FAKULTI TEKNOLOGI
KEJURUTERAAN KELAUTAN
DAN INFORMATIK

2019/2020

DATA STRUCTURE & ALGORITHM

Lab 6: Generic Class & Tree

VERSION 1

STUDENT INFORMATION

PLEASE FILL IN YOUR DETAILS:

NAME: OMAR ISMAIL ABDJALEEL ALOMORY

MATRIC NUMBER:S63955

GROUP:K2

LAB:MP3

DATE:13/12/2022

TABLE OF CONTENTS

INSTRUCTIONS

Manual makmal ini adalah untuk kegunaan pelajar-pelajar Fakulti Teknologi Kejuruteraan Kelautan dan Informatik, Universiti Malaysia Terengganu (UMT) sahaja. Tidak dibenarkan mencetak dan mengedar manual ini tanpa kebenaran rasmi daripada penulis.

Sila ikuti langkah demi langkah sebagaimana yang dinyatakan di dalam manual.

This laboratory manual is for use by the students of the Faculty of Ocean Engineering Technology and Informatics, Universiti Malaysia Terengganu (UMT) only. It is not permissible to print and distribute this manual without the official authorisation of the author.

Please follow step by step, as described in the manual.

TASK 1: UNDERSTANDING THE CONCEPT OF GENERIC CLASS

OBJECTIVE

In this task, students must be able to:

- Understand the concept of generic classes.
- Implement generic classes using Java.

ESTIMATED TIME

[60 Minutes]

DEFINITION OF GENERIC CLASSES

Generics in Java is similar to templates in C++. The idea is to allow type (Integer, String, .etc and user-defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// To create an instance of generic class

BaseType <Type> obj = new BaseType <Type>()
```

Note: In Parameter type we can not use primitives like `'int'`, `'char'` or `'double'`.

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterised classes or parameterised types because they accept one or more parameters.

NAMING CONVENTIONS FOR TYPE PARAMETERS

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- `E` - Element (used extensively by the Java Collections Framework)
- `K` - Key
- `N` - Number
- `T` - Type
- `V` - Value
- `S`, `U`, `V` etc. – 2nd, 3rd, 4th types

Important Notes:

Type Parameter and Type Argument Terminology: Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides "type arguments" in order to create a "parameterized type". Therefore, the `T` in `Foo<T>` is a type parameter, and the `String` in `Foo<String> f` is a type argument. This lesson observes this definition when using these terms.

STEPS:

1. Open NetBeans and create a new java application project.
2. Name your project as `GenericsExperiment` and click finish.
3. Change author profiles to :
    a. `Name :`
    b. `Program: <put your program. E.g., SMSK(SE) or SMSK with IM`
    c. `Course : CSF3104`
    d. `Lab : <enter lab number>`
    e. `Date : <enter lab date>`
4. In the same `GenericsExperiment` project's package, create a new file named `NonGenericClass.java`.
5. Add the following codes to the file:

```
public class NonGenericClass{

    String obj1;  // An object of type String
    String obj2;  // An object of type String

    // constructor
    NonGenericClass(String obj1, String obj2) {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print() {
        System.out.println(obj1);
        System.out.println(obj2);
    }

}
```

6. Now, create a test class to test the above class. Name your test class as `GenericClassDemo.java`. Your output should look like below:

```
run:
Final
Exam
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output

```
PS C:\Users\PC 48\Desktop\Omar>  & 'C:\Program Files\Java\jdk-11.0.13\bin\java.exe' '-cp' 'C:\Users\PC 48\AppData\Roaming\Code\User\workspaceStorage\17b2680b097dcedf91262f01328eaac7\redhat.java\jdt_ws\Omar_a21e8c9f\bin' 'GenericsExperiment.GenericClassDemo'
Final
Exam
```

