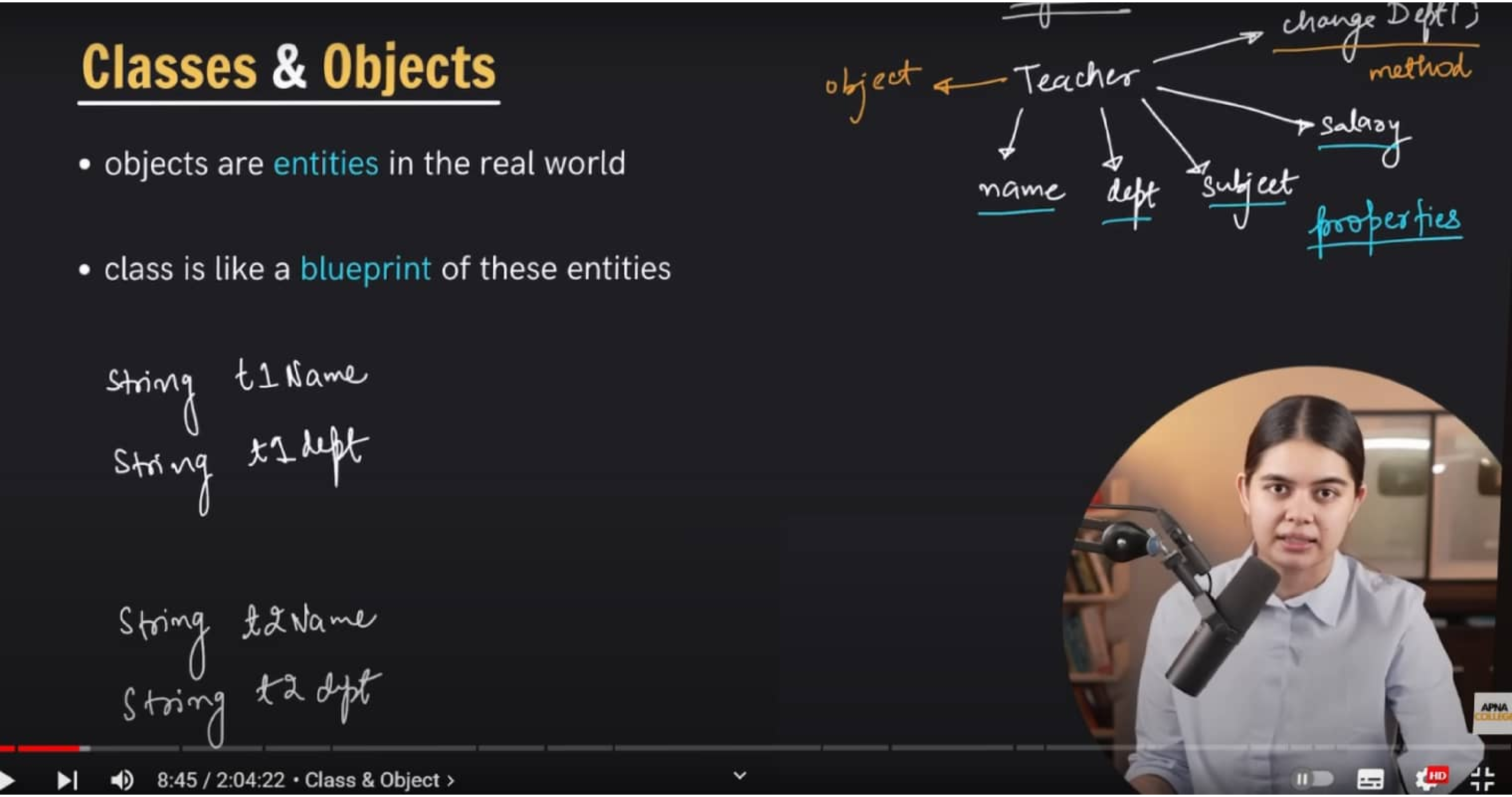
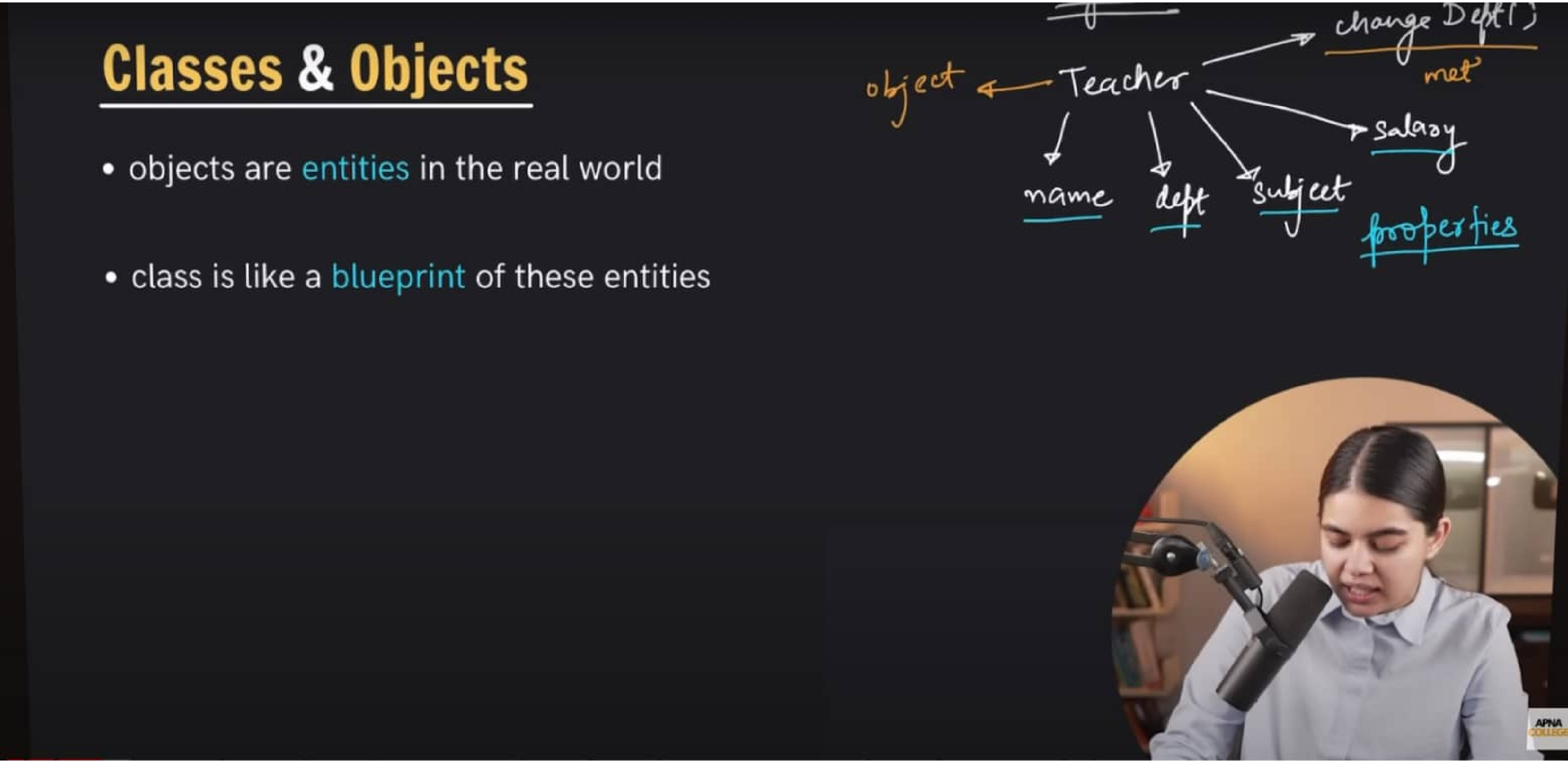


