

# 학습 내용 정리 - 2월 2주차

박시준

# Table of content

- Algorithm - Graph, Hashmap
- Algorithm - Monotonic Stack
- Algorithm - HashSet
- Spring annotation and Http Content-type
- UML – Class diagram
- Server-side Event implementation

Algorithm - Graph, HashMap

# [프로그래머스] 가장 먼 노드

## 가장 먼 노드

### 문제 설명

n개의 노드가 있는 그래프가 있습니다. 각 노드는 1부터 n까지 번호가 적혀있습니다. 1번 노드에서 가장 멀리 떨어진 노드의 갯수를 구하려고 합니다. 가장 멀리 떨어진 노드란 최단경로로 이동했을 때 간선의 개수가 가장 많은 노드들을 의미합니다.

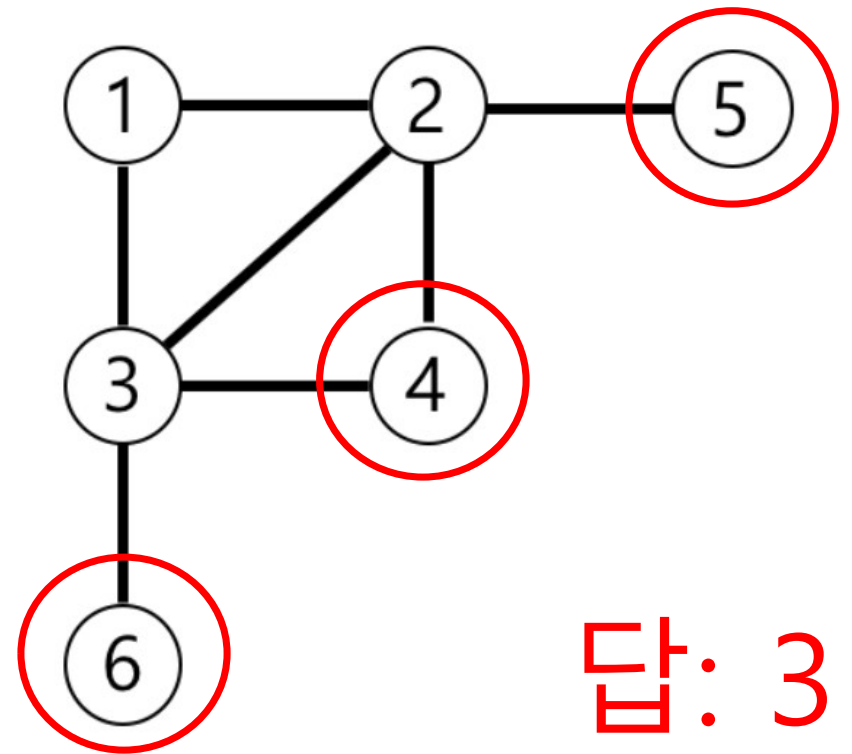
노드의 개수 n, 간선에 대한 정보가 담긴 2차원 배열 vertex가 매개변수로 주어질 때, 1번 노드로부터 가장 멀리 떨어진 노드가 몇 개인지를 return 하도록 solution 함수를 작성해주세요.

### 제한사항

- 노드의 개수 n은 2 이상 20,000 이하입니다.
- 간선은 양방향이며 총 1개 이상 50,000개 이하의 간선이 있습니다.
- vertex 배열 각 행 [a, b]는 a번 노드와 b번 노드 사이에 간선이 있다는 의미입니다.

### 입출력 예

n	vertex	return
6	[[3, 6], [4, 3], [3, 2], [1, 3], [1, 2], [2, 4], [5, 2]]	3



답: 3

# 일단 BFS

- 일단 보자마자 BFS인 걸 알아챈
- 하지만 계속되는 실패로 1시간 30분 만에 풀이 성공
- 첫번째 실패 요인:
  - ⇒노드가 크기 순으로 정렬되어 있지 않음
  - ⇒왼쪽 노드, 오른쪽 노드 모두 조사해줘야 함.

입출력 예

n	vertex	return
6	[[3, 6], [4, 3], [3, 2], [1, 3], [1, 2], [2, 4], [5, 2]]	3

```
if edge[i][0] == 1:
    q.append((edge[i][1], length))
    distance[edge[i][1]] = length
if edge[i][1] == 1:
    q.append((edge[i][0], length))
    distance[edge[i][0]] = length
```

# 시간 초과

- 두 번째 요인: 시간 초과
- 다음 조언이 결정적으로 도움이 됨

**[JS] 8, 9번이 시간초과나고 7번도 3000ms 넘어가시는 분 있으시면 보세요 (코드 x)**

아마 이러신 분들은 bfs 루프 안에서 매번 filter 등으로 edge에서 다음 노드값을 찾아주실 거라고 생각합니다.

그런데 edge는 50,000개 이상의 간선이 있기 때문에 매번 filter로 원하는 값을 찾아주는게 되게 비효율적입니다.

그러니 루프 밖에다가 다음 노드로 향할 값들을 Map 등으로 미리 정의해두는게 어떨까요?

```
{  
  1 => [2, 3]  
  2 => [1, 3, 4, 5]  
}
```

이런 식으로 정의하고 다음노드 찾을 때 저 다음 노드를 전부넣어줘봅시다.

최소 길이 갱신과 함께 하면서 continue를 적절히 이용한다면 금방 해결하실 수 있을거예요

# HashMap, Dictionary 활용

- Dictionary 사용하니 시간 해결됨
- Node Mapping으로 시간을 매우 절약한듯.
- 교훈

⇒ Graph 상태에서 거리를 따져야 하는 문제는 일단 무조건 Dictionary를 떠올려보자

Q. 이런 경우는 해답 안 봐도..?

```
for i in range(len(edge)):
    if edge[i][0] == start_node and distance[edge[i][1]] == -1:
        q.append((edge[i][1], length))
        distance[edge[i][1]] = length
    if edge[i][1] == start_node and distance[edge[i][0]] == -1:
        q.append((edge[i][0], length))
        distance[edge[i][0]] = length
```

```
dic = defaultdict(list)
distance[1] = 0
for i in range(len(edge)):
    dic[edge[i][0]].append(edge[i][1])
    dic[edge[i][1]].append(edge[i][0])
    if edge[i][0] == 1:
        q.append((edge[i][1], length))
        distance[edge[i][1]] = length
    if edge[i][1] == 1:
        q.append((edge[i][0], length))
        distance[edge[i][0]] = length

while q:
    (start_node, leng) = q.popleft()
    leng += 1
    arr = dic[start_node]
    for node in arr:
        if distance[node] == -1:
            distance[node] = leng
            q.append((node, leng))
```

# Algorithm – Monotonic stack



# [LeetCode] 739. Daily Temperature

## 739. Daily Temperatures

Hint 

Medium

 10.1K

 233



 Amazon

 Microsoft

 TikTok



Given an array of integers `temperatures` represents the daily temperatures, return *an array* `answer` such that `answer[i]` is the number of days you have to wait after the  $i^{\text{th}}$  day to get a warmer temperature. If there is no future day for which this is possible, keep `answer[i] == 0` instead.

**Input:** `temperatures = [73,74,75,71,69,72,76,73]`

**Output:** `[1,1,4,2,1,1,0,0]`

# [LeetCode] 739. Daily Temperature

- Stack and Hashmap
- But couldn't solve the problem
- There was a recommended problem before this level



amanchandna

Dec 18, 2022

Must Solve this question first for better fundamentals : <https://leetcode.com/problems/next-greater-element-i/>

Tip

6 Reply

```
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        answer = [0 for _ in range(len(temperatures))]
        dic = {}
        stack = []

        stack.append(temperatures[0])
        dic[temperatures[0]] = 1

        i = 1
        while i < len(temperatures):
            if stack:
                if temperatures[i] > stack[-1]:
                    print(stack, temperatures[i], 1)
                    temp = stack.pop()
                    print(stack, temperatures[i], 1)
                    del(dic[temp])
                    stack.append(temperatures[i])
                    dic[temperatures[i]] = 1
                    continue
                else:
                    stack.append(temperatures[i])
                    while
                        dic[temperatures[i]] = 1
                        i += 1
                    continue
            stack.append(temperatures[i])
```

# [LeetCode] 496. Next Greater Element I

## 496. Next Greater Element I



Easy



5.5K



369



Amazon

Adobe

Bloomberg



The **next greater element** of some element `x` in an array is the **first greater** element that is **to the right** of `x` in the same array.

You are given two **distinct 0-indexed** integer arrays `nums1` and `nums2`, where `nums1` is a subset of `nums2`.

For each  $0 \leq i < \text{nums1.length}$ , find the index `j` such that `nums1[i] == nums2[j]` and determine the **next greater element** of `nums2[j]` in `nums2`. If there is no next greater element, then the answer for this query is `-1`.

Return an array `ans` of length `nums1.length` such that `ans[i]` is the **next greater element** as described above.

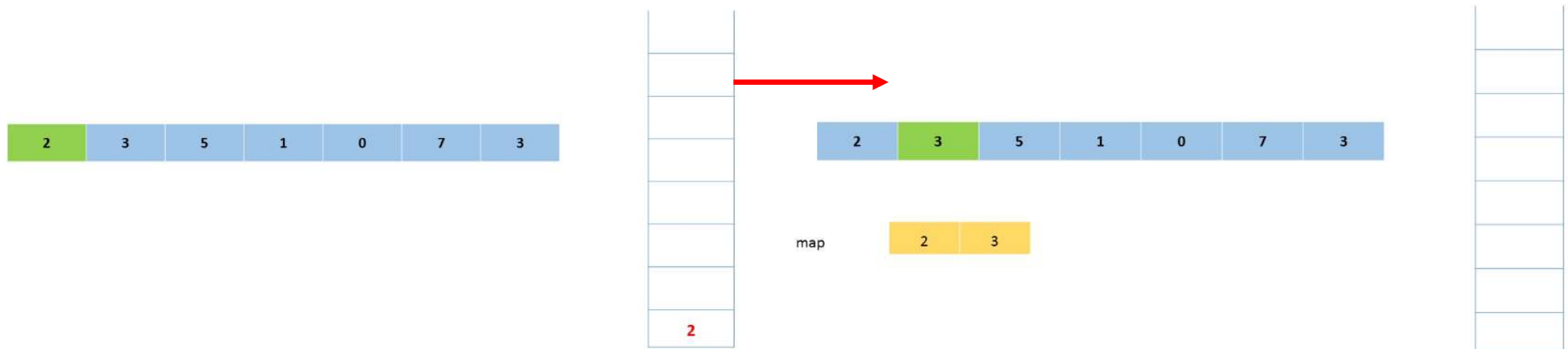
**Input:** `nums1 = [4,1,2]`, `nums2 = [1,3,4,2]`

**Output:** `[-1,3,-1]`

**Explanation:** The next greater element for each value of `nums1` is as follows:  
- 4 is underlined in `nums2 = [1,3,4,2]`. There is no next greater element, so the answer is `-1`.

