객체 없이 클래스만으로도 사용 가능한 필드
  - 정적 메소드: 객체가 없이 클래스만으로도 호출 가능한 메소드

인스턴스 멤버와 this

❖ 인스턴스 (instance) 멤버:

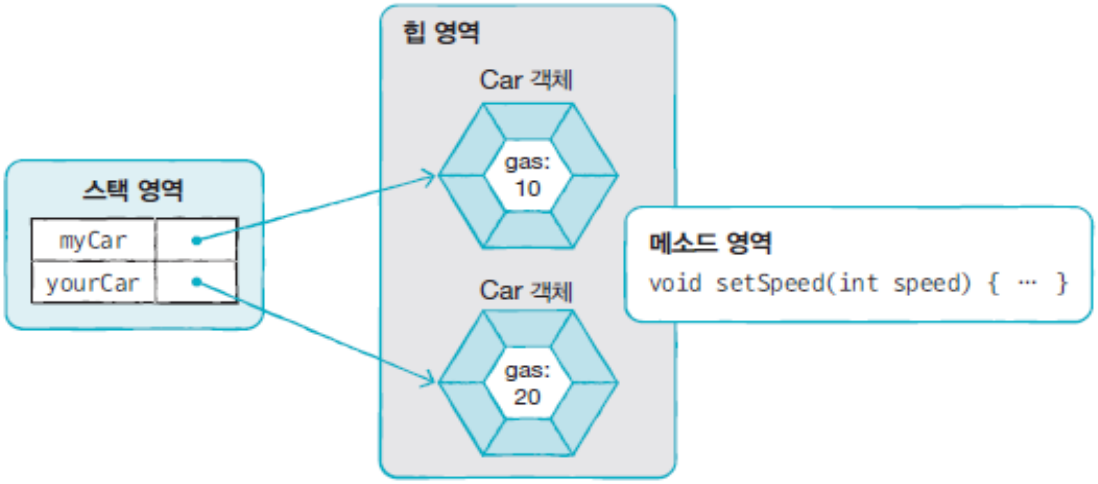
- 객체를 생성한 후 사용할 수 있는 필드와 메소드

```
public class Car {
  //필드
  int gas;

  //메소드
  void setSpeed(int speed) { … }
}
```

```
Car myCar = new Car();
myCar.gas = 10;
myCar.setSpeed(60);

Car yourCar = new Car();
yourCar.gas = 20;
yourCar.setSpeed(80);
```



인스턴스 멤버와 this

❖ this

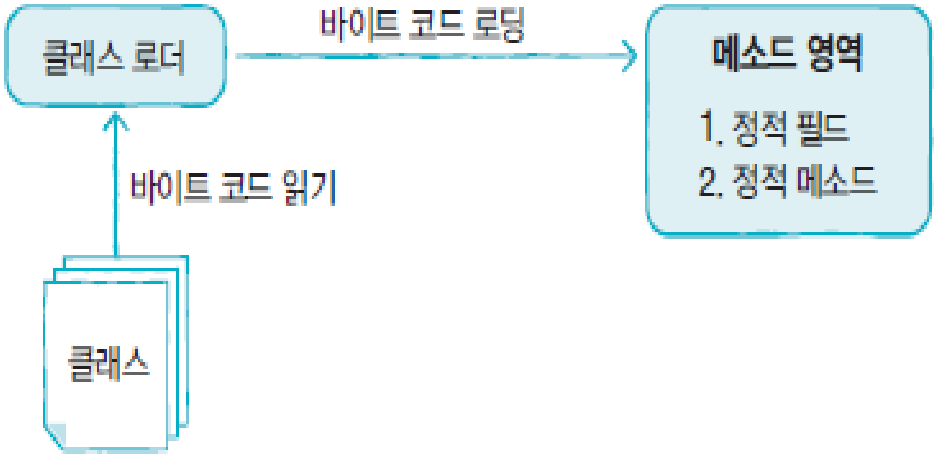
- 객체 내에서 인스턴스 멤버에 접근하기 위해 사용
- 생성자와 메소드의 매개 변수 이름이 필드와 동일할 경우, 필드 임을 지정하기 위해 주로 사용