Classes & Objects

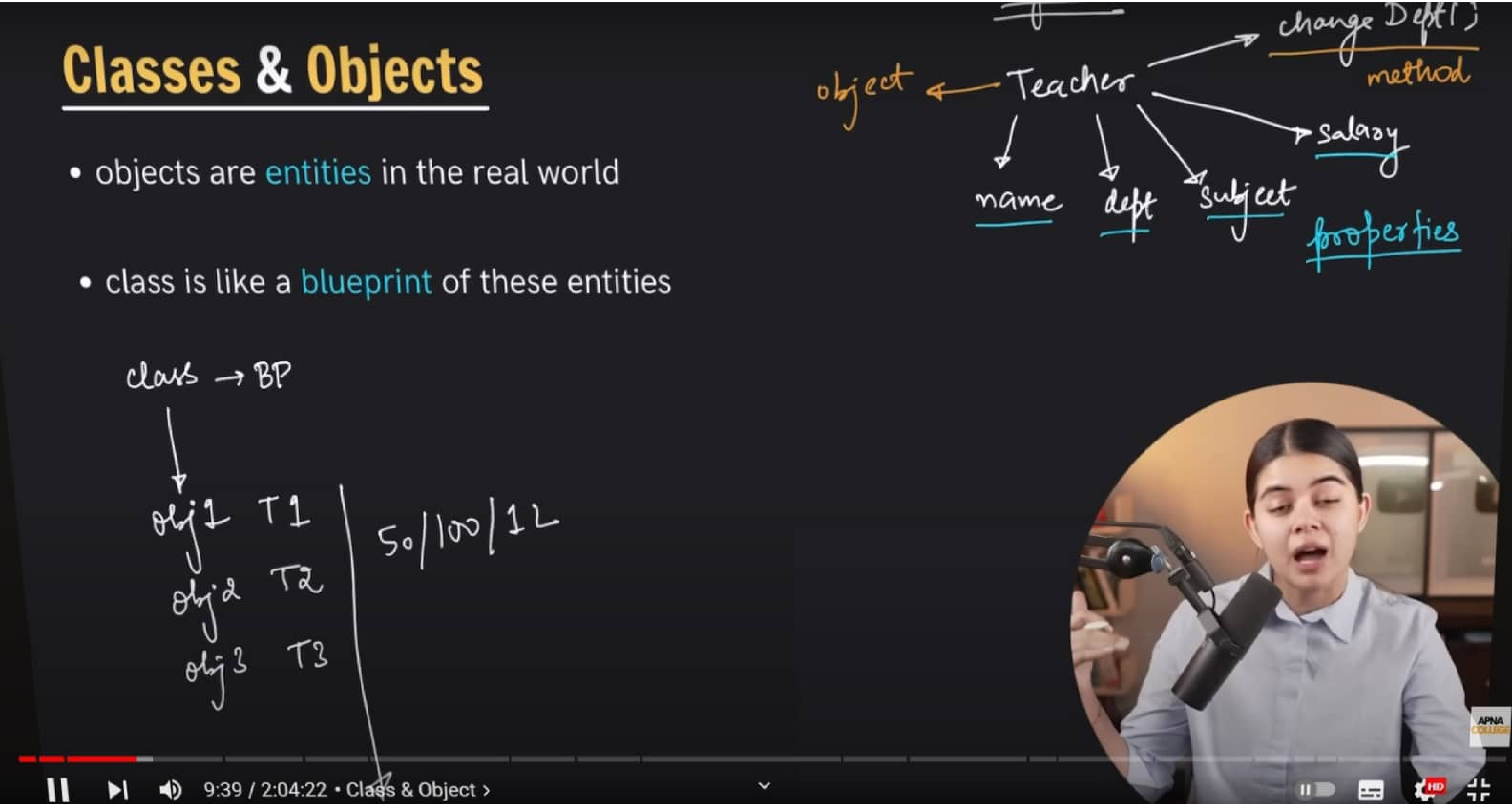
- objects are entities in the real world
- class is like a blueprint of these entities

```
String t1Name
String t1dept

String t2Name
String t2dpt
```