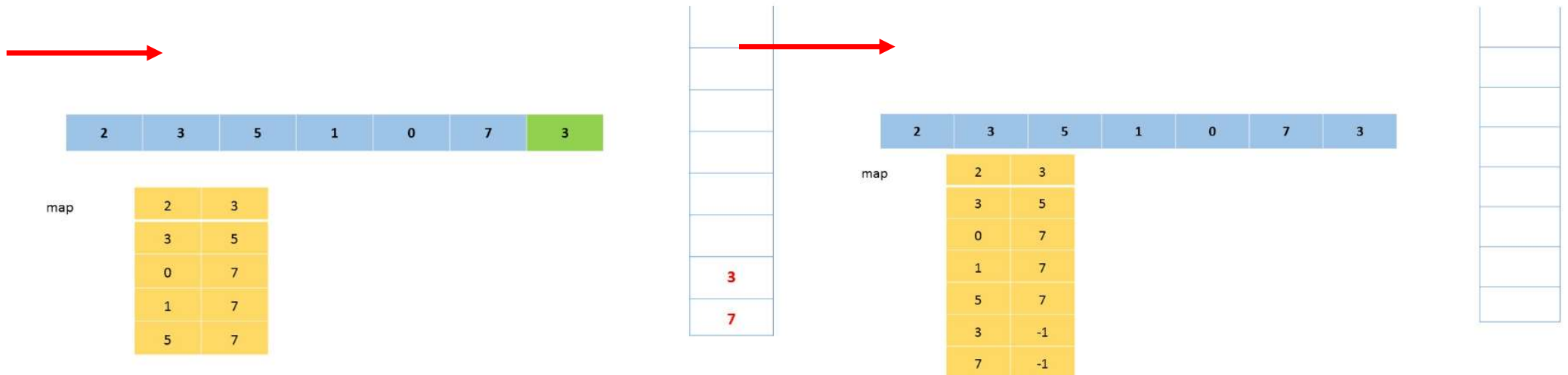
# [LeetCode] 496. Next Greater Element I

- Solved this problem, but with brute force.
- The key idea to solve this problem is “**monotonic stack**”
- Monotonic stack is stack where the elements are always in sorted order



# [LeetCode] 496. Next Greater Element I

- The key idea to solve this problem is "monotonic stack"
- It needs "Stack" and "HashMap(Dictionary)"



# [LeetCode] 496. Next Greater Element I

- The key idea to solve this problem is "monotonic stack"
- It needs "Stack" and "HashMap"
- Two loop strategy
- One for map initialization using stack
- One for making answer array

```
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) ->
        stack = []
        dic = {}
        answer = [-1 for _ in range(len(nums1))]

        for i in range(len(nums2)):
            while(stack and stack[-1] < nums2[i]):
                small = stack.pop()
                dic[small] = nums2[i]
            stack.append(nums2[i])

        while stack:
            leftover = stack.pop()
            dic[leftover] = -1

        for i in range(len(nums1)):
            answer[i] = dic[nums1[i]]

        return answer
```

Loop(1)  
Map 초기화

Loop(2)  
Answer 배열 작성

# [LeetCode] 739. Daily Temperature

- In this case, it also needs "Stack" array and "Hashmap"
- But I initialized "Hashmap" in the loop, while checking stack value
- Initialization of map before check loop is essential to solve this problem easily

- 실패요인:

(1) Dictionary를 개수 체크 용도로 사용했음

```
del(dic[temp])  
stack.append(temperatures[i])  
dic[temperatures[i]] = 1  
continue
```

- 해결책:

(1) Dictionary는 개수가 아닌 노드 값(혹은 노드 index)을 담아야 한다.

# [LeetCode] 739. Daily Temperature

- Monotonic stack
- Stack and Map
- 이 문제에선 Map이 필요 없음. 비교해야 하는 다른 array가 있는 것이 아니기 때문
- Map을 쓴다 해도 개수 count가 아닌, Value나 index를 넣는 용도로 사용해야한다.

```
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        answer = [0 for _ in range(len(temperatures))]
        stack = []

        for day in range(len(temperatures)):
            while stack and temperatures[stack[-1]] < temperatures[day]:
                prev_day = stack.pop()
                answer[prev_day] = day - prev_day
            stack.append(day)

        return answer
```



# Algorithm – HashSet

# [LeetCode] 128. Longest Consecutive sequence

## 128. Longest Consecutive Sequence

Medium



14.9K



614



Google



Amazon



Adobe



Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence*.

You must write an algorithm that runs in  $O(n)$  time.

**Input:** `nums = [100,4,200,1,3,2]`

**Output:** 4

**Explanation:** The longest consecutive elements sequence is `[1, 2, 3, 4]`.  
Therefore its length is 4.

- Given condition:  
 $O(n)$  time complexity
- HashSet과 적절한 조건문이 결합되어야 풀 수 있는 문제

# [LeetCode] 128. Longest Consecutive sequence

- Brute Force solution
- It takes  $O(n^3)$
- Time Limit Exceeded
- First loop -  $O(n)$
- Second loop -  $O(n)$
- 'in' search work -  $O(n)$

```
class Solution:
    def longestConsecutive(self, nums):
        longest_streak = 0

        O(n) for num in nums:
            current_num = num
            current_streak = 1

            O(n) while current_num + 1 in nums: O(n)
                current_num += 1
                current_streak += 1

            longest_streak = max(longest_streak, current_streak)

        return longest_streak
```

# [LeetCode] 128. Longest Consecutive sequence

- HashSet solution
- It takes  $O(n)$
- The key to make  $O(n)$  is "if (조건문)"

ex) [7, 6, 5, 4, 3, 2, 1]

원소 1일 때 조건문을 통과하면 while문에서 streak를 모두 쌓고 끝. 나머지 원소들은 조건문조차 통과 못함.

사실상  $O(n+n) = O(n)$

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        num_set = set(nums)
        longest_streak = 0

        for number in num_set:
            if number - 1 not in num_set:
                current_streak = 1
                current_number = number

                while current_number + 1 in num_set:
                    current_streak += 1
                    current_number += 1

                longest_streak = max(longest_streak, current_streak)

        return longest_streak
```

$O(n)?$

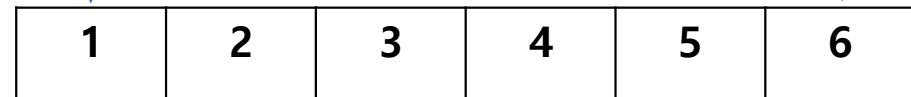
# List에서의 'in' 연산자 성능

- Time complexity of 'in' of List:  $O(n)$
- 첫 번째 인덱스는 1번이면 찾지만, 마지막 인덱스는  $n$ 번 루프 돌려야 찾을 수 있음

```
a = list(range(1, 10000001))  
print(1 in a) # 첫번째 값  
print(10000000 in a) # 마지막 값
```