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

정적 멤버와 static

❖ 정적 (static) 멤버

- 클래스에 고정된 멤버로서 객체 생성하지 않고 사용할 수 있는 필드와 메소드



❖ 정적 멤버 선언

```
public class 클래스 {
  //정적 필드
  static 타입 필드 [= 초기값];

  //정적 메소드
  static 리턴 타입 메소드( 매개변수선언, … ) { … }
}
```

정적 멤버와 static

❖ 정적 멤버 사용

- 클래스 이름과 함께 도트 연산자로 접근

```
클래스.필드;
클래스.메소드( 매개값, … );
```

```
public class Calculator {
  static double pi = 3.14159;
  static int plus(int x, int y) { … }
  static int minus(int x, int y) { … }
}
```

```
double result1 = 10 * 10 * Calculator.pi;
int result2 = Calculator.plus(10, 5);
int result3 = Calculator.minus(10, 5);
```

정적 멤버와 static

- 인스턴스 멤버와 정적 멤버 선택 기준
  - 객체마다 다를 수 있는 필드값 -> 인스턴스 필드로 선언
  - 그렇지 않고 객체마다 다를 필요가 없는 필드값 -> 정적 필드로 선언

```
public class Calculator {
  String color;                 //계산기별로 색깔이 다를 수 있습니다.
  static double pi = 3.14159;   //계산기에서 사용하는 파이(π) 값은 동일합니다.
}
```

  - 메소드 블록에 인스턴스 필드 또는 인스턴스 메소드를 사용할 경우 -> 인스턴스 메소드로 선언
    그렇지 않을 경우 -> 정적 메소드로 선언

```
public class Calculator {
  String color;                                          //인스턴스 필드
  void setColor(String color) { this.color = color; }    //인스턴스 메소드
  static int plus(int x, int y) { return x + y; }        //정적 메소드
  static int minus(int x, int y) { return x - y; }       //정적 메소드
}
```

정적 멤버와 static

❖ 정적 메소드 선언 시 주의할 점

- 정적 메소드 선언 시 그 내부에 인스턴스 필드 및 메소드 사용 불가
- 정적 메소드 선언 시 그 객체 자신 참조인 this 키워드 사용 불가

```
public class ClassName {
  //인스턴스 필드와 메소드
  int field1;
  void method1() { … }
  //정적 필드와 메소드
  static int field2;
  static void method2() { … }

  //정적 메소드
  static void Method3 {
    this.field1 = 10;  //(x)
    this.method1();    //(x)
    field2 = 10;       //(o)
    method2();         //(o)
  }
}
```

← 컴파일 에러

- 정적 메소드에서 인스턴스 멤버 사용하려는 경우

  객체 우선 생성 후 참조 변수로 접근

```
static void Method3() {
  ClassName obj = new ClassName();
  obj.field1 = 10;
  obj.method1();
}
```

정적 멤버와 static

- main() 메소드 정적 메소드이므로 동일 규칙 적용

```
public class Car {
  int speed;

  void run() { … }

  public static void main(String[] args) {
    speed = 60;   //(x)
    run();        //(x)
  }
}
```

컴파일 에러

```
public static void main(String[] args) {
  Car myCar = new Car();
  myCar.speed = 60;
  myCar.run();
}
```

싱글톤

❖ 싱글톤 (singleton)

