

제08장

접근제어자와 정적 멤버

구디아카데미 ▷ 민경태 강사

학습목표

1. 패키지와 import문에 대해서 알 수 있다.
2. 접근제어자의 의미와 종류에 대해서 알 수 있다.
3. Getter와 Setter 메소드에 대해서 알 수 있다.
4. 인스턴스 멤버와 정적 멤버에 대해서 알 수 있다.

목차

1. 패키지와 import
2. 접근제어자의 의미와 종류
3. Getter와 Setter 메소드
4. 인스턴스 멤버와 정적 멤버

```

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 (m) return m.call(t, e, n);
    for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : r; r--;)
      if (n in t && t[r] === e) return n
  }
}

```

1. 패키지와 import

패키지

- 자바의 모든 클래스들의 저장 경로를 관리하기 위해서 패키지가 도입됨
- 패키지를 다르게 지정하면 동일한 이름의 클래스들도 만들 수 있음
- 자바의 모든 클래스는 패키지에 저장해야 함
- 패키지는 계층 구조를 가질 수 있음

package

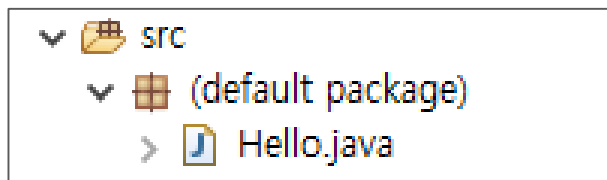
└─ sub package

└─ sub package

└─ class

default package

- 자바의 모든 클래스는 패키지에 저장해야 함
- 패키지 없이 클래스를 만들면 이클립스는 default package를 만든 뒤 그 곳에 클래스를 저장함
- default package는 실제로 존재하는 패키지가 아니기 때문에 항상 패키지를 직접 만들도록 해야 함



default package가 생성된 모습

패키지 작성 방법

- 실제 프로젝트에서는 패키지 이름을 회사 도메인으로 만듦
- 회사 도메인을 역순으로 작성한 뒤 프로젝트명을 추가하는 방식을 주로 사용함
- 예시
삼성에서 개발한 갤럭시 프로그램
com.samsung.galaxy
회사 도메인 역순 프로젝트명

패키지별 클래스

■ 자바 API 주요 패키지별 클래스와 인터페이스

패키지	주요 클래스 및 인터페이스
java.lang	System, String, StringBuilder, Math, Object, Integer 등
java.io	File, InputStream, OutputStream, Reader, Writer 등
java.util	Arrays, List, Set, Map, Calendar, Date, Optional, Random 등
java.time	LocalDate, LocalTime, LocalDateTime 등
java.net	InetAddress, Socket, URL, URLConnection, URLEncoder 등
java.text	SimpleDateFormat, DecimalFormat 등

패키지의 역할

- 패키지는 클래스를 구분하는 역할을 수행함
- 실제로 클래스는 패키지를 함께 작성해 주어야만 함
- 예시

```
com
├── example
│   └── service
│       └── BoardService.java
```

실제 BoardService 클래스의 명칭은 아래와 같다.

com.example.service.BoardService

패키지 생략

- 클래스는 패키지를 함께 작성해 주어야만 한다고 했으나 패키지를 작성하지 않고 생략할 수 있는 경우가 있음

- 클래스 이름 앞에 패키지를 생략할 수 있는 경우

1. 동일한 패키지에 저장된 클래스를 사용할 때
2. java.lang 패키지에 저장된 클래스를 사용할 때
(String, System 클래스 등)
3. import문을 작성해 둔 경우

지금까지 패키지를 생략한 이유

import

- 클래스를 사용하기 전에 한 번만 import문을 작성하면 클래스를 작성할 때 매번 패키지를 작성하는 불편함이 없어짐
- 이클립스에서는 자동완성기능(`Ctrl + Spacebar`)이나 자동 import(`Ctrl + Shift + O`)를 이용해 쉽게 import를 할 수 있음
- 형식
 - `import 패키지.클래스;`
 - `import 패키지.*;`

import 작성 방식1

■ import문 예시-1

```
com
├── example
│   └── service
│       └── BoardService.java
```