7. Next, we are going to modify the above class to become a generic class.
8. Create a new file and name it as `MyGenericClass.java`.
9. Add <T, U> after the name of the class (see below). For this example, we put T and U in the parameter section to represent the generic type.

```
public class MyGenericClass<T, U> {
```

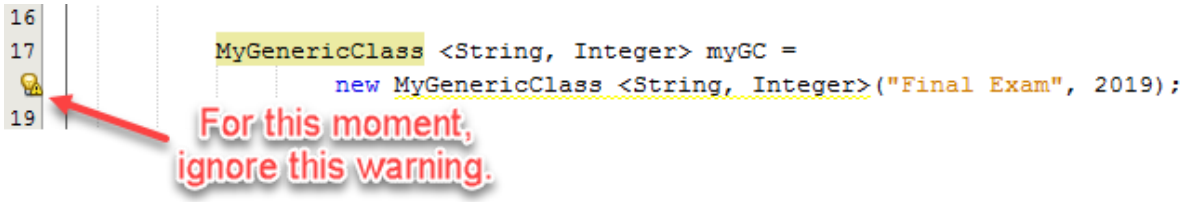
10. Copy the codes from `NonGenericClass.java` and paste it to the body of `MyGenericClass` class. Modify it to make it looks like follows:

```
T obj1;  // An object of type T
U obj2;  // An object of type U

// constructor
MyGenericClass(T obj1, U obj2) {
    this.obj1 = obj1;
    this.obj2 = obj2;
}

// To print objects of T and U
public void print() {
    System.out.println(obj1);
    System.out.println(obj2);
}
```

11. It's time to test your class. Use the previous test class to test `MyGenericClass`. Unlike previous steps, where we have supplied the class with `String` parameters, this time we are going to supply a `String` and an `Integer` as parameters to the class. So, part of the codes in the `main` method should look like below:



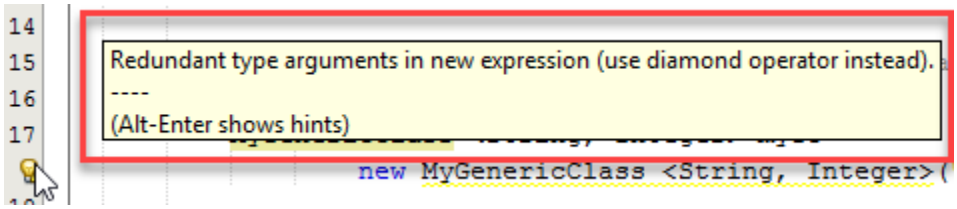
12. Save and compile your codes, and your output should look like below:

```
run:
Final Exam
2019
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output

```
PS C:\Users\PC 48\Desktop\Omar>  c:; cd 'c:\Users\PC 48\Desktop\Omar'; & 'C:\Progra
cp' 'C:\Users\PC 48\AppData\Roaming\Code\User\workspaceStorage\17b2680b097dcedf9126
8c9f\bin' 'GenericsExperiment.GenericClassDemo'
Final Exam
2019
```

13. Let's go back to discuss the warning appear in Netbeans:



Starting from Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle

brackets, <>, is informally called the **diamond**. For example, you can create an instance of `MyGenericClass <String, Integer>` with the following statement:

```
MyGenericClass <String, Integer> myGC =
                new MyGenericClass <>("Final Exam", 2019);
```

14. Fix the codes by following the above example and observe the output. Do the warnings disappear?
    Answer:
    Yes the Error gone

15. Copy and paste your Java codes into the text box below:
    Answer:
    None Generic Class

```
/*
 * NAME: OMAR ISMAIL ALOMORY
 * PROGRAM: SMSK (KMA) K2
 * LAB: MP3
 * DATE: 13/12/2022
 */
package GenericsExperiment;

public class NoneGenericClass{
    String obj1, obj2;

    public NoneGenericClass(String obj1, String obj2){
        this.obj1 =obj1;
        this.obj2 = obj2;
    }

    public void print(){
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

Generic Experiment Class

```java
/*
 * NAME: OMAR ISMAIL ALOMORY
 * PROGRAM: SMSK (KMA) K2
 * LAB: MP3
 * DATE: 13/12/2022
 */
package GenericsExperiment;

public class MyGenericClass <T , U> {
    T obj1;
    U obj2;

    public MyGenericClass(T obj1, U obj2){
        this.obj1 =obj1;
        this.obj2 = obj2;
    }

    public void print(){
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

Generics Demo Class