```
<걸린 시간>  
0.17809176445007324  
0.45179247856140137
```

1번째 탐색

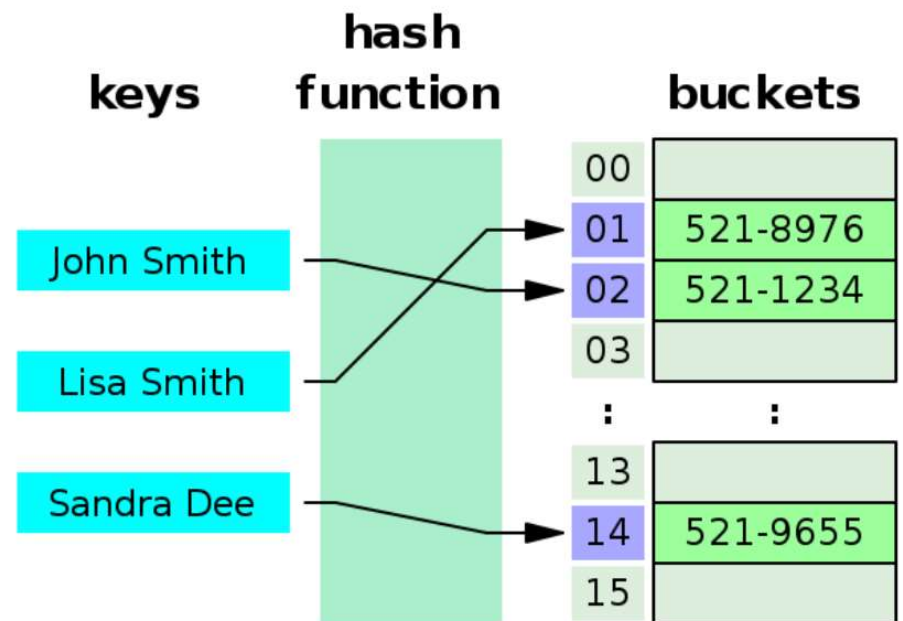


6번째 탐색



# Set, Map에서의 'in' 연산자 성능

- Time complexity of 'in' of Set or Dictionary:  $O(1)$
- 첫번째 index나 마지막 index나 동일한 검색 시간이 걸림
- Set과 Dictionary는 Hash Table Mapping 되어 메모리에 저장이 되고 반환 값으로 그것의 address나 index를 들고 있다.
- Set은 Key만 저장하고, Dictionary는 Key와 Value를 함께 저장함
- Key를 넣으면 해당 index 바로 찾음
- 가끔 다른 key들이 동일한 index를 사용할 경우 충돌 발생  $\rightarrow O(n)$



# List vs Set, Map







- Time complexity of 'in' of List:  $O(n)$
- Time complexity of 'in' of Set or Dictionary:  $O(1)$

# Spring annotation and Http Content-type



# @RequestBody, @RequestParam and @RequestPart

- Table of Experiment to check error

<p>@RequestPart MultipartFile @RequestBody VideoForm</p>  <p>HttpMediaNotSupportedException</p>	<p>@RequestParam MultipartFile @RequestParam VideoForm</p>  <p>MethodArgumentConversionNot SupportedException</p>	<p>@RequestPart MultipartFile @RequestPart VideoForm</p> 
<p>@RequestPart MultipartFile @RequestBody String</p>  <p>HttpMessageNotReadableException</p>	<p>@RequestParam MultipartFile @RequestParam String</p> 	<p>@RequestPart MultipartFile @RequestPart String</p> 

# @RequestBody, @RequestParam and @RequestPart

<https://middleearth.tistory.com/35>

## @RequestBody

HTTP 요청으로 넘어오는 body의 내용을 **HttpMessageConverter**를 통해 Java Object로 역직렬화한다.

## @RequestPart

Content-type이 'multipart/form-data'와 관련된 경우에 사용한다.

MultipartFile이 포함되는 경우에 **MutliPartResolver**가 동작하여 역직렬화를 하게 됨. MultipartFile이 포함되지 않는 경우는 @RequestBody와 같이 **HttpMessageConverter**가 동작하게 된다.

## @RequestParam

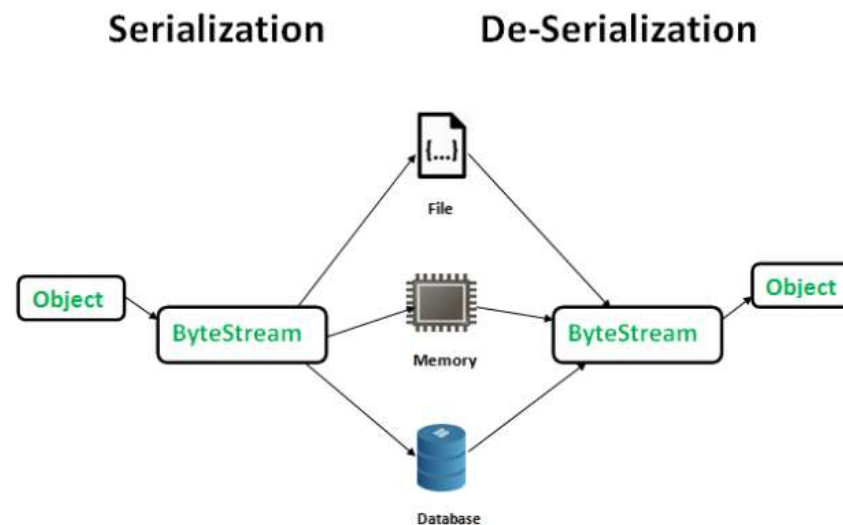
**하나의 파라미터**만을 받을 때 사용된다.

@RequestParam 또한 @RequestPart와 같이 **MultipartFile**을 받을 때 사용할 수 있다.

@RequestPart와의 다른 점은 @RequestParam의 경우 **파라미터가 String이나 MultipartFile이 아닌 경우 Converter나 PropertyEditor에 의해 처리** 되지만 @RequestPart는 **HttpMessageConverter**를 이용하여 Content-type을 참고하여 처리한다는 점이다.

# 역직렬화? 직렬화?

<https://www.geeksforgeeks.org/serialization-in-java>



Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

# @RequestBody

## @RequestBody

HTTP 요청으로 넘어오는 body의 내용을 **HttpMessageConverter**를 통해 Java Object로 역직렬화한다.

- **Spring** 공문서

Annotation indicating a method parameter should be bound **to the body** of the web request. The body of the request is passed through an **HttpMessageConverter** to resolve the method argument depending on the content type of the request.

# @RequestParam

## @RequestParam

Content-type이 'multipart/form-data'와 관련된 경우에 사용한다.

MultipartFile이 포함되는 경우에 **MutliPartResolver**가 동작하여 역직렬화를 하게 됨. MultipartFile이 포함되지 않는 경우는 @RequestBody와 같이 **HttpMessageConverter**가 동작하게 된다.

- **Spring 공문서**

Annotation that can be used to associate the part of a "multipart/form-data" request with a method argument.

Supported method argument types include MultipartFile in conjunction with Spring's **MultipartResolver** abstraction, jakarta.servlet.http.Part in conjunction with Servlet multipart requests, or otherwise for any other method argument, the content of the part is passed through an **HttpMessageConverter** taking into consideration the 'Content-Type' header of the request part. This is analogous to what @RequestBody does to resolve an argument based on the content of a non-multipart regular request.

# @RequestParam

## @RequestParam

하나의 파라미터만을 받을 때 사용된다.

@RequestParam 또한 @RequestPart와 같이 [MultipartFile](#)을 받을 때 사용할 수 있다.

@RequestPart와의 다른 점은 @RequestParam의 경우 파라미터가 [String](#)이나 [MultipartFile](#)이 아닌 경우 [Converter](#)나 [PropertyEditor](#)에 의해 처리 되지만 @RequestPart는 [HttpMessageConverter](#)를 이용하여 Content-type을 참고하여 처리한다는 점이다.

- **Spring** 공문서

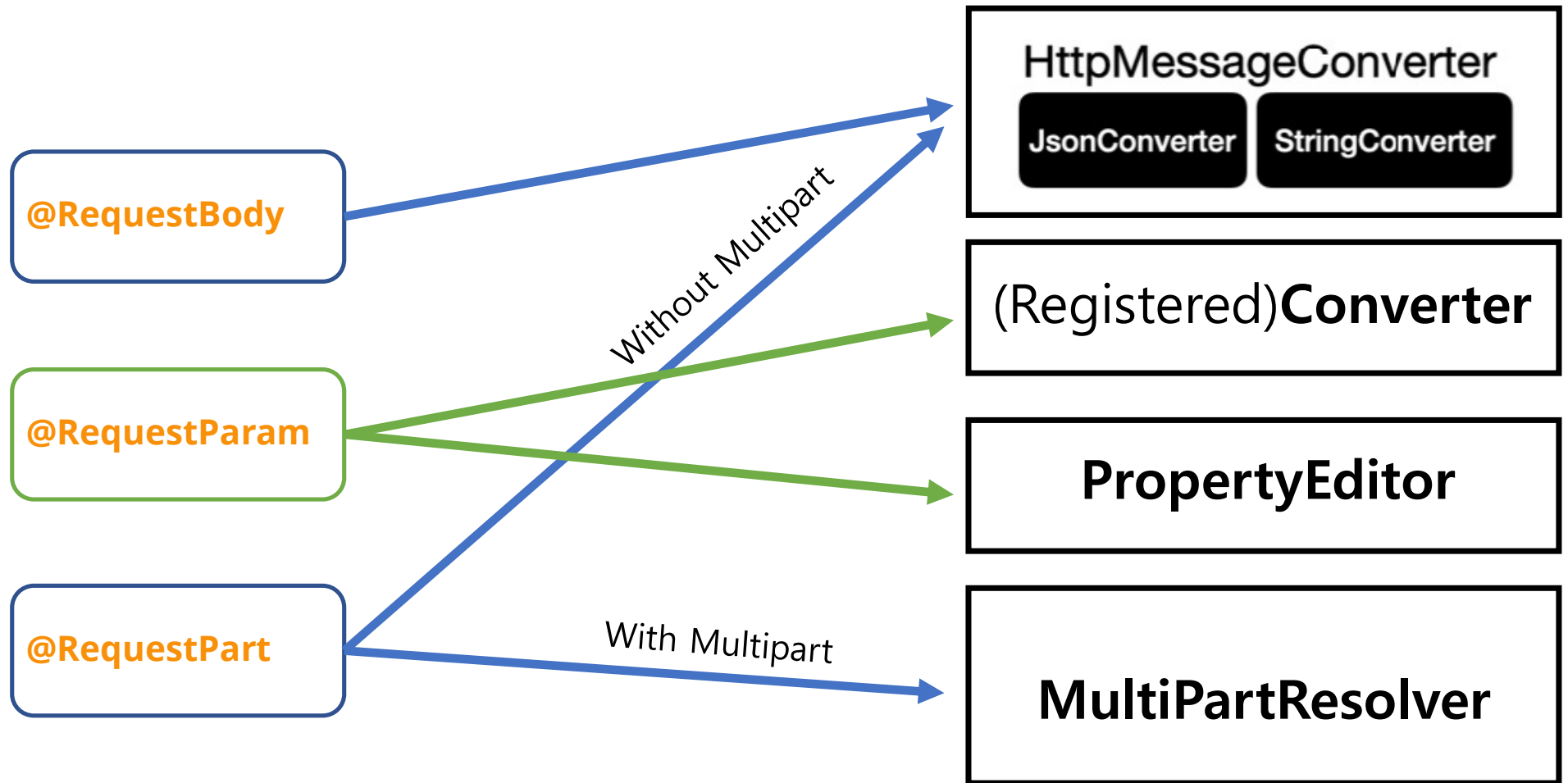
Annotation which indicates that a method parameter should be bound to a **web request parameter**. Note that @RequestParam annotation can also be used to associate the part of a "multipart/form-data" request with a method argument supporting the same method argument types.

The main difference is that when the method argument is not a String or raw MultipartFile / Part, @RequestParam relies on type conversion via a [registered Converter](#) or [PropertyEditor](#) while

RequestPart relies on [HttpMessageConverters](#) taking into consideration the 'Content-Type' header of the request part. RequestParam is likely to be used with name-value form fields while RequestPart is likely to be used with parts containing more complex content e.g. JSON, XML).

# @RequestBody, @RequestParam and @RequestPart

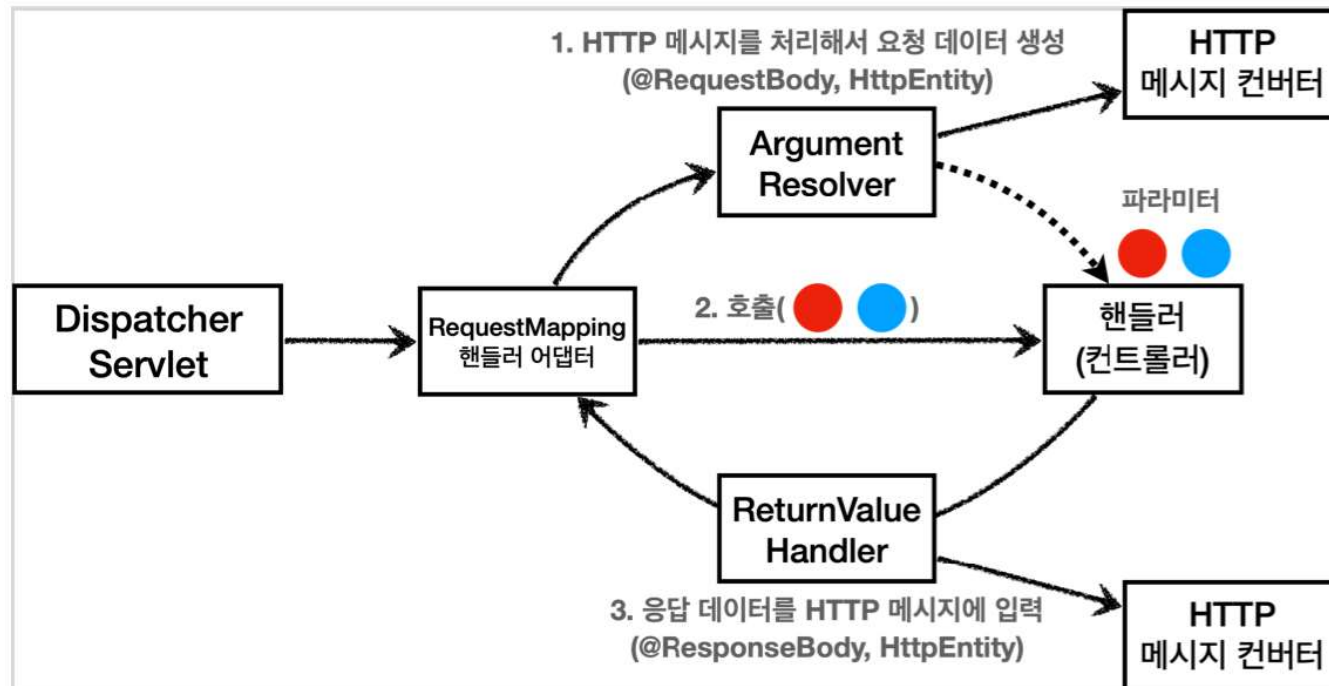
- 정리



# HttpMessageConverter

- RequestMappingHandlerAdapter controls HttpMessageConverter

HTTP 메시지 컨버터 위치





# DispatcherServlet – handler Adapter

- Go into handlerAdapter of doDispatch

