O-O implementation of tree traversals

Some considerations

- tree traversals allow computations on trees by recursively visiting their nodes
- in other words, a tree traversal implements some kind of operation on trees
- the visit of different types of nodes require different code

Few examples

- operations on Abstract Syntax Trees
  - returns the string corresponding to an expression in prefix notation
  - computes and returns the value of an expression
- operations on a file system
  - find files in (sub)folders
  - computes the total size of a folder

O-O implementation of tree traversals

Two different approaches to implement tree traversals

Object-oriented programming offers two different solutions to implement a tree traversal:

- data-oriented: object methods for visiting nodes are defined in every class implementing a type of node
- traversal-oriented: object methods for visiting nodes are defined in a single separate class

O-O implementation of tree traversals

A pictorial view

- in the data-oriented approach object methods are grouped by rows
- in the traversal-oriented approach object methods are grouped by columns

	traversal type 1	**traversal type 2**	**...**
node type 1	*object method 1*	*object method 2*	...
node type 2	*object method 3*	*object method 4*	...
...	...	...	...

Data-oriented approach

Pros

- new types of tree nodes can be added without code modification
- simpler solution and slightly more efficient solution

Cons

- the object methods for visiting nodes are scattered all over the classes implementing the different types of node
- defining new types of traversal requires modification of all the classes implementing the different types of node
- less general solution

Traversal-oriented approach

Pros

- new types of traversal can be implemented without code modification
- the object methods for visiting nodes are contained in the single class that implements the specific traversal
- more general solution

Cons

- adding new types of nodes requires modification of all the classes implementing the different types of traversals
- more complex solution and slightly less efficient code

Data-oriented approach

An example with file system trees with two types of nodes: `File` and `Folder`

Code

```
public interface FileSysTree {
    // traversal winch counts all files larger than minSize
    int countFilesLargerThan(int minSize);

    // other object methods could be added to implement other traversals
}

public class File implements FileSysTree { // nodes of type file
    private int size;

    public File(int size){
        if (size < 0)
            throw new IllegalArgumentException("File size cannot be negative");
        this.size = size;
    }
    public int countFilesLargerThan(int minSize){return size > minSize ? 1 : 0;}
}
```

Data-oriented approach

An example with file system trees with two types of nodes: `File` and `Folder`

Code

```
import java.util.LinkedList;
import java.util.List;
import static java.util.Objects.requireNonNull;

public class Folder implements FileSysTree { // nodes of type folder
    private final List<FileSysTree> children = new LinkedList<>();

    public Folder(FileSysTree... children) {
        for (var node : children) this.children.add(requireNonNull(node));
    }
    public int countFilesLargerThan(int minSize) {
        var res = 0;
        for (var node : children)
            res += node.countFilesLargerThan(minSize);
        return res;
    }
}
```

Data-oriented approach

An example with file system trees with two types of nodes: `File` and `Folder`

Code

```
public class Test {
    public static void main(String[] args) {
        var folder = new Folder(new File(10), new Folder(new File(2), new File
            (21)), new File(5), new File(42));
        assert folder.countFilesLargerThan(0) == 5;
        assert folder.countFilesLargerThan(20) == 2;
        assert folder.countFilesLargerThan(42) == 0;
        var f = new File(35);
        assert f.countFilesLargerThan(30) == 1;
        assert f.countFilesLargerThan(40) == 0;
    }
}
```

Traversal-oriented approach

Same example as before, but implemented with the visitor pattern

Code

```
public interface FileSysTree {
    // unique generic object method for any type of tree visit
    <T> T accept(Visitor<T> v);
}

import java.util.List;

public interface Visitor<T> {
    // an object method for each type of node
    T visitFile(int size);

    T visitFolder(List<FileSysTree> children);
}
```

Traversal-oriented approach

Same example as before, but implemented with the visitor pattern

Code

```
public class File implements FileSysTree {
    private int size;

    // constructor as before

    public <T> T accept(Visitor<T> v) { return v.visitFile(size); }
}

import java.util.LinkedList;
import java.util.List;

public class Folder implements FileSysTree {

    private final List<FileSysTree> children = new LinkedList<>();

    // constructor as before

    public <T> T accept(Visitor<T> v) { return v.visitFolder(children); }
}
```

Traversal-oriented approach

Same example as before, but implemented with the visitor pattern

Code

```
import java.util.List;

public class CountFilesLargerThan implements Visitor<Integer> {
    private final int minSize;

    public CountFilesLargerThan(int minSize) {this.minSize = minSize;}

    public Integer visitFile(int size) {return size > minSize ? 1 : 0;}

    public Integer visitFolder(List<FileSysTree> children) {
        var res = 0;
        for (var node : children)
            res += node.accept(this);
        return res;
    }
}
```

Traversal-oriented approach

Same example as before, but implemented with the visitor pattern

Code

```
public class Test {
    public static void main(String[] args) {
        var folder = new Folder(new File(10), new Folder(new File(2), new File
            (21)), new File(5), new File(42));
        assert folder.accept(new CountFilesLargerThan(0)) == 5;
        assert folder.accept(new CountFilesLargerThan(20)) == 2;
        assert folder.accept(new CountFilesLargerThan(42)) == 0;
        var f = new File(35);
        assert f.accept(new CountFilesLargerThan(30)) == 1;
        assert f.accept(new CountFilesLargerThan(40)) == 0;
    }
}
```

Traversal-oriented approach

Demo with the AST for a simple language of expressions

```
Exp ::= INTLIT | '-' Exp | Exp '*' Exp
```

Operations we would like to implement on ASTs

- returns the string corresponding to the expression in prefix notation
- returns the value of the expression

Remark: both operations require a traversal of the ASTs