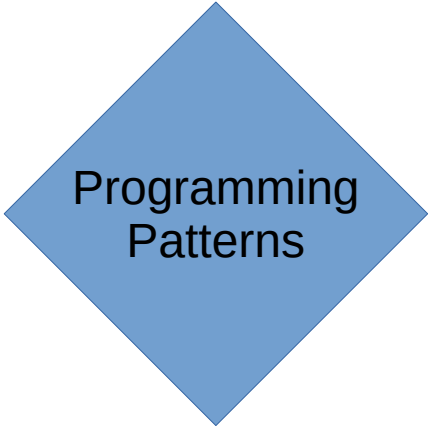
SWEN423 - Lecture 9

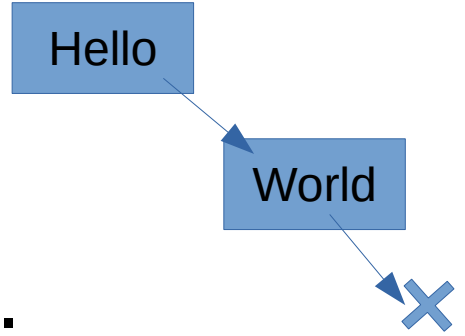


Stack: a collection of patterns

How to NOT implement a Stack

```
public class Stack<T>{
  public T elem;
  public Stack<T> tail;
  public Stack(T elem,Stack<T> tail){this.elem=elem;this.tail=tail;}
  }
...
public static void user(){
  var ss=new Stack<String>("Hello",new Stack<String>("World",null));
  }
```

- Counting the anti-patterns:
  - null, public fields, no methods
- This is how we coded in C in the 70s.



Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Better Stack implementation

```
public class Stack<T>{
  public boolean isEmpty(){return true;}//base case

  public T top(){throw new UnsupportedOperationException();}//take control of errors
  public Stack<T> pop(){throw new UnsupportedOperationException();}

  public Stack<T> push(T elem){
    Stack<T> self=this;//explicit this naming
    return new Stack<T>(){//closure
      public boolean isEmpty(){return false;}//avoid instanceof
      public T top(){return elem;}//is elem a field?
      public Stack<T> pop(){return self;}//is self a field?
      };
    }

  private static Stack<Object> empty=new Stack<Object>();//null object pattern
  @SuppressWarnings("unchecked")//sadly needed
  public static <T> Stack<T> empty(){return (Stack<T>)empty;}//generic singleton pattern

  private Stack(){}//private constructor == sealed class
  //please add toString,equals and hashCode too!
  }
//can a user define a usable MyStack extends Stack?
```

Counting patterns (9?):

- singleton pattern
- null object pattern
- sealed class/sealed subtyping
- explicit this naming (micropattern)
- boolean methods vs instanceof (micropattern)
- hiding concrete types (micropattern)
- encapsulation, take control of errors (micropatterns)
- closures instead of fields (micropattern)

Are we forgetting any pattern?

Are we forgetting any pattern?

- The Stack uses the COMPOSITE pattern!
- The type Stack is a component
- The empty stack is a leave,
- The anonymous closure is a composite.

- Now we can also apply the visitor to the Stack!

A Better Stack

```
public class Stack<T>{
  public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
    return onEmpty.get();
    }
  .. isEmpty .. top .. pop ..//as before
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);
        }
       .. isEmpty .. top .. pop ..//as before
      };
    }
  ..empty.. empty().. private Stack(){} .. //as before

  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){
    return (o instanceof Stack<?>) && eq((Stack<?>)o);
    }
  public boolean eq(Stack<?> o){
    return match(
      ()->o.isEmpty(),
      (e1,t1)->!o.isEmpty() && e1.equals(o.top())&&t1.eq(o.pop()));
    }  }
```

A Better Stack

```
public class Stack<T>{
  public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
    return onEmpty.get();
    }
  .. isEmpty .. top .. pop ..//as before
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);
        }
       .. isEmpty .. top .. pop ..//as before
      };
    }
  ..empty.. empty().. private Stack(){} .. //as before

  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){
    return (o instanceof Stack<?>) && eq((Stack<?>)o);
    }
  public boolean eq(Stack<?> o){
    return match(
      ()->o.isEmpty(),
      (e1,t1)->!o.isEmpty() && e1.equals(o.top())&&t1.eq(o.pop()));
    }  }
```