패키지 구조

```
import com.example.service.BoardService;
```

하나의 클래스 import 하기

import 작성 방식2

■ import문 예시-2

```
com
├── example
│   └── service
│       ├── BoardService.java
│       ├── UserService.java
│       └── ReserveService.java
```

패키지 구조

import com.example.service.*;

여러 개의 클래스 한 번에 import 하기

```

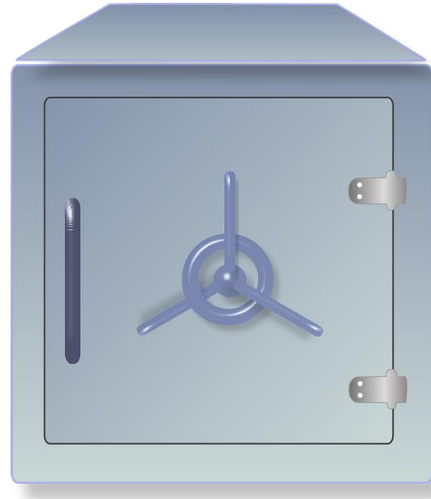
each: function(e, t, n) {
  var r, i = 0,
      o = e.length,
      a = M(e);
  if (n) {
    if (a) {
      for (; o > i; i++)
        if (r = t.apply(e[i], n), r === !1) break;
    } else
      for (i in e)
        if (r = t.apply(e[i], n), r === !1) break;
  } else if (a) {
    for (; o > i; i++)
      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
      if (n in t && t[n] === e) return n;
  }
}

```

2. 접근제어자의 의미와 종류

- Information Hiding
- 객체 지향 프로그래밍의 주요 특징 중 하나
- 객체 내부의 정보를 다른 곳에 보여주지 않고 숨기는 것을 의미함
- 객체 외부에서는 객체 내부 정보에 직접 접근할 수 없음을 의미함
- 객체가 가지고 있는 정보는 필드이므로 필드를 외부로부터 숨기는 것을 정보 은닉이라고 할 수 있음

객체



객체 외부에서는 객체 내부 값에 직접 접근할 수 없다.

접근제어자

- Access Modifier

- 클래스, 필드, 생성자, 메소드와 같은 클래스의 구성 요소마다 접근할 수 있는 권한을 다르게 지정할 수 있음

- 접근제어자를 통해서 접근 권한을 조정함

- 접근제어자 종류

1. private
2. default (기본값)
3. protected
4. public

접근제어자 허용 범위

■ 접근제어자의 종류와 접근 허용 범위

접근제어자	동일 클래스	동일 패키지	자식 클래스	그 외 영역
public	○	○	○	○
protected	○	○	○	×
default	○	○	×	×
private	○	×	×	×

지금까지는 접근제어자를 사용하지 않았으므로 default 접근제어자를 사용한 상황이다.

public

- 모든 영역에서 접근을 허용함
- 일반적으로 클래스와 메소드가 가지는 접근제어자임
- 즉, 클래스와 메소드는 모든 영역에서의 접근을 허용함

```
public class Apple {  
    public void eat() {  
        System.out.println("먹는다");  
    }  
}
```

private

- 오직 클래스 내부에서만 접근을 허용함
- 일반적으로 필드가 가지는 접근제어자임
- 즉, 필드에 접근할 수 있는 건 같은 클래스에 있는 생성자나 메소드임
- 필드를 private 처리를 하는 것이 정보 은닉의 기본 개념임

private


■ private 예시

Apple 클래스 내부에서만 접근 가능한 kind 필드

```
public class Apple {  
  
    private String kind;  
  
    public void info() {  
        System.out.println(kind);  
    }  
}
```

Apple 클래스 외부에서는
Apple 클래스의 private 멤버에 접근 불가

```
public class AppleFactory {  
  
    public static void main(String[] args) {  
        Apple a = new Apple();  
        a.kind = "부사";  
    }  
}
```