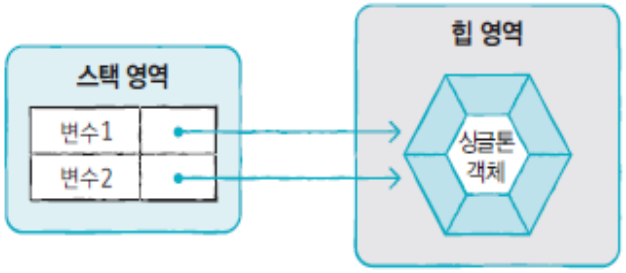
- 전체 프로그램에서 단 하나의 객체만 만들도록 보장하는 코딩 기법
- 싱글톤 작성 방법
  - 클래스 외부에서 new 연산자 통해 생성자 호출하는 것 불가하도록 private 접근 제한자 사용
  - 자신의 타입인 정적 필드 선언 후 자신의 객체 생성해 초기화
  - 외부에서 호출할 수 있는 getInstance() 선언
  - 정적 필드에서 참조하는 자신의 객체 리턴

```
public class 클래스 {
  //정적 필드
  private static 클래스 singleton = new 클래스();

  //생성자
  private 클래스() {}

  //정적 메소드
  static 클래스 getInstance() {
    return singleton;
  }
}
```

```
클래스 변수1 = 클래스.getInstance();
클래스 변수2 = 클래스.getInstance();
```



final 필드와 상수

❖ final 필드

- 초기값이 저장되면 최종값이 되어 프로그램 실행 도중 수정 불가

- final 필드의 초기값 주는 방법
  - 단순 값일 경우 필드 선언 시 초기화(주로 정적 필드(상수)일 경우)
  - 객체 생성 시 외부 데이터로 초기화 필요한 경우 생성자에서 초기화(주로 인스턴스 필드일 경우)

- 인스턴스 final 필드
  - 객체에 한번 초기화된 데이터를 변경 불가로 만들 경우: ex) 주민 번호

```
final 타입 필드 [= 초기값];
```

final String ssn; //생성자에서 초기화

- 정적 final 필드 (관례적으로 모두 대문자로 작성)
  - 불편의 값인 상수를 만들 경우: ex)

```
static final 타입 상수 = 초기값;
```

```
static final double PI = 3.14159;
static final double EARTH_RADIUS = 6400;
static final double EARTH_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
```

키워드로 끝내는 핵심 포인트

- 인스턴스 멤버 : 객체를 생성한 후 사용할 수 있는 필드와 메소드.
  인스턴스 필드. 인스턴스 메소드

- this : 객체 내부에서도 인스턴스 멤버에 접근하기 위해 this를 사용할 수 있음.
  주로 생성자와 메소드의 매개 변수 이름이 필드와 동일한 경우, 인스턴스 멤버인 필드임을 명시

- 정적 멤버 : 클래스에 고정된 멤버로서 객체 생성하지 않고 사용할 수 있는 필드와 메소드

- static : 정적 멤버를 선언할 때 사용하는 키워드입니다.

- 싱글톤 : 전체 프로그램에서 단 하나의 객체만 만들도록 보장해야 하는 경우 사용하는 코드 패턴

- final 필드 : 초기값 저장되면 이것이 최종값이 되어 프로그램 실행 도중 수정할 수 없는 필드.

- 상수 : 불변의 값을 저장하는 정적 필드. final static 키워드로 선언

Chapter 06

클래스

06-6. 패키지와 접근 제한자

혼자 공부하는 자바 (신용권 저)

목차

- 시작하기 전에
- 패키지 선언
- 접근 제한자
- 클래스의 접근 제한
- 생성자의 접근 제한
- 필드와 메소드의 접근 제한
- Getter와 Setter 메소드
- 키워드로 끝내는 핵심 포인트

시작하기 전에

[핵심 키워드] : 패키지 선언, import문, 접근 제한자, Getter/Setter

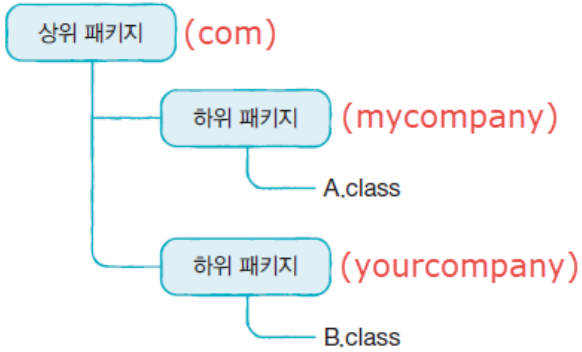
[핵심 포인트]

프로젝트 개발 시 클래스를 체계적으로 관리하기 위해 패키지를 사용한다.
클래스와 클래스의 멤버를 사용 범위에 맞게 접근 제한자를 활용한다.

