복습 1

**public** **class** TreeNode {

Object data;

TreeNode left;

TreeNode right;

}

**public** **class** BT {

TreeNode root;

BT() {

root = **null**;

}

BT(BT left, Object data, BT right){

root = **new** TreeNode();

root.data = data;

root.left = left.root;

root.right = right.root;

}

**boolean** isEmpty(){

**if**(root == **null**)

**return** **true**;

**else**

**return** **false**;

}

BT leftTree(){

BT newTree = **new** BT();

newTree.root = root.left;

**return** newTree;

}

BT rightTree(){

BT newTree = **new** BT();

newTree.root = root.right;

**return** newTree;

}

Object rootData(){

**if**(isEmpty() == **true**)

**return** **null**;

**else**

**return** root.data;

}

}

**public** **class** BinaryTreeTest {

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

BT bt;

BT lt;

BT rt;

BT leftT;

BT rightT;

lt = **new** BT(**new** BT(), "D", **new** BT());

rt = **new** BT(**new** BT(), "E", **new** BT());

bt = **new** BT(lt, "B", rt);

lt = bt;

rt = **new** BT(**new** BT(), "C", **new** BT());

bt = **new** BT(lt, "A", rt);

System.***out***.println("최상위 노드 데이터값");

System.***out***.println(bt.rootData());

System.***out***.println("좌측 서브트리의 루트 노드 데이터값");

leftT = bt.leftTree();

System.***out***.println(leftT.rootData());

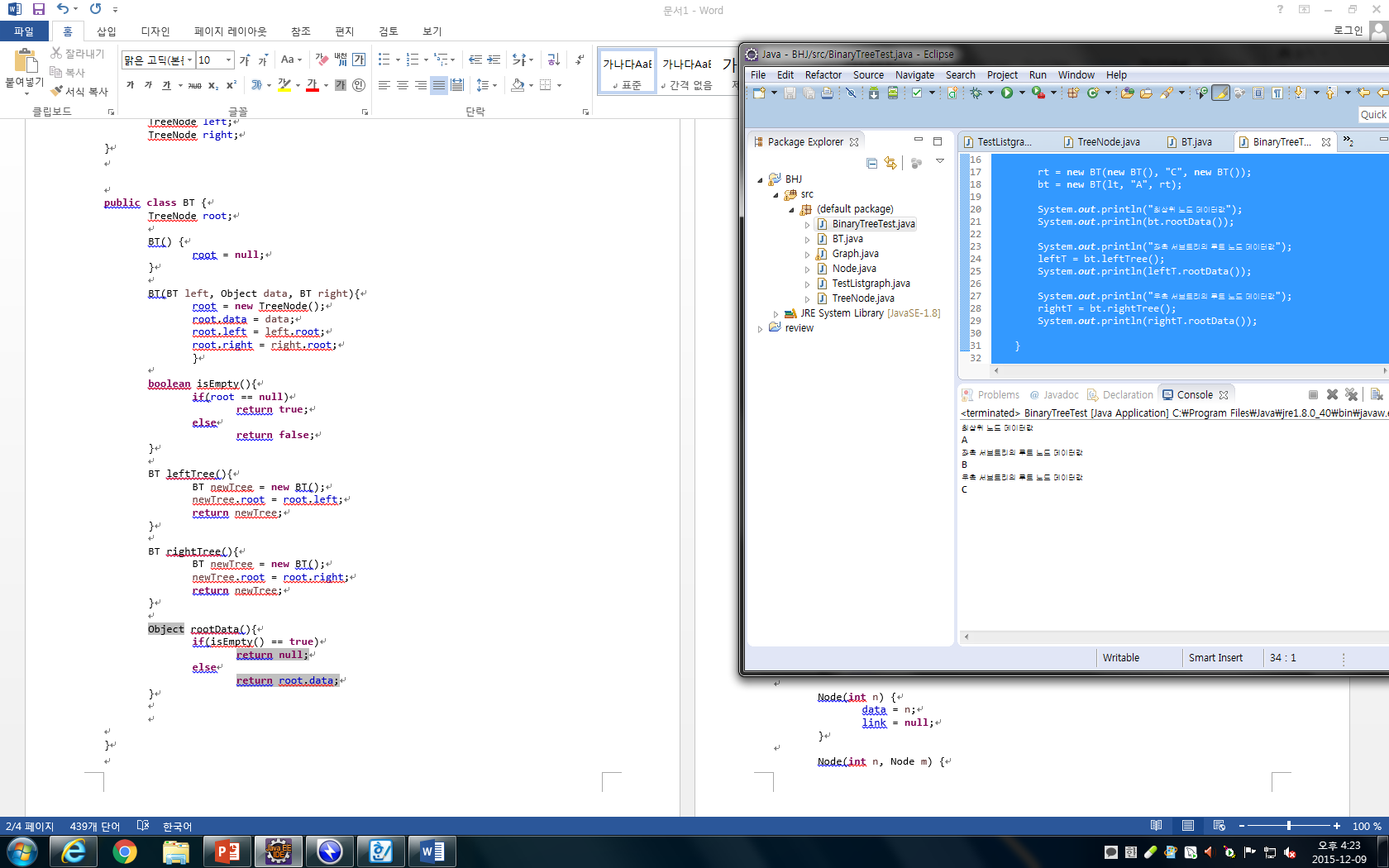
System.***out***.println("우측 서브트리의 루트 노드 데이터값");

