Termpaper 1

Name	Student ID	Course	University
Andrii Yatsura	19387	Programming Languages and Paradigms	Wrocław Academy of Business

This essay delves into the conceptualization and implementation of a queue data structure, emphasizing the differences between procedural and object-oriented programming (OOP) paradigms. Among various data structures structures, the queue stands out as a fundamental entity with a First-In-First-Out (FIFO) principle.

Queue

Queue is an abstract data type, implementing the *First In, First Out (FIFO)* principle. It is a linear data type, meaning we can represent it with an array or a linked list, both being structures often used for its implementation. In a queue, elements are added at the rear end, action known as *enqueue*, and removed from the front end, action called *dequeue*. It operates just like a real-world queue or line – the first person who joins the line is the first one to be served.

There are 5 main operations supported by the queue data type:

Operation	**Explanation**
`enqueue(e)` or `push(e)`	Appends an element `e` to the rear of the queue
`dequeue()`	Removes and returns the element at the front of the queue
`front()`	Returns a reference or a value of the element at the front of the queue
`size()`	Returns the number of elements in the queue
`empty()`	Returns `true` in case if queue is empty, otherwise `false`

Notice that there are two main ways to interact with the frontmost element of the queue: `dequeue()` and `front()`. Both let you access the reference to the element, but one does it without removing the element from the queue.

The simplicity of a queue makes it versatile in various applications, such as task scheduling, print job management, and breadth-first search algorithms. Now let's look at the implementations of this data structure in two different programming paradigms: *procedural* and *object-oriented*.

Procedural

In procedural paradigm, the focus goes to procedures, or operations that manipulate the data, being the queue in this case. A typical implementation would define a function for each operation the data structure has along with an initialization function. The data structure itself may be represented as a structure with attributes like `array`, `front`, and `rear` indexes.

This implementation let's user freely use the data, bypassing defined operations of the queue, which may not always be desirable. Also, this approach may require more effort when extending or maintaining functionality.

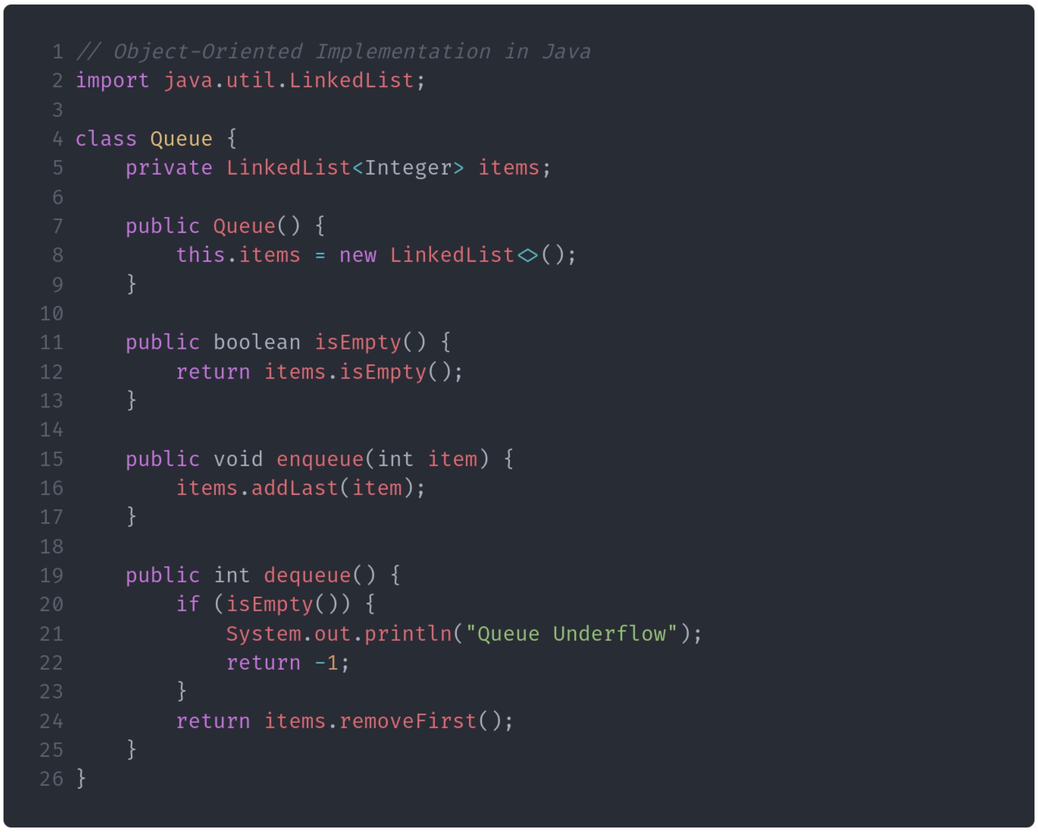
```c
// Procedural Queue Implementation in C
#include <stdio.h>

#define MAX_SIZE 100

struct Queue {
    int arr[MAX_SIZE];
    int front, rear;
};

void initializeQueue(struct Queue *q) {
    q->front = -1;
    q->rear = -1;
}

int isEmpty(struct Queue *q) {
    return (q->front == -1);
}

void enqueue(struct Queue *q, int item) {
    if (q->rear == MAX_SIZE - 1) {
        printf("Queue Overflow\n");
        return;
    }
    if (isEmpty(q))
        q->front = 0;
    q->arr[++q->rear] = item;
}

int dequeue(struct Queue *q) {
    if (isEmpty(q)) {
        printf("Queue Underflow\n");
        return -1;
    }
    int item = q->arr[q->front++];
    if (q->front > q->rear)
        initializeQueue(q);
    return item;
}
```

Object-Oriented

In contrast, this paradigm shifts the focus to an object, containing all of the properties and methods the queue might have, fostering a higher level of abstraction.

A `Queue` class provides a clean, encapsulated implementation of the queue data structure. User interacting with the object only has access to public methods, preventing unwanted changes to internal data or state of the object.

New methods can be easily added to the class, making for a better development environment.



```java
// Object-Oriented Implementation in Java
import java.util.LinkedList;

class Queue {
    private LinkedList<Integer> items;

    public Queue() {
        this.items = new LinkedList<>();
    }

    public boolean isEmpty() {
        return items.isEmpty();
    }

    public void enqueue(int item) {
        items.addLast(item);
    }

    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow");
            return -1;
        }
        return items.removeFirst();
    }
}
```

Comparison

This table provides a concise overview of the key differences between procedural and object-oriented implementations of a queue, Each paradigm has its strengths and weaknesses, and the choice between them depends on the specific requirements and design goals of the application.

	Procedural	Object-Oriented
Focus	Functions implementing the operations	An object, containing all properties and methods
Abstraction	Lower, user can access the data	Higher, only public interface is accessible from outside
Maintainability	Worse, doesn’t allow for easy extension	Better, encapsulation resolves many issues
Code reusability	Good, methods can be used wherever needed	Good, methods are accessible from every instance