A Better Stack

```
public class Stack<T>{
  public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
    return onEmpty.get();
    }
  .. isEmpty .. top .. pop ..//as before
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);
        }
       .. isEmpty .. top .. pop ..//as before
      };
    }
  ..empty.. empty().. private Stack(){} .. //as before

  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){
    return (o instanceof Stack<?>) && eq((Stack<?>)o);
    }
  public boolean eq(Stack<?> o){
    return match(
      ()->o.isEmpty(),
      (e1,t1)->!o.isEmpty() && e1.equals(o.top())&&t1.eq(o.pop()));
    }  }
```

A Better Stack

```
public class Stack<T>{
  public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
    return onEmpty.get();
    }
  .. isEmpty .. top .. pop ..//as before
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);
        }
       .. isEmpty .. top .. pop ..//as before
      };
    }
  ..empty.. empty().. private Stack(){} .. //as before

  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){
    return (o instanceof Stack<?>) && eq((Stack<?>)o);
    }
  public boolean eq(Stack<?> o){
    return match(
      ()->o.isEmpty(),
      (e1,t1)->!o.isEmpty() && e1.equals(o.top())&&t1.eq(o.pop()));
    }  }
```

A Better Stack

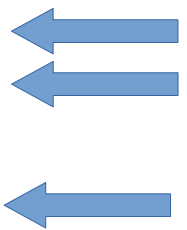
```
public class Stack<T>{
  public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
    return onEmpty.get();
    }
  .. isEmpty .. top .. pop ..//as before
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);
        }
       .. isEmpty .. top .. pop ..//as before
      };
    }
  ..empty.. empty().. private Stack(){} .. //as before

  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){
    return (o instanceof Stack<?>) && eq((Stack<?>)o);
    }
  public boolean eq(Stack<?> o){
    return match(
      ()->o.isEmpty(),
      (e1,t1)->!o.isEmpty() && e1.equals(o.top())&&t1.eq(o.pop()));
    }  }
```

A Better Stack

```
public class Stack<T>{
  public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
    return onEmpty.get();
    }
  .. isEmpty .. top .. pop ..//as before
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);
        }
       .. isEmpty .. top .. pop ..//as before
      };
    }
  ..empty.. empty().. private Stack(){} .. //as before

  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){
    return (o instanceof Stack<?>) && eq((Stack<?>)o);
    }
  public boolean eq(Stack<?> o){
    return match(
      ()->o.isEmpty(),
      (e1,t1)->!o.isEmpty() && e1.equals(o.top())&&t1.eq(o.pop()));
    }  }
```



isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

Focusing on toString

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

  <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();
    }
  ...
```

isEmpty, top and pop as derived

```
public class Stack<T>{
  public<R> R match(Supplier<R>onEmpty,BiFunction<T,Stack<T>,R>onElem){return onEmpty.get();}
  public Stack<T> push(T elem){
    Stack<T> self=this;
    return new Stack<T>(){//this is now very close to just be a lambda
      public <R> R match(Supplier<R> onEmpty,BiFunction<T,Stack<T>,R> onElem){
        return onElem.apply(elem,self);}};}
  public int hashCode(){return match(()->31,(e,t)->e.hashCode()+t.hashCode()*31);}
  public boolean equals(Object o){return (o instanceof Stack<?>) && eq((Stack<?>)o);}
  public boolean eq(Stack<?> o){return match(
    ()->o.match(()->true,(e,t)->false),
    (e1,t1)->o.match(()->false,(e2,t2)->e1.equals(e2)&&t1.eq(t2)));
    }
  public boolean isEmpty(){return match(()->false,(e,t)->true);}
  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);}

  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();}

  private static Stack<Object> empty=new Stack<Object>();//singleton pattern
  @SuppressWarnings("..")public static <T> Stack<T> empty(){return (Stack<T>)empty;}
  private Stack(){}  }
```

Example of external operation

```
abstract class StackWalk<A,B,C>{//This class implements the template method pattern
  public abstract C from(A a,B b);//abstract handler
  public abstract C fromA(A a);//abstract handler
  public abstract C fromB(B b); //abstract handler
  public Stack<C> of(Stack<A> as,Stack<B> bs){//This is the template method
    return as.match(
      ()->ofBs(bs),//case as is empty
      (a,at)->bs.match(
        ()->ofAs(as),//case bs is empty (but as is not)
        (b,bt)->of(at,bt).push(from(a,b))//case both not empty
        ));}
  private Stack<C> ofBs(Stack<B> bs){//what to do if as are empty
    return bs.match(Stack::empty,(e,t)->ofBs(t).push(fromB(e)));
    }
  private Stack<C> ofAs(Stack<A> as){//what to do if bs are empty
    return as.match(Stack::empty,(e,t)->ofAs(t).push(fromA(e)));
    }}
var as=Stack.<String>empty().push("x").push("y").push("z").push("w");
var bs=Stack.<Integer>empty().push(1).push(2).push(3);
var cs=new StackWalk<String,Integer,String>(){
  public String from(String a,Integer b){return a+b;}
  public String fromA(String a){return a;}
  public String fromB(Integer b){throw new Error();}
  }.of(as,bs);
System.out.println(cs);}//prints [w3; z2; y1; x]
```

Example of external operation