rightT = bt.rightTree();

System.***out***.println(rightT.rootData());

}

}



복습 2

**public** **class** Node {

**int** data;

Node link;

Node(**int** n) {

data = n;

link = **null**;

}

Node(**int** n, Node m) {

data = n;

link = m;

}

}

**import** java.util.Stack;

**public** **class** Graph {

Node header[] = **new** Node[10];

**boolean** visited[] = **new** **boolean**[10];

Graph() {

**for** (**int** i = 0; i < 10; i++) {

visited[i] = **false**;

}

}

**public** **void** bulid() {

header[0] = **new** Node(0, **new** Node(1, **new** Node(2, **new** Node(3))));

header[1] = **new** Node(1, **new** Node(0, **new** Node(4)));

header[2] = **new** Node(2, **new** Node(0, **new** Node(4, **new** Node(5))));

header[3] = **new** Node(3, **new** Node(0, **new** Node(5)));

header[4] = **new** Node(4, **new** Node(1, **new** Node(2, **new** Node(6))));

header[5] = **new** Node(5, **new** Node(2, **new** Node(3, **new** Node(6))));

header[6] = **new** Node(6, **new** Node(4, **new** Node(5)));

}

**public** **void** display(**int** n) {

Node m = header[n];

**while** (m != **null**) {

System.***out***.print(m.data + " ");

m = m.link;

}

}

**public** **void** DFS(**int** m) {

**for** (**int** i = 0; i < 10; i++) {

visited[i] = **false**;

}

Stack s = **new** Stack();

Node n = header[m];

s.push(n);

Node p;

**while** (!s.isEmpty()) {

p = (Node) s.pop();

**if** (visited[p.data] == **false**) {

System.***out***.print(header[p.data].data + "");

visited[p.data] = **true**;

}

**while** (p.link != **null**) {

p = p.link;

**if** (visited[p.data] == **false**) {

s.push(header[p.data]);

}

}

}

}

}

**public** **class** TestListgraph {

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

Graph g = **new** Graph();

g.bulid();

System.***out***.println("그래프의 인접 리스트 표현");