```
/deprecation/  
protected void doDispatch(HttpServletRequest request, HttpServletResponse  
    HttpServletRequest processedRequest = request;  
    HandlerExecutionChain mappedHandler = null;  
    boolean multipartRequestParsed = false;
```

```
}
```

```
// Determine handler adapter for the current request.
```

```
HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
```

```
// Process last-modified header, if supported by the handler.
```

```
String method = request.getMethod();
```

```
boolean isGet = HttpMethod.GET.matches(method);
```

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws  
    if (this.handlerAdapters != null) {  
        for (HandlerAdapter adapter : this.handlerAdapters) {  
            if (adapter.supports(handler)) {  
                return adapter;  
            }  
        }  
    }
```

```
meController 50 public interface HandlerAdapter {  
    Cho  
    AbstractHandlerMethodAdapter (org.springframework.web.servlet.m  
    HandlerFunctionAdapter (org.springframework.web.servlet.function  
    HttpRequestHandlerAdapter (org.springframework.web.servlet.mvc)  
    RequestMappingHandlerAdapter (org.springframework.web.servlet.m  
    SimpleControllerHandlerAdapter (org.springframework.web.servlet  
    SimpleServletHandlerAdapter (org.springframework.web.servlet.ha
```

# RequestMappingHandlerAdapter

- HandlerAdapter <- RequestMappingHandlerAdapter

```
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
    implements BeanFactoryAware, InitializingBean {
```

```
    private List<HandlerMethodArgumentResolver> getDefaultArgumentResolvers() {
        List<HandlerMethodArgumentResolver> resolvers = new ArrayList<>(initialCapacity: 30);

        // Annotation-based argument resolution
        resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), useDefaultResolution: false));
        resolvers.add(new RequestParamMapMethodArgumentResolver());
    }
```

```
    public void setArgumentResolvers(@Nullable List<HandlerMethodArgumentResolver>
        argumentResolvers) {
        if (argumentResolvers == null) {
            this.argumentResolvers = null;
        }
        else {
            this.argumentResolvers = new HandlerMethodArgumentResolverComposite();
            this.argumentResolvers.addResolvers(argumentResolvers);
        }
    }
}
```

# RequestMappingHandlerAdapter

- List of Argument Resolver

```
private List<HandlerMethodArgumentResolver> getDefaultArgumentResolvers() {  
    List<HandlerMethodArgumentResolver> resolvers = new ArrayList<>(initialCapacity: 30);  
  
    // Annotation-based argument resolution  
    resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), useDefaultResolution: false));  
    resolvers.add(new RequestParamMapMethodArgumentResolver());  
    resolvers.add(new PathVariableMethodArgumentResolver());  
    resolvers.add(new PathVariableMapMethodArgumentResolver());  
    resolvers.add(new MatrixVariableMethodArgumentResolver());  
    resolvers.add(new MatrixVariableMapMethodArgumentResolver());  
    resolvers.add(new ServletModelAttributeMethodProcessor(annotationNotRequired: false));  
    resolvers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(), this.requestRes);  
    resolvers.add(new RequestPartMethodArgumentResolver(getMessageConverters(), this.requestResp);  
    resolvers.add(new RequestHeaderMethodArgumentResolver(getBeanFactory()));  
    resolvers.add(new RequestHeaderMapMethodArgumentResolver());  
    resolvers.add(new ServletCookieValueMethodArgumentResolver(getBeanFactory()));  
    resolvers.add(new ExpressionValueMethodArgumentResolver(getBeanFactory()));  
    resolvers.add(new SessionAttributeMethodArgumentResolver());  
    resolvers.add(new RequestAttributeMethodArgumentResolver());  
}
```

# ArgumentResolver

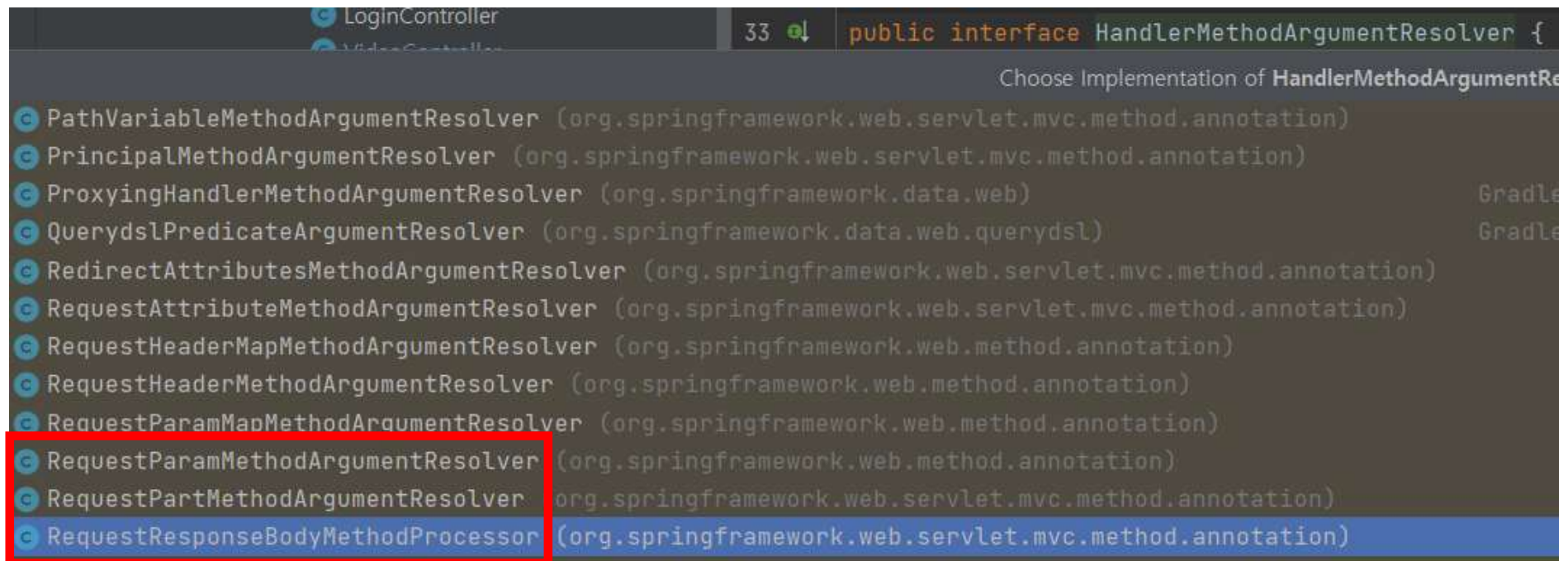
- HandlerMethodArgumentResolver interface

```
public interface HandlerMethodArgumentResolver {  
  
    Whether the given method parameter is supported by this resolver.  
    Params: parameter – the method parameter to check  
    Returns: true if this resolver supports the supplied parameter; false otherwise  
  
    boolean supportsParameter(MethodParameter parameter);  
}
```



# ArgumentResolver

- HandlerMethodArgumentResolver implementation



The screenshot shows an IDE window with the `HandlerMethodArgumentResolver` interface and a list of its implementations. The interface is defined as `public interface HandlerMethodArgumentResolver`. Below it, a list of implementations is shown, each with a radio button for selection. The `RequestParamMethodArgumentResolver` is highlighted with a red box, and the `RequestBodyMethodProcessor` is highlighted with a blue box.

```
public interface HandlerMethodArgumentResolver {  
    // ...  
}
```

Choose Implementation of `HandlerMethodArgumentResolver`

- ☐ `PathVariableMethodArgumentResolver` (`org.springframework.web.servlet.mvc.method.annotation`)
- ☐ `PrincipalMethodArgumentResolver` (`org.springframework.web.servlet.mvc.method.annotation`)
- ☐ `ProxyingHandlerMethodArgumentResolver` (`org.springframework.data.web`) Gradle
- ☐ `QuerydslPredicateArgumentResolver` (`org.springframework.data.web.querydsl`) Gradle
- ☐ `RedirectAttributesMethodArgumentResolver` (`org.springframework.web.servlet.mvc.method.annotation`)
- ☐ `RequestAttributeMethodArgumentResolver` (`org.springframework.web.servlet.mvc.method.annotation`)
- ☐ `RequestHeaderMapMethodArgumentResolver` (`org.springframework.web.method.annotation`)
- ☐ `RequestHeaderMethodArgumentResolver` (`org.springframework.web.method.annotation`)
- ☐ `RequestParamMapMethodArgumentResolver` (`org.springframework.web.method.annotation`)
- ☒ `RequestParamMethodArgumentResolver` (`org.springframework.web.method.annotation`)
- ☐ `RequestPartMethodArgumentResolver` (`org.springframework.web.servlet.mvc.method.annotation`)
- ☐ `RequestBodyMethodProcessor` (`org.springframework.web.servlet.mvc.method.annotation`)

# RequestMappingHandlerMethodProcessor

- HandlerMethodArgumentResolver implementation

```
*/
public class RequestResponseBodyMethodProcessor extends AbstractMessageConverterMethodPr

    Basic constructor with converters only. Suitable for resolving @RequestBody. For handling
    @ResponseBody consider also providing a ContentNegotiationManager.
    public RequestResponseBodyMethodProcessor(List<HttpMessageConverter<?>> converters)

    Basic constructor with converters and ContentNegotiationManager. Suitable for resolving
    @RequestBody and handling @ResponseBody without Request~ or ResponseBodyAdvice.
    public RequestResponseBodyMethodProcessor(List<HttpMessageConverter<?>> converters,
        @Nullable ContentNegotiationManager manager) {

        super(converters, manager);
    }
}
```

- Super: 부모 클래스 생성자를 작동시킴

# HttpMessageConverter

- HttpMessageConverter interface

```
public interface HttpMessageConverter<T> {
```

```
    read(Class<? extends T> clazz, HttpInputMessage inputMessage)  
        throws IOException, HttpMessageNotReadableException;
```

```
    void write(T t, @Nullable MediaType contentType, HttpOutputMessage outputMessage)  
        throws IOException, HttpMessageNotWritableException;
```

# AbstractJackson2HttpMessageConverter

```
public abstract class AbstractJackson2HttpMessageConverter extends  
  
    private static final Map<String, JsonEncoding> ENCODINGS;
```

```
protected JsonEncoding getJsonEncoding(@Nullable MediaType contentType) {  
    if (contentType != null && contentType.getCharset() != null) {  
        Charset charset = contentType.getCharset();  
        JsonEncoding encoding = ENCODINGS.get(charset.name());  
        if (encoding != null) {  
            return encoding;  
        }  
    }  
    return JsonEncoding.UTF8;  
}
```

```
public void setObjectMapper(ObjectMapper objectMapper) {  
    Assert.notNull(objectMapper, message: "ObjectMapper must not be null");  
    this.defaultObjectMapper = objectMapper;  
    configurePrettyPrint();  
}
```