```
abstract class StackWalk<A,B,C>{//This class implements the template method pattern
  public abstract C from(A a,B b);//abstract handler
  public abstract C fromA(A a);//abstract handler
  public abstract C fromB(B b); //abstract handler
  public Stack<C> of(Stack<A> as,Stack<B> bs){//This is the template method
    return as.match(
      ()->ofBs(bs),//case as is empty
      (a,at)->bs.match(
        ()->ofAs(as),//case bs is empty (but as is not)
        (b,bt)->of(at,bt).push(from(a,b))//case both not empty
        ));}
  private Stack<C> ofBs(Stack<B> bs){//what to do if as are empty
    return bs.match(Stack::empty,(e,t)->ofBs(t).push(fromB(e)));
    }
  private Stack<C> ofAs(Stack<A> as){//what to do if bs are empty
    return as.match(Stack::empty,(e,t)->ofAs(t).push(fromA(e)));
    }}
var as=Stack.<String>empty().push("x").push("y").push("z").push("w");
var bs=Stack.<Integer>empty().push(1).push(2).push(3);
var cs=new StackWalk<String,Integer,String>(){
  public String from(String a,Integer b){return a+b;}
  public String fromA(String a){return a;}
  public String fromB(Integer b){throw new Error();}
  }.of(as,bs);
System.out.println(cs);}//prints [w3; z2; y1; x]
```

Example of external operation

```
abstract class StackWalk<A,B,C>{//This class implements the template method pattern
  public abstract C from(A a,B b);//abstract handler
  public abstract C fromA(A a);//abstract handler
  public abstract C fromB(B b); //abstract handler
  public Stack<C> of(Stack<A> as,Stack<B> bs){//This is the template method
    return as.match(
      ()->ofBs(bs),//case as is empty
      (a,at)->bs.match(
        ()->ofAs(as),//case bs is empty (but as is not)
        (b,bt)->of(at,bt).push(from(a,b))//case both not empty
        ));}
  private Stack<C> ofBs(Stack<B> bs){//what to do if as are empty
    return bs.match(Stack::empty,(e,t)->ofBs(t).push(fromB(e)));
    }
  private Stack<C> ofAs(Stack<A> as){//what to do if bs are empty
    return as.match(Stack::empty,(e,t)->ofAs(t).push(fromA(e)));
    }}
var as=Stack.<String>empty().push("x").push("y").push("z").push("w");
var bs=Stack.<Integer>empty().push(1).push(2).push(3);
var cs=new StackWalk<String,Integer,String>(){
  public String from(String a,Integer b){return a+b;}
  public String fromA(String a){return a;}
  public String fromB(Integer b){throw new Error();}
  }.of(as,bs);
System.out.println(cs);}//prints [w3; z2; y1; x]
```

Example of external operation

```
abstract class StackWalk<A,B,C>{//This class implements the template method pattern
  public abstract C from(A a,B b);//abstract handler
  public abstract C fromA(A a);//abstract handler
  public abstract C fromB(B b); //abstract handler
  public Stack<C> of(Stack<A> as,Stack<B> bs){//This is the template method
    return as.match(
      ()->ofBs(bs),//case as is empty
      (a,at)->bs.match(
        ()->ofAs(as),//case bs is empty (but as is not)
        (b,bt)->of(at,bt).push(from(a,b))//case both not empty
        ));}
  private Stack<C> ofBs(Stack<B> bs){//what to do if as are empty
    return bs.match(Stack::empty,(e,t)->ofBs(t).push(fromB(e)));
    }
  private Stack<C> ofAs(Stack<A> as){//what to do if bs are empty
    return as.match(Stack::empty,(e,t)->ofAs(t).push(fromA(e)));
    }}
var as=Stack.<String>empty().push("x").push("y").push("z").push("w");
var bs=Stack.<Integer>empty().push(1).push(2).push(3);
var cs=new StackWalk<String,Integer,String>(){
  public String from(String a,Integer b){return a+b;}
  public String fromA(String a){return a;}
  public String fromB(Integer b){throw new Error();}
  }.of(as,bs);
System.out.println(cs);}//prints [w3; z2; y1; x]
```

Example of external operation