**for** (**int** i = 0; i < 7; i++) {

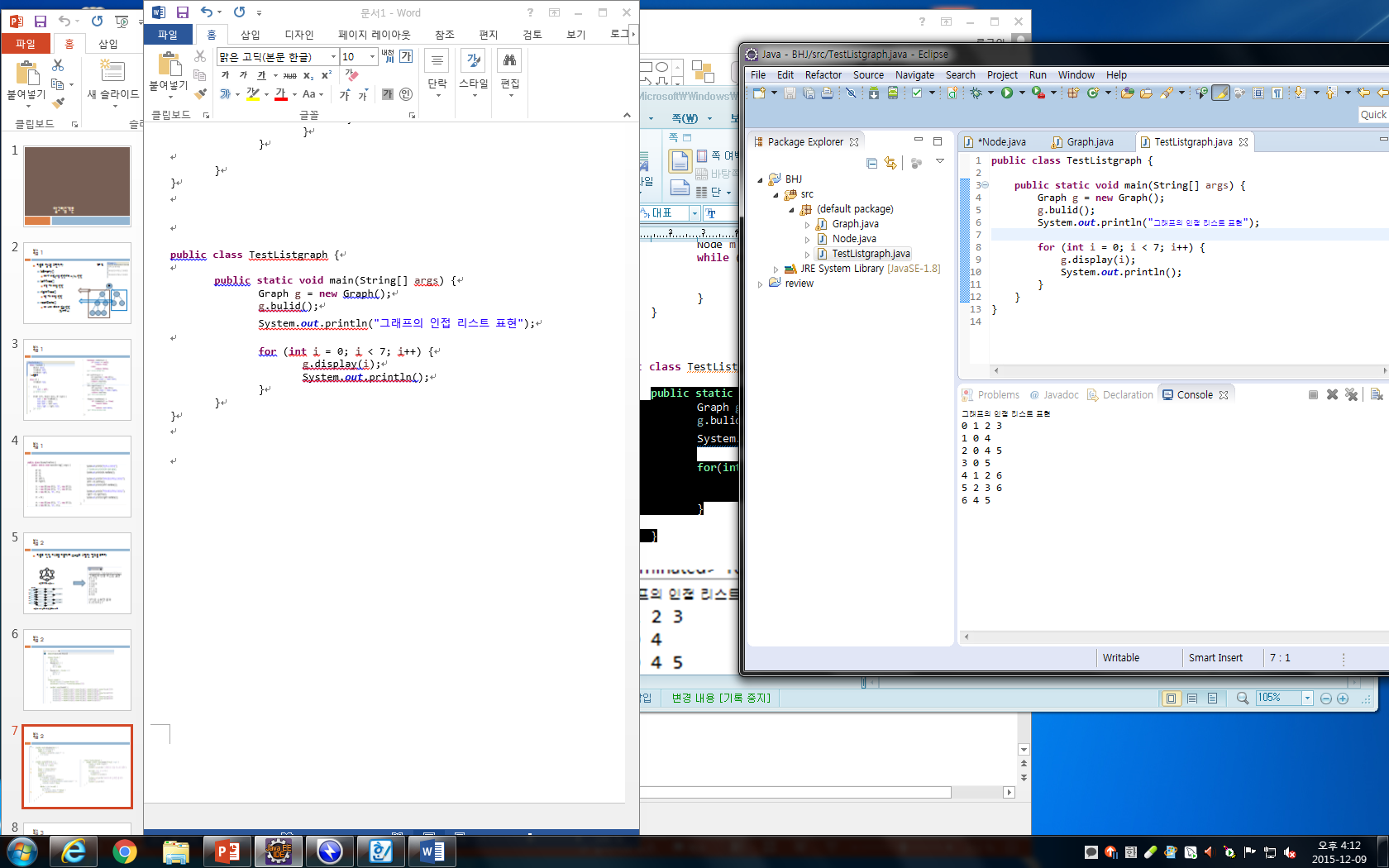
g.display(i);

System.***out***.println();

}

}

}



복습 3

**public** **class** Dijkstral {

**int** weight[][] = **new** **int**[5][5];

**int** Dist[] = **new** **int**[5];

**int** num, u;

**boolean** S[] = **new** **boolean**[5];

**public** Dijkstral(**int** w[][], **int** nv) {

num = nv;

**for** (**int** i = 0; i < num; i++) {

**for** (**int** j = 0; j < num; j++) {

weight[i][j] = w[i][j];

}

}

}

**public** **void** shortestPath(**int** v){

**for**(**int** i=0; i<num; i++){ //S[], Dist[] 초기화

S[i] = **false**;

Dist[i] = weight[v][i];

}

S[v] = **true** ;

Dist[v] = 0;

//시작값

**for**(**int** i =0; i<num-2; i++){

**int** mindist = 999;

//시작 정점에 연결되어 있는 가중치 중 가장 작은 값 선택

**for**(**int** j=0; j<num; j++){

**if**(Dist[j] < mindist && S[j] == **false**){ //최단거리가 발견되지 않았고, Dist의 값이 mindist보다 작을 때

u = j; //찾은 정점을 u에 저장

mindist = Dist[j]; //찾은 Dist값으로 초기화

}

}

S[u] = **true**; //선택된 정점 표시

//위에서 선택된 가장 작은 Dist값 + 그 값을 가진 정점(A)에서 다른 정점(B)로

//연결된 값이 지금 가지고 있는 값(B의 Dist값)보다 더 작을 경우 값 변경

**for**(**int** w = 0; w<num; w++){

**if**(S[w] == **false**){

**if**(Dist[w] > (Dist[u] + weight[u][w]))

Dist[w] = (Dist[u] + weight[u][w]);

}

}

}

}

**public** **void** print\_ds() { // 출력

**for** (**int** i = 0; i < num; i++)

System.***out***.print(S[i] + "\t");

System.***out***.println();

**for** (**int** i = 0; i < num; i++)

System.***out***.print(Dist[i] + "\t");

System.***out***.println();

}

}

**class** Dijkstra {

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

**int** nv = 5; // 접점 수

**int** weight[][] = { { 0, 2, 5, 999, 3 }, { 999, 0, 999, 4, 10 },

{ 999, 999, 0, 6, 2 }, { 999, 999, 999, 0, 999 },

{ 999, 999, 1, 2, 0 } }; // 가중치 인접행렬

Dijkstral ds = **new** Dijkstral(weight, nv);

ds.shortestPath(0);

ds.print\_ds();

}

}