❖ 패키지

- 패키지의 물리적인 형태는 파일 시스템의 폴더
- 패키지는 클래스의 일부분으로, 클래스를 유일하게 만들어주는 식별자 역할
- 클래스 이름이 동일하더라도 패키지가 다르면 다른 클래스로 인식
- 클래스의 전체 이름은 패키지+클래스 사용해서 다음과 같이 표현

  - 상위패키지.하위패키지.클래스
  - com.mycompany.A
  - com.yourcompany.B



패키지 선언

❖ 패키지 선언

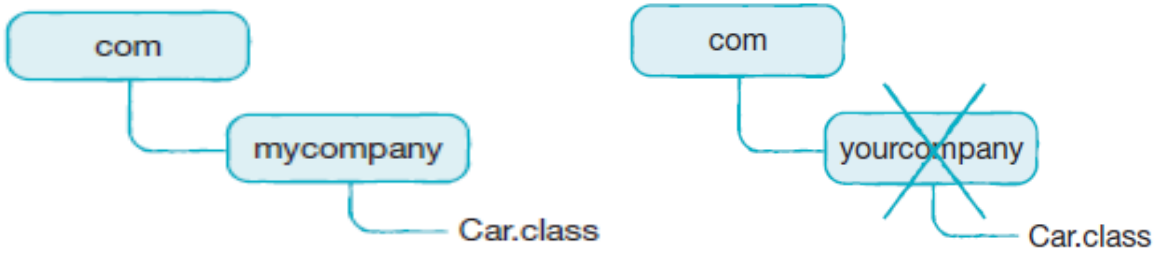
- 클래스 작성 시 해당 클래스가 어떤 패키지에 속할 것인지를 선언

```
package 상위패키지.하위패키지;

public class ClassName { … }
```

```
package com.mycompany;

public class Car { … }
```



- 패키지 이름 규칙
  - - 숫자로 시작 불가
  - - _ 및 $ 제외한 특수문자 사용 불가
  - - java로 시작하는 패키지는 자바 표준 API 에서만 사용하므로 사용 불가
  - - 모두 소문자로 작성하는 것이 관례

패키지 선언

❖ import문

- 사용하고자 하는 클래스 또는 인터페이스가 다른 패키지에 소속된 경우
- 해당 패키지 클래스 또는 인터페이스 가져와 사용할 것임을 컴파일러에 통지

```
import 상위패키지.하위패키지.클래스이름;
import 상위패키지.하위패키지.*;
```

- 패키지 선언과 클래스 선언 사이에 작성

```
package com.mycompany;

import com.hankook.Tire;
[ 또는 import com.hankook.*; ]

public class Car {
 Tire tire = new Tire();
}
```

- 하위 패키지는 별도로 import를 해야함

```
import com.hankook.*;
import com.hankook.project.*;
```

