Chapter 3 - Stacks & Queues

// **Three in One** → describe how you could use a single array to implement three stacks

// **Solution A** → Fixed Division

```
public class FixedMultiStack {

	private int numberOfStacks = 3;
	private int stackCapacity;
	private int[] values;
	private int[] sizes;

	public FixedMultiStack(int stackSize) {
		stackCapacity = stackSize;
		values = new int[stackSize * numberOfStacks];
		sizes = new int[numberOfStacks];
	}

	public void push(int stackNum, int value) {
		if (isFull(stackNum)) {
			throw new FullStackException();
		}
		sizes[stackNum]++;
		values[indexOfTop(stackNum)] = value;
	}

	public int pop(int stackNum) {
		if (isEmpty(stackNum)) {
			throw new EmptyStackException();
		}
		int topIndex = indexOfTop(stackNum);
		int value = values[topIndex];
		values[topIndex] = 0;  // clear
		sizes[stackNum]--;
		return value;
	}

	public int peek(int stackNum) {
		if (isEmpty(stackNum)) {
			throw new EmptyStackException();
```

```
		}
		return values[indexOfTop(stackNum)];
	}

	public boolean isEmpty(int stackNum) {
		return sizes[stackNum] == 0;
	}

	public boolean isFull(int stackNum) {
		return sizes[stackNum] == stackCapacity;
	}

	public int indexOfTop(int stackNum) {
		int offset = stackNum * stackCapacity;
		return offset + sizes[stackNum] - 1;
	}
}
```

// **Stack Min** → How would you desing a stack that has a function min that returns the minimum element?

// **SolutionA** → each item keeps tracks of its own minimum

```
public class StackWithMin extends Stack<NodeWithMin> {
	public void push(int value) {
		int newMin = Math.min(value, min());
		super.push(new NodeWithMin(value, newMin));
	}

	public int min() {
		if (this.isEmpty()) {
			return Integer.MAX_VALUE;
		}
		return peek().min;
	}
}

class NodeWithMin {
	public int value;
	public int min;

	public NodeWithMin(int value, int min) {
		this.value = value;
```

```
                this.min = min;
        }
}
```

// **Solution B** → if we have a large stack we waste a lot of space by keeping track of the min for every single element

```
public class StackWithMin2 extends Stack<Integer> {

        Stack<Integer> s2;

        public StackWithMin2() {
                s2 = new Stack<>();
        }

        public void push(int value) {
                if (value <= min()) {
                        s2.push(value);
                }
                super.push(value);
        }

        public Integer pop() {
                int value = super.pop();
                if (value == min()) {
                        s2.pop();
                }
                return value;
        }

        public int min() {
                if (s2.isEmpty()) {
                        return Integer.MAX_VALUE;
                }
                return s2.peek();
        }
}
```

// **Stack of Plates** → Implement a set of stacks

```
public class SetOfStacks {

        public int capacity;
```

```
private List<Stack> stacks = new ArrayList<>();

public SetOfStacks(int capacity) {
	this.capacity = capacity;
}

public void push(int value) {
	Stack last = getLastStack();

	if (last != null && !last.isFull()) {
		last.push(value);
	} else {
		Stack stack = new Stack(capacity);
		stack.push(value);
		stacks.add(stack);
	}
}

public int pop() {
	Stack last = getLastStack();
	if (last == null) {
		throw new EmptyStackException();
	}
	int value = last.pop();
	if (last.size == 0) {
		stacks.remove(stacks.size() - 1);
	}
	return value;
}

public Stack getLastStack() {
	if (stacks.size() == 0) {
		return null;
	}
	return stacks.get(stacks.size() - 1);
}

public boolean isEmpty() {
	Stack last = getLastStack();
	return last != null || last.isEmpty();
}

public int popAt(int index) {
```

```
		return leftShift(index, true);
	}

	public int leftShift(int index, boolean removeTop) {
		Stack stack = stacks.get(index);
		int removedItem;
		if (removeTop) {
			removedItem = stack.pop();
		} else {
			removedItem = stack.removeBottom();
		}
		if (stack.isEmpty()) {
			stacks.remove(index);
		} else if (stacks.size() > index + 1) {
			int v = leftShift(index + 1, false);
			stack.push(v);
		}
		return removedItem;
	}
}
```

// **Queue vis Stacks** → Implement a MyQueue class using two stacks

```
public class MyQueue<Item> {

	Stack<Item> stackNewest, stackOldest;

	public MyQueue() {
		stackNewest = new Stack<>();
		stackOldest = new Stack<>();
	}

	public int size() {
		return stackNewest.size() + stackOldest.size();
	}

	public void add(Item item) {
		// Always has the newest elements on top
		stackNewest.push(item);
	}

	public void shiftStacks() {
		if (stackOldest.isEmpty()) {
```

```
				while (!stackNewest.isEmpty()) {
					stackOldest.push(stackNewest.pop());
				}
			}
		}

		public Item peek() {
			shiftStacks();
			return stackOldest.peek();
		}

		public Item remove() {
			shiftStacks();
			return stackOldest.pop();
		}
	}
```

// **Sort Stack** → write a program to sort a stack such that the smallest elements are on top

```
public class Question {

	public static void sort(Stack<Integer> s) {
		Stack<Integer> r = new Stack<>();
		while (!s.isEmpty()) {
			int tmp = s.pop();
			while (!r.isEmpty() && r.peek() > tmp) {
				s.push(r.pop());
			}
			r.push(tmp);
		}

		while (!r.isEmpty()) {
			s.push(r.pop());
		}
	}
}
```

// **Animal Shelter**

```
public abstract class Animal {

	private int order;
	protected String name;
```

```
	public Animal(String name) {
		this.name = name;
	}

	public void setOrder(int order) {
		this.order = order;
	}

	public int getOrder() {
		return order;
	}

	public boolean isOlderThan(Animal a) {
		return this.order > a.getOrder();
	}
}

public class AnimalQueue {
	LinkedList<Dog> dogs = new LinkedList<>();
	LinkedList<Cat> cats = new LinkedList<>();
	private int order = 0;  // instead of timestamp

	public void enqueue(Animal a) {
		a.setOrder(order);
		order++;

		if (a instanceof Dog) {
			dogs.addLast((Dog) a);
		} else if (a instanceof Cat) {
			cats.addLast((Cat) a);
		}
	}

	public Animal dequeueAny() {
		if (dogs.size() == 0) {
			return dequeueCats();
		} else if (cats.size() == 0) {
			return dequeueDogs();
		}

		Dog dog = dogs.peek();
		Cat cat = cats.peek();
```

```
		if (dog.isOlderThan(cat)) {
			return dequeueDogs();
		} else {
			return dequeueCats();
		}
	}

	public Dog dequeueDogs() {
		return dogs.poll();
	}

	public Cat dequeueCats() {
		return cats.poll();
	}
}

public class Dog extends Animal {
	public Dog(String name) {
		super(name);
	}
}

public class Cat extends Animal {
	public Cat(String name) {
		super(name);
	}
}
```