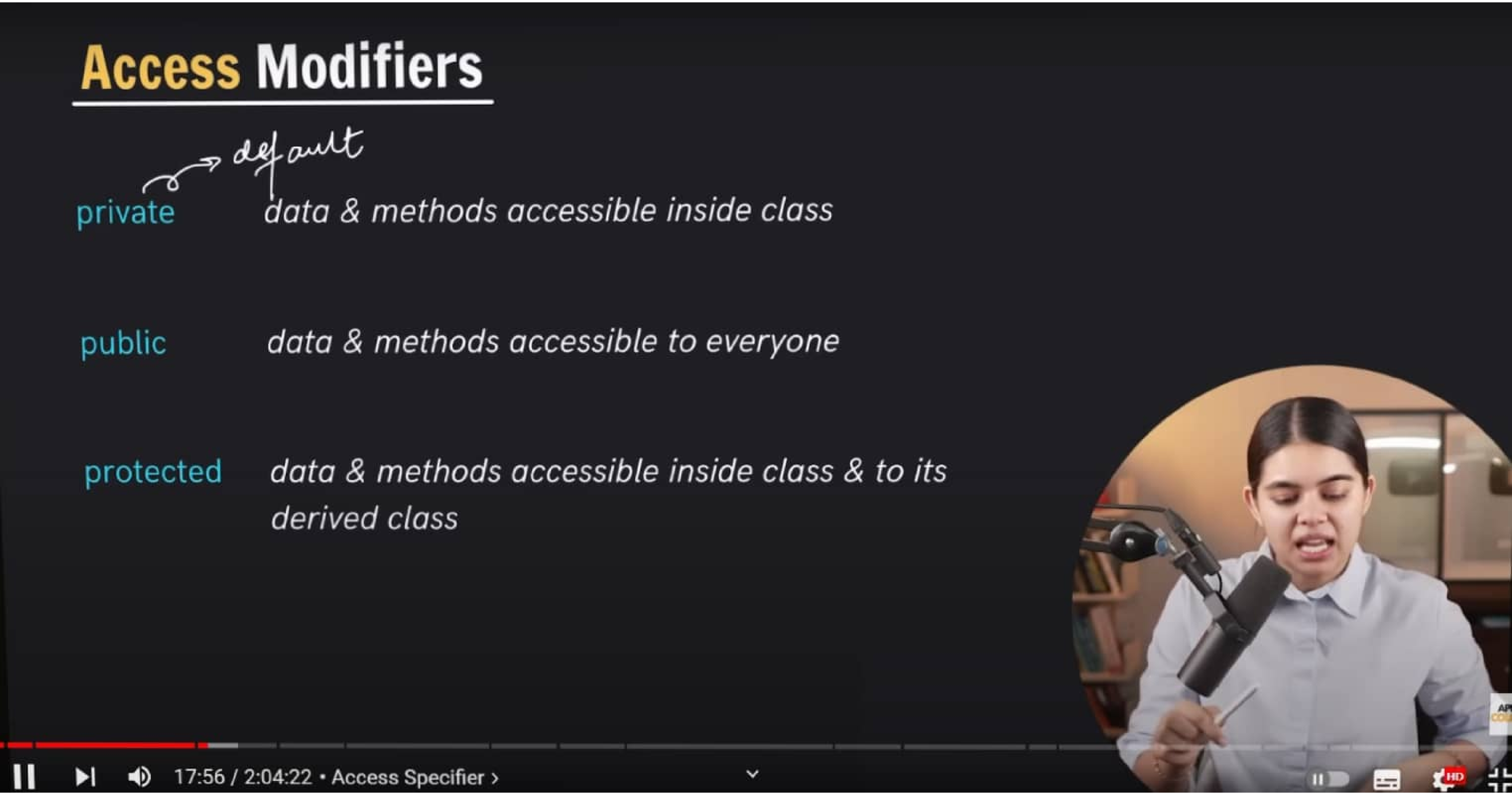
Classes & Objects

- objects are entities in the real world
- class is like a blueprint of these entities



Classes & Objects

- objects are entities in the real world
- class is like a blueprint of these entities



Access Modifiers

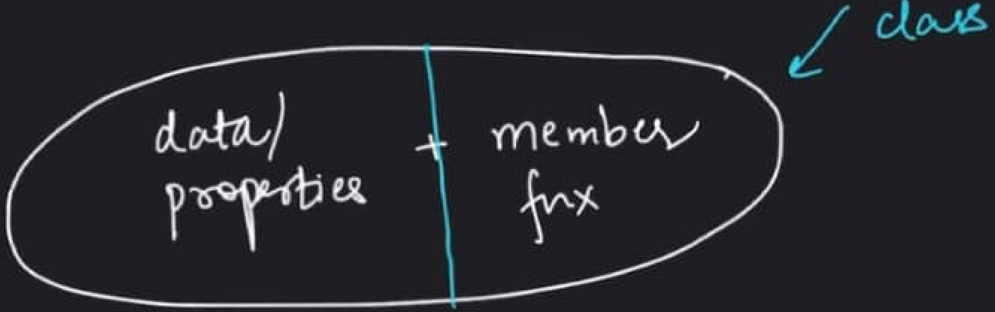
private $\xrightarrow{}$ default — data & methods accessible inside class

