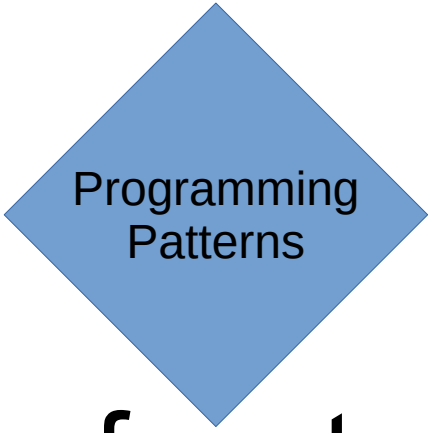
SWEN423 - Lecture 10



Many patterns for stack
Many patterns for Html Nodes

Stack

- Many patterns
- Fully (shallow) immutable
- Match/visitor: easier to define new operations
- Flyweight: best == hasCode + memory efficiency?
- Exceptions/default results delegate to the caller
- Real OO code!

Stack from lecture 9

```
public class Stack<T>{
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();//one for each stack object
  public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
    return onEmpty.get();
    }
  public Stack<T> push(T elem){return cache.computeIfAbsent(elem,this::newPush);}
  private Stack<T> newPush(T elem){//same implementation as before, but in a private method
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);
      }};}
  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("unchecked")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  public <R> R match(Supplier<R> onEmpty,Function<T,R>onLast,BiFunction<T,Stack<T>,R>onElem){
    return match(onEmpty,(e,t)->t.isEmpty()?onLast.apply(e):onElem.apply(e,t));
    }
  private String toStrAux(){return match(()->"]",e->e+"]",(e,t)->e+"; "+t.toStrAux());}
  public String toString(){return "["+toStrAux();}
  public boolean isEmpty(){return match(()->true,(e,t)->false);}
  public T topOrElse(Supplier<T> s){return match(s,(e,t)->e);}
  public Stack<T> popOrElse(Supplier<Stack<T>> s){return match(s,(e,t)->t);}
  //note: no need of equals/hashcode any more!! The identity is now the right one!
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  private Stack<T> _push(T e){return (a,b)->b.of(e,this);}// it would be so simple!
  //if only generic method lambdas was allowed, we could just do this!

  //instead of all of this!
  ##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>() {//looks like a lambda since we only implement one method!
  ##    public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return b.of(e,self);}
  ##  };}
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Another way to write the Stack

```
public class Stack<T> {
  private Stack(){}
  private WeakHashMap<T, Stack<T>> cache = new WeakHashMap<>();
  public interface OnEmpty<R>{R of();}
  public interface OnElem<T,R>{R of(T top,Stack<T>tail);}
  public <R> R match(OnEmpty<R> a, OnElem<T,R> b){return a.of();}
  public Stack<T> push(T e){return cache.computeIfAbsent(e,this::_push);}
  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);}
    };}
  private static Stack<?> empty = new Stack<>();
  @SuppressWarnings("unchecked")
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private String toStrAux(){return match(
    () -> "]",
    (e, t) -> e + (t.match(()->"",(a,b)->"; ") + t.toStrAux()));
    }
  public String toString(){return "[" + toStrAux();}
  public boolean isEmpty(){return match(()->true, (e,t)->false);}
  public T top(OnEmpty<T> a){return match(a, (e,t)->e);}
  public Stack<T> pop(OnEmpty<Stack<T>> a){return match(a, (e,t)->t);}
  }
```

Stack, where to use it?

- This stack is not indented to be a replacement for all your ArrayLists<T>
- Works the best when also the elements are deeply immutable or even flyweights
- Great when you need to transform data while keeping the old version

WeakHashMap

- WeakHashMap is just one of the options;
  it may not be the right one for your application.
- Consider using Google Guava Interner / Cache:
  - com.google.common.collect.Interners.newWeakInterner()
  - com.google.common.cache.CacheBuilder.newBuilder().softValues()
- You should also learn the detailed behaviour of
  - String.intern(String)
  - PhantomReference
  - SoftReference
  - WeakReference

Can we flyweight everything?

- Stacks have a very simple structure, so we can afford to make each node into a cache for the possible next nodes.
- It is possible to build up any structure in this way, by making new object by using operations that takes up only a single extra parameter.
- Next we show instead how to keep all the caches as static variables. Both ways are useful patterns.

Can we flyweight Node?

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  public static class Head extends Node{...}
  public static class P extends Node{...}
  public static class H1 extends Node{...}
  public static class Li extends Node{...}
  public static class Ul extends Node{...}
  public static class Ol extends Node{...}
  public static class Div extends Node{...}
  public static class Body extends Node{...}
  public static class Html extends Node{...}
  public static class A extends Node{...}
  }
```

```
interface Visitor<T>{
  T visitP(Node.P e);
  T visitH1(Node.H1 e);
  T visitHtml(Node.Html e);
  T visitHead(Node.Head e);
  T visitBody(Node.Body e);
  T visitDiv(Node.Div e);
  T visitA(Node.A e);
  T visitUl(Node.Ul e);
  T visitOl(Node.Ol e);
  T visitLi(Node.Li e);
  }
```

