

SWEN 423 Mock Term Test 2

Released 8am Friday 18 September Due 8am Saturday 19 August

Total 100 points

Tree data structure

Similarly to the `Stack<T>` and the `BinaryTree<T>` that we have seen in the lectures, you shall now make a tree where nodes contains either a label of type `T` or a non empty sequence of children.

You must submit code formatted in a similar way to what is shown on the slides.

The code must be correct and as compact as possible.

Correct but not compact code will not receive full marks.

Your code must rely on pure object oriented features as much as possible, as shown in the `Stack`.

Your code must compile without warnings in Java 14 (with preview features enabled)

To this aim you can use `@SuppressWarnings("unchecked")` when strictly needed.

We suggests that you run and test all your code extensively before submitting it.

In the required tree,

- Every node have either a label of type `T` or a non empty sequence of children, but not both
- Serialization/deserialization are supported as shown in the lecture.
- No mutation is allowed. Some operations will instead create new trees.
- The flyweight pattern is correctly implemented
- A proper match is supported

The Tree should offer (at least) those specific methods:

- `label`: returns the label of the node.
- `children`: returns a sequence of all the children of the node. If the node have no children, the empty sequence is returned.
- `withChild`: returns a tree that looks like the current one, but adds a new child at the end of the sequence of children. If the node had a label, such label is discarded.
- `toString`: produces a good `String` representation of the tree.

Method `'label'` may be unable to produce a result; similarly to `Stack.top` and `Stack.pop`. Design this method in the best way, according to the lectures of this course.

Finally, the code must be safe to use in a library setting; the properties of flyweight and immutability can not be broken by library users.

Tasks:

Q1[50 marks]

Provide the full code of the tree
up to 40 marks if you do not cover serialization
The code of the Tree must be in the file Tree.Java
If you reuse any code presented in the lectures, please include such code too.
Your code must compile correctly in Java 14 (with preview features enabled)

Q2[15 marks]

Write a full description of your code using correct terminology, and mentioning all the used patterns. In the next page you can find an example of the correct terminology reading the code of Stack<T> presented in the lectures.

This will be marked focusing on the use of correct terminology, and on how many patterns has been identified.

The text must be in the file TreeText.txt

Q3[15 marks]

Provide Unit Testing code testing the tree

We expect at least 6 JUnit tests creating different trees and trying all the different operations.

The code must be in the file TreeTest.Java

If you discover bugs that you are not able to fix, please, include tests for those, they will be considered favorably. The worst is if we think that there is some bug you are unaware of.

Q4[10 marks]

Using match, implement an operation “collectLabels(Tree<T> tree)”, returning a sequence containing all labels present in the tree.

The code must be in the file CollectLabels.Java

If you can not fully complete the code of the tree, you can still try to implement this operation using match. If the operation in itself would be correct, you will receive marks for it.

Q5[10 marks]

Using match, implement an operation “Tree<Integer> twice(Tree<Integer> tree){..}” producing a new tree, where all the Integer labels hold the double of the value.

The code must be in the file TreeOpTwice.Java

If you can not fully complete the code of the tree, you can still try to implement this operation using match. If the operation in itself would be correct, you will receive marks for it.

Correct description of Stack<T>, so that it could be read out loud.

```
public class Stack<T> {
```

We define the generic class Stack of T. It has the following members:

```
    private Stack(){}
```

A private empty no-arg constructor, this supports the sealed hierarchy pattern.

```
    private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
```

Cache: A private field of type Map from T to Stack of T.

This field is used to implement the flyweight pattern.

```
    public interface OnEmpty<R>{R of();}
```

```
    public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
```

Two nested generic interfaces; OnEmpty of result, and OnElem of T and result.

The method OnEmpty dot of takes no argument and produces a result.

It is similar to the Supplier as defined in the standard library

The method OnElem dot of takes two arguments: the top and the tail of a stack.

```
    public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
```

Match: A generic result method with two arguments, of type OnEmpty and OnElem.

The method body delegates to the result of the OnEmpty argument.

```
    public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
```

Push: a method implementing the flyweight pattern using the field cache and delegating to the underscore push method.

```
    private Stack<T> _push(T e){return _push(this,e);}
```

```
    private static <T> Stack<T> _push(Stack<T> self,T e){
```

```
        return new Stack<T>() {
```

```
            public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return b.of(e,self);}
```

```
        };}
```

Two underscore push private methods; they take care of generating a stack with elements.

A new Stack object is allocated, where the method match is overridden to delegate the result on the OnElem parameter. OnElem dot of receives the pushed element as the new top of the stack, and the outer stack object as the tail.

```
    private static Stack<?> empty = new Stack<>();
```

```
    @SuppressWarnings("unchecked")
```

```
    public static <T> Stack<T> empty()return (Stack<T>)empty;
```

empty: a static no-arg method returning a singleton instance of any Stack type.

The singleton object is stored in the private static field empty.

This shows us that all of the places where Stack objects are allocated are properly abstracted away from the user. Thus the flyweight pattern is properly implemented.

```
    private String toStrAux()return match(
```

```
        () -> "]",
```

```
        (e, t) -> e + (t.match(()->"",(a,b)->">") + t.toStrAux());
```

```
    }
```

```
    public boolean isEmpty()return match(()->true, (e,t)->false);}
```

IsEmpty: a method using match to discover if the stack is empty or not.

This works thanks to dynamic dispatch; the empty stack will delegate the behavior to OnEmpty, returning true; while instances of the anonymous nested class defined in the method underscore push will delegate to OnElem, returning false.

```
    public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
```

Top: a method using the second argument of match to extract the element at the top of the stack. If the stack is empty, the behavior is delegated to the method parameter.

```
    public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
```

Pop: similar to top, pop is a method using the second argument of match to extract the tail of the stack. If the stack is empty, the behavior is delegated to the method parameter.

```
    public String toString()return "[" + toStrAux();}
```

ToString: a method overriding the default implementation from object, printing the content of the stack between square brackets.

It delegates the bulk of the behavior to the private recursive method toStrAux, that in turn relies on match to compose the result; an inner match is used to correctly generate the separator characters.