**기술 면접 – 기초 지식**

**[ 자료구조 및 알고리즘 ]**

**1. 자료구조**

**1.1. 큐(대기열, Queue)**

**1.2. 스택(Stack)**

**1.3. 이진트리(Binary Tree, B-Tree)**

**1.4. 연결리스트(Linked List)**

**1.5 그래프**

**2. 알고리즘**

**2.1. 재귀호출/재귀함수(Recursive Call)**

**2.2. 정렬(Sort)**

**2.2.1 거품 정렬(버블정렬, bubble sort)**

**2.2.2 퀵 정렬(quick sort)**

**2.3. 탐색(Search)**

**2.3.1 순차 탐색(sequential search 또는 선형탐색, linear search)**

**2.3.2 이진 탐색(이분 탐색, binary search)**

**[ 프로그래밍 ]**

**1. 자바 객체지향 프로그래밍**

**1.1 캡슐화(encapsulation), 정보은닉(information hiding)**

**1.2 상속(inheritance), 오버라이딩(overriding)**

**1.3 다형성(polymorphism), 제네릭(generic), 오버로딩(overloading)**

**1.4 어노테이션(Annotation)**

**1.5 객체 생성 패턴 (Creational Pattern)**

**1.5.1 싱글톤(Singleton)**

**1.5.2 의존성 주입(Dependency Injection, DI)**

**1.5.3 빌더(Builder)**

**2. 자바스크립트 함수형 프로그래밍**

**2.1 변수의 범위, 환경(variable scope/context/environment)**

**2.1.1 var, let/const**

**2.1.2 호이스팅(Hoisting)**

**2.1.3 클로저(Closure)**

**2.2 and/or, short-circuit evaluation**

**2.3 AJAX(Asyncronous Javascript And XML)**

**2.4 callback, promise, async-await**

**[ 데이터베이스 ]**

**1. 정규화(Normalization)와 역정규화(De-normalization)**

**2. 트랜잭션(Transaction)**

**3. 인덱스**

**[ 소프트웨어공학 ]**

**1. Agile 소프트웨어 개발 방법론**

**2. 스크럼(Scrum)**

**3. XP(eXtreme Programming)**

**[ 운영체제 ]**

**1. 커널, 쉘, 프로세스(process)**

**2. 가상메모리, 페이징, 스왑(swap)**

**[ 네트워크 ]**

**1. 네트워크 7-Layer, 네트워크 Class, Class별 Private IP 대역**

**2. 라우팅 및 서브넷**

**3. TCP/UDP의 차이, 특징, 용도, TCP 3-way handshake**

**4. HTTP, Methods, Request/Response Header**

**[ 자료구조 및 알고리즘 ]**

**1. 자료구조**

**1.1. 큐(대기열, Queue)**

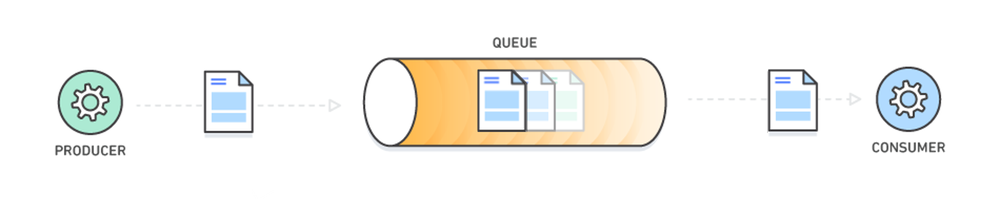
|  |  |
| --- | --- |
| 자료구조] Queue(큐) | - 먼저 집어넣은 데이터가 먼저 나오는 FIFO (First In First Out)구조로 저장하는 형식을 말한다. |
| - 자료구조의 앞쪽에서는 데이터를 꺼내올 수 있고(dequeue) 뒤쪽에서는 데이터를 추가할 수 있다.(enqueue) |

- 큐가 꽉 차서 더 이상 자료를 넣을 수 없는 경우를 오버플로우(Overflow), 큐가 비어 있어 자료를 꺼낼 수 없는 경우를 언더플로우(Underflow)라고 한다.

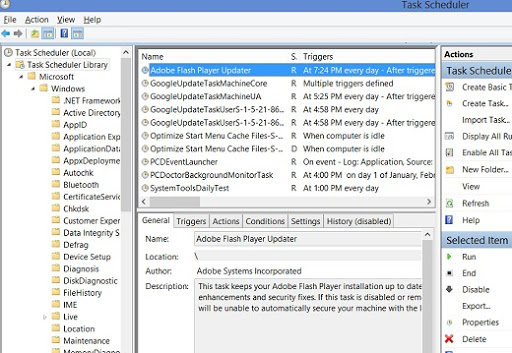
**- 사용사례**

: 메시징 시스템, 운영체제의 작업 스케쥴러, 인쇄 대기열

: 메시징 시스템



: 작업 스케줄러



**1.2. 스택(Stack)**

|  |  |
| --- | --- |
|  | 한 쪽 끝에서만 자료를 넣거나 뺄 수 있는 선형 구조(LIFO - Last In First Out)으로 되어 있는 자료구조이다. |
| 자료를 넣는 것을 '밀어넣는다' 하여 푸쉬(push)라고 하고 반대로 넣어둔 자료를 꺼내는 것을 팝(pop)이라고 하는데, 이때 꺼내지는 자료는 가장 최근에 푸쉬한 자료부터 나오게 된다. |

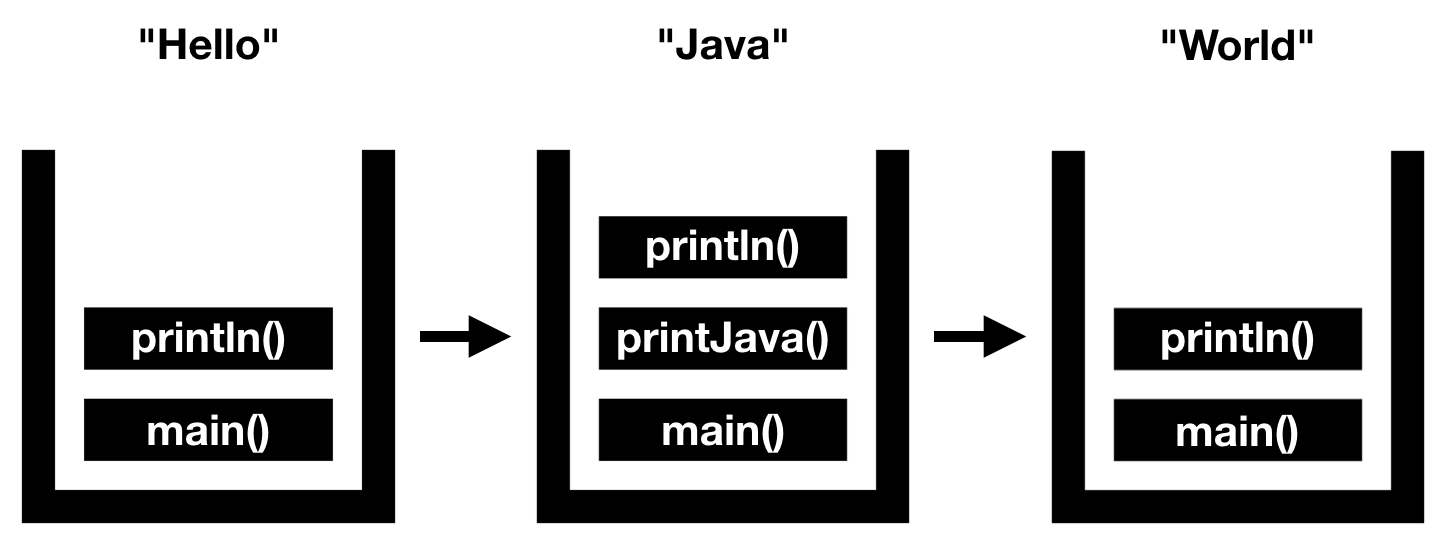
[**https://ko.wikipedia.org/wiki/%EC%8A%A4%ED%83%9D**](https://ko.wikipedia.org/wiki/%EC%8A%A4%ED%83%9D)

**- 사용사례**

: 애플리케이션 네비게이션(뒤로가기), 실행 취소(undo), 시스템 스택(함수호출 및 복귀)

: 시스템 스택(함수호출 및 복귀) 예)

|  |
| --- |
| public class MainTest {  public static void main(String[] args) {  System.out.println("Hello");  printJava();  System.out.println("World");  }  public static void printJava() {  System.out.println("Java");  }  } |
| Hello  Java  World |

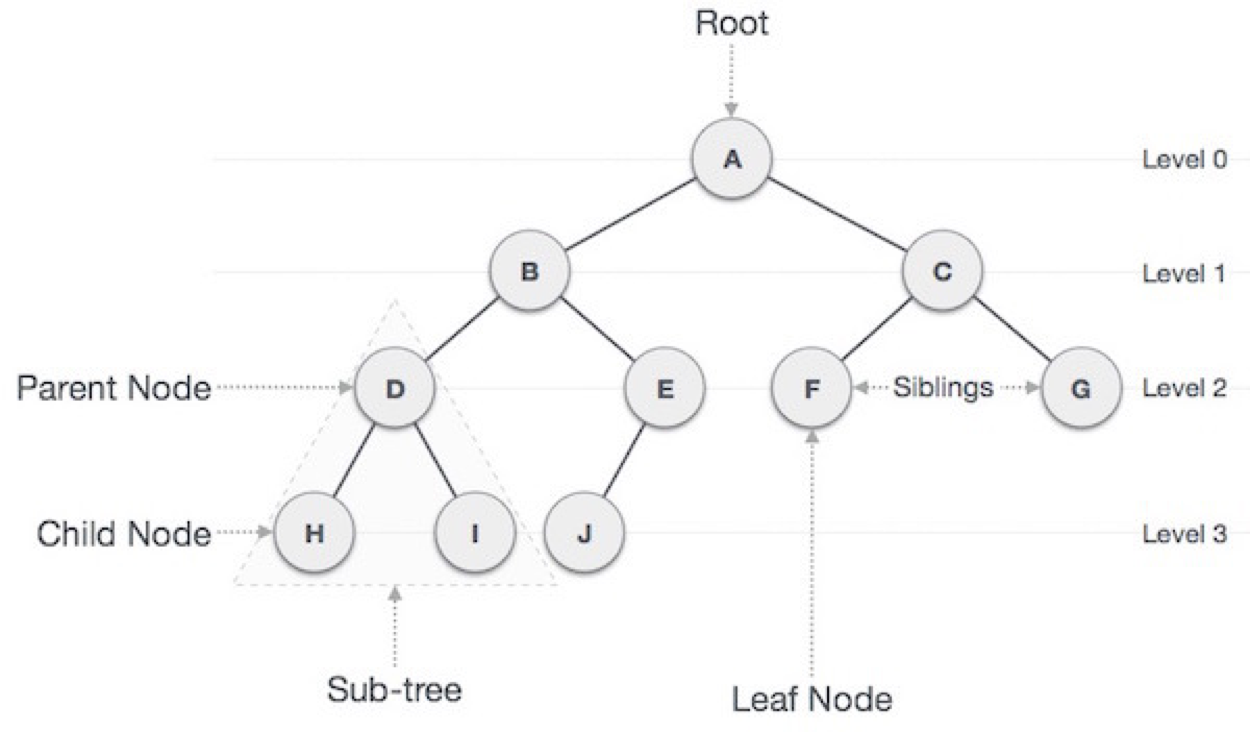


<https://cloudstudying.kr/lectures/141>

**1.3. 이진트리(Binary Tree, B-Tree)**

|  |  |
| --- | --- |
|  | 이진 트리는 각각의 노드가 최대 두 개의 자식 노드를 가지는 트리 자료 구조이다. |
| 자식 노드를 각각 왼쪽 자식 노드와 오른쪽 자식 노드라고 한다. |

: 용어



루트 노드(root node): 부모가 없는 노드, 트리는 하나의 루트 노드만을 가진다.