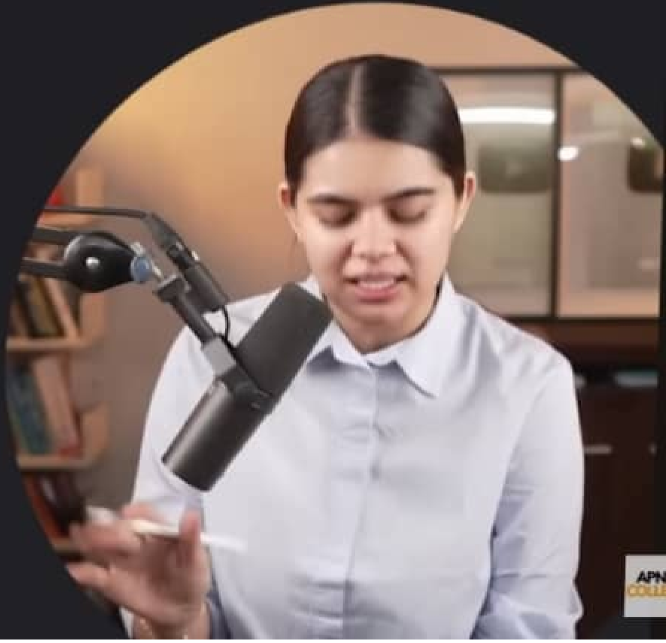
public — data & methods accessible to everyone

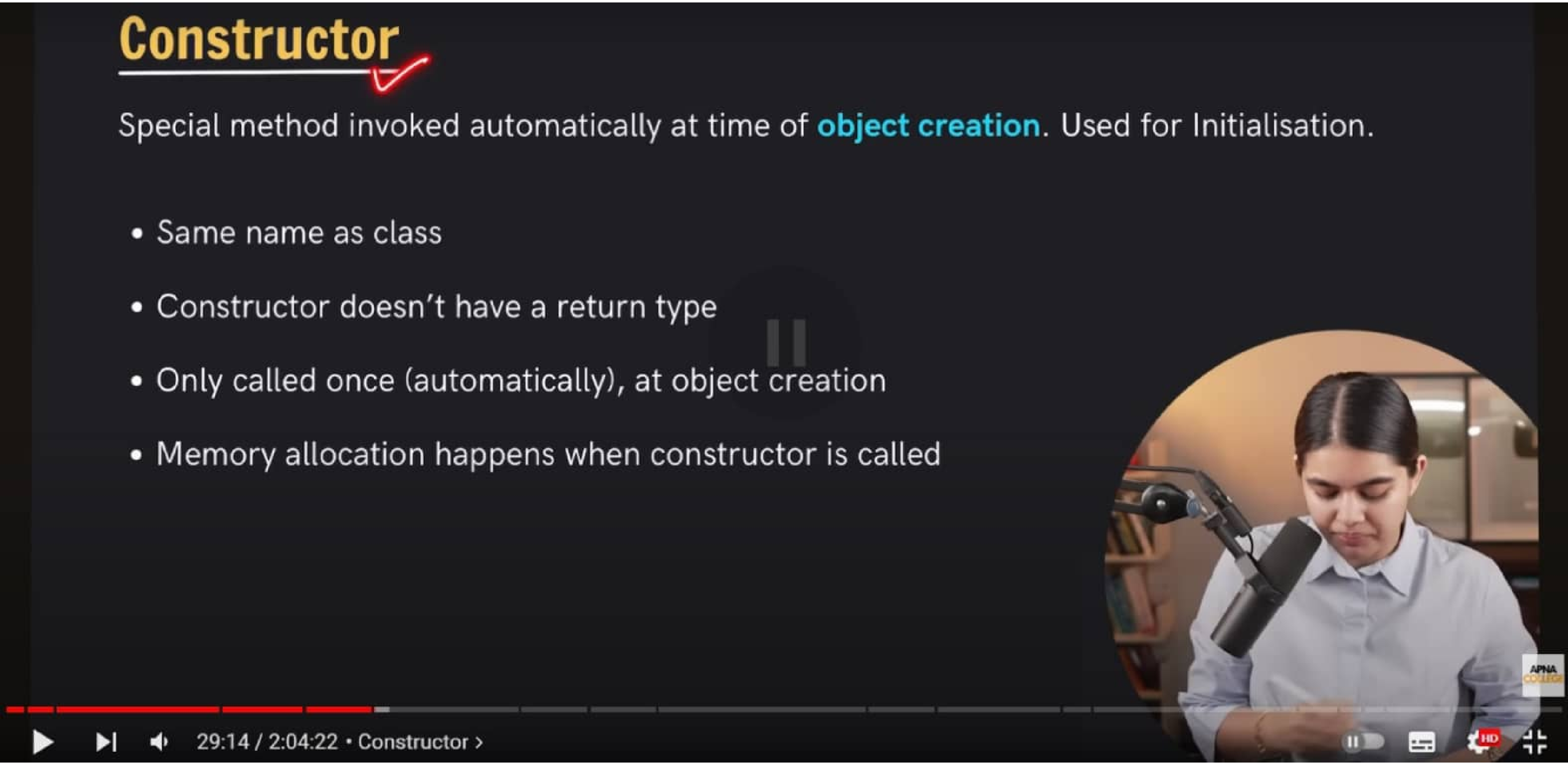
protected — data & methods accessible inside class & to its derived class

Encapsulation

Encapsulation is **wrapping up** of data & member functions in a single unit called class.