Can we flyweight Node?

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  public static class Head extends Node{
    private static final Head instance=new Head();
    private Head() {}
    public static Head of() {return instance;}
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}
    }
  public static class P extends Node{...}
  public static class H1 extends Node{...}
  public static class Li extends Node{...}
  public static class Ul extends Node{...}
  public static class Ol extends Node{...}
  public static class Div extends Node{...}
  public static class Body extends Node{...}
  public static class Html extends Node{...}
  public static class A extends Node{...}
  }
```

Can we flyweight Node?

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  public static class Head extends Node{...}
  public static class P extends Node{
    private static WeakHashMap<String,P> cache = new WeakHashMap<>();
    public final String text;
    private P(String text){this.text=text;}
    public static P of(String text){return cache.computeIfAbsent(text,P::new);}
    public <T> T accept(Visitor<T> v){return v.visitP(this);}
    }
  public static class H1 extends Node{
    private static WeakHashMap<String,H1> cache = new WeakHashMap<>();
    public final String text;
    private H1(String text){this.text=text;}
    public static H1 of(String text){return cache.computeIfAbsent(text,H1::new);}
    public <T> T accept(Visitor<T> v){return v.visitH1(this);}
    }
  public static class Li extends Node {...}   ...}
```

Private constructor+'of' method: creation is under control

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}
  /* ... Head, P, H1 ... */
  public static class Li extends Node {
    private static WeakHashMap<Node,Li> cache = new WeakHashMap<>();
    public final Node node;
    private Li(Node node){this.node=node;}
    public static Li of(Node node){return cache.computeIfAbsent(node,Li::new);}
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}
    }
  public static class Ul extends Node{
    private static WeakHashMap<Stack<Li>,Ul> cache = new WeakHashMap<>();
    public final Stack<Li> lis;
    private Ul(Stack<Li> lis){this.lis=lis;}
    public static Ul of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ul::new);}
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}
    }
  public static class Ol extends Node{
    private static WeakHashMap<Stack<Li>,Ol> cache = new WeakHashMap<>();
    public final Stack<Li> lis;
    private Ol(Stack<Li> lis){this.lis=lis;}
    public static Ol of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ol::new);}
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}
    }
  public static class Div extends Node{...}  ...}
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}
  /* ... Head, P, H1 ... */
  public static class Li extends Node {
    private static WeakHashMap<Node,Li> cache = new WeakHashMap<>();
    public final Node node;
    private Li(Node node){this.node=node;}
    public static Li of(Node node){return cache.computeIfAbsent(node,Li::new);}
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}
    }
  public static class Ul extends Node{
    private static WeakHashMap<Stack<Li>,Ul> cache = new WeakHashMap<>();
    public final Stack<Li> lis;
    private Ul(Stack<Li> lis){this.lis=lis;}
    public static Ul of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ul::new);}
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}
    }
  public static class Ol extends Node{
    private static WeakHashMap<Stack<Li>,Ol> cache = new WeakHashMap<>();
    public final Stack<Li> lis;
    private Ol(Stack<Li> lis){this.lis=lis;}
    public static Ol of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ol::new);}
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}
    }
  public static class Div extends Node{...}  ...}
```