```java
/*
 * NAME: OMAR ISMAIL ALOMORY
 * PROGRAM: SMSK (KMA) K2
 * LAB: MP3
 * DATE: 13/12/2022
 */
package GenericsExperiment;


public class GenericClassDemo {
    public static void main(String[] args) {
        // NoneGenericClass test = new NoneGenericClass("Final","Exam");
        // test.print();
        MyGenericClass<String, Integer> myGC =
        new MyGenericClass<>("Final Exam", 2019);
        myGC.print();
    }
}
```

QUESTIONS

1. Discuss the differences between Non-generic and generic classes.

   Answer:

- With help of Generics, one needs to write a method/class/interface only once and use it for any type whereas, in non-generics, the code needs to be written again and again whenever needed.
- Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time).

Simple "Solution" class example to show the declaration of both:

Generic	Non-Generic

public class Solution<T>
{
T data;
public static T getData(){
return data;
}
}
	public class Solution
	{
	T data;
	public static String getData(){
	return data;
	}
	}

The key different here is that we can put any data here(String,Integer,Double ,etc)	Here is limited to only String value.

This is a simple example to show the different in syntax declaration.

2. Beside Class, where else you can apply Generics?

   Answer:

Methods and interfaces

3. List the advantages of using Generics in Java.

   Answer:

1. Code usability

   The same code can be used for several classes rather than just use it in one. Example for that is the code we just made, algorithm can be implemented with ease since they can be used to work with different types of object.

2. Type safeliy

   It means that error would arise on compile time ranther than run time.

TASK 2: IMPLEMENTING A GENERAL TREE

OBJECTIVE

In this task, students must be able to:

- Understand the concept of Tree data structure.
- Implement generic classes for Tree.

ESTIMATED TIME

[45 Minutes]

INTRODUCTION TO TREE

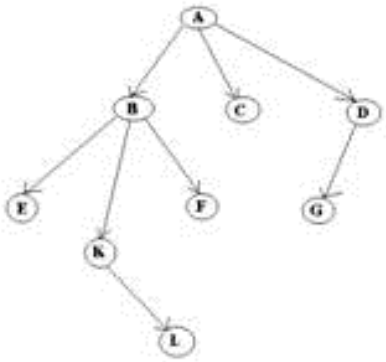
There are many basic data structures that can be used to solve application problems. Array is a good static data structure that can be accessed randomly and is fairly easy to implement. Linked Lists on the other hand is dynamic and is ideal for application that requires frequent operations such as add, delete, and update. One drawback of linked list is that data access is sequential. Then there are other specialized data structures like, stacks and queues that allows us to solve complicated problems (e.g.: Maze traversal) using these restricted data structures. One other data structure is the hash table that allows users to program applications that require frequent search and updates. They can be done in O(1) in a hash table.