단말 노드(leaf node): 자식이 없는 노드, ‘말단 노드’ 또는 ‘잎 노드’라고도 부른다.

간선(edge): 노드를 연결하는 선 (link, branch 라고도 부름)

노드의 레벨(level): 트리의 특정 깊이를 가지는 노드의 집합

노드의 차수(degree): 하위 트리 개수 / 간선 수 (degree) = 각 노드가 지닌 가지의 수

트리의 차수(degree of tree): 트리의 최대 차수

트리의 깊이(height): 트리에 속한 노드의 최대 레벨

<https://gmlwjd9405.github.io/2018/08/12/data-structure-tree.html>

<http://dblab.duksung.ac.kr/ds/pdf/Chap08.pdf>

**- 사용사례**

: 데이터베이스의 인덱스

**1.4. 연결리스트(Linked List)**

|  |  |
| --- | --- |
|  | 각 노드가 데이터와 포인터를 가지고 한 줄로 연결되어 있는 방식으로 데이터를 저장하는 자료 구조이다. |
| 이름에서 말하듯이 데이터를 담고 있는 노드들이 연결되어 있는데, 노드의 포인터가 다음이나 이전의 노드와의 연결을 담당하게 된다. |
|  | **단일 연결 리스트**는 각 노드에 자료 공간과 한 개의 포인터 공간이 있고, 각 노드의 포인터는 다음 노드를 가리킨다. |
|  | **이중 연결 리스트의 구조**는 단일 연결 리스트와 비슷하지만, 포인터 공간이 두 개가 있고 각각의 포인터는 앞의 노드와 뒤의 노드를 가리킨다. |
|  | **원형 연결 리스트**는 일반적인 연결 리스트에 마지막 노드와 처음 노드를 연결시켜 원형으로 만든 구조이다. |

**- 사용사례**

: 데이터베이스 인덱스의 가장 하위 노드



<https://m.blog.naver.com/mirine_11/221768199155>

**1.5 그래프**

|  |  |
| --- | --- |
|  | 그래프는 vertex와 edge로 구성된 한정된 자료구조를 의미한다 |
| vertex는 정점, edge는 정점과 정점을 연결하는 간선이다. |

**- 그래프에서 사용하는 용어**

정점(vertice) : 노드(node)라고도 하며 정점에는 데이터가 저장됩니다. (0, 1, 2, 3)

간선(edge): 링크(arcs)라고도 하며 노드간의 관계를 나타냅니다.

인접 정점(adjacent vertex) : 간선에 의해 연결된 정점으로 위에서 (정점0과 정점1은 인접 정점)

단순 경로(simple-path): 경로 중 반복되는 정점이 없는것, 같은 간선을 자나가지 않는 경로

차수(degree): 무방향 그래프에서 하나의 정점에 인접한 정점의 수 (정점 0의 차수는 3)

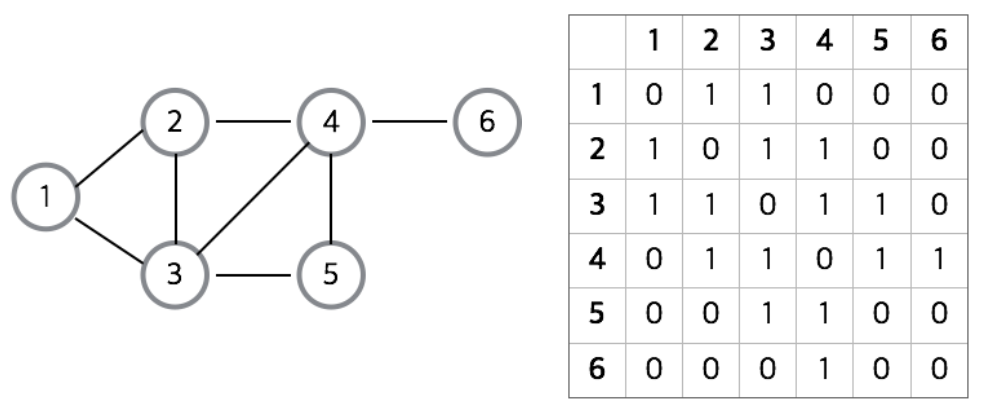
진출 차수(out-degree) : 방향그래프에서 사용되는 용어로 한 노드에서 외부로 향하는 간선의 수를 뜻합니다.

진입차수(in-degree) : 방향그래프에서 사용되는 용어로 외부 노드에서 들어오는 간선의 수를 뜻합니다.

**- 그래프 구현 방법**

그래프를 구현하는 방법에는 인접행렬(Adjacency Matrix)와 인접리스트(Adjacency List)방식이 있습니다. 두개의 구현 방식은 각각의 상반된 장단점을 가지고 있습니다.

**[인접행렬]**



인접행렬은 그래프의 노드를 2차원 배열로 만든 것입니다. 완성된 배열의 모양은 1, 2, 3, 4, 5, 6의 정점을 연결하는 노드에 다른 노드들이 인접 정점이라면 1, 아니면 0을 넣어줍니다.

**인접행렬 장점**

1. 2차원 배열 안에 모든 정점들의 간선 정보를 담기 때문에 배열의 위치를 확인하면 두 점에 대한 연결 정보를 조회할 때 O(1) 의 시간 복잡도면 가능합니다.

2. 구현이 비교적 간편합니다.

**인접행렬 단점**

1. 모든 정점에 대해 간선 정보를 대입해야 하므로 O(n²)의 시간복잡도가 소요됩니다.

2. 무조건 2차원 배열이 필요하기에 필요 이상의 공간이 낭비됩니다.

**[인접리스트]**



인접리스트란 그래프의 노드들을 리스트로 표현한것입니다. 주로 정점의 리스트 배열을 만들어 관계를 설정해줌으로써 구현합니다.

**인접리스트 장점**

1. 정점들의 연결 정보를 탐색할 때 O(n) 의 시간이면 가능합니다. (n: 간선의 갯수)

2. 필요한 만큼의 공간만 사용하기때문에 공간의 낭비가 적습니다.

**인접리스트 단점**

1. 특정 두 점이 연결되었는지 확인하려면 인접행렬에 비해 시간이 오래 걸립니다. (배열보다 search 속도느림)

2. 구현이 비교적 어렵습니다.

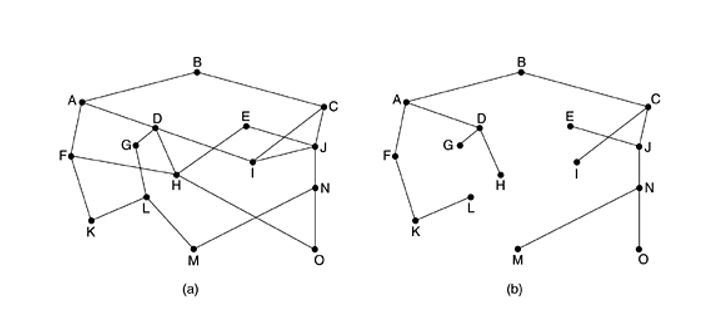
**그래프의 종류**

<https://ko.wikipedia.org/wiki/%EA%B7%B8%EB%9E%98%ED%94%84_(%EC%9E%90%EB%A3%8C_%EA%B5%AC%EC%A1%B0)>

<https://coding-factory.tistory.com/610>

|  |  |
| --- | --- |
|  | **무방향그래프**  무방향 그래프는 두 정점을 연결하는 간선에 방향이 없는 그래프입니다. |
|  | **방향그래프**  방향 그래프는 두 정점을 연결하는 간선에 방향이 존재하는 그래프입니다. 간선의 방향으로만 이동할 수 있습니다. |
|  | **가중치그래프**  가중치 그래프는 두 정점을 이동할때 비용이 드는 그래프입니다. |

- 사용사례: 네트워크 라우팅 정보(Shortest path routing) - https://doorbw.tistory.com/52



**2. 알고리즘**

**2.1. 재귀호출/재귀함수(Recursive Call)**

- 재귀함수란 어떤 함수에서 자신을 다시 호출하여 작업을 수행하는 방식의 함수. 반복문을 사용하는 코드는 재귀함수를 통해 구현 가능하고 그 반대도 가능

- 재귀함수를 작성할 때는 함수내에서 다시 자신을 호출한 후 그 함수가 끝날 때까지 함수 호출 이후의 명령문이 수행되지 않는다(스택구조를 사용하기 때문에), 종료조건이 꼭 포함되어있어야 무한루프를 방지할 수 있다.

- 사용 예) 팩토리얼 계산

f(n) = n \* f(n-1)

단, f(1) = 1

public long factorial(int n) {

long result = 0;

if (n == 1) {

result = 1;

} else {

result = n \* factorial(n - 1);

}

return result;