```
abstract class StackWalk<A,B,C>{//This class implements the template method pattern
  public abstract C from(A a,B b);//abstract handler
  public abstract C fromA(A a);//abstract handler
  public abstract C fromB(B b); //abstract handler
  public Stack<C> of(Stack<A> as,Stack<B> bs){//This is the template method
    return as.match(
      ()->ofBs(bs),//case as is empty
      (a,at)->bs.match(
        ()->ofAs(as),//case bs is empty (but as is not)
        (b,bt)->of(at,bt).push(from(a,b))//case both not empty
        ));}
  private Stack<C> ofBs(Stack<B> bs){//what to do if as are empty
    return bs.match(Stack::empty,(e,t)->ofBs(t).push(fromB(e)));
    }
  private Stack<C> ofAs(Stack<A> as){//what to do if bs are empty
    return as.match(Stack::empty,(e,t)->ofAs(t).push(fromA(e)));
    }}
var as=Stack.<String>empty().push("x").push("y").push("z").push("w");
var bs=Stack.<Integer>empty().push(1).push(2).push(3);
var cs=new StackWalk<String,Integer,String>(){
  public String from(String a,Integer b){return a+b;}
  public String fromA(String a){return a;}
  public String fromB(Integer b){throw new Error();}
  }.of(as,bs);
System.out.println(cs);}//prints [w3; z2; y1; x]
```

Example of external operation

```
abstract class StackWalk<A,B,C>{//This class implements the template method pattern
  public abstract C from(A a,B b);//abstract handler
  public abstract C fromA(A a);//abstract handler
  public abstract C fromB(B b); //abstract handler
  public Stack<C> of(Stack<A> as,Stack<B> bs){//This is the template method
    return as.match(
      ()->ofBs(bs),//case as is empty
      (a,at)->bs.match(
        ()->ofAs(as),//case bs is empty (but as is not)
        (b,bt)->of(at,bt).push(from(a,b))//case both not empty
        ));}
  private Stack<C> ofBs(Stack<B> bs){//what to do if as are empty
    return bs.match(Stack::empty,(e,t)->ofBs(t).push(fromB(e)));
    }
  private Stack<C> ofAs(Stack<A> as){//what to do if bs are empty
    return as.match(Stack::empty,(e,t)->ofAs(t).push(fromA(e)));
    }}
var as=Stack.<String>empty().push("x").push("y").push("z").push("w");
var bs=Stack.<Integer>empty().push(1).push(2).push(3);
var cs=new StackWalk<String,Integer,String>(){
  public String from(String a,Integer b){return a+b;}
  public String fromA(String a){return a;}
  public String fromB(Integer b){throw new Error();}
  }.of(as,bs);
System.out.println(cs);}//prints [w3; z2; y1; x]
```

Immutable data structures

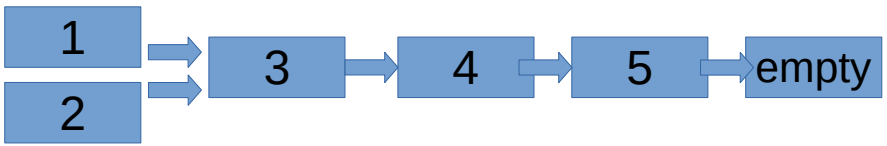
- The stack as shown is an immutable data structure
- Shallow immutability == T may be mutable
- pop/push does not mutate the Stack, but returns an object that 'looks like' the receiver, but with some difference
  (like having an extra element)

Immutable data structures

```
assert ns.push(1).pop() == ns;
```

- Same pointer! reuse of memory space

```
var ns1=ns.push(1);
var ns2=ns.push(2);
```



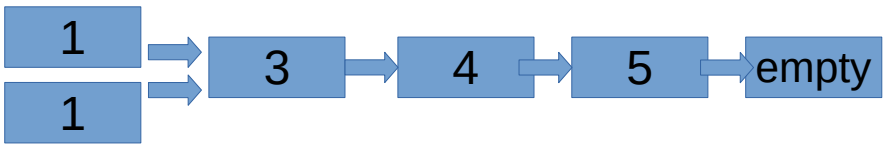
- Even if ns is a very long stack, having ns, ns1, ns2 only costs 2 extra objects w.r.t. just having ns.

Immutable data structures