One of the disadvantages of using an array or linked list to store data is the time necessary to search for an item. Since both the arrays and Linked Lists are **linear structures** the time required to search a "linear" list is proportional to the size of the data set. For example, if the size of the data set is n, then the number of comparisons needed to find (or not find) an item may be as bad as some multiple of n. So, imagine doing the search on a linked list (or array) with $n = 10^6$ nodes. Even on a machine that can do million comparisons per second, searching for m items will take roughly m seconds. This not acceptable in today's world where speed at which we complete operations is extremely important. Time is money. Therefore it seems that better (more efficient) data structures are needed to store and search data.

In this chapter, we can extend the concept of linked data structure (linked list, stack, queue) to a structure that may have multiple relations among its nodes. Such a structure is called a **tree**. A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a *nonlinear* data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more subtrees. A tree has following general properties:

- One node is distinguished as a **root;**

- Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent -> children*



- A is a parent of B, C, D,
  B is called a child of A.
  on the other hand, B is a parent of E, F, K

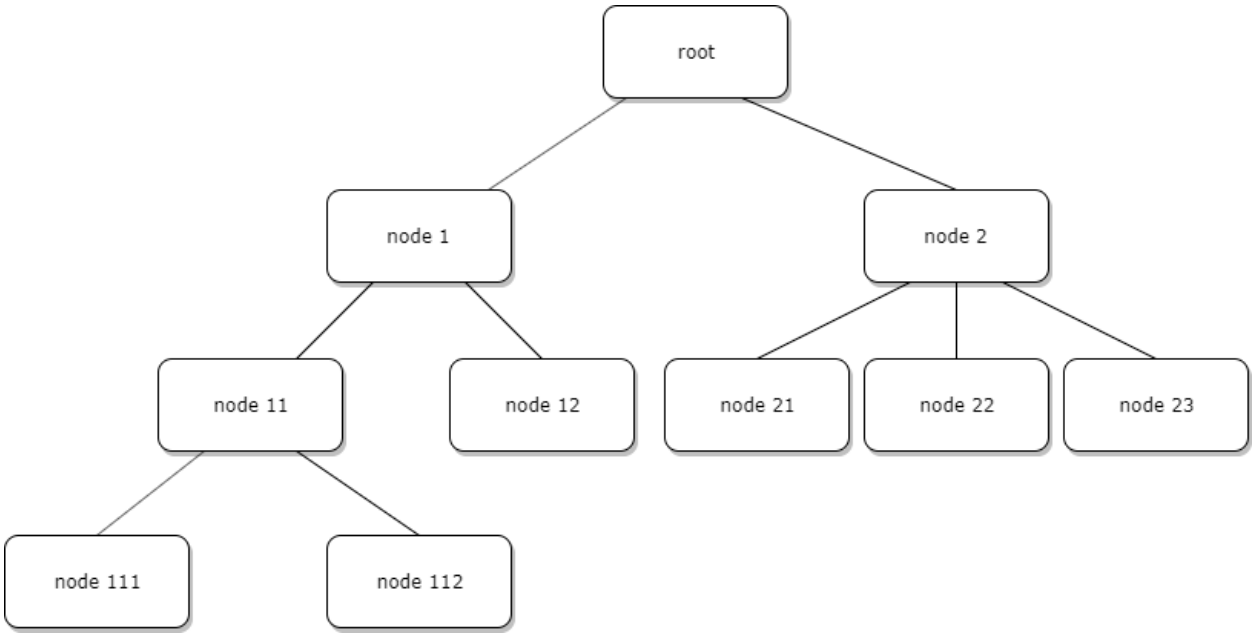
In the above picture, the root has 3 subtrees.

Each node can have *arbitrary* number of children. Nodes with no children are called **leaves**, or **external** nodes. In the above picture, C, E, F, L, G are leaves. Nodes, which are not leaves, are called **internal** nodes. Internal nodes have at least one child.

Nodes with the same parent are called **siblings**. In the picture, B, C, D are called siblings. The **depth of a node** is the number of edges from the root to the node. The depth of K is 2. The **height of a node** is the number of edges from the node to the deepest leaf. The height of B is 2. The **height of a tree** is a height of a root.

STEPS:

1. Given below is the example of a general Tree structure. In this task, we will do the implementation of a tree data structure using generic class.



2. First, open NetBeans and create new java application project.
3. Name your project as `TreeExperiment` and click finish.
4. Change author profiles to :
    a. `Name :`
    b. `Program: <put your program. E.g., SMSK(SE) or SMSK with IM`
    c. `Course : CSF3104`
    d. `Lab : <enter lab number>`
    e. `Date : <enter lab date>`

5. In the same `TreeExperiment` project's package, create a new file named `Node.java` and insert the following codes:

```java
import java.util.ArrayList;
import java.util.List;

public class Node <T> {

    private T data = null;

    private List<Node<T>> children = new ArrayList<>();

    private Node<T> parent = null;

    public Node(T data) {
        this.data = data;
    }

    public Node<T> addChild(Node<T> child) {
        child.setParent(this);
        this.children.add(child);
        return child;
    }

    public void addChildren(List<Node<T>> children) {
        children.forEach(each -> each.setParent(this));
        this.children.addAll(children);
    }


    public List<Node<T>> getChildren() {
        return children;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }

    private void setParent(Node<T> parent) {
        this.parent = parent;
    }

    public Node<T> getParent() {
        return parent;
    }
} // End of Node Class
```

6. Next, create a test class for `Node.java`. This test class will create a tree as described in Step 1.

```
public class TreeDemo {

    public static void main(String[] args) {
        Node<String> root = new Node<>("root");

        Node<String> node1 = root.addChild(new Node<String>("node 1"));

        Node<String> node11 = node1.addChild(new Node<String>("node 11"));
        Node<String> node111 = node11.addChild(new Node<String>("node 111"));
        Node<String> node112 = node11.addChild(new Node<String>("node 112"));

        Node<String> node12 = node1.addChild(new Node<String>("node 12"));

        Node<String> node2 = root.addChild(new Node<String>("node 2"));

        Node<String> node21 = node2.addChild(new Node<String>("node 21"));
        Node<String> node211 = node2.addChild(new Node<String>("node 22"));
        Node<String> node212 = node2.addChild(new Node<String>("node 23"));

        root.printTree(root, " ");
    }
}
```

7. Save, compile and run your source code. Observe the output. A correct output should look like below:

```
root
  node 1
    node 11
        node 111
        node 112
    node 12
  node 2
    node 21
    node 22
    node 23
```

8. Print screen your output from NetBeans and upload it using the following control box:

```
  root
   node1
     node11
          node111
          node112
     node12
   node2
     node21
     node22
     node23
PS C:\Users\komar\OneDrive\Desktop\Omar>
```

9. Copy and paste your Java codes from NetBeans into text box below:
   Node.java

```java
/*
 * NAME: OMAR ISMAIL ALOMORY
 * PROGRAM: SMSK (KMA) K2
 * COURSE: CSF3013
 * LAB: MP3
 * DATE: 13/12/2022
 */
package TreeExperiment;

import java.util.ArrayList;
import java.util.List;

public class Node<T> {
    private T data =null;

    private List<Node<T>> children = new ArrayList<>();

    private Node<T> parent = null;

    public Node(T data){
        this.data= data;
    }
    public Node<T> addChild(Node<T> child){
        child.setParent(this);
        this.children.add(child);
        return child;
```

```
    }
    public void addChildren(List<Node<T>> children){
        children.forEach(each -> each.setParent(this));
        this.children.addAll(children);
    }
    public List<Node<T>> getChildren(){
        return children;
    }
    public T getData(){
        return this.data;
    }

    public void setData(T data){
        this.data = data;
    }
    private void setParent(Node<T> parent){
        this.parent =parent;
    }
    public Node<T> getParent(){
        return this.parent;
    }

    public void printTree(Node<T> root ,String gap){
        if(root == null){
            return;
        }

        System.out.println(gap+ root.data);
        for ( Node<T> child : root.children) {
            printTree(child, gap+gap);
        }
    }
}
```

TreeDemo.java