- 다른 패키지에 동일한 이름의 클래스가 있을 경우

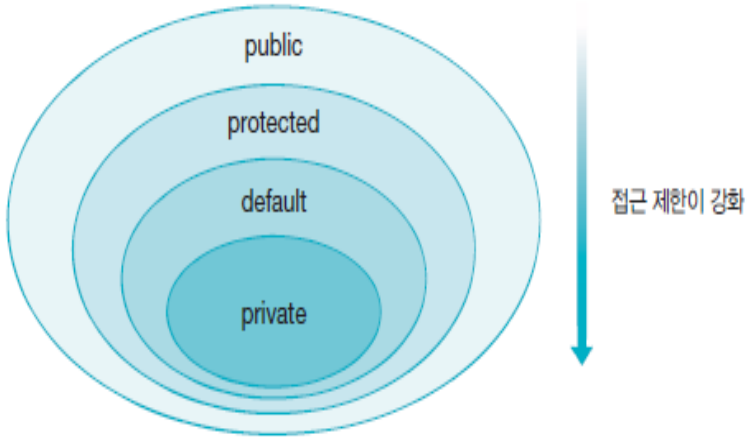
  import와 상관없이 클래스 전체 이름을 기술

접근 제한자

❖ 접근 제한자 (access modifier)

- 클래스와 인터페이스 및 이들이 가진 멤버의 접근 제한

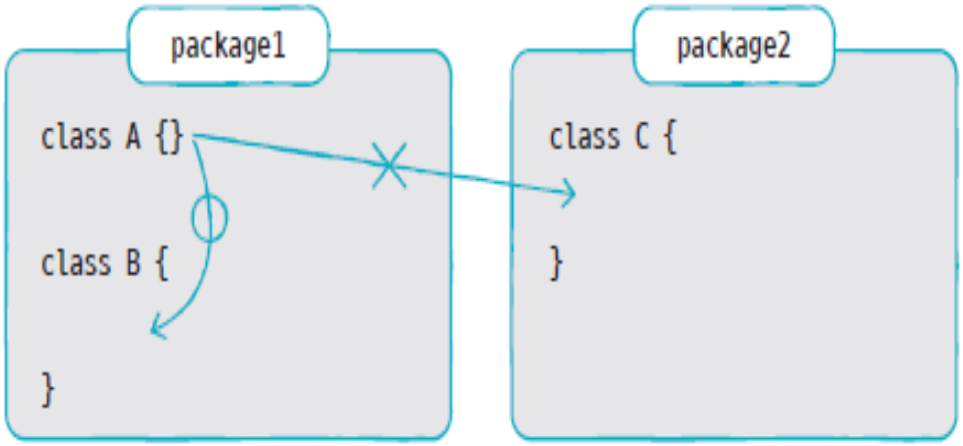
- public 접근 제한자
  - 외부 클래스가 자유롭게 사용할 수 있도록 함
- protected 접근 제한자
  - 같은 패키지 또는 자식 클래스에서 사용할 수 있도록 함
- private 접근 제한자
  - 외부에서 사용할 수 없도록 함
- default 접근 제한
  - 같은 패키지에 소속된 클래스에서만 사용할 수 있도