# AbstractJackson2HttpMessageConverter

- Transforms JSON data in letters into Java objects using objectMapper, a Jackson library.

```
@Override
protected void writeInternal(Object object, @Nullable Type type, HttpOutputMessage outputMessage)
    throws IOException, HttpMessageNotWritableException {

    MediaType contentType = outputMessage.getHeaders().getContentType();
    JsonEncoding encoding = getJsonEncoding(contentType);

    Class<?> clazz = (object instanceof MappingJacksonValue mappingJacksonValue ?
        mappingJacksonValue.getValue().getClass() : object.getClass());
    ObjectMapper objectMapper = selectObjectMapper(clazz, contentType);
    Assert.state( expression: objectMapper ≠ null, () → "No ObjectMapper for " + clazz.getName());

    OutputStream outputStream = StreamUtils.nonClosing(outputMessage.getBody());
```

# AbstractJackson2HttpMessageConverter

```
try (JsonGenerator generator = objectMapper.getFactory().createGenerator(outputStream, encoding))
    writePrefix(generator, object);

Object value = object;
Class<?> serializationView = null;
FilterProvider filters = null;
JavaType javaType = null;

if (object instanceof MappingJacksonValue mappingJacksonValue) {
    value = mappingJacksonValue.getValue();
    serializationView = mappingJacksonValue.getSerializationView();
    filters = mappingJacksonValue.getFilters();
}

if (type != null && TypeUtils.isAssignable(type, value.getClass())) {
    javaType = getJavaType(type, contextClass: null);
}

ObjectWriter objectWriter = (serializationView != null ?
    objectMapper.writerWithView(serializationView) : objectMapper.writer());
```

```
    }
    objectWriter = customizeWriter(objectWriter, javaType);
    objectWriter.writeValue(generator, value);

    writeSuffix(generator, object);
    generator.flush();
}
```

# overall flow (@RequestBody의 경우)

DispatcherServlet의 .getHandlerAdapter()

-> HandlerAdapter

-> RequestMappingHandlerAdapter의 .getDefaultArgumentResolver()

(분기점 @RequestBody, @RequestParam, @RequestPart)

(@RequestBody의 경우)

-> **new** RequestResponseBodyMethodProcessor() ->

getMessageConverters() -> HttpMessageConverters

-> AbstractJackson2HttpMessageConverters의 **.writeInternal()**

-> value = mappingJacksonValue.getValue(),

objectWriter.writeValue(generator, value)

# @RequestBody

## @RequestBody

HTTP 요청으로 넘어오는 body의 내용을 [HttpMessageConverter](#)를 통해 Java Object로 역직렬화한다.

- **Spring** 공문서

Annotation indicating a method parameter should be bound **to the body** of the web request. The body of the request is passed through an [HttpMessageConverter](#) to resolve the method argument depending on the content type of the request.

# RequestMappingHandlerAdapter

- Comeback to..
- HandlerAdapter <- ArgumentResolver <- HttpConverter
- RequestBody vs RequestPart vs RequestParam

```
resolvers.add(new MatrixVariableMethodArgumentResolver());  
resolvers.add(new ServletModelAttributeMethodProcessor(annotationNotRequired: false));  
resolvers.add(new RequestResponseBodyMethodProcessor(getMessageConverters(), this.requestResponseBodyAdvice));  
resolvers.add(new RequestPartMethodArgumentResolver(getMessageConverters(), this.requestResponseBodyAdvice));  
resolvers.add(new RequestHeaderMethodArgumentResolver(getBeanFactory()));
```

```
resolvers.add(new PrincipalMethodArgumentResolver());  
resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(),  
resolvers.add(new ServletModelAttributeMethodProcessor(annotationNotRequired:
```

# DispatcherServlet - checkMultipart

- There

```
protected HttpServletRequest checkMultipart(HttpServletRequest request) throws
    if (this.multipartResolver != null && this.multipartResolver.isMultipart(
        if (WebUtils.getNativeRequest(request, MultipartHttpServletRequest.class) != null) {
            if (DispatcherType.REQUEST.equals(request.getDispatcherType())) {
                logger.trace("Request already resolved to MultipartHttpServletRequest");
            }
        }
    }
```

UML – class diagram

# UML

- UML(Unified Modeling Language)
- Object Management Group에서 표준으로 채택한 통합 모델링 언어
- 무작정 쓰면 안 되고, 시험해볼 구체적인 것이 있고, 코드로 시험해보는 것보다 UML로 시험해보는 것이 비용이 덜 들 때 사용해야 한다.
- UML 용도
  - 다른 사람들과 의사소통하고 설계에 대해 논의할 때
  - 전체 시스템의 구조 및 클래스의 의존성 파악
  - 유지보수를 위한 설계의 백엔드 문서



# UML - 다이어그램

- UML은 구조 다이어그램 7개, 행위 다이어그램 7개로 구성되어 있음
- 구조 다이어그램은 개념, 관계 측면에서 요소들을 나타내고, 요소들의 정적인 면을 보기 위한 것임
- 행위 다이어그램은 각 요소들 혹은 요소들 간의 변화나 흐름, 주고받는 데이터 등의 동작을 보기위한 것임.

# UML - 차원

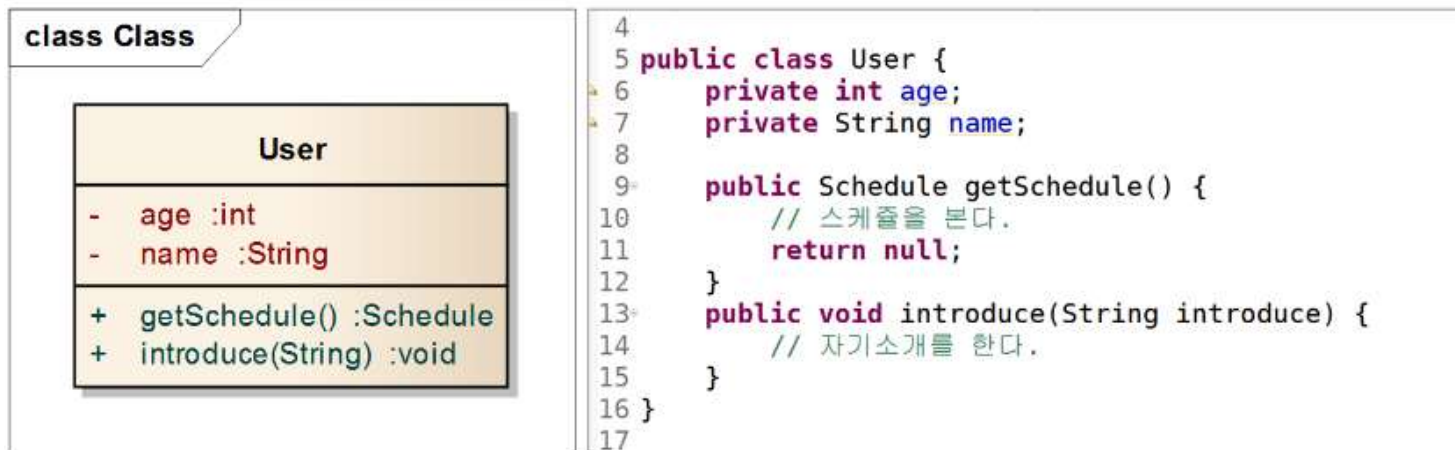
- UML은 목적에 따라 다르게 사용되는데, 크게 3가지로 개념, 명세, 구현의 차원들이다.
- 개념 차원의 UML은 문제 도메인의 구조를 나타낸다. 사람이 풀고자 하는 문제 도메인 안에 있는 개념과 추상적 개념을 기술하기 위한 것이다. 사람의 언어와 더 관련 있다.
- 명세와 구현 차원의 UML은 소프트웨어 구현 설명 목적 등으로 사용하며 소스코드와 관계가 깊다.
- 클래스 다이어그램은 둘 다 가능하지만, 차후 설명은 명세와 구현 차원의 용도로 사용됨

# 클래스 다이어그램

- 클래스 다이어그램은 클래스 내부의 정적인 내용 + 클래스 사이의 관계를 표기하는 다이어그램. 시스템의 일부 또는 전체의 구조를 나타낼 수 있다.
- 순환 의존이 발생하는 지점을 찾아서 순환고리를 어떻게 깨는 것이 좋은지 결정할 수 있게 해준다.

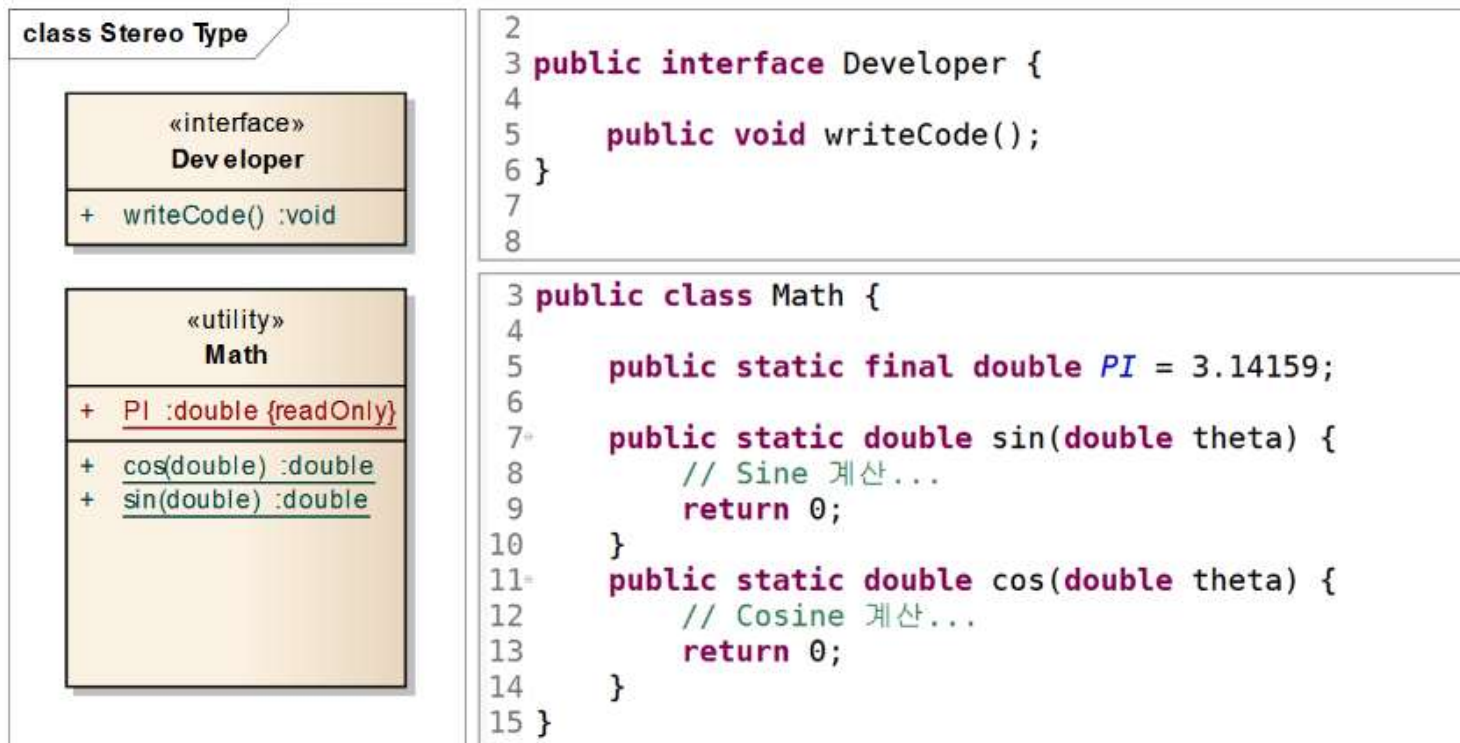
# 클래스 다이어그램 요소

- 클래스는 3개의 compartment(구획)으로 나눈다. 이름, 속성, 기능
- 속성과 기능은 옵션이지만, 이름은 필수.
- 속성에는 필드의 개요를 적고, 기능에는 메서드의 개요를 적는다.
- UML 다이어그램에는 모든 필드와 메서드를 적을 필요는 없다.



