~~~
# H01 - OOP Worksheet
**Name:** Hornel Cloubou
**Date:** Feb 23, 2026

---

## Part 1 — Stacks & Queues
* **Implementation:** Array-based stacks offer $O(1)$ access and are cache-friendly. Linked-
lists avoid resizing but have higher memory overhead per element.
* **Vector as a Queue:** Inefficient because `pop_front` requires shifting all elements,
resulting in $O(n)$ time complexity.
* **Invariants:** * **Stack:** Last-In, First-Out (LIFO).
    * **Queue:** First-In, First-Out (FIFO).

---

## Part 2 — Overloading vs Overriding
| Feature | Overloading | Overriding |
| :--- | :--- | :--- |
| **Definition** | Same name, different parameters | Same name, same parameters |
| **Binding** | Static (Compile-time) | Dynamic (Runtime) |
| **Requirement** | Within same scope/class | Requires Inheritance |

---

## Part 3 — Constructors & Initialization Lists
In the `Widget` class, `const int id` **must** be initialized in the initialization list
because constants cannot be assigned a value within the constructor body.
* **Correct Syntax:** `Widget(int val) : id(val) {}`

---

## Part 4 — Point2D Class
```cpp
#include <iostream>
class Point2D {
private:
    int x, y;
public:
    Point2D() : x(0), y(0) {}
    Point2D(int _x, int _y) : x(_x), y(_y) {}

    Point2D operator+(const Point2D& other) const {
        return Point2D(x + other.x, y + other.y);
    }

    bool operator==(const Point2D& other) const {
        return (x == other.x && y == other.y);
    }

    friend std::ostream& operator<<(std::ostream& os, const Point2D& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};
```

---

## Part 5 — Composition & Inheritance
* **Composition (Has-A):** A class contains an instance of another class (e.g., a Car has an
Engine).
* **Inheritance (Is-A):** A class derives from another to inherit properties (e.g., a Dog is
an Animal).
~~~

Part 6 — Access Modifiers
* **Public:** Accessible from anywhere in the program.
* **Private:** Accessible only within the class itself.
* **Protected:** Accessible within the class and its child (derived) classes.

Part 7 — Reflection
1. **Operator Overloading:** It makes custom types feel like native data types, making code more readable.
2. **Initialization Lists:** Essential for performance and required for `const` or reference members.
3. **Data Structures:** Choosing the right structure (Stack vs Queue) is vital for algorithm efficiency.