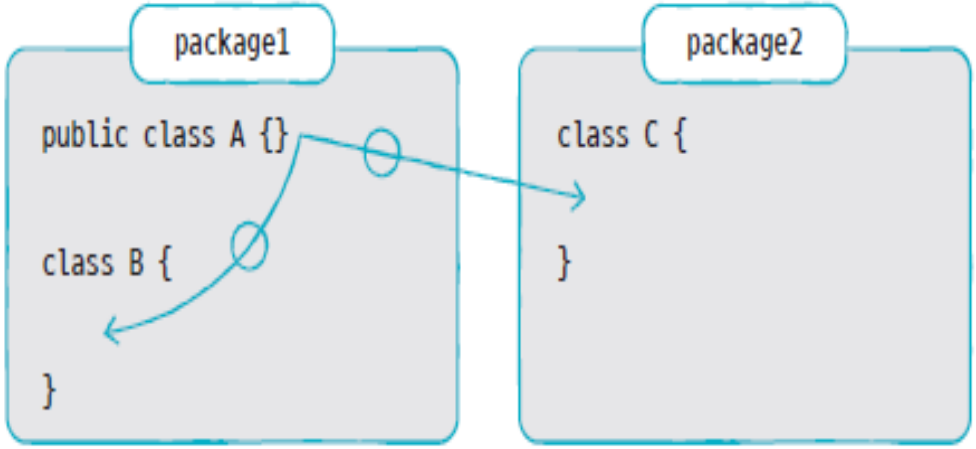


클래스의 접근 제한

❖ 클래스 접근 제한

- 같은 패키지 내에서만 사용할 것인지 다른 패키지 내에서도 사용할 수 있도록 할 것인지 결정





생성자의 접근 제한

❖ 생성자 접근 제한에 따라 생성자 호출 가능 여부 결정

```
public class ClassName {
  //public 접근 제한
  public ClassName(…) { … }

  //protected 접근 제한
  protected ClassName(…) { … }

  //default 접근 제한
  ClassName(…) { … }

  //private 접근 제한
  private ClassName(…) { … }
}
```

필드와 메소드의 접근 제한

❖ 필드와 메소드의 접근 제한

```
//필드 선언
[ public ¦ protected ¦ private ] [static] 타입 필드;

//메소드 선언
[ public ¦ protected ¦ private ] [static] 리턴 타입 메소드(…) { … }
```

Getter와 Setter 메소드

- 외부에서 객체에 마음대로 접근할 경우 객체의 무결성 깨질 수 있음
- Setter 메소드
  - 외부의 값을 받아 필드의 값을 변경하는 것이 목적
  - 매개값 검증하여 유효한 값만 필드로 저장할 수 있음

```
void setSpeed(double speed) {
  if(speed < 0) {
    this.speed = 0;
    return;
  } else {
    this.speed = speed;
  }
}
```

매개값이 음수일 경우 speed 필드에 0으로 저장하고, 메소드 실행 종료

Getter와 Setter 메소드

❖ Getter 메소드

- 외부로 필드값을 전달하는 것이 목적
- 필드값을 가공해서 외부로 전달할 수도 있음

```
double getSpeed() {
  double km = speed*1.6;
  return km;
}
```

필드값인 마일을 km 단위로 환산 후 외부로 리턴

키워드로 끝내는 핵심 포인트

- 패키지 선언 : 해당 클래스 또는 인터페이스가 어떤 패키지에 속할 것인지를 선언.

```
package 상위패키지.하위패키지;
```

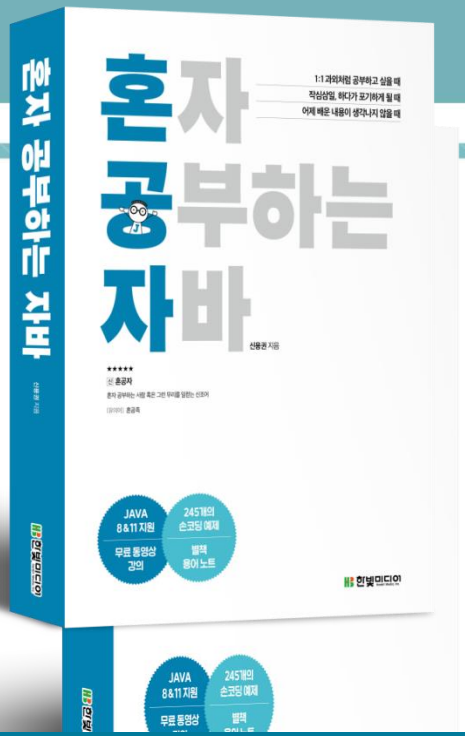
- import문 : 다른 패키지에 소속하는 클래스와 인터페이스를 사용할 경우 필요

```
import 상위패키지.하위패키지.클래스이름;
import 상위패키지.하위패키지.*;
```

- 접근 제한자 : 클래스, 인터페이스, 그리고 멤버들을 사용을 제한할 경우 사용

접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스

- Getter/Setter :
  - 필드 값을 외부로 리턴하는 메소드를 Getter (getXXX(), isXXX())
  - 외부에서 값을 받아 필드값을 변경하는 메소드를 Setter (setXXX())



Thank You!

혼자 공부하는 자바 (신용권 저)

한빛미디어 Hanbit Media, Inc.