```
/*
 * NAME: OMAR ISMAIL ALOMORY
 * PROGRAM: SMSK (KMA) K2
 * COURSE: CSF3013
 * LAB: MP3
 * DATE: 13/12/2022
 */
package TreeExperiment;

public class TreeDemo{
    public static void main(String[] args) {
        Node<String> root = new Node<>("root");
        Node<String> node1 =root.addChild(new Node<String>("node1"));
        Node<String> node11 =node1.addChild(new Node<String>("node11"));
        Node<String> node111=node11.addChild(new Node<String>("node111"));
        Node<String> node112 =node11.addChild(new Node<String>("node112"));
        Node<String> node12 =node1.addChild(new Node<String>("node12"));
        Node<String> node2 =root.addChild(new Node<String>("node2"));
        Node<String> node21 =node2.addChild(new Node<String>("node21"));
        Node<String> node211 =node2.addChild(new Node<String>("node22"));
        Node<String> node212 =node2.addChild(new Node<String>("node23"));
        root.printTree(root , " ");
    }
}
```

QUESTIONS

1. List the basic operations in tree data structure.

   Answer:

   1. Insertion
   2. Deletion
   3. Searching
   4. Traversal (Inorder, Preorder, Postorder)
   5. Printing the Tree
   8. Printing Level Order Traversal of Tree