 오류 발생

```

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;
  }
}

```

3. Getter와 Setter 메소드

누구나 접근 가능한 public

- 클래스 외부에서는 누구나 public에 접근 가능함



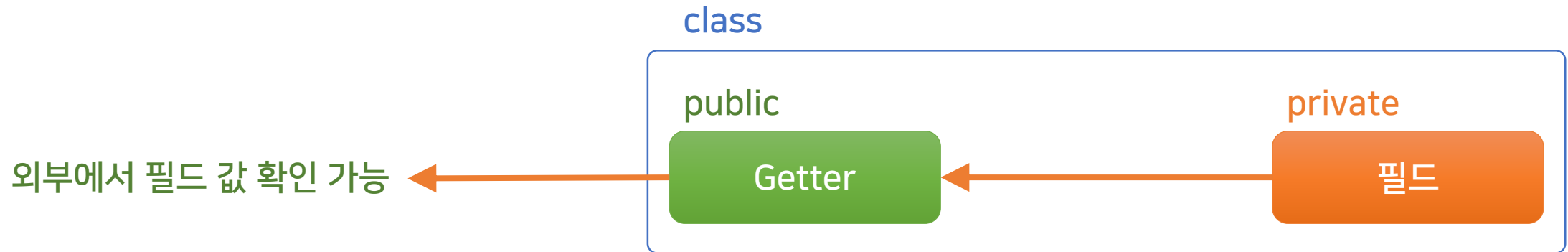
누구에게나 보이는 public 메소드

필드에 접근하기

- 필드는 private 접근제어자를 가지므로 클래스 내부에서만 접근 가능
- 메소드는 public 접근제어자를 가지므로 클래스 외부에서도 접근 가능
- 클래스 외부에서는 메소드에 접근할 수 있으므로 메소드를 통한 필드 접근이 가능함

Getter

- 클래스의 필드 값을 외부에서 확인할 때 사용하는 메소드
- get + 필드이름 형식으로 메소드이름을 만듦
(boolean 자료형인 필드는 is + 필드이름 형식으로 만듦)



Getter

■ Getter 예시

```
public class Member {
```

```
    private String grade;  
    private boolean vip;
```

```
    public String getGrade() {  
        return grade;  
    }
```

```
    public boolean isVip() {  
        return vip;  
    }
```

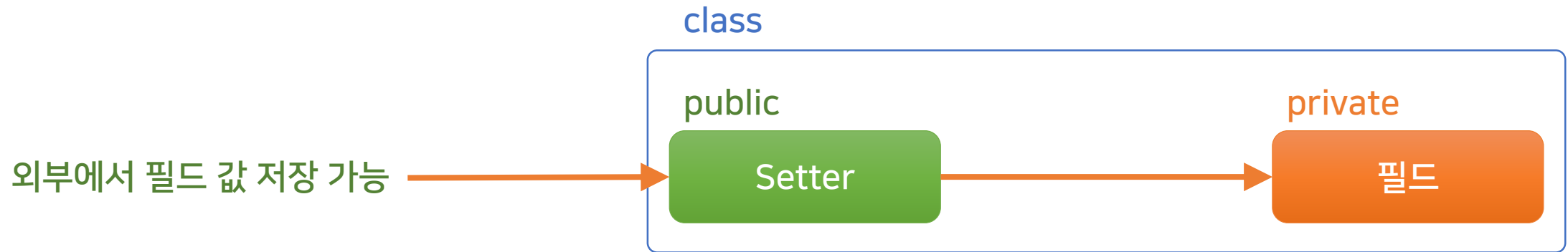
```
}
```

Getter는 필드 값을 반환하는 형식으로 메소드를 구성해야 한다.

boolean 타입의 Getter는 is로 시작한다.

Setter

- 클래스의 필드 값을 저장할 때 사용하는 메소드
- set + 필드이름 형식으로 메소드이름을 만듦



Setter

■ Setter 예시

```
public class Member {  
  
    private String grade;  
    private boolean vip;  
  
    public void setGrade(String grade) {  
        this.grade = grade;  
    }  
  
    public void setVip(boolean vip) {  
        this.vip = vip;  
    }  
  
}
```

Setter는 매개변수를 필드로 전달하는 형식으로 메소드를 구성해야 한다.