}

<https://gomguard.tistory.com/111>

<https://cremazer.github.io/What-is-a-recursive-call/>

**2.2. 정렬(Sort)**

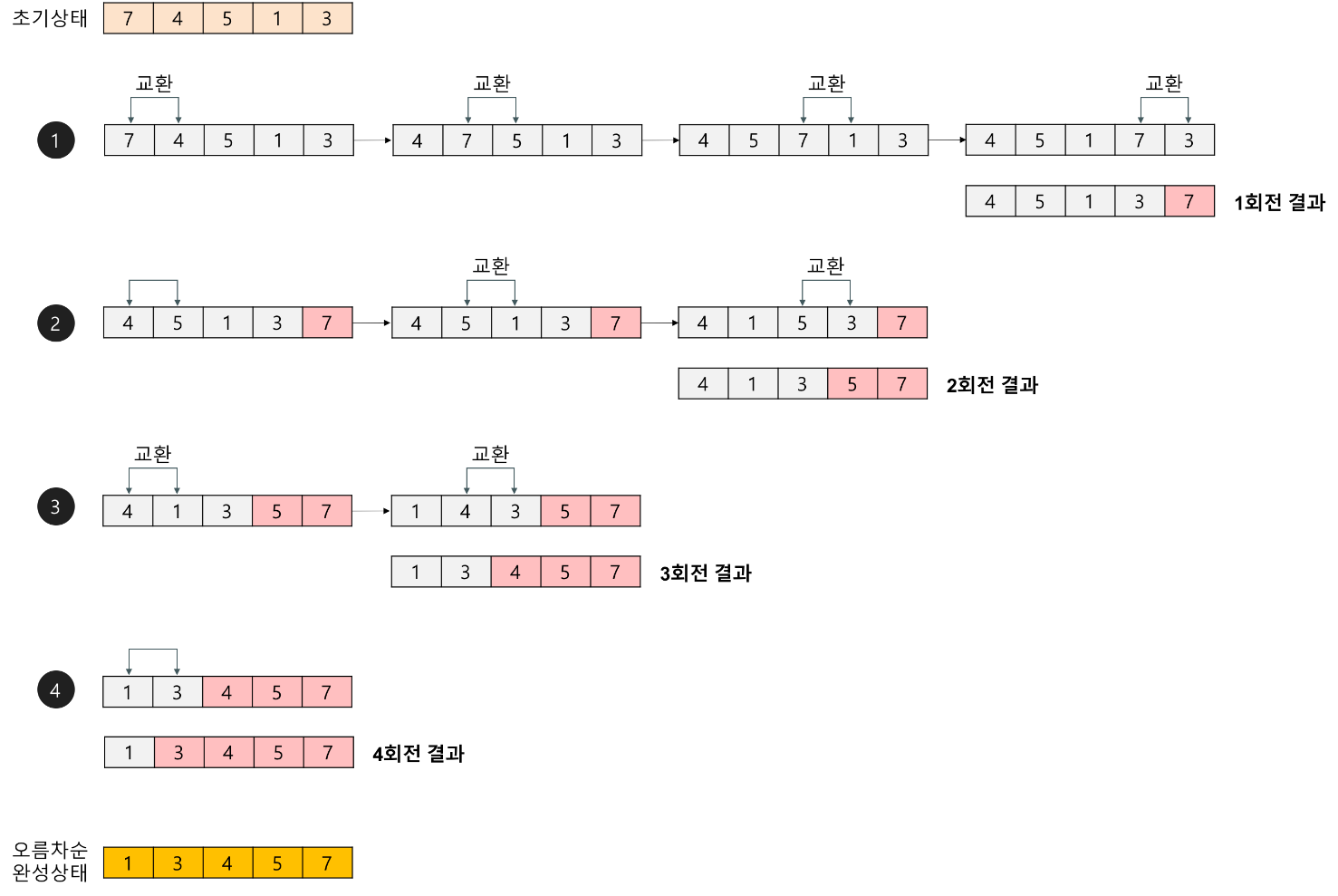
**2.2.1 거품 정렬(버블정렬, bubble sort)**

-  두 인접한 요소를 비교하여 정순정렬(ascending)일 때는 큰 요소를 뒤로, 역순정렬(descending)일 때는 작은 요소를 뒤로 이동하여 정렬하는 방식이다.

- 요소간 비교 및 교환이 끝나면 다음 인덱스의 요소로 이동하여 비교 및 교환이 진행된다.

- 한번 반복이 끝나면 가장 끝 요소가 가장 크거나 작은 요소가 되므로 반복할 때마다 비교 및 교환되는 횟수가 줄어든다.

- 성능은 안 좋지만 알고리즘 구현이 단순하여 많이 사용한다.

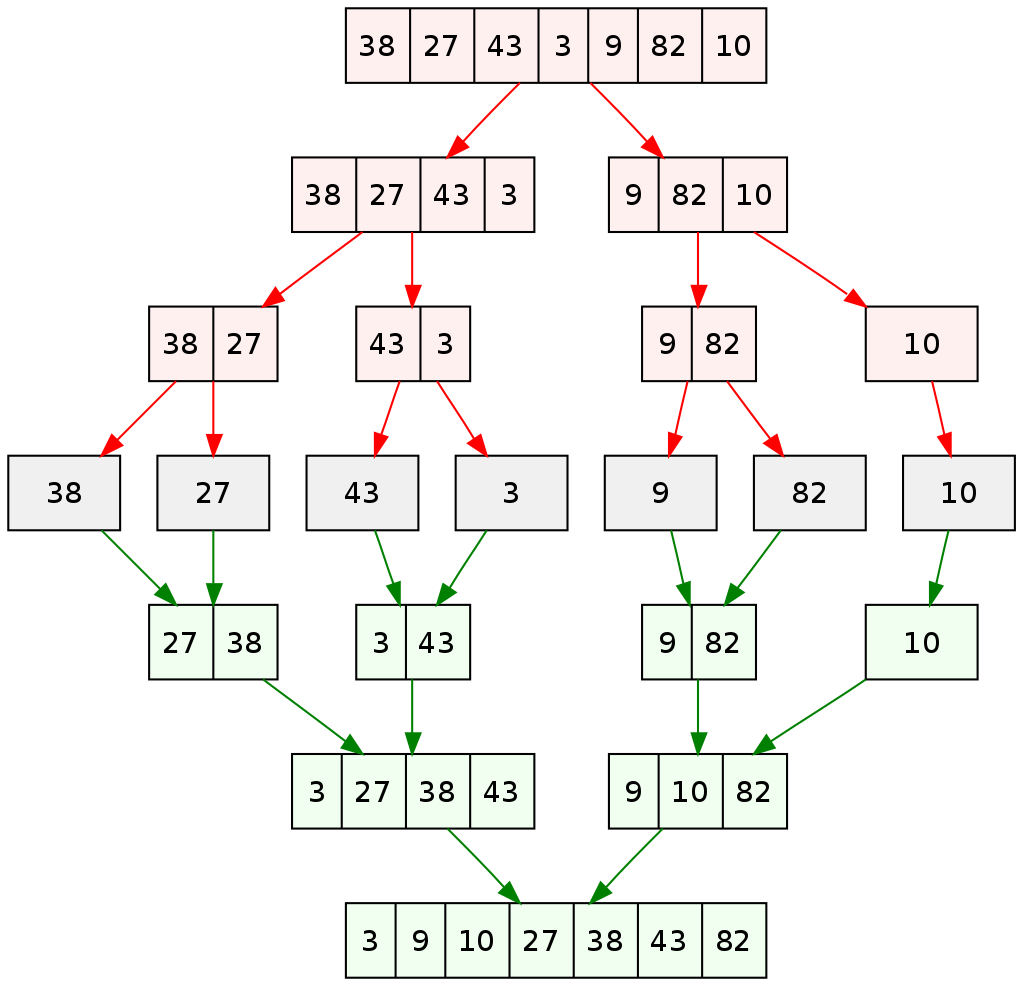


|  |
| --- |
| public static void bubleSort(int[] arr) {      for(int i = 0; i < arr.length; i++) {          for(int j = 0 ; j < arr.length - i - 1 ; j++) {              if(arr[j] > arr[j+1]) {                  int temp = arr[j+1];                  arr[j+1] = arr[j];                  arr[j] = temp;              }          }      }  } |
| public static void main(String[] args) {      int[] arr = new int[]{3,6,41,2,4,1,5,743};      Sort.bubleSort(arr);      for(int i : arr) {          System.out.print(i+" ");      }  } |
| 1 2 3 4 5 6 41 743 |

**2.2.2 퀵 정렬(Quick Sort)**

- 병합 정렬은 여러 개의 정렬된 자료의 집합을 결합해 하나의 집합으로 만드는 정렬 방법이다. 이 정렬은 전체를 상대로 수행하지 않고 분할(Divide)한 뒤 각 부분 집합들에 대해 정렬한 후 다시 결합(Combine)하는 분할 정복(Divide and Conquer) 기법을 사용한다.

- 병합 정렬은, 원소가 저장되어 있는 배열을 계속 쪼개서 길이가 1인 배열을 만들고, 그 이후 정렬하면서 합치는 알고리즘이다.



https://yunmap.tistory.com/entry/알고리즘-Java로-구현하는-쉬운-Merge-Sort-병합-정렬-합병-정렬

https://heekim0719.tistory.com/289 [별토끼 DEVLOG]

\*\* 합병정렬은 재귀함수로 처리되며, 각 재귀함수호출은 합병할 리스트의 인덱스 범위를 축소하는 과정(분할)이 일어나고, 리스트를 합병하는 형태로 진행된다.

\*\* 재귀함수로 처리되므로 가장 마지막에 호출되는 함수에서 1개의 리스트간 합병이 먼저 일어난다.

\*\* 합병은 분할된 리스트의 요소 간의 비교하여 작은 요소부터 임시저장공간에 저장하고 왼쪽 또는 오른쪽에 요소들이 남아있으면 남은 요소들을 남은 임시저장공간에 저장 후 실제저장공간으로 복사한다.

****

|  |
| --- |
| \*\*   \* 부분리스트는 a배열의 left ~ right 까지이다.   \*   \* @param a   정렬할 배열   \* @param left  배열의 시작점   \* @param mid 배열의 중간점   \* @param right 배열의 끝 점   \*/  private static void merge(int[] a, int left, int mid, int right) {    int l = left;   // 왼쪽 부분리스트 시작점    int r = mid + 1;  // 오른쪽 부분리스트의 시작점    int idx = left;   // 채워넣을 배열의 인덱스        while(l <= mid && r <= right) {      /\*       \*  왼쪽 부분리스트 l번째 원소가 오른쪽 부분리스트 r번째 원소보다 작거나 같을 경우       \*  왼쪽의 l번째 원소를 새 배열에 넣고 l과 idx를 1 증가시킨다.       \*/      if(a[l] <= a[r]) {        sorted[idx] = a[l];        idx++;        l++;      }      /\*       \*  오른쪽 부분리스트 r번째 원소가 왼쪽 부분리스트 l번째 원소보다 작거나 같을 경우       \*  오른쪽의 r번째 원소를 새 배열에 넣고 r과 idx를 1 증가시킨다.       \*/      else {        sorted[idx] = a[r];        idx++;        r++;      }    }      /\*     \* 왼쪽 부분리스트가 먼저 모두 새 배열에 채워졌을 경우 (l > mid)     \* = 오른쪽 부분리스트 원소가 아직 남아있을 경우     \* 오른쪽 부분리스트의 나머지 원소들을 새 배열에 채워준다.     \*/    if(l > mid) {      while(r <= right) {        sorted[idx] = a[r];        idx++;        r++;      }    }      /\*     \* 오른쪽 부분리스트가 먼저 모두 새 배열에 채워졌을 경우 (r > right)     \* = 왼쪽 부분리스트 원소가 아직 남아있을 경우     \* 왼쪽 부분리스트의 나머지 원소들을 새 배열에 채워준다.     \*/    else {      while(l <= mid) {        sorted[idx] = a[l];        idx++;        l++;      }    }      /\*     \* 정렬된 새 배열을 기존의 배열에 복사하여 옮겨준다.     \*/    for(int i = left; i <= right; i++) {      a[i] = sorted[i];    }  }    **https://st-lab.tistory.com/233** |

**2.3. 탐색(Search)**

**2.3.1 순차 탐색(sequential search 또는 선형탐색, linear search)**

|  |  |
| --- | --- |
|  | - 순차 검색 알고리즘(sequential search algorithm), 또는 선형 검색 알고리즘(linear search algorithm)은 리스트에서 특정한 값을 찾는 알고리즘의 하나다. 이것은 리스트에서 찾고자 하는 값을 맨 앞에서부터 끝까지 차례대로 찾아 나가는 것이다. 검색할 리스트의 길이가 길면 비효율적이지만, 검색 방법 중 가장 단순하여 구현이 쉽고 **정렬되지 않은 리스트에서도 사용할 수 있다는 장점**이 있다. |