2. Where can we apply tree data structure in a software development?

   Answer:

   Database Indexing, File Systems, Algorithm Design, Graphs and Machine Learning etc.

3. Write algorithm for finding the root of tree from any node.
   Answer:

   1. If there is no parent for the node, then it is the root.

   2. Else, set the current node to its parent.

   3. Repeat step 2 until there is no parent for the node.

TASK 3: DEVELOP A TREE WITH DOUBLE DATA TYPE

OBJECTIVE

During this activity, students will apply a general tree concept in a simple programming task. A student should understand how tree works.

ESTIMATED TIME

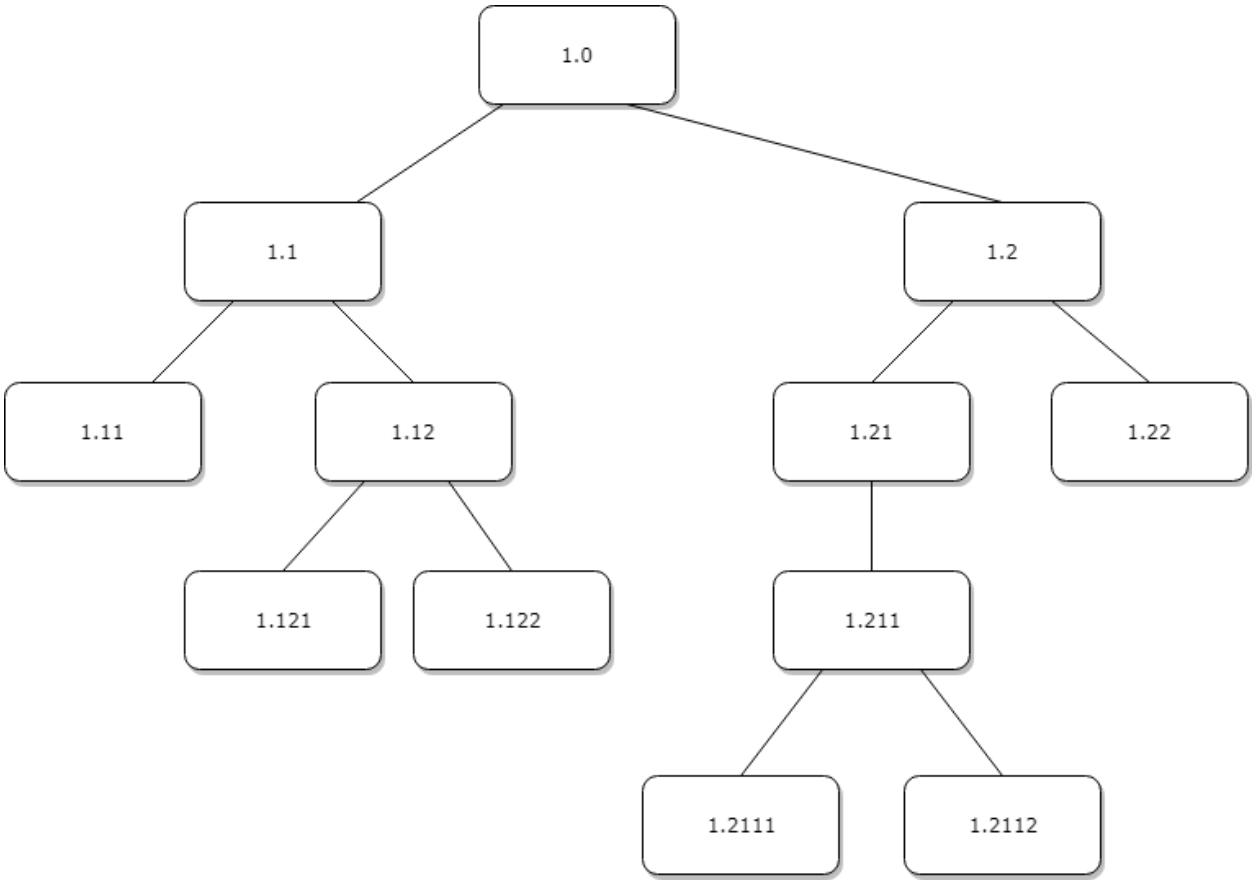
[30 Minutes]

INTRODUCTION

For this task, every node in a tree holds data with a double data type.

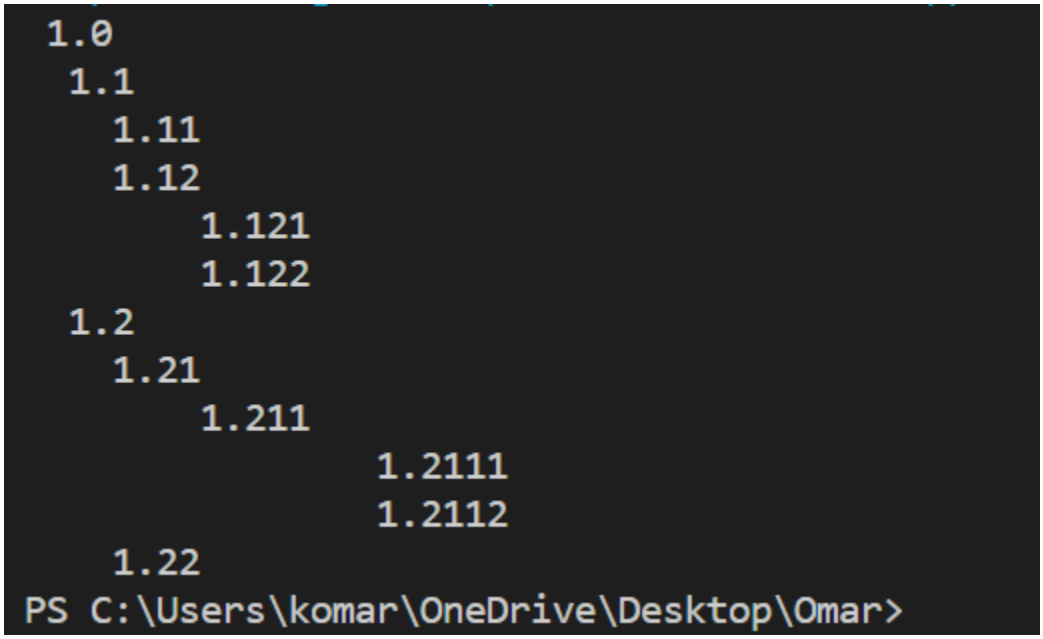
STEPS:

1. Open previously created `Netbeans` project.
2. Create a new class named `DoubleTypeTreeDemo`.
3. In `DoubleTypeTreeDemo.java`, create a test class for the following tree structure:



4. Save, compile and execute your codes.

5. Upload your output using the control box below:



6. Copy and paste your Java codes from NetBeans into text box below:
   Since we have the first class Node.java as generic class, we use it.
   Node.java