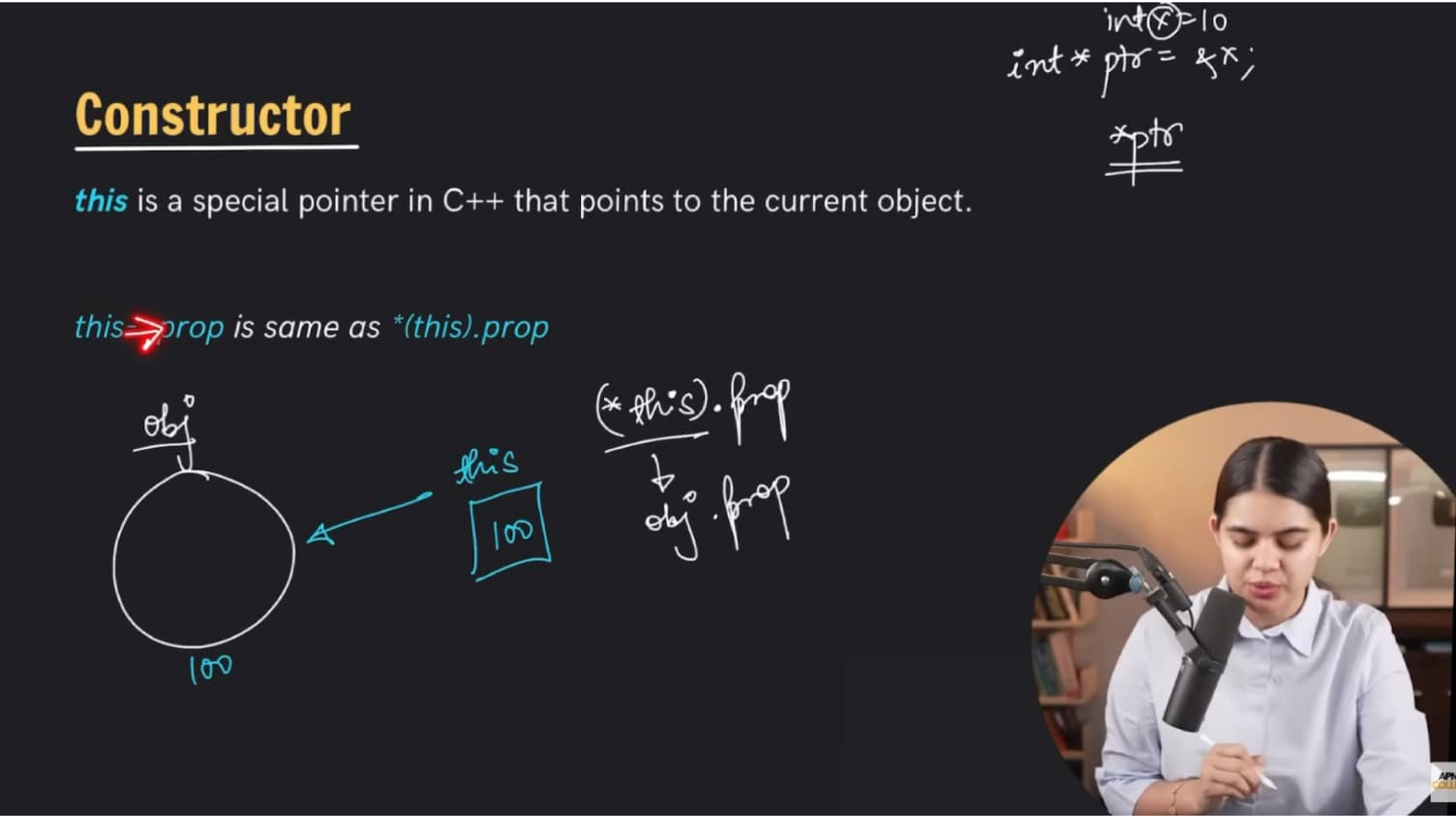




Constructor

Special method invoked automatically at time of **object creation**. Used for Initialisation.

- Same name as class
- Constructor doesn't have a return type
- Only called once (automatically), at object creation
- Memory allocation happens when constructor is called



Constructor

this is a special pointer in C++ that points to the current object.