<https://ko.wikipedia.org/wiki/%EC%88%9C%EC%B0%A8_%EA%B2%80%EC%83%89_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98>

<https://blog.hexabrain.net/245>

**2.3.2 이진 탐색(이분 탐색, binary search)**

|  |  |
| --- | --- |
|  | - 이진 검색 알고리즘(binary search algorithm)은 오름차순으로 정렬된 리스트에서 특정한 값의 위치를 찾는 알고리즘이다. 처음 중간의 값을 임의의 값으로 선택하여, 그 값과 찾고자 하는 값의 크고 작음을 비교하는 방식을 채택하고 있다. 처음 선택한 중앙값이 만약 찾는 값보다 크면 그 값은 새로운 최댓값이 되며, 작으면 그 값은 새로운 최솟값이 된다. 검색 원리상 **정렬된 리스트에만 사용할 수 있다는 단점**이 있지만, **검색이 반복될 때마다 목표값을 찾을 확률은 두 배가 되므로 속도가 빠르다는 장점**이 있다. |

<https://ko.wikipedia.org/wiki/%EC%9D%B4%EC%A7%84_%EA%B2%80%EC%83%89_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98>

<https://m.blog.naver.com/PostView.nhn?blogId=beaqon&logNo=221255293930&proxyReferer=https:%2F%2Fwww.google.com%2F>

**[프로그래밍]**

**1. 자바 객체지향 프로그래밍**

**1.1 캡슐화(encapsulation), 정보은닉(information hiding)**

- 캡슐화는 객체들의 구조적인 부분과 기능적인 부분을 추상화하여, 구조적인 부분은 필드(field)로 기능적인 부분은 메소드로 하여 클래스로 만드는 작업이다

- 정보은닉은 클래스 내의 구현 내용 중 공개범위에 따라서 일부를 외부 클래스에서 접근할 수 없도록 감추어 은닉하는 방법이다. 공개범위를 지정하는 방법은 access modifier (접근제한자, private/protected/ public)를 이용하여 처리한다.

**1.2 상속(inheritance), 오버라이딩(overriding)**

- **부모 클래스의 필드와 메소드를 자식 클래스에서 재사용할 수 있도록 하는 방법**이다.

- 부모 클래스는 자식 클래스들의 공통적인 속성과 기능을 가지는 클래스로 설계한다.

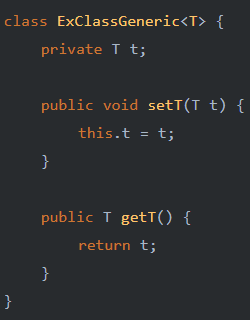
- 자바에서는 자식 클래스에서 extends라는 키워드를 이용하여 부모 클래스를 상속받아 사용한다.

- 메소드 오버라이딩은 자식 클래스가 자신의 부모 클래스에 의해 이미 제공된 메소드의 구현내용을 변경하는 것이나. 부모클래스의 메소드 시그니처(signature, 함수명/매개변수타입 및 순서) 및 반환형식(return type)이 동일해야 한다.

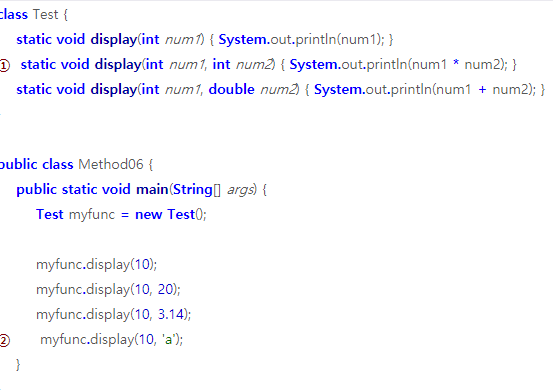
**1.3 다형성(polymorphism), 제네릭(generic), 오버로딩(overloading)**

- 프로그램 언어의 각 요소들(상수, 변수, 식, 오브젝트, 함수, 메소드 등)이 다양한 자료형(type)에 속하는 것이 허가되는 성질을 가리킨다. 대표적으로 자바에서 지원하는 타입인 제네릭이 있다.

- 제네릭은 데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있는 기술에 중점을 두어 재사용성을 높일 수 있는 프로그래밍 방식이다. <https://coding-factory.tistory.com/573>

- 메소드 오버로딩은 다형성의 원리에 따라서 메소드를 구현하는 방법으로 메소드 시그니처 중에서 메소드 이름은 같으나 매개변수의 개수 및 타입이 다른 메소드를 정의하는 것을 말한다.



<https://tcpschool.com/java/java_usingMethod_overloading>

**1.4 어노테이션(Annotation)**

- 자바 소스 코드에 추가하여 사용할 수 있는 메타데이터의 일종이다. @기호를 앞에 붙여서 사용하며 JDK 1.5 버전 이상에서부터 사용이 가능하다. 어노테이션마다 어노테이션을 추가할 수 있는 대상이 클래스, 메소드로 다르다.

- 어노테이션의 사용처

: 애플리케이션 실행 시에 특정 기능을 수행하도록 정보를 제공

-> @Scheduler, @EurekaServer, @Entity

: 툴이나 프레임워크에서 자바객체나 구현체 코드를 자동으로 생성할 수 있도록 정보를 제공

-> @Controller, @Repository

: 컴파일러에게 문법체크나 메소드 생성과 같은 추가적인 정보를 제공한다.

-> @SuppressWarnings, @Data,

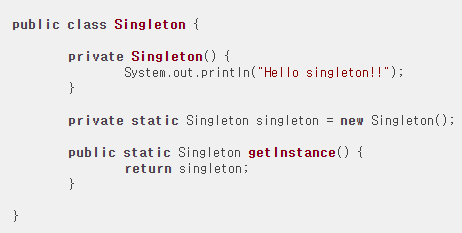
<https://coding-factory.tistory.com/575>

**1.5 객체 생성 패턴 (Creational Pattern)**

**1.5.1 싱글톤(Singleton)**

- 싱글톤 패턴(Singleton pattern)은 클래스에 해당하는 인스턴스를 하나만 생성하여 사용할 수 있도록 하는 방법이다. 방법적으로 기본 생성자를 private으로하여 외부에서 생성자로 객체를 생성하지 못하게 하며, getInstance()와 같은 static 메소드에서만 사전에 생성된 객체를 리턴받아 사용할 수 있게 한다.

- 주로 애플리케이션에서 단일적으로 존재하는 객체들을 선언하는 방식으로 사용하며, 데이터베이스 접근객체, 애플리케이션 설정 객체들에서 사용할 수 있다.

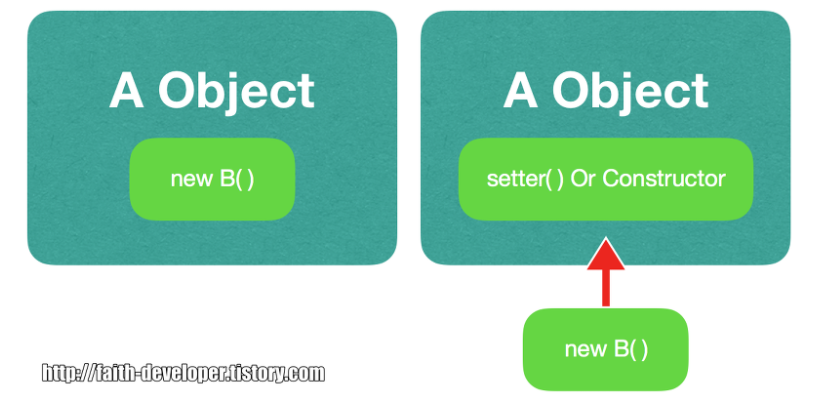


<https://gofnrk.tistory.com/63>

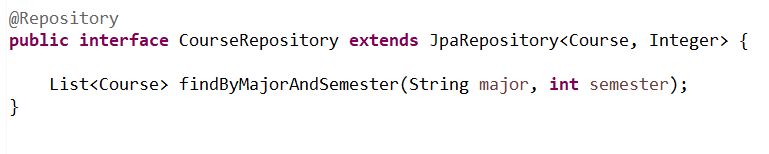
**1.5.2 의존성 주입(Dependency Injection, DI)**

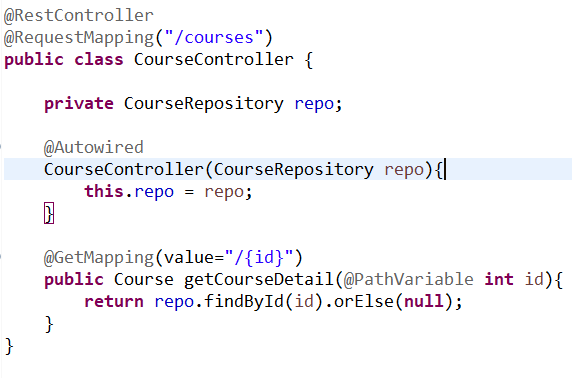
- 의존성 주입은 외부에서 의존객체를 생성하여 사용하는 객체로 제공해주는 방법이다.

- 의존성 주입의 의도는 객체의 생성과 사용의 관심을 분리하는 것이다.



- 의존성 주입은 주로 DI 프레임워크에 의하여 의존객체가 생성되고 사용하는 객체로 의존객체가 주입된다.

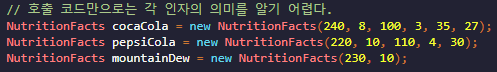
****

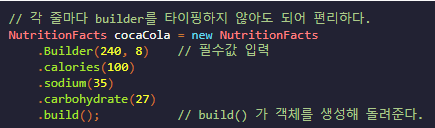
****

- 스프링 프레임워크에서는 의존성 주입을 광범위하게 사용하며 예로 애플리케이션 실행 시에 @Repository 인터페이스 구현 객체를 생성하고 해당 인터페이스가 @Autowired 어노테이션으로 매개변수나 필드로 존재하면 주입해준다.

**1.5.3 빌더(Builder)**

- 빌더 패턴은 객체의 생성과정과 표현방법을 분리하는 방법으로, 주로 매개변수가 있는 생성자를 통해 객체를 생성할 때 매개변수의 순서나 항목과 상관없이 다양한 방법으로 생성할 수 있도록 하는 방법에 사용된다.

 <생성자로 생성>

 <빌더로 생성>

**<빌더 코드>**

****

****

-lombok 라이브러리를 사용하여 @Builder 어노테이션을 추가하면 컴파일 타임에 클래스에 맞는 Builder가 생성되어 손쉽게 사용할 수 있다.