# 클래스 다이어그램 요소

- 스트레오 타입: 추가적인 확장요소. "<<>>" guillemet 사이에 적는다.
- 많이 사용되는 것은 <<interface>>, <<utility>>, <<abstract>>



# 클래스 다이어그램 요소

- 추상 클래스/ 메서드: 구현체가 없고 명세만 존재하는 클래스
- 추상 클래스/ 메서드는 italic체나 {abstract}를 사용하여 표기한다.

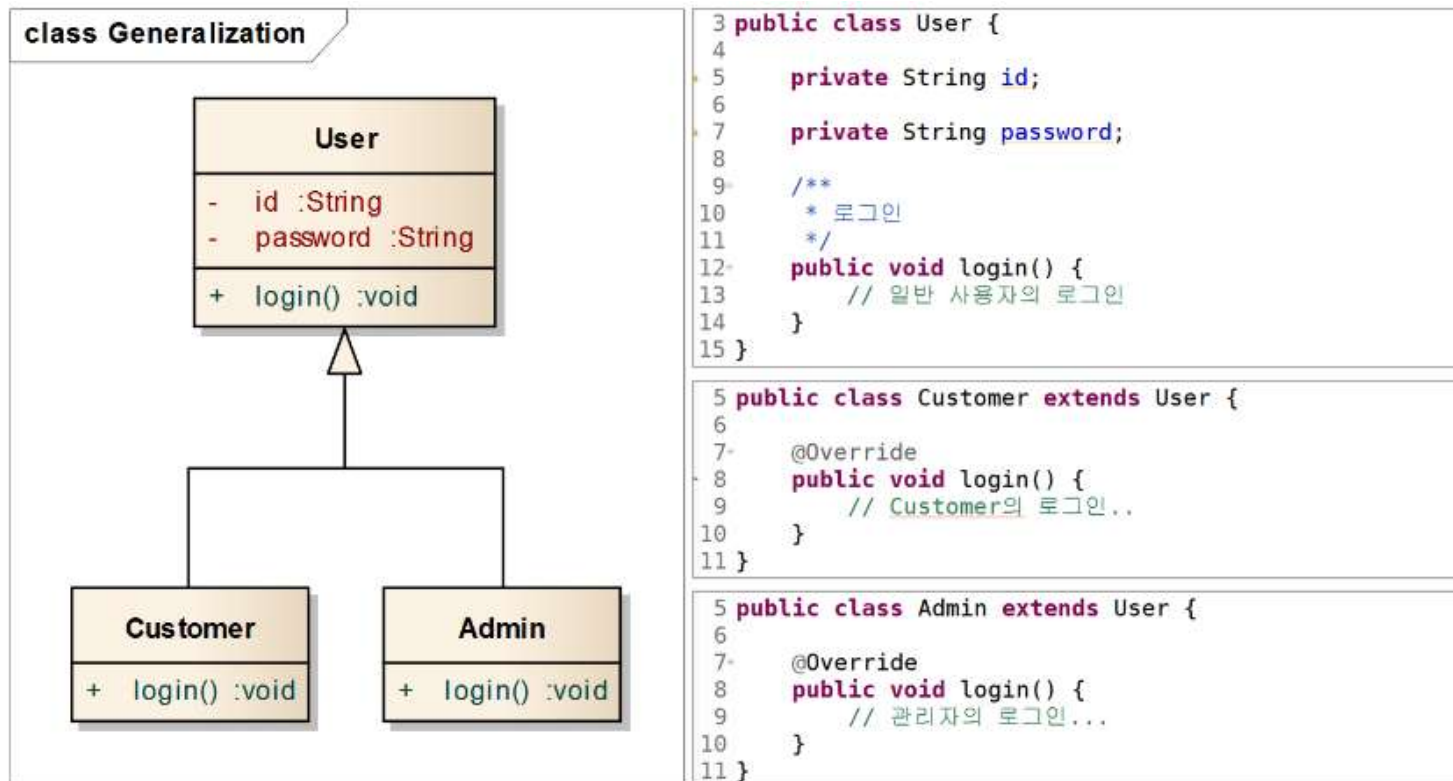
# 클래스 간 관계

- 클래스 간 관계

관계	UML 표기
Generalization (일반화)	
Realization (실체화)	
Dependency (의존)	
Association (연관)	
Directed Association (직접연관)	
Aggregation (집합, 집합연관)	
	
Composition (합성, 복합연관)	
	

# 클래스 간 관계

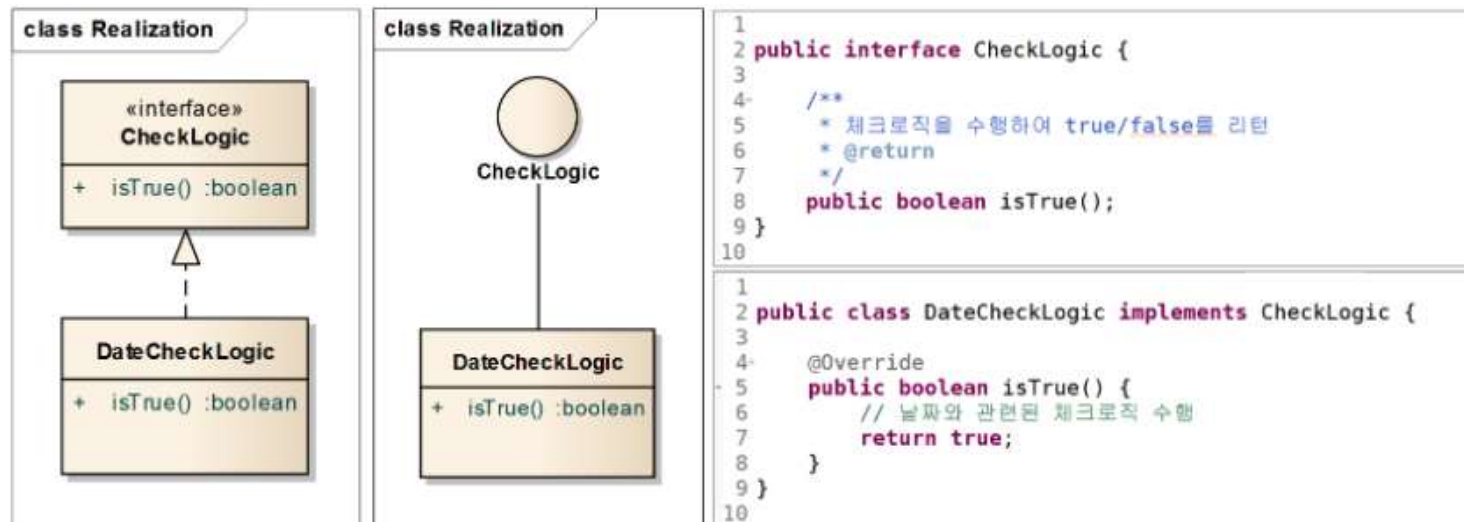
- Generalization 일반화: 부모클래스와 자식클래스 간의 Inheritance를 나타낸다.





# 클래스 간 관계

- Realization 실체화: interface 에서 spec 만 있는 메서드를 오버라이딩 하여 구현하는 것을 의미.



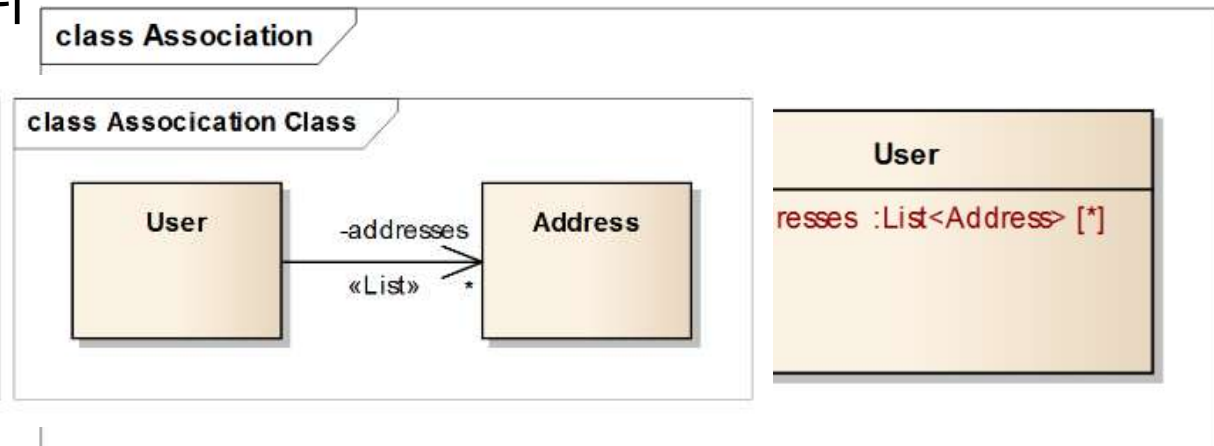
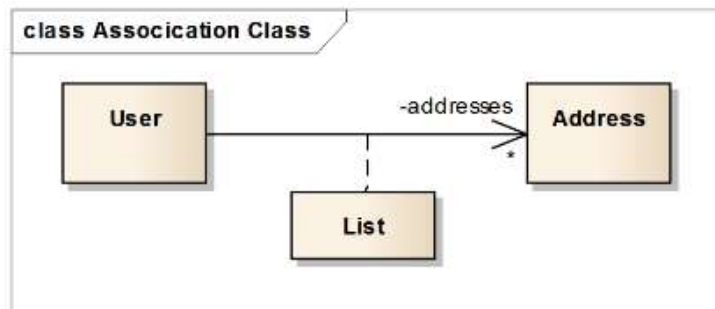
# 클래스 간 관계

- Dependency(의존): 어떤 클래스가 다른 클래스를 참조하는 것을 의미
- 메서드 내에서 대상 클래스의 객체 생성, 객체 사용, 메서드 호출, 객체 리턴, 매개변수로 해당 객체를 받는 것 전부 다 해당한다.