```java
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  /* ... Head, P, H1, Li, Ul, OL ... */
  public static class Div extends Node{
    private static WeakHashMap<Stack<Node>,Div> cache = new WeakHashMap<>();
    public final Stack<Node> ns;
    private Div(Stack<Node> ns){this.ns=ns;}
    public static Div of(Stack<Node> ns){return cache.computeIfAbsent(ns,Div::new);}
    public <T> T accept(Visitor<T> v){return v.visitDiv(this);}
    }
  public static class Body extends Node{
    private static WeakHashMap<Stack<Div>,Body> cache = new WeakHashMap<>();
    public final Stack<Div> divs;
    private Body(Stack<Div> divs){this.divs=divs;}
    public static Body of(Stack<Div> divs){return cache.computeIfAbsent(divs,Body::new);}
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}
    }
  public static class Html extends Node{...}
  ...}
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}
  /* ... Head, P, H1, Li, Ul, OL, Div, Body ... */
  public static class Html extends Node{
    static record State(Head head, Body body){}
    private static WeakHashMap<State,Html> cache = new WeakHashMap<>();
    public final State state;
    private Html(State state){this.state=state;}
    public static Html of(Head head, Body body){return of(new State(head,body));}
    public static Html of(State state){return cache.computeIfAbsent(state,Html::new);}
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}
    }
  public static class A extends Node{
    static record State(String href, String text){}
    private static WeakHashMap<State,A> cache = new WeakHashMap<>();
    public final State state;
    private A(State state){this.state=state;}
    public static A of(String href, String text){return of(new State(href,text));}
    public static A of(State state){return cache.computeIfAbsent(state,A::new);}
    public <T> T accept(Visitor<T> v){return v.visitA(this);}
    }
  }
```

```
abstract class Node{  public abstract <T> T accept(Visitor<T> v);  private Node(){}
  public static class Head extends Node{    private static final Head instance=new Head();
    private Head() {}    public static Head of() {return instance;}
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}    }
  public static class P extends Node{    private static WeakHashMap<String,P> cache = new WeakHashMap<>();
    public final String text;    private P(String text){this.text=text;}
    public static P of(String text){return cache.computeIfAbsent(text,P::new);}
    public <T> T accept(Visitor<T> v){return v.visitP(this);}     }
  public static class H1 extends Node{    private static WeakHashMap<String,H1> cache = new WeakHashMap<>();
    public final String text;    private H1(String text){this.text=text;}
    public static H1 of(String text){return cache.computeIfAbsent(text,H1::new);}
    public <T> T accept(Visitor<T> v){return v.visitH1(this);}    }
  public static class Li extends Node {    private static WeakHashMap<Node,Li> cache = new WeakHashMap<>();
    public final Node node;    private Li(Node node){this.node=node;}
    public static Li of(Node node){return cache.computeIfAbsent(node,Li::new);}
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}    }
  public static class Ul extends Node{    private static WeakHashMap<Stack<Li>,Ul> cache = new WeakHashMap<>();
    public final Stack<Li> lis;    private Ul(Stack<Li> lis){this.lis=lis;}
    public static Ul of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ul::new);}
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}    }
  public static class Ol extends Node{    private static WeakHashMap<Stack<Li>,Ol> cache = new WeakHashMap<>();
    public final Stack<Li> lis;    private Ol(Stack<Li> lis){this.lis=lis;}
    public static Ol of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ol::new);}
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}    }
  public static class Div extends Node{    private static WeakHashMap<Stack<Node>,Div> cache = new WeakHashMap<>();
    public final Stack<Node> ns;    private Div(Stack<Node> ns){this.ns=ns;}
    public static Div of(Stack<Node> ns){return cache.computeIfAbsent(ns,Div::new);}
    public <T> T accept(Visitor<T> v){return v.visitDiv(this);}     }
  public static class Body extends Node{    private static WeakHashMap<Stack<Div>,Body> cache = new WeakHashMap<>();
    public final Stack<Div> divs;    private Body(Stack<Div> divs){this.divs=divs;}
    public static Body of(Stack<Div> divs){return cache.computeIfAbsent(divs,Body::new);}
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}    }
  public static class Html extends Node{    static record State(Head head, Body body){}
    private static WeakHashMap<State,Html> cache = new WeakHashMap<>();
    public final State state;    private Html(State state){this.state=state;}
    public static Html of(Head head, Body body){return of(new State(head,body));}
    public static Html of(State state){return cache.computeIfAbsent(state,Html::new);}
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}    }
  public static class A extends Node{    static record State(String href, String text){}
    private static WeakHashMap<State,A> cache = new WeakHashMap<>();
    public final State state;    private A(State state){this.state=state;}
    public static A of(String href, String text){return of(new State(href,text));}
    public static A of(State state){return cache.computeIfAbsent(state,A::new);}
    public <T> T accept(Visitor<T> v){return v.visitA(this);}    }}
```