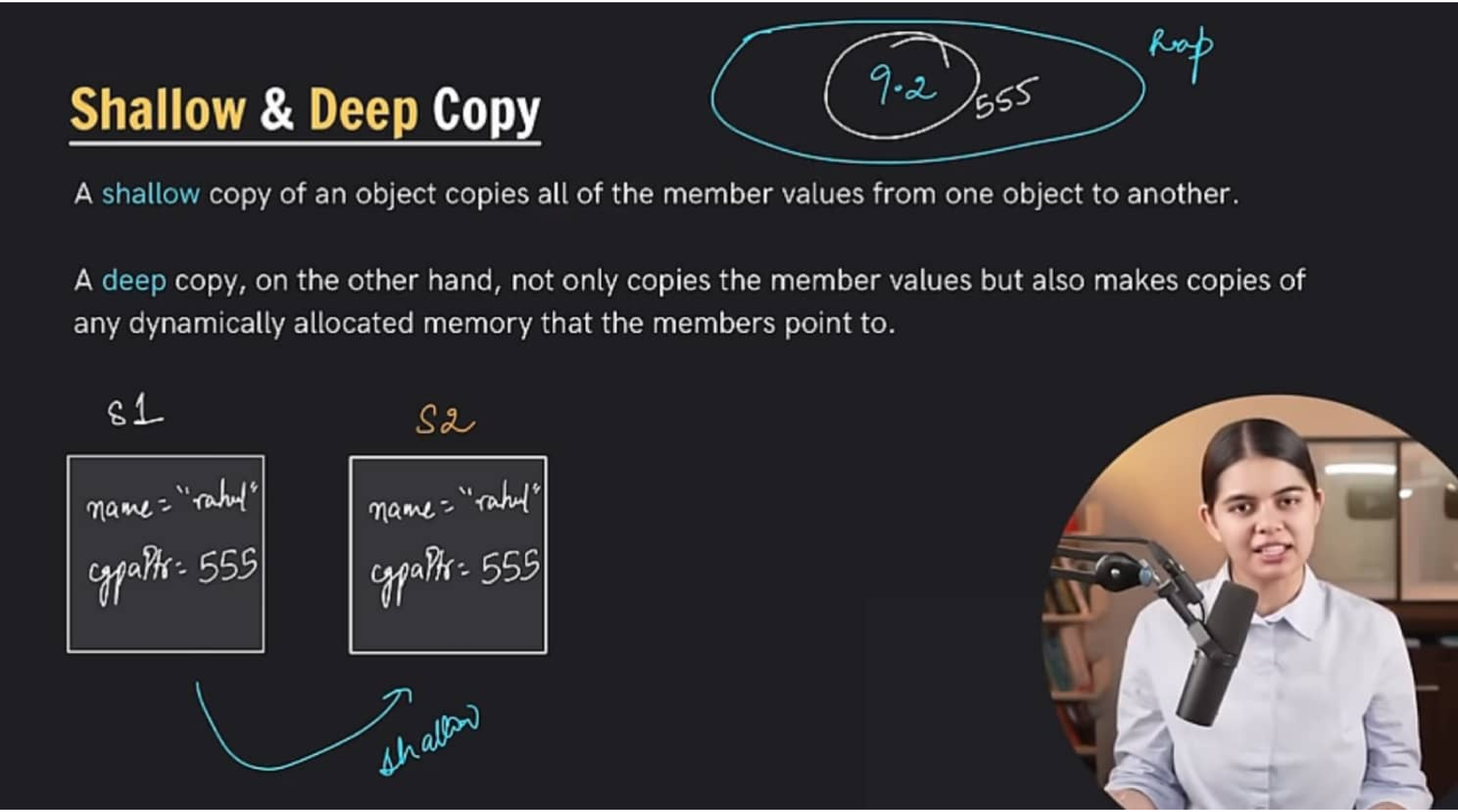
*this->prop is same as *(this).prop*

int $\boxed{x}$ = 10

int * ptr = &x;

$\underline{\underline{*ptr}}$

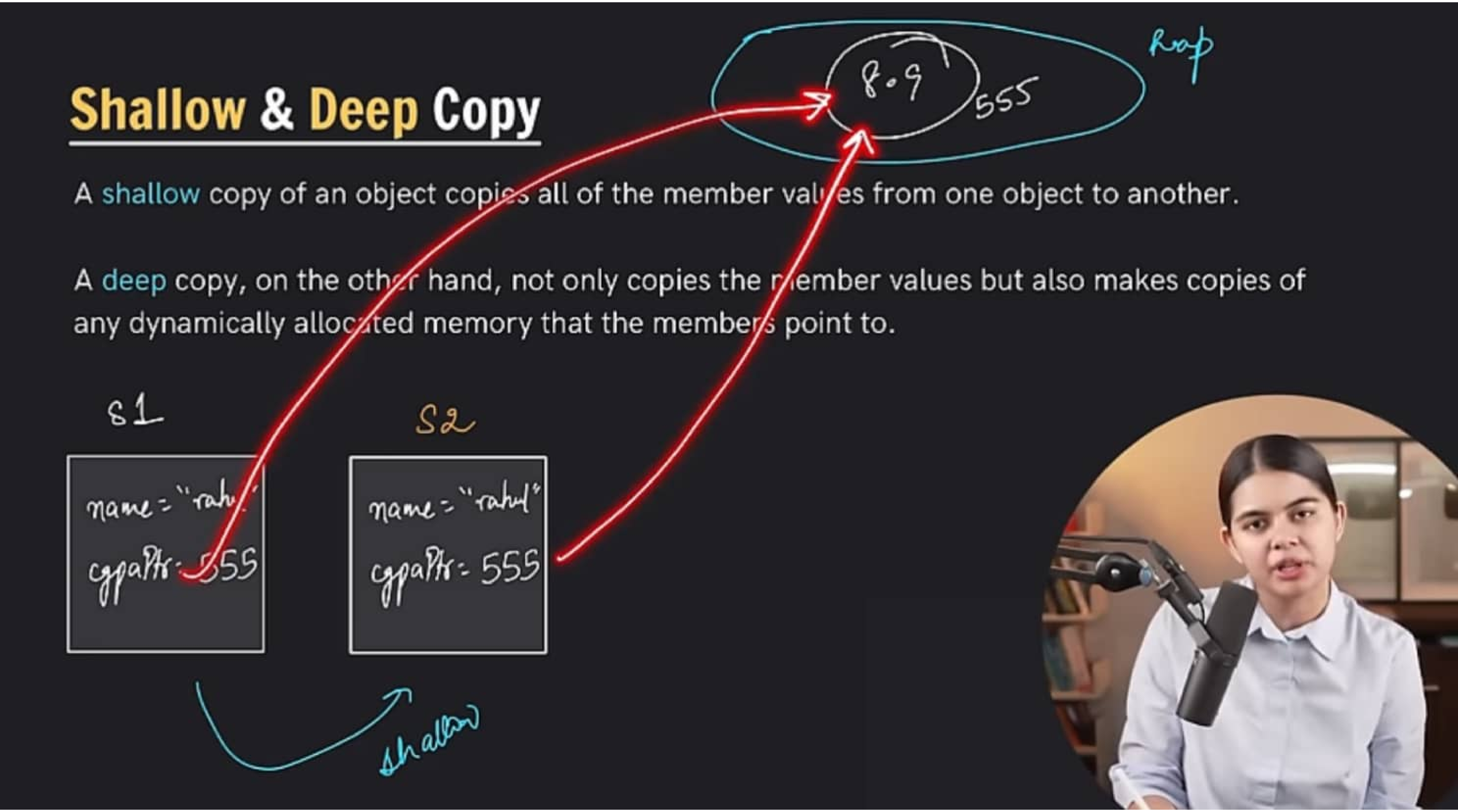
$$\begin{tikzcd}\underline{(*this)}.prop \arrow[d] \\ obj.prop\end{tikzcd}$$