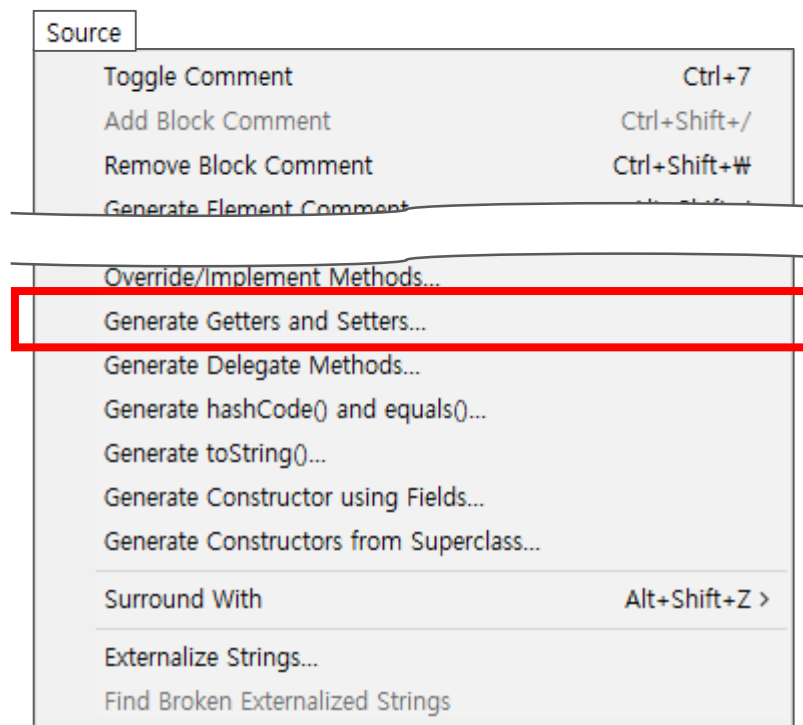
boolean 타입의 Setter도 set으로 시작한다.

Getter와 Setter 주의사항

- Getter와 Setter의 이름은 반드시 규칙을 지켜서 작성해야 함
- 자바 애플리케이션을 개발할 때는 일반적으로 Java, DataBase, 뷰(Jsp, Template engine 등) 등이 함께 사용되는데 이들은 모두 Getter와 Setter를 이용해서 필드에 접근하도록 설계되어있음
- Getter와 Setter의 이름이 규칙과 다르다면 필드에 접근할 수 없으니 반드시 정해진 이름을 사용하도록 미리 노력해야 함

Generate Getters and Setters

- 이클립스는 자동으로 Getter와 Setter를 만들어 줌(적극 활용 권장)
- [Source] - [Generate Getters and Setters...]



```

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;
  }
}

```

4. 인스턴스 멤버와 정적 멤버

인스턴스화

- 메모리를 할당 받아 객체가 생성되는 것을 객체의 인스턴스화(instantiate)라고 함
- 객체 인스턴스화 과정
 1. 메모리 스택(stack) 영역에 객체의 참조 변수 생성
 2. 메모리 힙(heap) 영역에 객체가 저장될 메모리 공간 할당
 3. 힙(heap) 영역에 할당된 메모리 주소값(참조값)을 스택 영역(stack)의 참조 변수에 저장

그래서 객체를 인스턴스라고도 한다.

인스턴스 멤버

- Instance Member
- 인스턴스(객체)가 사용할 수 있는 클래스의 멤버(필드, 메소드)
- 인스턴스를 통해서만 인스턴스 멤버에 접근할 수 있음
- 인스턴스 멤버 접근 과정
 1. 인스턴스 생성
 2. "인스턴스.멤버" 방식으로 호출

인스턴스 멤버 생성

- 인스턴스 멤버를 만드는 별도의 키워드는 없음
- 지금까지 배운 모든 클래스 멤버는 인스턴스 멤버임

```
public class Apple {
```

```
    String kind; 인스턴스 멤버 → kind 필드
```

```
    void eat() {  
        System.out.println(kind + " 먹는다");  
    }
```

```
    인스턴스 멤버 → eat() 메소드
```

```
}
```

```
public class MainClass {
```

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

```
        Apple apple = new Apple();
```

```
        apple.kind = "부사"; 인스턴스 apple을 이용해  
        apple.eat(); kind 필드와 eat() 메소드에 접근
```

```
    }
```

```
}
```