# 클래스 간 관계

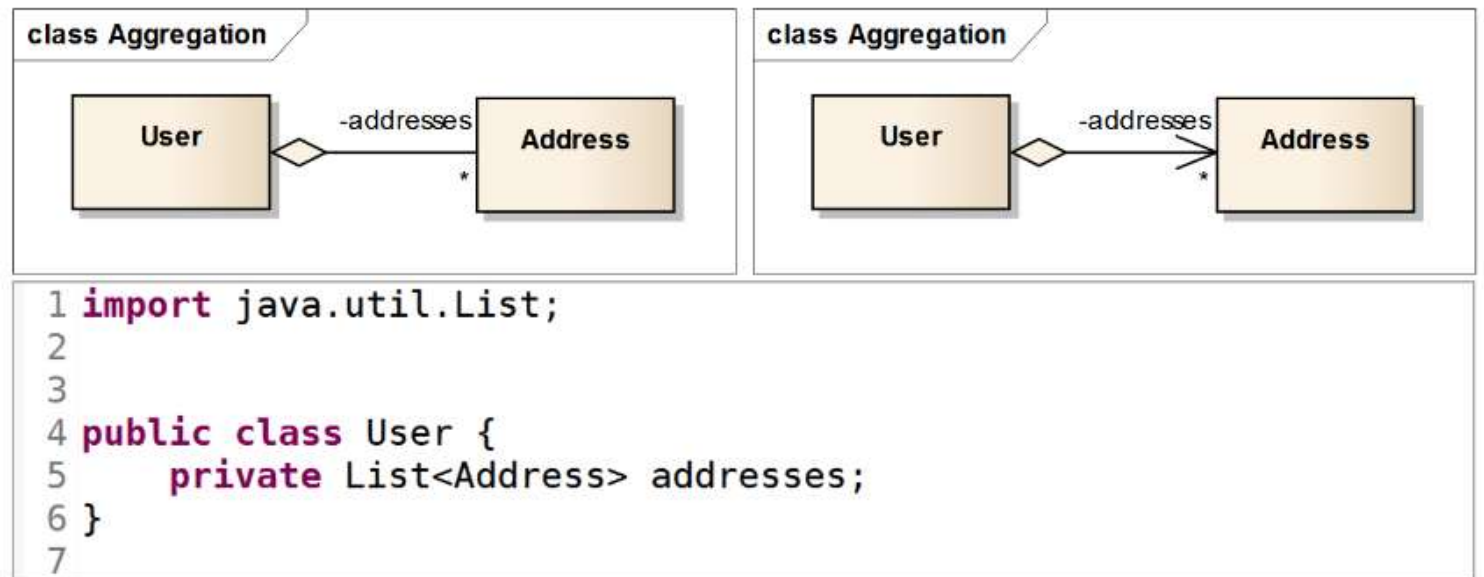
- Association(연관): 다른 객체의 참조를 가지는 필드를 의미.
- navigability(방향성)가 있을 수도 있고 없을 수도 있다.
- A -> B 이면 A가 B를 참조한다



```
1 import java.util.List;
2
3
4 public class User {
5     private List<Address> addresses;
6 }
7
```

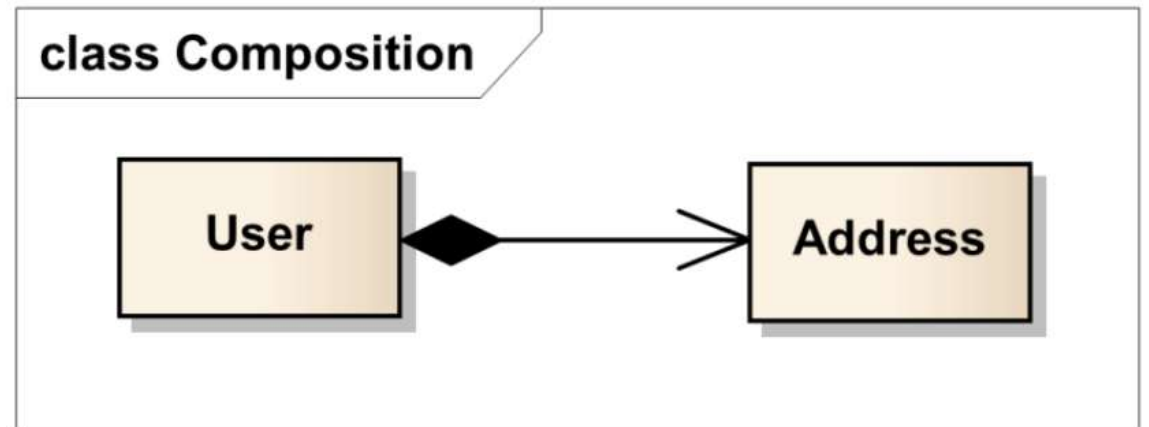
# 클래스 간 관계

- Aggregation(집합): 전체와 부분의 관계를 나타낸다.
- Whole쪽에 다이아몬드 표기
- Part쪽에는 화살표 표시해도 되고 안 해도 됨
- 하지만 코드는 Association과 달라지는 게 없음.
- 이걸 강 쓰지 마라



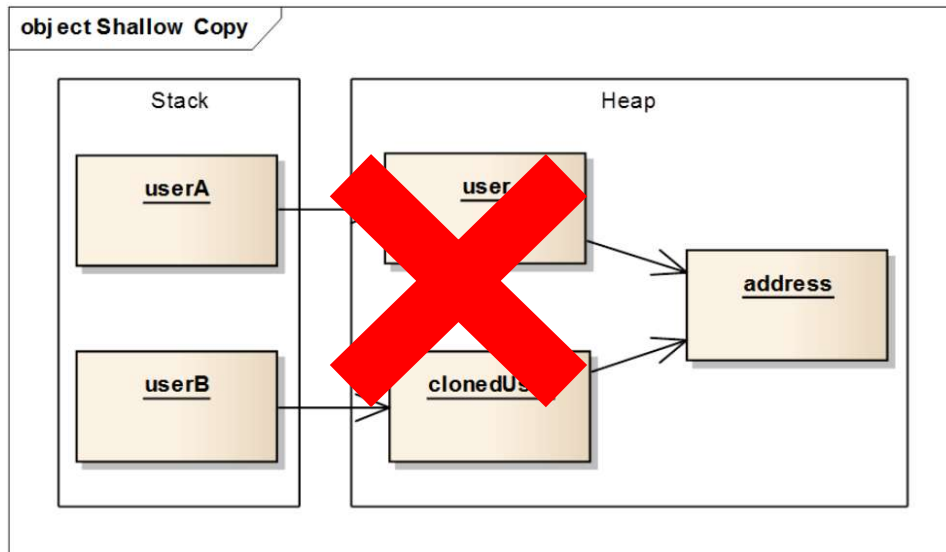
# 클래스 간 관계

- Composition(합성): 전체와 부분의 집합관계, Aggregation 보다 더 강한 집합관계.
- 표기법에는 다이아몬드 내부가 채워져 있다.
- 강한집합이란, Whole이 Part를 소유한다. Part에 해당하는 인스턴스는 공유될 수 없다.
- Whole 인스턴스가 Part 인스턴스를 생성, 소멸도 함께, 복사도 함께

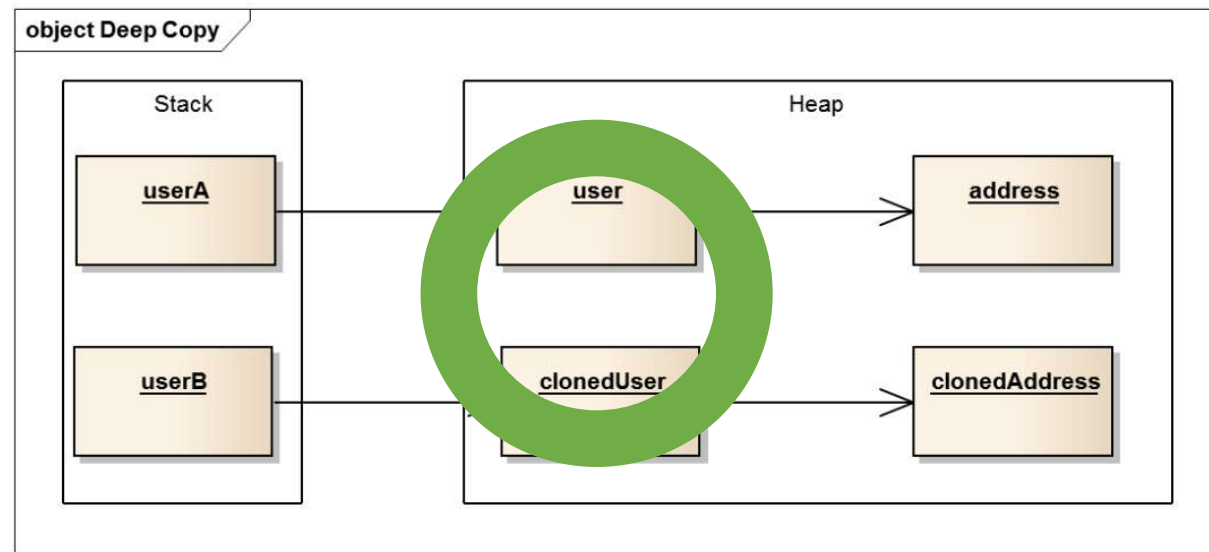


# Composition

- Part에 해당하는 인스턴스는 공유될 수 없다.
- Composition에서 Part들의 인스턴스는 Deep-Copy가 되어야한다.
- 아래는 객체 다이어그램. 객체 다이어그램에는 이름에 밑줄이 있다.



Part가 공유됨



Part가 공유되지 않음

# Composition 구현

- Composition을 코드로 구현하려면 신경 써야 할 부분이 있다.
- 일단, Whole 클래스에서 Part 인스턴스를 생성해야 한다.
- 또한, 외부에서 접근해서 Part를 임의적으로 생성하지 못하도록 Whole클래스에는 Part인스턴스에 대한 setter가 있으면 안 된다.
- 인스턴스 소멸은 Java에서는 GC가 알아서 처리해주므로 신경쓰지 않아도 됨.

Server Side event



# Real-time web application

- The typical interactions between browsers and servers consist of browsers requesting resources and servers providing responses.
- There are a few way to make our servers send data to clients at any time without explicit requests. That is..