```java
/*
 * NAME: OMAR ISMAIL ALOMORY
 * PROGRAM: SMSK (KMA) K2
 * COURSE: CSF3013
 * LAB: MP3
 * DATE: 13/12/2022
 */
package TreeExperiment;

import java.util.ArrayList;
import java.util.List;

public class Node<T> {
    private T data =null;

    private List<Node<T>> children = new ArrayList<>();

    private Node<T> parent = null;

    public Node(T data){
        this.data= data;
    }
    public Node<T> addChild(Node<T> child){
```

```
        child.setParent(this);
        this.children.add(child);
        return child;
    }
    public void addChildren(List<Node<T>> children){
        children.forEach(each -> each.setParent(this));
        this.children.addAll(children);
    }
    public List<Node<T>> getChildren(){
        return children;
    }
    public T getData(){
        return this.data;
    }

    public void setData(T data){
        this.data = data;
    }
    private void setParent(Node<T> parent){
        this.parent =parent;
    }
    public Node<T> getParent(){
        return this.parent;
    }

    public void printTree(Node<T> root ,String gap){
        if(root == null){
            return;
        }

        System.out.println(gap+ root.data);
        for ( Node<T> child : root.children) {
            printTree(child, gap+gap);
        }
    }
}
```

TreeDemo.java

```java
/*
 * NAME: OMAR ISMAIL ALOMORY
 * PROGRAM: SMSK (KMA) K2
 * COURSE: CSF3013
 * LAB: MP3
 * DATE: 13/12/2022
 */
package TreeExperiment;

public class TreeDemo{
    public static void main(String[] args) {
        Node<Double> root = new Node<>(1.0);
        Node<Double> node1 =root.addChild(new Node<Double>(1.1));
        Node<Double> node11 =node1.addChild(new Node<Double>(1.11));
        Node<Double> node111=node1.addChild(new Node<Double>(1.12));
        Node<Double> node112 =node111.addChild(new Node<Double>(1.121));
        Node<Double> node12 =node111.addChild(new Node<Double>(1.122));
        Node<Double> node2 =root.addChild(new Node<Double>(1.2));
        Node<Double> node21 =node2.addChild(new Node<Double>(1.21));
        Node<Double> node211 =node21.addChild(new Node<Double>(1.211));
        Node<Double> node212 =node211.addChild(new Node<Double>(1.2111));
        Node<Double> node213 = node211.addChild( new Node<Double> (1.2112));
        Node<Double> node214 = node2.addChild(new Node<Double>(1.22));
        root.printTree(root , " ");
    }
}
```

QUESTIONS

1. Explain about 3 types tree traversals.

Answer:

1. Preorder Traversal: Preorder traversal is a type of tree traversal in which each node is visited before its children. The preorder traversal visits the root node first, then its left child, then its right child.

2. Inorder Traversal: Inorder traversal is a type of tree traversal in which each node is visited between its two children. The inorder traversal visits the left child first, then the root node, then the right child.

3. Postorder Traversal: Postorder traversal is a type of tree traversal in which each node is visited after its children. The postorder traversal visits the left child first, then the right child, then the root node.

2. What is a binary tree?

Answer:

A binary tree is a type of data structure that stores data in a hierarchical format, with each node having two child nodes (or branches). This structure allows for quick searching, insertion, and deletion of data.

3. How binary differs from a general tree?

Answer:

A binary tree is a type of tree in which each node has at most two children. In contrast, a general tree can have any number of children for each node. Binary trees are used for data storage and retrieval, whereas general trees are used to represent hierarchical structures.

Finally, read the instruction regarding submission carefully. Submit your answer using the link provided in Oceania UMT. Please ensure your codes are submitted to the correct group.