Shallow & Deep Copy

A shallow copy of an object copies all of the member values from one object to another.

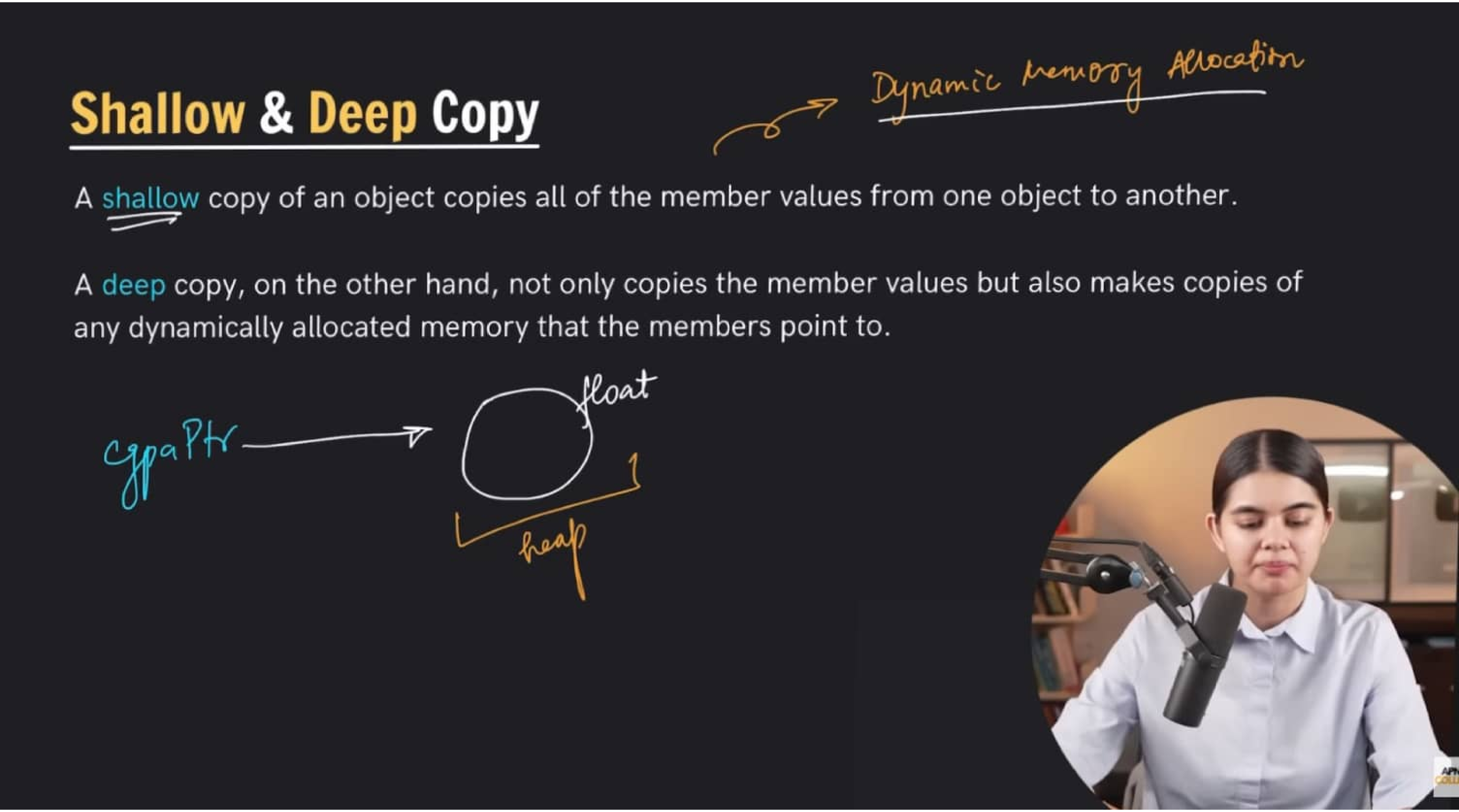
A deep copy, on the other hand, not only copies the member values but also makes copies of any dynamically allocated memory that the members point to.