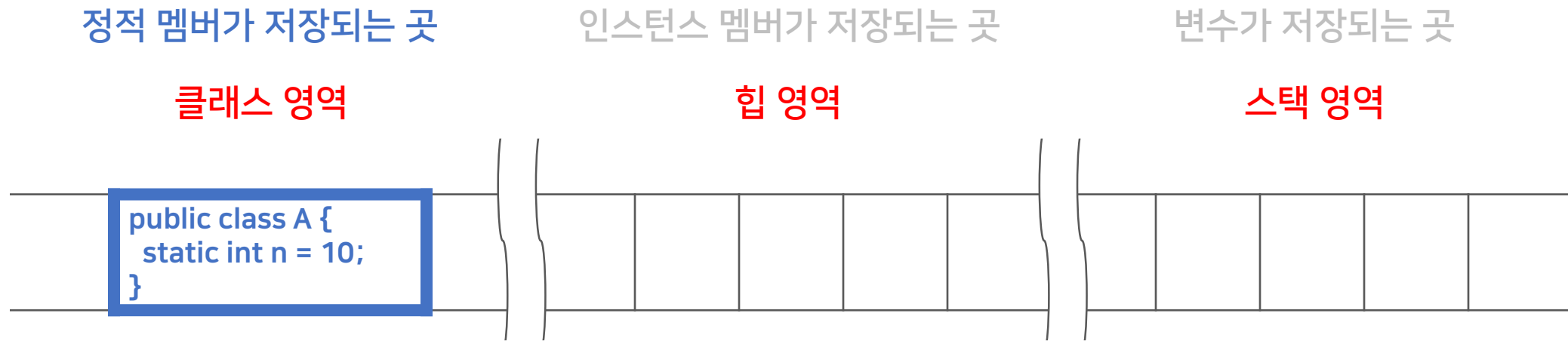
* 인스턴스 멤버 접근 확인을 위해서 필드와 메소드의 접근제어자는 default를 사용하였다.

정적 멤버

- static member
- 인스턴스(객체)가 없어도 사용할 수 있는 클래스의 멤버(필드, 메소드)
- 클래스를 통해서 정적 멤버에 접근할 수 있음
(인스턴스를 이용한 정적 멤버 접근도 허용되지만 비추천)
- static 키워드를 추가하면 정적 멤버가 됨

정적 멤버의 메모리 할당

- 정적 멤버는 메모리의 클래스 영역*을 할당 받음



* 클래스 영역은 메소드 영역이라고도 한다.

메모리 할당 순서

- 클래스 영역 → 스택 영역 → 힙 영역 순서로 메모리가 할당됨

정적 멤버가 저장되는 곳

클래스 영역

인스턴스 멤버가 저장되는 곳

힙 영역

변수가 저장되는 곳

스택 영역



정적 메소드

- 정적 메소드는 가장 먼저 메모리를 할당 받음
- 정적 메소드가 메모리에 생기는 시점에는 인스턴스 멤버(필드, 메소드)가 존재하지 않음
- 따라서 정적 메소드는 인스턴스 멤버를 사용할 수 없음

**정적 메소드 내부에서는
정적 멤버(필드, 메소드)만 사용할 수 있다.**

main() 메소드와 static

- main() 메소드는 static 처리되어 있음
- main() 메소드를 가진 클래스의 인스턴스가 없어도 main() 메소드가 실행가능하도록 main() 메소드는 반드시 정적 메소드로 작성해야 함

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

실행할 때 new MainClass()는 하지 않는다.

즉 MainClass의 인스턴스 생성 없이 main() 메소드가 실행되는 것이다.

인스턴스 생성 없이 main() 메소드가 호출되기 위해서는 반드시 static 키워드가 필요하다.

static 예시

■ static 필드와 메소드

모든 멤버가 static 처리되어 있다.

```
public class MyMath {  
  
    public static final double PI = 3.14;  
  
    public static int max(int a, int b) {  
        return (a > b) ? a : b;  
    }  
  
}
```

MyMath 클래스로 직접 멤버에 접근한다.

```
public class MainClass {  
  
    public static void main(String[] args) {  
        System.out.println(MyMath.PI);  
        System.out.println(MyMath.max(1,2));  
    }  
  
}
```