CodInClub
Powered By BridgeLabz

Wk6 /S2/ Lecture # : DSOOPS-26

Topics Covered:

- OOP Inspiration from the Real World
- Class vs Object (Instance)
- How Basic Data Types Differ from Reference/Object Types
- Accessing Class Members
- Understanding Encapsulation
- Benefits of Encapsulation: Data Hiding and Organization

OOP Inspiration from the Real World

Object-Oriented Programming (OOP) borrows its approach from the way we view and organize complex real-life systems.

Think Real World:
Imagine a library. It has many books. Each book has properties (title, author, pages) and behaviors (borrow, return). You might also think of students, cars, phones, or animals—each with unique data and actions.

OOP in Code:
We represent real-world things as objects in programming—each object bundles together both data (attributes/properties) and behaviors (functions/actions).

Why?

- Helps model complex systems naturally
- Makes big problems easier to break down

Class vs Object (Instance)

Class:

- A “blueprint” or “template” describing what an object should be made of.
- Specifies both data (called members or attributes) and actions (member functions/methods).
- Example: The Book blueprint.

Object (Instance):

- A specific, concrete thing created using a class.
- Example: A particular copy of "The Hobbit" on your desk.

Code Example:

```
class Book {
public:
    std::string title;
    int pages;
};

int main() {
    Book myBook;          // Object (instance)
    myBook.title = "The Hobbit";
    myBook.pages = 310;
}
```

Analogy:
If the class is the recipe, an object is the actual cake you bake with it.

How Basic Data Types Differ from Reference/Object Types

Basic Data Types (Primitive Types):

- Built-in: `int`, `double`, `char`, `float`, `bool`, etc.
- Store single values directly.
- Simple allocation (fixed size), fast to use.

Reference/Object Types:

- Created from classes or structs (your own or from libraries).
- Can store multiple values and functions together (even of different types!).
- Accessed via their members.
- Objects can be large or complex and may be allocated dynamically on the heap (with pointers).

Key Difference:

- Primitives hold just a number or symbol.
- Objects are collections of data *and* behaviors, often tied together and designed to represent real concepts.

Example:

```
int age = 17;     // Primitive (basic type)
Book book1;       // Reference/Object (complex type)
```

Accessing Class Members

To access or set the data inside an object, use the dot operator (`.`) for regular (stack) objects and the arrow operator (`->`) for pointers.

Example:

```
Book b;
b.title = "1984";
b.pages = 328;
std::cout << b.title << std::endl; // using dot operator

Book* ptr = new Book;
ptr->title = "C++ Primer";
ptr->pages = 900;
std::cout << ptr->title << std::endl; // using arrow operator
delete ptr; // Don't forget to free memory if using new
```

Understanding Encapsulation

Encapsulation means “wrapping up” data and functions that work on that data into a single unit (the class). It is a central OOP idea.
Why encapsulate?

- Keeps related data and actions together
- Reduces confusion in large codebases

How?

- Use `private:` and `public:` specifiers:
  - public: accessible from outside the class
  - private: only accessible by the class itself (hidden from outside)

Example:

```
class BankAccount {
private:
    double balance; // hidden

public:
    void deposit(double amount) { balance += amount; }
    double getBalance() { return balance; }
};
```

Benefits of Encapsulation: Data Hiding and Organization

1. Data Hiding

- Details (like the `balance` variable above) are *protected from direct outside access*.
- Prevents careless or malicious parts of the program from changing the data in unsafe ways.

Example:

```
BankAccount acct;
// acct.balance = 9999; // ERROR: can't access private member!
acct.deposit(100);
std::cout << acct.getBalance() << std::endl;
```

2. Better Organization

- All data and the code that works on it live *together* in a class.
- This makes programs easier to:
  - Design

- Understand
- Manage and update

Practice Problems and Activities

Easy 1

Write a class called Student with two public members: name (string) and rollNo (int). In main, create an object, assign values to both members, and print them.

Easy 2

What's the output? Briefly explain why.

```
class Number {
public:
    int value;
};

int main() {
    Number a;
    a.value = 20;
    Number b = a;
    b.value = 50;
    std::cout << a.value << " " << b.value << std::endl;
    return 0;
}
```

Hint: Are these the same object or two separate ones?

Medium

Below is a partially defined class. Make balance a private member, and provide a public function setBalance to change it, and a public function getBalance to read it. In main, show that you can't directly access balance:

```
class Wallet {
    // Your code here
};

int main() {
    Wallet w;
    // w.balance = 500; // Should be ERROR!
    w.setBalance(500);
    std::cout << w.getBalance() << std::endl; // Should print 500
    return 0;
}
```

Hard

Suppose you're designing a class BankAccount with a private variable balance. Only deposit and withdraw member functions are allowed to change the balance. Write this class, create two BankAccount objects, and:

1. Deposit 1000 in one, 500 in the other.
2. Withdraw 200 from both.
3. Print both balances using a getBalance method.
4. Attempt to directly set balance from main (should result in an error; comment this out).

Show your code and briefly explain why this approach is more secure than making balance public.

Wrap-Up & Key Takeaways

- OOP models code after real-world objects, making big problems easier to manage.
- Class = blueprint, Object = actual thing you use
- Primitives store single values; objects bundle together data and behavior.
- Access class members with `.` for objects and `->` for pointers.
- Encapsulation keeps data and methods together, hiding details and reducing bugs.
- Data hiding and organization from encapsulation make your programs safer, cleaner, and easier to understand.