```
assert ns.push(1) != ns.push(1);
```

- Different pointer, but same structure

```
var ns1=ns.push(1);
var ns2=ns.push(1);
```



- We may end up having different copies of the same stack. This may consume too much memory

The Flyweight pattern

- Working with immutable structures allows to reuse a lot of memory thanks to aliasing.
- Can we do even better, and make so that all objects are always in a normalized form?

Introducing the Flyweight pattern

Web assessment: Overriding 5 Peano identity [Challenging]

```
//The answer must have balanced parentesis and not contain 'static'
class Num{
  static Num zero(){return instance;}
  static Num instance=new Num();
  [???]
  }
public class Exercise{
  public static void main(String[]arg){
    Num zero=Num.zero();
    assert zero==Num.zero();
    Num one=zero.succ();
    assert one!=zero;
    Num two=one.succ();
    assert one!=two;
    assert two.pred()==one;
    //up to here it was like last question, but now more checks:
    Num two2=one.succ();
    Num tree=two.succ();
    assert two==two2;
    assert tree==two.succ();
    assert tree.pred().pred()==one;
    assert two.pred()==zero.succ();
  }
}
```

Web assessment: Overriding 5 Peano identity [Challenging]

```
//The answer must have balanced parentesis and not contain 'static'
class Num{
  static Num zero(){return instance;}
  static Num instance=new Num();
  Num succ(){//Answer
    Num self=this;//explicit this naming
    return new Num(){Num pred(){return self;}};//re implementing pred
    }
  Num pred(){return this;}//pred of zero==0
  }
public class Exercise{public static void main(String[]arg){
    Num zero=Num.zero();
    assert zero==Num.zero();
    Num one=zero.succ();
    assert one!=zero;
    Num two=one.succ();
    assert one!=two;
    assert two.pred()==one;     //Ok up to here, but FAILS below
    //up to here it was like last question, but now more checks:
    Num two2=one.succ();
    Num tree=two.succ();
    assert two==two2;//FAILS HERE!
    assert tree==two.succ();
    assert tree.pred().pred()==one;
    assert two.pred()==zero.succ(); } }
```

Web assessment: Overriding 5 Peano identity [Challenging]

```
class Num{
  static Num zero(){return instance;}
  static Num instance=new Num();
  Num mySucc;//adding a field to cache the result of succ()
  Num succ(){//implementing succ() with dynamic programming
    if(mySucc!=null){return mySucc;}//return cached result
    Num self=this;//explicit this naming
    return mySucc=new Num(){Num pred(){return self;}};//return+initialize cache
    }
  Num pred(){return this;}//pred of zero==0
  }
public class Exercise{ public static void main(String[]arg){
    Num zero=Num.zero();
    assert zero==Num.zero();
    Num one=zero.succ();
    assert one!=zero;
    Num two=one.succ();
    assert one!=two;
    assert two.pred()==one;//Now it passes all the tests
    Num two2=one.succ();
    Num tree=two.succ();
    assert two==two2;
    assert tree==two.succ();
    assert tree.pred().pred()==one;
    assert two.pred()==zero.succ(); }}
```

This is the Flyweight pattern over peano numbers. Peano numbers look like Stacks.

Can we do the Flyweight pattern over Stacks?

Flyweight for Stack (12 patterns total)

```
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!
  }
```

Flyweight for Stack (12 patterns total)

```
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!
  }
```

Flyweight for Stack (12 patterns total)

```
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!
  }
```

Flyweight for Stack (12 patterns total)

```
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,e->newPush(e));}
  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!
  }
```

Flyweight for Stack (12 patterns total)

```
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!
  }
```

Flyweight for Stack (12 patterns total)

```
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!
  }
```

Stack usage

```
public static void main(String[]a){

  var ss1=Stack.<String>empty().push("Hello").push("World").push("!!");
  var ss2=Stack.<String>empty().push("Hello").push("World").push("!!");

  assert ss1==ss2;

  assert ss1.popOrElse(err())==ss2.popOrElse(err());

  assert ss1.popOrElse(err()).popOrElse(err())==Stack.<String>empty().push("Hello");
  }
}

public static <T> Supplier<T> err(){return ()->{throw new Error();};}
```

Concluding

- Please, do not forget everything during the break!
- I organized the course so that you had no assignment/long tasks in the break!

- Take care, sleep well.
- Reminder: Tomorrow you have to complete the first chunk of readings: They are strongly connected with the visitor!