<https://tomining.tistory.com/180>

**2. 자바스크립트 함수형 프로그래밍**

**2.1 변수의 범위, 환경(variable scope/context/environment)**

**2.1.1 var, let/const**

- var는 전역 범위(global)와 함수 범위의 지역(local) 변수를 선언할 때 사용하는 키워드이다. ES5(2009)

<https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Statements/var#%EC%84%A4%EB%AA%85>

- let은 블록 범위의 지역 변수를 선언할 때 사용하는 키워드이다. ES6(2015)

<https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Statements/let#description>

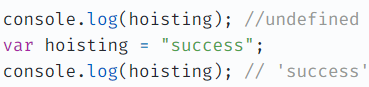
- const는 블록 범위의 상수를 선언할 때 사용하는 키워드이다. ES6(2015)

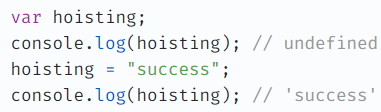
<https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Statements/const#%EC%84%A4%EB%AA%85>

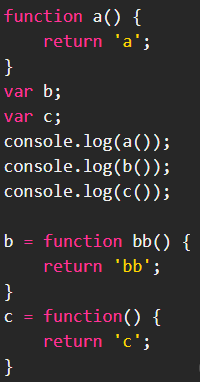
**2.1.2 호이스팅(Hoisting)**

- Hoisting은 자바스크립트 Parser가 코드 실행 전에 선언되지 않은 상태로 사용되는 변수나 함수가 존재하면 하단에 있는 선언을 유효범위에 최상단으로 **선언을 이동하여** 주는 것을 말한다.

- 실제로 코드가 끌어올려지는 건 아니며, 자바스크립트 Parser 내부적으로 끌어올려서 처리하는 것이라 실제 메모리에서는 변화가 없다.

 <실제 코드>

 <내부적인 hoisting 결과>

<실제코드>  <내부적인hoisting 결과>

function hoist() {

a = 20;

var b = 100;

}

hoist();

console.log(a);

/\*

Accessible as a global variable outside hoist() function

Output: 20

\*/

console.log(b);

/\*

Since it was declared, it is confined to the hoist() function scope.

We can't print it out outside the confines of the hoist() function.

Output: ReferenceError: b is not defined

\*/

<https://gmlwjd9405.github.io/2019/04/22/javascript-hoisting.html>

<https://velog.io/@stampid/Hoisting%ED%98%B8%EC%9D%B4%EC%8A%A4%ED%8C%85%EC%9D%B4%EB%9E%80>

<https://developer.mozilla.org/ko/docs/Glossary/Hoisting>

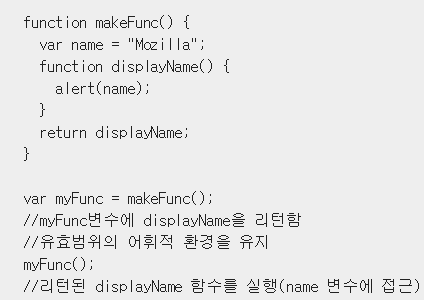
<https://gmlwjd9405.github.io/2019/04/22/javascript-hoisting.html>

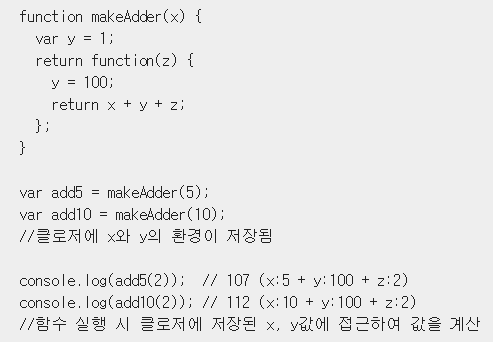
<https://deftkang.tistory.com/17>

**2.1.3 클로저(Closure)**

- 클로저는 **생성된 함수**와 **함수가 생성된 시점의 유효 범위 내에 있는 모든 지역변수**로 구성된 조합을 말한다. 함수 안에서 함수를 생성하여 리턴할 때 클로저가 생성된다. **(함수+변수목록)**

- 클로저 쓰는 이유가 함수가 실행이 종료되도, 함수 안에 상태를 기역해서 차후에 처리하고 싶을 때 사용

****

****

[**https://developer.mozilla.org/ko/docs/Web/JavaScript/Closures**](https://developer.mozilla.org/ko/docs/Web/JavaScript/Closures)

**2.2 and/or, short-circuit evaluation**

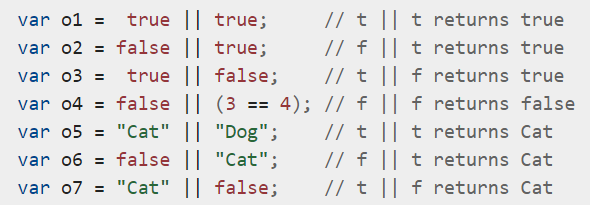
- expr1 &&expr2: expr1을 true로 변환할 수 있는 경우 expr2을 반환하고, 그렇지 않으면 expr1을 반환

- expr1 || expr2: expr1을 true로 변환할 수 있으면 expr1을 반환하고, 그렇지 않으면 expr2를 반환

- false로 변환될 수 있는 값: null, 0, NaN, 빈 문자열 (""), 또는 undefined

다음의 코드는 && (논리 곱) 연산자의 예제를 보여주고 있습니다.





[**https://developer.mozilla.org/ko/docs/Web/JavaScript/Guide/Expressions\_and\_Operators#%EB%85%BC%EB%A6%AC\_%EC%97%B0%EC%82%B0%EC%9E%90**](https://developer.mozilla.org/ko/docs/Web/JavaScript/Guide/Expressions_and_Operators#%EB%85%BC%EB%A6%AC_%EC%97%B0%EC%82%B0%EC%9E%90)

**<React: Short-Circuit을 이용한 조건부 렌더링>**

****

[**https://ko.reactjs.org/docs/conditional-rendering.html**](https://ko.reactjs.org/docs/conditional-rendering.html)

**2.3 AJAX(Asyncronous Javascript And XML)**

- Ajax는 XMLHttpRequest, fetch 등의 **자바스크립트** API를 이용하여 **서버와 비동기적으로 통신**하기 위한 방법이다. 비동기적인 이유는 AJAX가 처리될 대 다른 **context가 생성**되어 처리기되기 때문이기 때문에 **DOM 처리나 다른 자바스크립트 처리와 병렬적**으로 실행되기 때문이다.



**2.4 callback, promise, async-await**

- **Callback**: **다른 함수의 매개변수로서 넘겨주는 실행 가능한 함수를** 말한다. 자바스크립트의 이벤트 핸들러, AJAX로 API를 호출 후 **결과를 처리**하는 함수가 대표적인 callback 함수라고 할 수 있다.

<https://ko.wikipedia.org/wiki/%EC%BD%9C%EB%B0%B1>

- 함수에 넘겨주는 매개변수를 함수를 익명(anonymous)의 함수로 넘겨주게 되면 callback hell이 발생할 수 있어 되도록이면 함수를 선언하여 함수의 이름을 넘겨줄 수 있도록 하는 것이 좋다.

fs.readdir(source, **function** (err, files) {

**if** (err) {

console.log('Error finding files: ' **+** err)

} **else** {

files.**forEach**(**function** (filename, fileIndex) {

console.log(filename)

gm(source **+** filename).size(**function** (err, values) {

**if** (err) {

console.log('Error identifying file size: ' **+** err)

} **else** {

console.log(filename **+** ' : ' **+** values)

aspect **=** (values.width / values.height)

widths.**forEach**(**function** (width, widthIndex) {

height **=** Math.round(width / aspect)

console.log('resizing ' **+** filename **+** 'to ' **+** height **+** 'x' **+** height)

**this**.resize(width, height).write(dest **+** 'w' **+** width **+** '\_' **+** filename, **function**(err) {

**if** (err) console.log('Error writing file: ' **+** err)

})

}.bind(**this**))

}

})

})

}

})

<익명함수로 처리>

**var** form **=** document.querySelector('form')

form.onsubmit **=** **function** (submitEvent) {

**var** name **=** document.querySelector('input').value

request({

uri: "http://example.com/upload",

body: name,

method: "POST"

}, **function** (err, response, body) {

**var** statusMessage **=** document.querySelector('.status')

**if** (err) **return** statusMessage.value **=** err

statusMessage.value **=** body

})

}

<함수의 이름을 주고 선언하여 처리>

document.querySelector('form').onsubmit **=** formSubmit

**function** **formSubmit** (submitEvent) {

**var** name **=** document.querySelector('input').value

request({

uri: "http://example.com/upload",

body: name,

method: "POST"

}, postResponse)

}

**function** **postResponse** (err, response, body) {

**var** statusMessage **=** document.querySelector('.status')

**if** (err) **return** statusMessage.value **=** err

statusMessage.value **=** body

}

<http://callbackhell.com/>

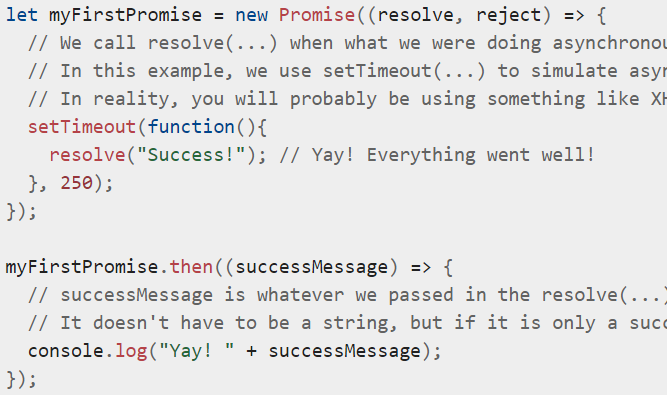
-**Promise**는 비동기 작업의 완료 또는 실패에 대한 결과 값을 나타내는 객체이며, promise 객체를 호출하면 Promise 객체 내부적으로 완료된 결과값은 resolve로 처리되며 오류가 발생했을 때는 reject로 처리된다. 또한 Promise 객체를 호출하고 완료된 결과는 then에 함수로 처리한다 요류로인한 결과는catch에 오류 결과값을 처리하는 함수로 처리한다.

- Promise는 3가지 상태를 가진다.

: 대기(pending) – 이행하거나 거부되지 않은 초기 상태.

: 이행(fulfilled) – 연산이 성공적으로 완료됨.

: 거부(rejected) – 연산이 실패함.