```
abstract class Node{  public abstract <T> T accept(Visitor<T> v);  private Node(){}
  public static class Head extends Node{    private static final Head instance=new Head();
    private Head() {}    public static Head of() {return instance;}
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}    }
  public static class P extends Node{    private static WeakHashMap<String,P> cache = new WeakHashMap<>();
    public final String text;    private P(String text){this.text=text;}
    public static P of(String text){return cache.computeIfAbsent(text,P::new);}
    public <T> T accept(Visitor<T> v){return v.visitP(this);}     }
  public static class H1 extends Node{    private static WeakHashMap<String,H1> cache = new WeakHashMap<>();
    public final String text;    private H1(String text){this.text=text;}
    public static H1 of(String text){return cache.computeIfAbsent(text,H1::new);}
    public <T> T accept(Visitor<T> v){return v.visitH1(this);}    }
  public static class Li extends Node {    private static WeakHashMap<Node,Li> cache = new WeakHashMap<>();
    public final Node node;    private Li(Node node){this.node=node;}
    public static Li of(Node node){return cache.computeIfAbsent(node,Li::new);}
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}    }
  public static class Ul extends Node{    private static WeakHashMap<Stack<Li>,Ul> cache = new WeakHashMap<>();
    public final Stack<Li> lis;    private Ul(Stack<Li> lis){this.lis=lis;}
    public static Ul of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ul::new);}
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}    }
  public static class Ol extends Node{    private static WeakHashMap<Stack<Li>,Ol> cache = new WeakHashMap<>();
    public final Stack<Li> lis;    private Ol(Stack<Li> lis){this.lis=lis;}
    public static Ol of(Stack<Li> lis){return cache.computeIfAbsent(lis,Ol::new);}
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}    }
  public static class Div extends Node{    private static WeakHashMap<Stack<Node>,Div> cache = new WeakHashMap<>();
    public final Stack<Node> ns;    private Div(Stack<Node> ns){this.ns=ns;}
    public static Div of(Stack<Node> ns){return cache.computeIfAbsent(ns,Div::new);}
    public <T> T accept(Visitor<T> v){return v.visitDiv(this);}     }
  public static class Body extends Node{    private static WeakHashMap<Stack<Div>,Body> cache = new WeakHashMap<>();
    public final Stack<Div> divs;    private Body(Stack<Div> divs){this.divs=divs;}
    public static Body of(Stack<Div> divs){return cache.computeIfAbsent(divs,Body::new);}
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}    }
  public static class Html extends Node{    static record State(Head head, Body body){}
    private static WeakHashMap<State,Html> cache = new WeakHashMap<>();
    public final State state;    private Html(State state){this.state=state;}
    public static Html of(Head head, Body body){return of(new State(head,body));}
    public static Html of(State state){return cache.computeIfAbsent(state,Html::new);}
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}    }
  public static class A extends Node{    static record State(String href, String text){}
    private static WeakHashMap<State,A> cache = new WeakHashMap<>();
    public final State state;    private A(State state){this.state=state;}
    public static A of(String href, String text){return of(new State(href,text));}
    public static A of(State state){return cache.computeIfAbsent(state,A::new);}
    public <T> T accept(Visitor<T> v){return v.visitA(this);}    }}
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);  private Node(){}