**1. Web Socket**

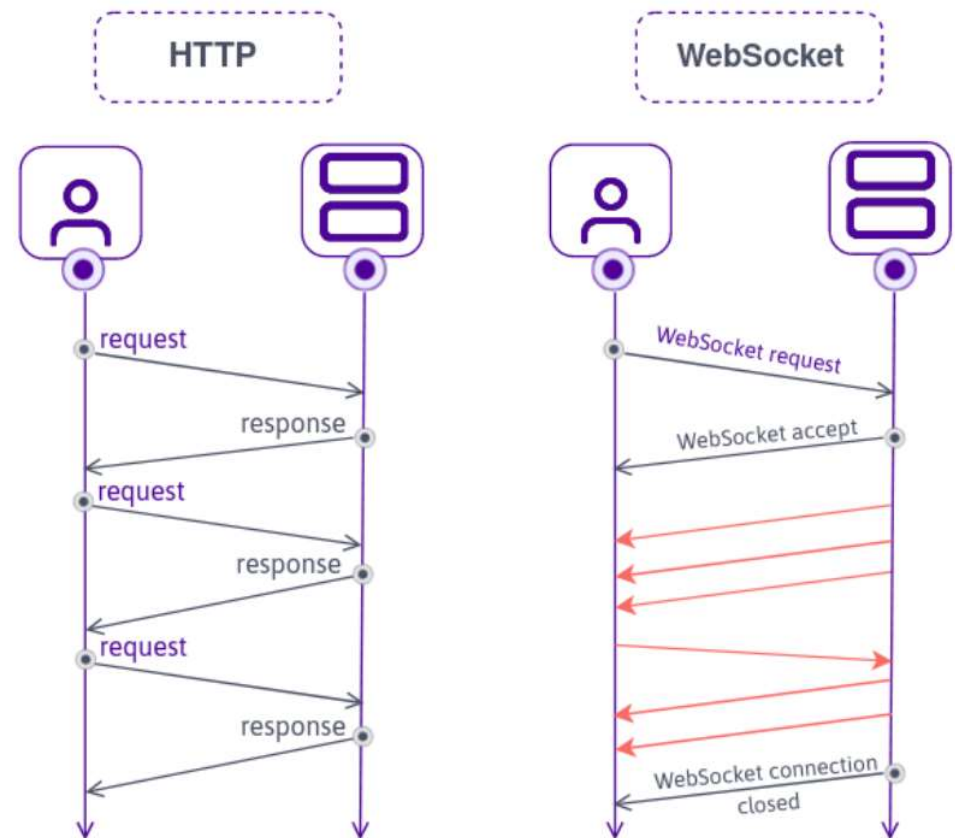
**2. Server-Sent Event**

# Web Socket

- WebSocket is kind of protocol of application just like as HTTP
- It also uses TCP/IP protocol for its transport layer (same as HTTP)
- WebSocket uses the URI starting from "wss://" or "ws://"
- WebSocket client and server 'handshake' at the first of the connection.
- Server doesn't have to wait the request. It can send message without client request. => bidirectional connection

# WebSocket vs HTTP

- The difference between HTTP and WebSocket is that **HTTP is stateless while WebSocket is stateful**
- HTTP: 클라이언트 쪽에서만 대화를 시작할 수 있는 한 방향 통신
- WebSocket: 서버도 언제든지 클라이언트에게 정보를 전송할 수 있음 (계속해서 handshake를 유지함.)



# WebSocket 단점

1. Complicated Code - WebSocket은 HTTP와 달리 Stateful protocol 이기 때문에 서버와 클라이언트 간의 연결을 항상 유지해야 하며 비정상적으로 연결이 끊어졌을 때 적절하게 대응해야 함. 그래서 HTTP에 비해 코드가 복잡할 수 있다.
2. High Cost - 서버와 클라이언트 간의 Socket 연결 유지에 비용이 많이 들어 감. 특히나 트래픽 양이 많은 서버 같은 경우에는 CPU에 큰 부담 발생

# Server-Sent Event

- Server-Sent Events (SSE) is a standard that enables Web servers to push data in real time to clients.
- It uses HTTP as its application protocol so there isn't additional complex protocol setting.
- SSE response has to contain this setting in HTTP header

'Content-Type': 'text/event-stream',

'Connection': 'keep-alive',

'Cache-Control': 'no-cache',

- Server send JSON data format to client when the event happen, and it has "~~W~~n~~W~~n" at the end of the string data.

```
client.response.write(`data: ${JSON.stringify(payload)}\n\n`)
```

# WebSocket vs Server-Sent Event

	WebSocket	Server-Sent Event
direction	양방향	단방향(server -> client)
Real time	Yes	Yes
Protocol	WebSocket	HTTP
Battery consume	High	Low
Usage	Chat, Trading char	SNS feed, friend request
Maximum concurrent connections	No limit (Browser)	HTTP: 6개, HTTP2: 100개 (Browser)

# Server-Sent Event Header

- Client 측 설정

```
GET /connect HTTP/1.1  
Accept: text/event-stream  
Cache-Control: no-cache
```

- Server 측 설정

```
HTTP/1.1 200  
Content-Type: text/event-stream; charset=UTF-8  
Transfer-Encoding: chunked
```

Text/event-stream은 표준이다.

Chunked Encoding 이유: 본문의 크기를 알 수 없어서

# Server-Sent Event 구현하기 – 연결

- Client code

=> JS provides EventSource which is used for SSE connection

```
const sse = new EventSource("http://localhost:8080/connect");
```

- Server code

⇒ Make the produces of controller GetMapping method to  
MediaType.TEXT\_EVENT\_STREAM\_VALUE

```
@GetMapping(value = "/connect", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
```



# Server-Sent Event 구현하기 – 연결

- SseEmitter

⇒ Spring provides SseEmitter API which developer can use to make response. SseEmitter is kind of class.

```
SseEmitter emitter = new SseEmitter(60 * 1000L);
```

We can set the expiration time which is used for reconnection term.

- Server Store the connection

⇒ Store the emitter for connection to send SSE later

```
sseEmitters.add(emitter);
```

- Send the dummy data to avoid 503 Error.

```
emitter.send(SseEmitter.event()  
    .name("connect")  
    .data("connected!"));
```

# Server-Sent Event 구현하기 – SSE 보내기

- When somebody sent the request of count increment, sseEmitter will be activated and send the date to All the client who opened SSE connection.

```
@PostMapping("/count")
public ResponseEntity<Void> count() {
    sseEmitters.count();
    return ResponseEntity.ok().build();
}
```

- This controller refers to another class 'SseEmitters' to help the function of SSE sending.

```
private final SseEmitters sseEmitters;
```

# Server-Sent Event 구현하기 – SSE 보내기

- SseEmitter.class

=> Send SSE just like when it sent the dummy data

```
public void count() {  
    long count = counter.incrementAndGet();  
    emitters.forEach(emitter -> {  
        try {  
            emitter.send(SseEmitter.event()  
                .name("count")  
                .data(count));  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    });  
}
```

# Server-Sent Event Test - backend

```
private static final AtomicLong emitterCounter = new AtomicLong();
4 usages
private final List<SseEmitter> emitterList = new CopyOnWriteArrayList<>();
no usages new *
@CrossOrigin
@GetMapping(value="/sse", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public ResponseEntity<SseEmitter> connect() {
    SseEmitter sseEmitter = new SseEmitter();
    emitterList.add(sseEmitter);
    log.info("new emitter added: {}", sseEmitter);
    log.info("emitter list:{}", emitterList);
    sseEmitter.onTimeout(sseEmitter::complete);
    sseEmitter.onCompletion(()->{
        emitterList.remove(sseEmitter);
        log.info("emitter deleted");
    });
    try {
        sseEmitter.send(SseEmitter.event().name("connect").data("connected!"));
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return ResponseEntity.ok(sseEmitter);
}
```

SSE emitter의  
연결 링크 기억하기

30초 후 만료되면  
emitter 기록을 제거

연결 성공했다고  
dummy data 보내기

# Server-Sent Event Test - backend

List를 사용한 이유는 해당 URL에 연결 되어있는 모든 클라이언트에게 발송하기 위해서임.

```
@CrossOrigin
@PostMapping("/sse")
public ResponseEntity<Void> order(){
    long count = emitterCounter.incrementAndGet();
    log.info("sseEmitter counter size={}", count);
    emitterList.forEach(sseEmitter -> {
        try {
            sseEmitter.send(SseEmitter.event().name( eventName: "count").data(count));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    });
    return ResponseEntity.ok().build();
}
```

SSE emitter 객체의 .send()를 사용하면 SSE가 브라우저로 전송된다.

# Server-Sent Event Test - frontend

```
const sse = new EventSource("http://localhost:8080/sse");

sse.addEventListener('connect', (e) => {
  const { data: receivedConnectData } = e;
  console.log('connect event data: ', receivedConnectData);
  const connectData = document.getElementById("connect");
  connectData.innerText = receivedConnectData;
});

sse.addEventListener('count', e => {
  const { data: receivedCount } = e;
  console.log("count event data", receivedCount);
  const countData = document.getElementById("count");
  countData.innerText = `count: ${receivedCount}`;
});
```

연결 유지의 핵심  
만료기한이 지나면  
서버로 자동으로 연결  
재요청을 보낸다.

# Server-Sent Event Test result - browser

The diagram illustrates the state of a web browser during an SSE test at two different points in time. A blue arrow indicates the progression from the initial state to the updated state.

**Initial State (Top):**

- Page content: **Welcome to SSE test!**  
connected!  
count: 6
- Browser console: Shows two log entries:
  - connect event data: connected!
  - count event data 6

**Updated State (Bottom):**

- Page content: **Welcome to SSE test!**  
connected!  
count: 8
- Form: Includes two input fields and a **Submit** button.
- Browser console: Shows four log entries:
  - connect event data: connected!
  - count event data 6
  - connect event data: connected!
  - count event data 7
  - count event data 8

- 브라우저를 새로고침하지 않아도 알아서 count가 늘어나는 것을 볼 수 있음



# Server-Sent Event Test result - log

- 주기적으로 SSE emitter 객체가 생성되고 30초 후에 삭제되는 것을 볼 수 있다.
- 만료시간(30초)가 지나면 자동으로 브라우저에서 서버로 연결 재요청을 보낸다. EventSource 덕분에

```
const sse = new EventSource("http://localhost:8080/sse");
```

```
exec-7] c.e.d.c.c.CustomerOrderController : new emitter added: SseEmitter@5bea722c
exec-7] c.e.d.c.c.CustomerOrderController : emitter list:[SseEmitter@5bea722c]
exec-8] c.e.d.c.c.CustomerOrderController : emitter deleted
exec-5] c.e.d.c.c.CustomerOrderController : new emitter added: SseEmitter@4cf12509
exec-5] c.e.d.c.c.CustomerOrderController : emitter list:[SseEmitter@4cf12509]
exec-1] c.e.d.c.c.CustomerOrderController : emitter deleted
exec-3] c.e.d.c.c.CustomerOrderController : new emitter added: SseEmitter@2780f7da
exec-3] c.e.d.c.c.CustomerOrderController : emitter list:[SseEmitter@2780f7da]
exec-10] c.e.d.c.c.CustomerOrderController : emitter deleted
```