****

****

[**https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global\_Objects/Promise**](https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Promise)

[**https://github.com/axios/axios**](https://github.com/axios/axios)

**// ES2015**

**const result = api.list(); // 바로 promise 객체를 리턴함**

**result.then(response => { console.log(response.data); })**

**-> 네트워크 통신이 완료(resolve)되면 .then의 콜백함수를 호출**

**// ES2017**

**async function a (**

**const result = await api.list(); // 네트워크 통신이 끝나면 완료(resolve)값 반환**

**console.log(result.data);**

**)**

let a = 'a';

const result = api.list(); // 실행해서 promise 받기 1

result.then(response => {

console.log(response); // 실행해서 출력하기 3

}); // 콜백함수 정의

console.log(a); // 실행해서 출력하기 2

-> 출력결과:

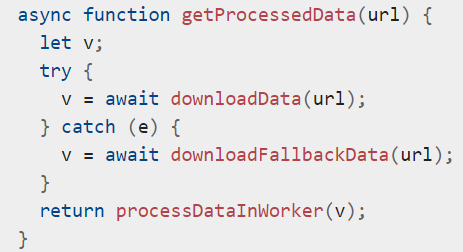
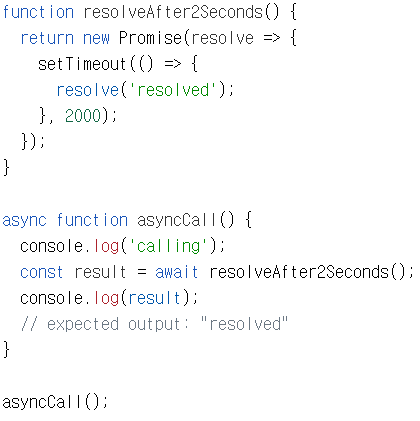
1 - ‘a’

2 - response: {}

**- async-await:**

: **async** 키워드는 비동기 함수를 정의할 때 사용하며, 비동기 함수는 이벤트 루프를 통해 비동기적으로 작동하는 함수로, 암시적으로 Promise를 사용하여 결과를 반환합니다.

: **await** 키워드는 비동기 함수의 Promise에서 resolve가 처리되기까지 대기하는 키워드이며 resolve된 값을 동기적인 함수호출의 리턴 값처럼 처리할 수 있으며, reject된 오류는 try catch 구문으로 감싸서 처리할 수 있다.



<https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Statements/async_function>

async function a (

let a = 'a';

const response = await api.list(); // 실행 대기 결과 반환 1

console.log(response); // 실행 결과 출력 2

console.log('a'); // 실행 출력 3

)

**[ 데이터베이스 ]**

**1. 정규화(Normalization)와 역정규화(De-normalization)**

- 관계형 데이터베이스의 설계에서 **중복을 최소화하게** 데이터를 구조화하는 프로세스를 **정규화**라고 한다.

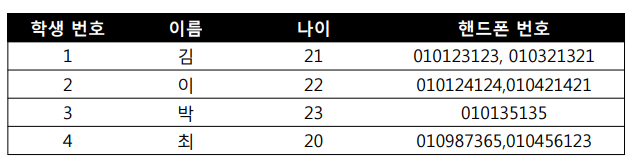
- 일반적으로 정규화란 크고, 제대로 조직되지 않은 테이블들과 관계들을 작고 잘 조직된 **테이블과 관계들로 나누는 것**을 포함한다.

**- 나눈 테이블에는 해당 테이블에 필요한 속성 존재하게 함.**

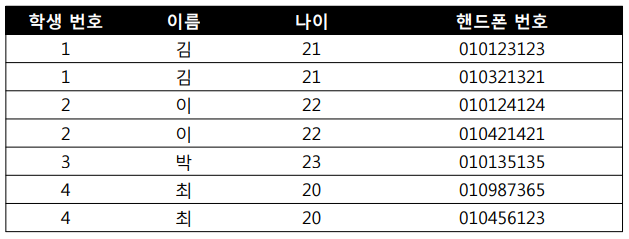
<https://ko.wikipedia.org/wiki/%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%B2%A0%EC%9D%B4%EC%8A%A4_%EC%A0%95%EA%B7%9C%ED%99%94>

**- 제1정규화:** 도메인(컬럼값)은 모두 원자 값이어야 함

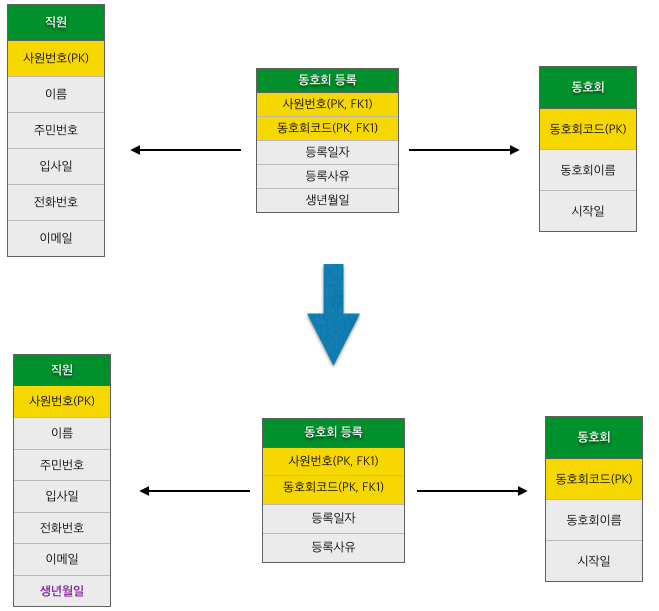
<정규화 전>



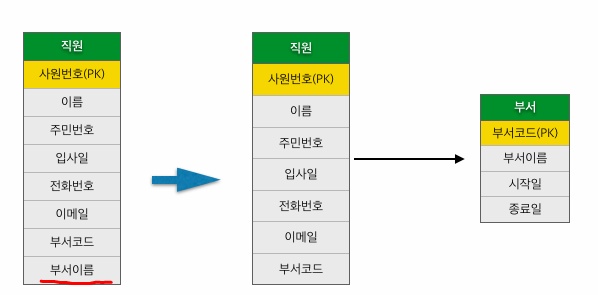
<정규화 후>



**- 제2정규화:** 제1정규화이면서 부분함수적 종속성 제거. 기본키가 복합 키로 구성되었을 때, 복합 키 전체에 의존하지 않고 일부에 의존적인 열이 있으면 이를 제거해야 하는 것을 의미합니다.



**- 제3정규화:** 제2정규화이면서 이행적 함수종속 없음. 기본 키에 의존하지 않고 일반 열에 의존하는 열이 있다면 이를 제거하는 것

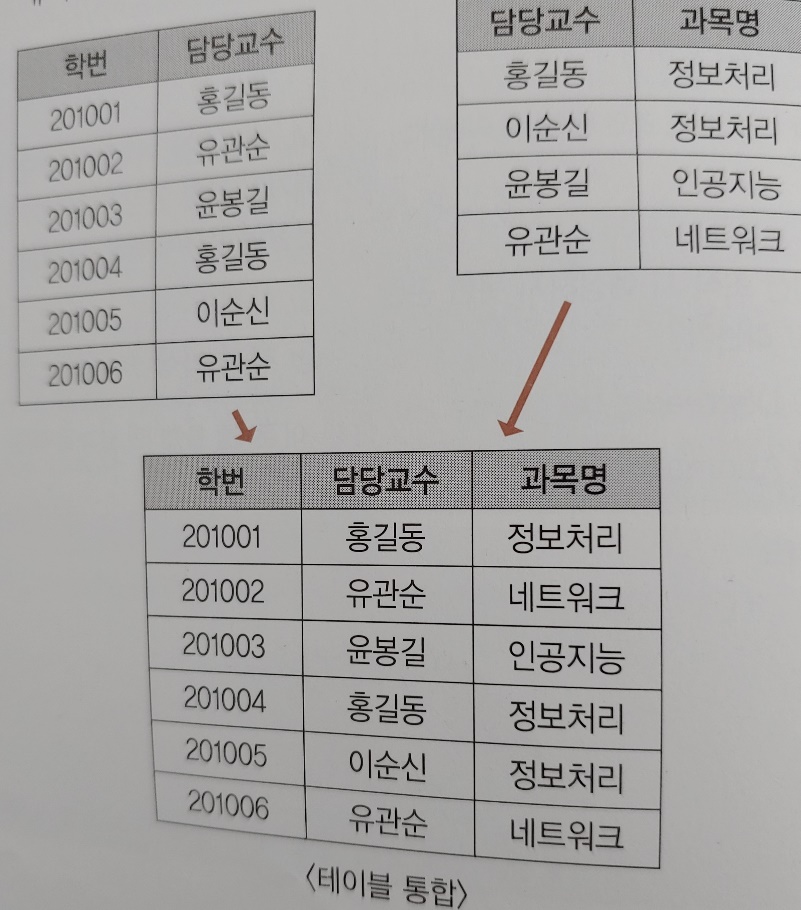


<https://nirsa.tistory.com/107>

<https://lee-mandu.tistory.com/478>

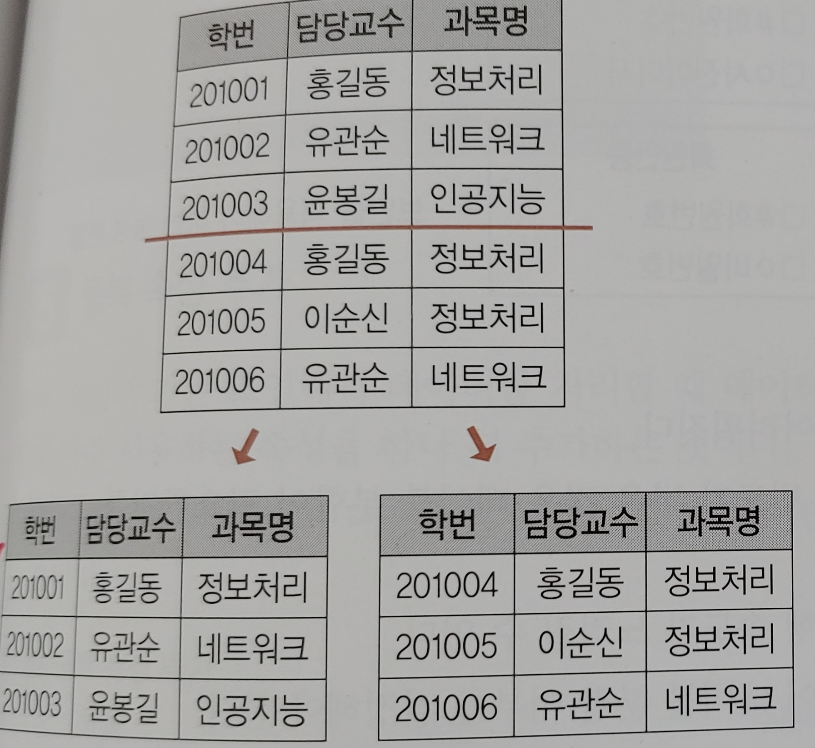
- **역정규화**는 주로 애플리케이션의 성능 향상, 개발 및 운영 편의성의 목적에 따라서 정규화된 데이터모델을 통합, 분리하는 과정으로 의도적으로 정규화 원칙을 위배하는 행위이다. 방법적으로는 테이블 통합, 테이블 분할, 중복 테이블 추가, 중복 속성 추가 등의 방법이 있다.