  public static class Head extends Node{
    private static final Head instance=new Head();
    private Head() {}    public static Head of() {return instance;}
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}
    }
  public static class P extends Node{
    private static WeakHashMap<String,P> cache = new WeakHashMap<>();
    public final String text;    private P(String text){this.text=text;}
    public static P of(String text){return cache.computeIfAbsent(text,P::new);}
    public <T> T accept(Visitor<T> v){return v.visitP(this);}
    }...
public static class Body extends Node{
    private static WeakHashMap<Stack<Div>,Body> cache = new WeakHashMap<>();
    public final Stack<Div> divs;    private Body(Stack<Div> divs){this.divs=divs;}
    public static Body of(Stack<Div> divs){return cache.computeIfAbsent(divs,Body::new);}
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}
    }
  public static class Html extends Node{
    static record State(Head head, Body body){}
    private static WeakHashMap<State,Html> cache = new WeakHashMap<>();
    public final State state;    private Html(State state){this.state=state;}
    public static Html of(Head head, Body body){return of(new State(head,body));}
    public static Html of(State state){return cache.computeIfAbsent(state,Html::new);}
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}    }...}
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  Head{accept visitHead
    private static final Head instance=new Head();
    Head of() {return instance;}
    }
  P{String text;    accept visitP
    <String,P> cache
    P of(String text){return cache.computeIfAbsent(text,P::new);}
    }
..
  Body{Stack<Div> divs;    accept visitBody
    <Stack<Div>,Body> cache
    Body of(Stack<Div> divs){return cache.computeIfAbsent(divs,Body::new);}
  }
  Html{State state;    accept visitHtml
    static record State(Head head, Body body){}
    <State,Html> cache
    Html of(Head head, Body body){return of(new State(head,body));}
    Html of(State state){return cache.computeIfAbsent(state,Html::new);}
    }
...}
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  Head{accept visitHead
    private static final Head instance=new Head();
    Head of() {return instance;}
    }
  P{String text;    accept visitP
    <String,P> cache
    P of(String text){return cache.computeIfAbsent(text,P::new);}
    }
..
  Body{Stack<Div> divs;    accept visitBody
    <Stack<Div>,Body> cache
    Body of(Stack<Div> divs){return cache.computeIfAbsent(divs,Body::new);}
  }
  Html{State state;    accept visitHtml
    static record State(Head head, Body body){}
    <State,Html> cache
    Html of(Head head, Body body){return of(new State(head,body));}
    Html of(State state){return cache.computeIfAbsent(state,Html::new);}
    }
...}
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  Head{Head of() {return instance;}
    private static final Head instance=new Head();}
  P{String text;  P of(_) from cache    }
..
  Body{Stack<Div> divs;  Body of(_) from cache  }
  Html{State state;  Html of(_) from cache
    static record State(Head head, Body body){}
    Html of(Head head, Body body){return of(new State(head,body));}
    }
...}
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}