**1. 테이블 통합:** 두 개의 테이블이 조인되는 경우가 많고 추가/수정 보다 조회가 많은 경우



**2. 테이블 분할:** 테이블을 수직 또는 수평으로 분할

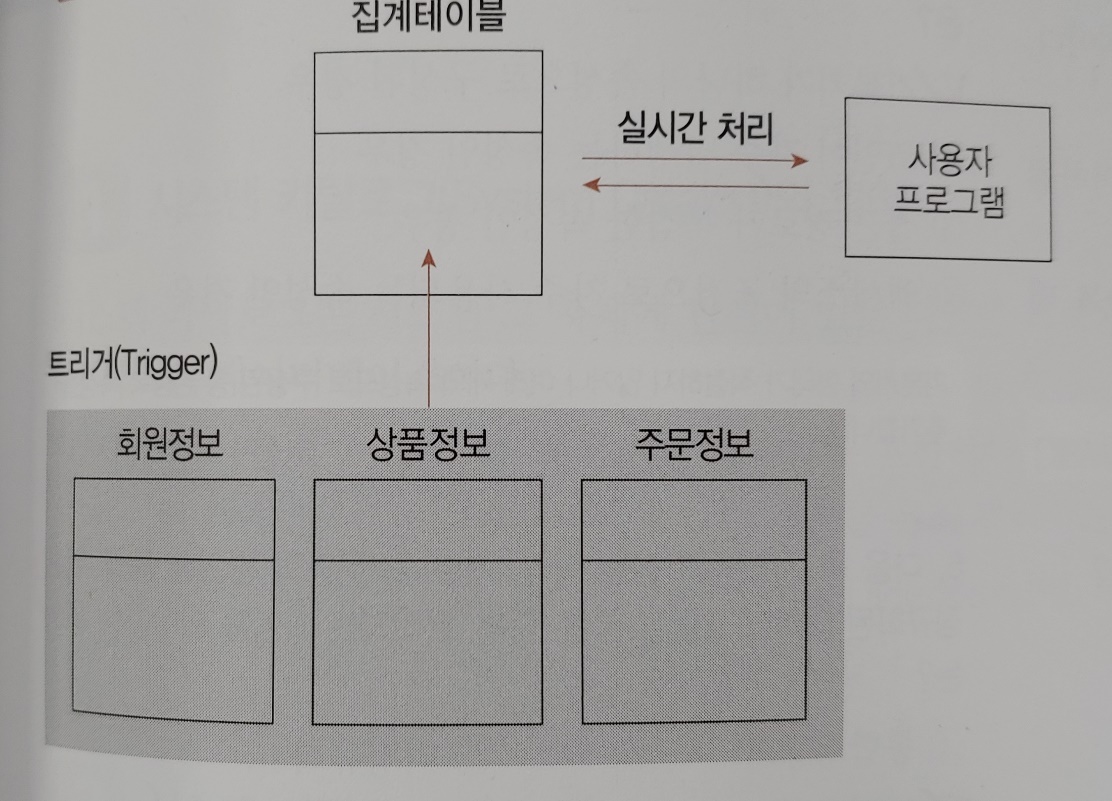
<수평분할> - 레코드를 기준으로 분할, 레코드별 사용빈도 차이가 클 때 분할( ex> 최신 글 조회)

****

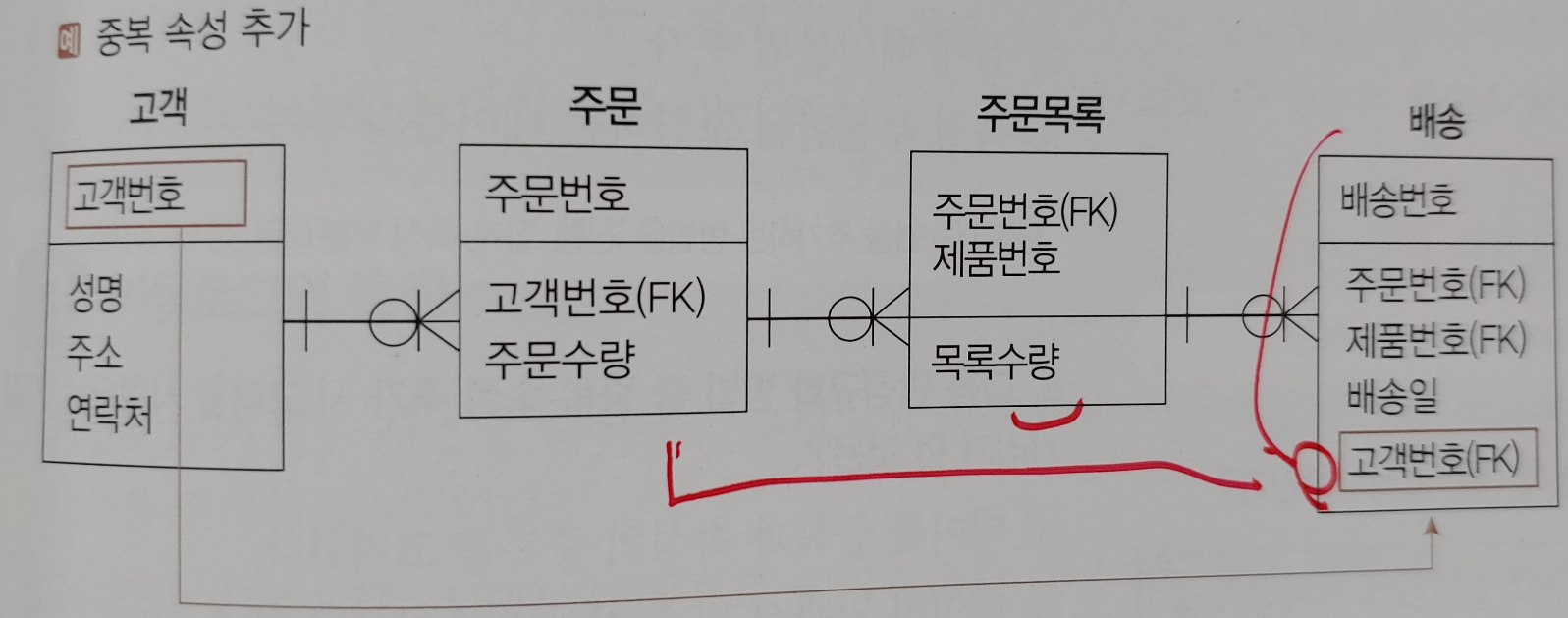
<수직분할> - 갱신위주의 속성, 자주 조회되는 속성, 크기가 큰 속성을 분할

****

**3. 중복 테이블 추가:** 여러 테이블의 데이터를 추출 또는 다른 서버에 저장된 테이블을 이용해야 하는 경우 중복 테이블을 추가한다.



**4. 중복 속성 추가:** 데이터를 조회하는 경로를 단축하기 위해 자주 사용하는 속성을 추가하는 것



[**https://book.naver.com/bookdb/book\_detail.nhn?bid=17769459**](https://book.naver.com/bookdb/book_detail.nhn?bid=17769459)

**2. 트랜잭션(Transaction)**

- 트랜잭션(Transaction)은 **데이터베이스의 상태를 변화시키는 하나의 논리적** 기능을 수행하기 위한 **작업의 단위** 또는 한꺼번에 모두 수행되어야 할 일련의 연산들을 의미한다.

- **ACID:** (Atomicity: 원자성, Consistency: 일관성, Isolation: 고립성, Durability: 지속성)는 데이터베이스 트랜잭션이 안전하게 수행된다는 것을 보장하기 위한 성질을 가리키는 약어이다.

- **원자성(Atomicity)**은 트랜잭션과 관련된 작업들이 부분적으로 실행되다가 중단되지 않는 것을 보장하는 능력이다. 예를 들어, 자금 이체는 성공할 수도 실패할 수도 있지만 보내는 쪽에서 돈을 빼 오는 작업만 성공하고 받는 쪽에 돈을 넣는 작업을 실패해서는 안된다. 원자성은 이와 같이 중간 단계까지 실행되고 실패하는 일이 없도록 하는 것이다. (하나의 처리, 나눠서 처리)

- **일관성(Consistency)**은 트랜잭션이 실행을 성공적으로 완료하면 언제나 일관성 있는 데이터베이스 상태로 유지하는 것을 의미한다. **무결성 제약이** 모든 계좌는 잔고가 있어야 한다면 이를 **위반**하는 트랜잭션은 중단된다.

- **독립성(Isolation)**은 트랜잭션을 수행 시 **다른 트랜잭션의 연산 작업이 끼어들지 못하도록** 보장하는 것을 의미한다. 이것은 트랜잭션 밖에 있는 어떤 연산도 중간 단계의 데이터를 볼 수 없음을 의미한다. 은행 관리자는 이체 작업을 하는 도중에 쿼리를 실행하더라도 특정 계좌간 이체하는 양 쪽을 볼 수 없다. 공식적으로 고립성은 트랜잭션 실행내역은 연속적이어야 함을 의미한다. 성능관련 이유로 인해 이 특성은 가장 유연성 있는 제약 조건이다. 자세한 내용은 관련 문서를 참조해야 한다.

- **지속성(Durability)**은 성공적으로 수행된 트랜잭션은 **영원히 반영되어야 함**을 의미한다. 시스템 문제, DB 일관성 체크 등을 하더라도 유지되어야 함을 의미한다. 전형적으로 모든 트랜잭션은 **로그**로 남고 시스템 장애 발생 전 상태로 되돌릴 수 있다

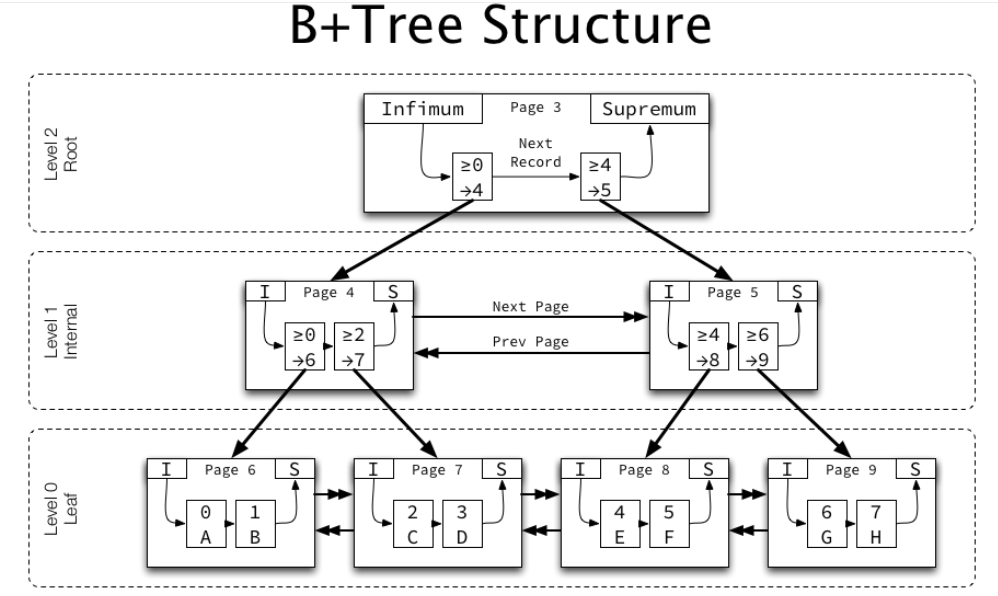
<https://ko.wikipedia.org/wiki/ACID>

<https://coding-factory.tistory.com/226>

**3. 인덱스**

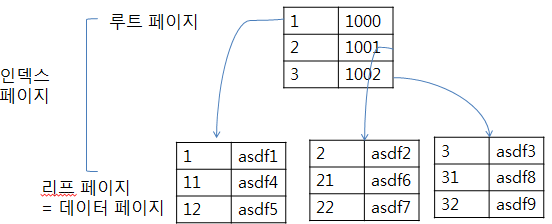
- 인덱스란 추가적인 쓰기 작업과 저장 공간을 활용하여 데이터베이스 테이블의 검색 속도를 향상시키기 위한 자료구조이다. 주로 B+ Tree 구조를 많이 사용한다.

- B+ Tree 특징: 1) 리프노드(데이터노드)만 인덱스와 함께 데이터(Value)를 가지고 있고, 2) 나머지 노드(인덱스노드)들은 데이터를 위한 인덱스(Key)만을 갖는다. 3) 리프노드들은 LinkedList로 연결되어 있다.데이터 노드 크기는 인덱스 노드의 크기와 같지 않아도 된다.

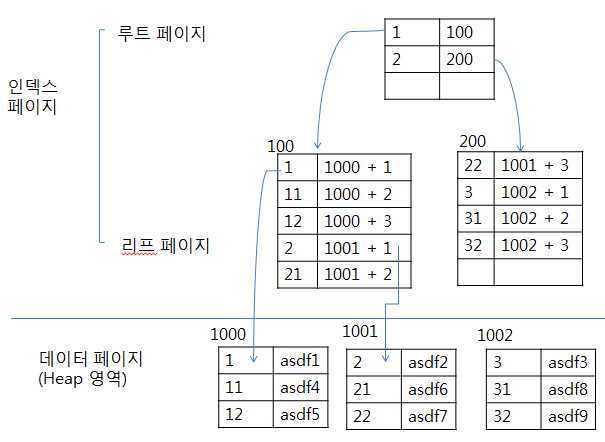


- Clustered Index와 Non-clustred Index: 테이블당 한 개만 생성가능하며 인덱스의 순서에 따라서 실제 데이터가 정렬되어 있다.

<Clustered index> - 인덱스의 가장 하위 페이지(노드)에 데이터가 인덱스 순서대로 정렬되어 있다.



<Non-Clustered index> - 인덱스의 가장 하위 페이지(노드)는 데이터가 있는 메모리 번지만 참조하고 있다.



<https://mangkyu.tistory.com/96>

<https://mongyang.tistory.com/75>

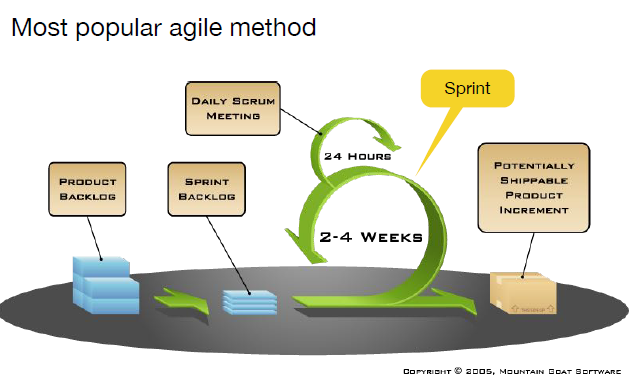
**[ 소프트웨어공학 ]**

**1. Agile 소프트웨어 개발 방법론**

- 애자일은 ‘민첩합’, ‘기민한’이라는 의미로, 고객의 요구사항 변화에 유연하게 대응할 수 있도록 일정한 주기를 반복하면서 개발과정을 진행한다.

- 애자일 모형은 **스프린트(Sprint)** 또는 **이터레이션(Iteration)**이라고 불리는 짧은 개발 주기를 반복하며, 반복되는 주기마다 만들어지는 **결과물(product increment)**에 대한 고객의 평가와 요구를 적극 수용한다.

[**https://book.naver.com/bookdb/book\_detail.nhn?bid=17769459**](https://book.naver.com/bookdb/book_detail.nhn?bid=17769459)



- 실천방법으로는 **Scrum**과 **XP(eXtreme Programming)**이 있다.

**2. 스크럼(Scrum)**

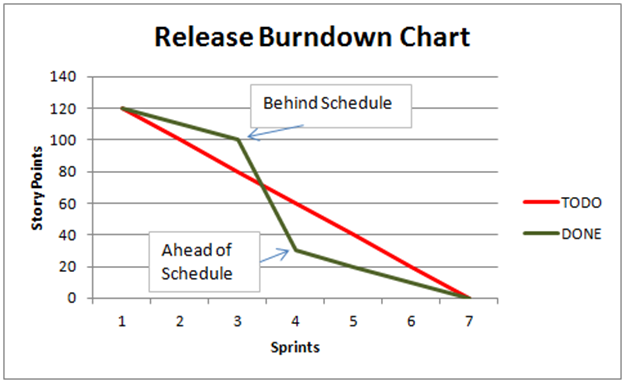
- **스크럼(Scrum)**의 어원은 럭비에서 반칙으로 경기가 중단된 경우 양팀의 선수들이 럭비공을 가운데 두고 상대팀을 밀치기 위해 서로 대치해 있는 대형을 말한다. 스크럼은 초기 위험 관리(early risk management)에 초점을 맞춘 관리 및 제어 프로세스이다. 매일 스크럼 미팅이 진행되며, 스프린트가 끝날 때마다 각 스프린트 검토하고 우리가 달성한 것과 향후 2 ~ 4 주 내에 무엇을 할지를 계획하는 과정으로 진행된다.

: **제품 백로그**(product backlog) – 제품 개발에 필요한 모든 요구사항(user story)을 우선순위에 따라 나열한 목록이다.

: 스프린트 계획 회의(Planning) – 스프린트에 처리할 요구사항을 개발자들이 나눠서 작업할 수 있도록 태스크(task) 단위로 나누고 **스프린트 백로그(sprint backlog)**를 작성한다.

: **스프린트** – 실제 개발 작업을 진행하는 과정, 2-4주정도의 기간, 태스크는 할 일(to-do), 진행중(in-progress), 완료(done)의 상태를 갖는다.

: **일일 스크럼 미팅(Daily Scrum Meeting)** – 모든 팀원이 매일 약속된 시간에 약 15분 정도의 짧은 시간동안 진행 상황을 점검한다. 남은 작업 시간은 **burn-down chart**에 표시한다. **스크럼 마스터(master)**는 발견된 장애 요소를 해결할 수 있도록 지원하는 역할을 한다. Product Owner(요구사항 관리자) -> 요구사항에 구현 진척상황 관리, 추가적이거나 변경되는 요구사항 관리. Risk(문제가 발생할 수 있는 상황), Issue(문제 발생해서 해결해야되는 상황)



: 스프린트 검토 회의(Review) – 부분 또는 전체 완성 제품이 요구사항에 잘 부합되는지 사용자가 포함된 참석자 앞에서 테스팅을 수행한다. 스프린트의 한 주당 한 시간 정도로 진행한다. 제품 책임자(Product Owner)는 개선할 사항에 대한 피드백을 정리한 후 다음 스프린트에 반영할 수 있도록 제품 백로그를 업데이트한다.

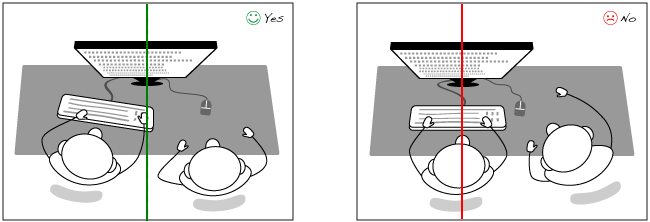
: 스프린트 회고(Retrospective) – 스프린트 주기를 되돌아보며 정해놓은 규칙을 잘 준수했는지, 개선할 점은 없는지 등을 확인하고 기록한다. 스프린트가 끝난 시점에 수행하거나 일정 주기로 수행한다.

**3. XP(eXtreme Programming)**

: **XP(eXtreme Programming)**에서는 반복주기를 이터레이션(iteration)이라고 하면 1~3주정도의 기간으로 진행된다.

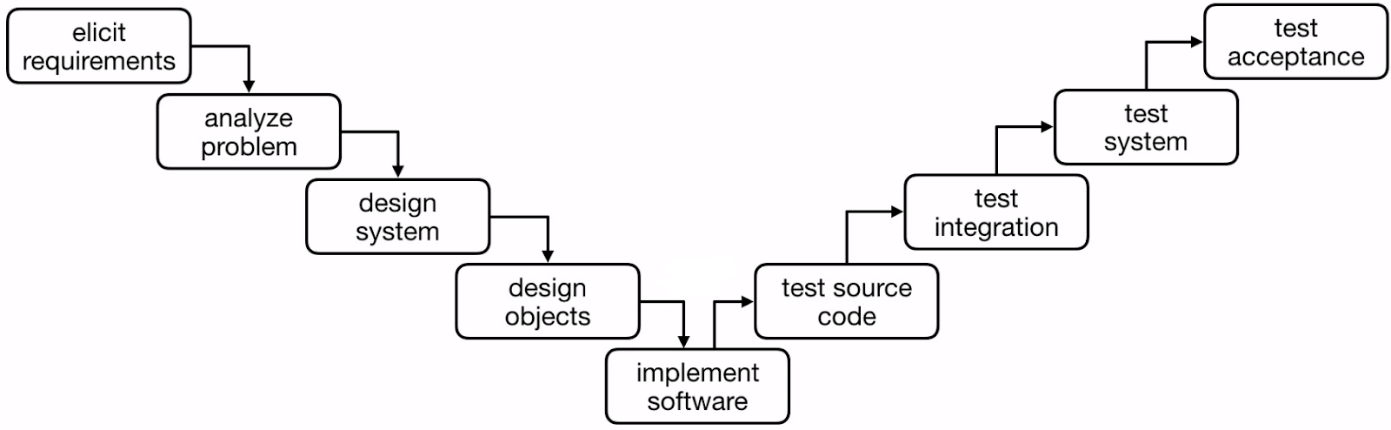
**: Pair-Programming(짝프로그래밍)** – 짝으로 나눠서 교대로 프로그래밍 수행. Driver(코드짜는사람)와 Navigator(옆에서보는사람). 서로 코드 리뷰하는 것, Peer-Review

<https://gmlwjd9405.github.io/2018/07/02/agile-pair-programming.html>



**: TDD(Test Driven Development)** – 실제 구현 코드를 작성하기 전에 **요구사항에 따라 테스트케이스를 먼저 작성**하고 **개발하는 방식**을 말한다. 따라서 요구사항이 무엇인지 좀더 정확히 파악한 후 개발을 진행할 수 있으며, 개발이 진행될 때 **테스트가 지속으로 진행**될 수 있도록 자동화된 테스팅 도구를(JUnit, Cucumber, Jest) 사용한다.

\*\* 테스트 단계에 따른 소프트웨어 생명주기 모델(V-Model)



\*\* 단위 테스트(소스코드) -> 통합 테스트(서비스간 연동 테스트) -> 시스템 테스트 -> 인수 테스트

**: CI/CD(Continuous Integration/Continuous Delivery)** – **나눠서 개발된 모듈들을 지속적으로 통합**하고 주요 **의사결정권자들이 확인할 수 있는 배포환경에 전달**하는 것을 말한다.