  Head{Head of() {return instance;}
    private static final Head instance=new Head();}
  P{String text;  P of(_) from cache  }
  H1{String text;  H1 of(_) from cache  }
  Li{Node node;  Li of(_) from cache  }
  Ul{Stack<Li> lis;  Ul of(_) from cache  }
  Ol{Stack<Li> lis;  Ol of(_) from cache  }
  Div{Stack<Node> ns;  Div of(_) form cache  }
  Body{Stack<Div> divs;  Body of(_) from cache  }
  Html{State state;  Html of(_) from cache
    static record State(Head head, Body body){}
    Html of(Head head, Body body){return of(new State(head,body));}
    }
  A{State state; A(_) from cache
    static record State(String href, String text){}
    A of(String href, String text){return of(new State(href,text));}
    }
  }
```

```
abstract class Node{//Composite Pattern
  public abstract <T> T accept(Visitor<T> v);//Visitor Pattern
  private Node(){}//Sealed class pattern

  Head{Head of() {return instance;} //Singleton pattern
    private static final Head instance=new Head();}
  P{String text;  P of(_) from cache  }//flyweight pattern
  H1{String text;  H1 of(_) from cache  }
  Li{Node node;  Li of(_) from cache  }
  Ul{Stack<Li> lis;  Ul of(_) from cache  }
  Ol{Stack<Li> lis;  Ol of(_) from cache  }
  Div{Stack<Node> ns;  Div of(_) form cache  }
  Body{Stack<Div> divs;  Body of(_) from cache  }
  Html{State state;  Html of(_) from cache
    static record State(Head head, Body body){}//state for flyweight pattern
    Html of(Head head, Body body){return of(new State(head,body));}//factory
    }
  A{State state; A(_) from cache
    static record State(String href, String text){}
    A of(String href, String text){return of(new State(href,text));}
    }
  }
```

A Eagle eye view of the code

- Still, you can read it as the much longer Java.
- All/Most detail are implicitly required by the patterns.

- May be a macro pre-processor may help to write more compact Java code? Is M4 the right macro expander for this task?

Macros in M4

```
define(flyweight,`
  public static class $1 extends Node{
    private static WeakHashMap<$2,$1> cache = new WeakHashMap<>();
    public final $2 $3;
    private $1($2 $3){this.$3=$3;}
    public static $1 of($2 $3){
      return cache.computeIfAbsent($3,$1::new);}
    public <T> T accept(Visitor<T> v){return v.visit$1(this);}
    $4
  }
')
define(singleton,`
  public static class $1 extends Node{
    private static final $1 instance=new $1();
    private $1(){}
    public static $1 of(){return instance;}
    public <T> T accept(Visitor<T> v){return v.visit$1(this);}
    $4
  }
')
```

```
abstract class Node{
  public abstract <T> T accept(Visitor<T> v);
  private Node(){}
define(flyweight,`...')
define(singleton,`...')
singleton(Head)
flyweight(P,String,text)
flyweight(H1,String,text)
flyweight(Li,Node,node)
flyweight(Ul,Stack<Li>,lis)
flyweight(Ol,Stack<Li>,lis)
flyweight(Div,Stack<Node>,ns)
flyweight(Body,Stack<Div>,divs)
flyweight(Html,State,statecache,`
    static record State(Head head, Body body){}
    Html of(Head head, Body body){return of(new State(head,body));}
')
flyweight(A,State,state,`
    static record State(String href, String text){}
    A of(String href, String text){return of(new State(href,text));}
')
undefine(flyweight,singleton)
}
```

Memoization / Dynamic programming

- Flyweights allows for efficient caching of computations too.

- This technique was originally called “dynamic programming” by two USA military engineers that was searching of a term that “felt like had no drawbacks” to not displease their general. It have no connection with what it is doing.

Memoization / Dynamic programming

```
class HtmlValidator{

  private static WeakHashMap<Node,String> cache = new
WeakHashMap<>();

  public static String allErrors(Node n){
    //fast cache check even for massive htmls, using pointer ==
    return cache.computeIfAbsent(n,node->_allErrors(node));
    }
  private static String _allErrors(Node n){
    //massive code doing intensive computation
    return ...;
    }
  }
```