Shallow & Deep Copy

A shallow copy of an object copies all of the member values from one object to another.

A deep copy, on the other hand, not only copies the member values but also makes copies of any dynamically allocated memory that the members point to.

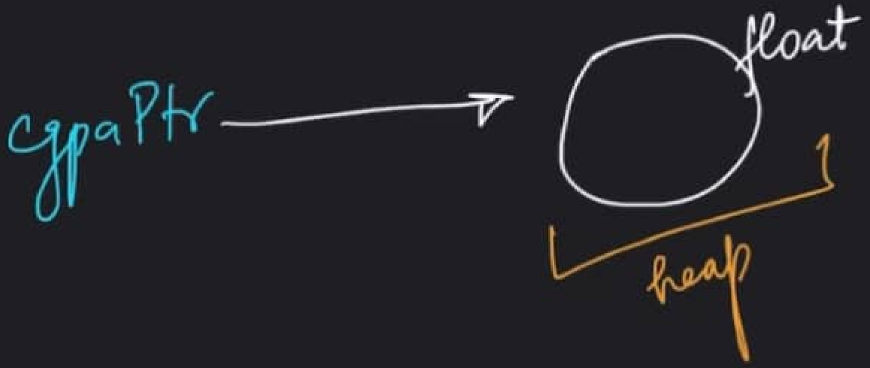


Shallow & Deep Copy

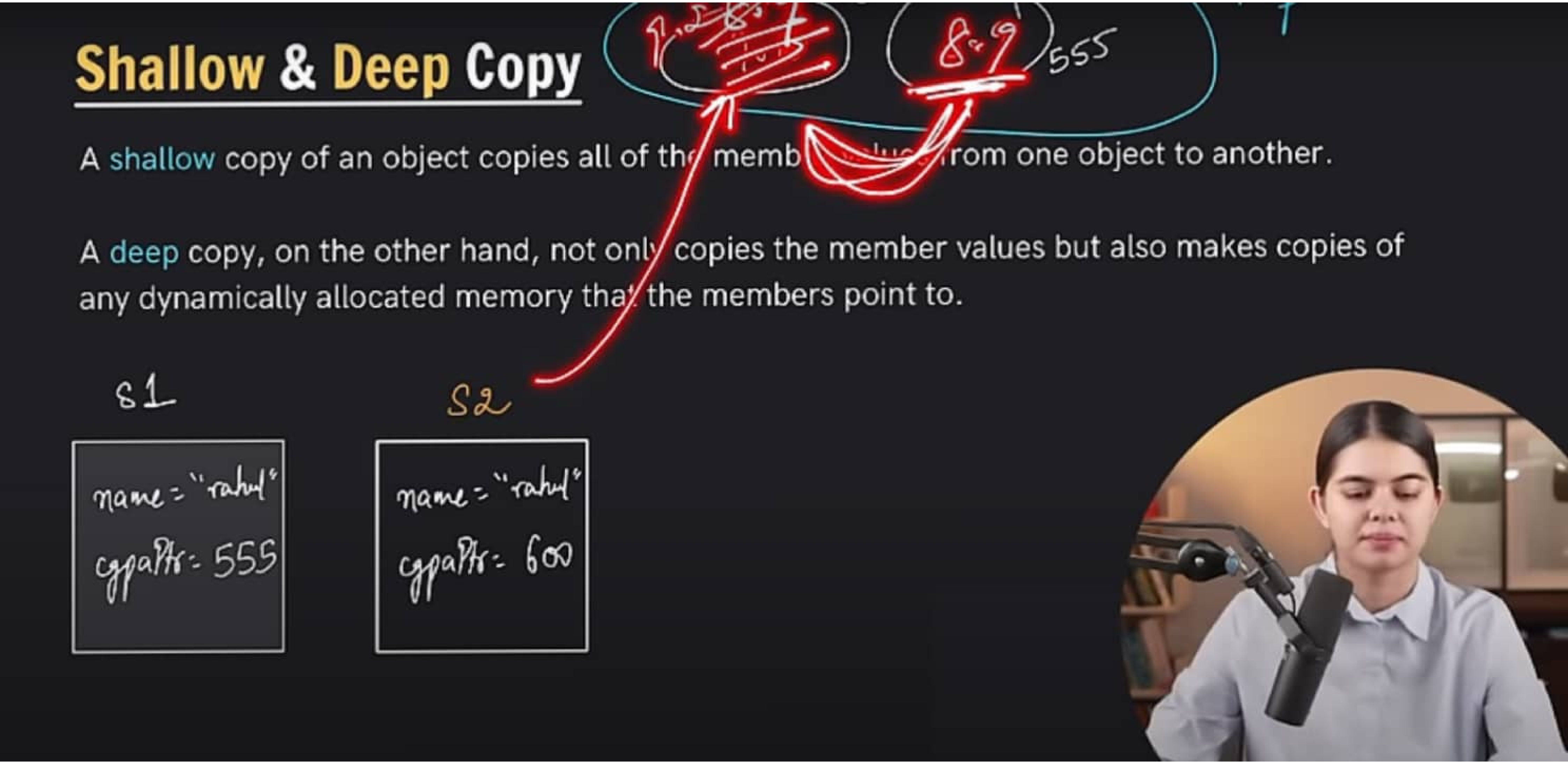
Dynamic Memory Allocation

A shallow copy of an object copies all of the member values from one object to another.

A deep copy, on the other hand, not only copies the member values but also makes copies of any dynamically allocated memory that the members point to.



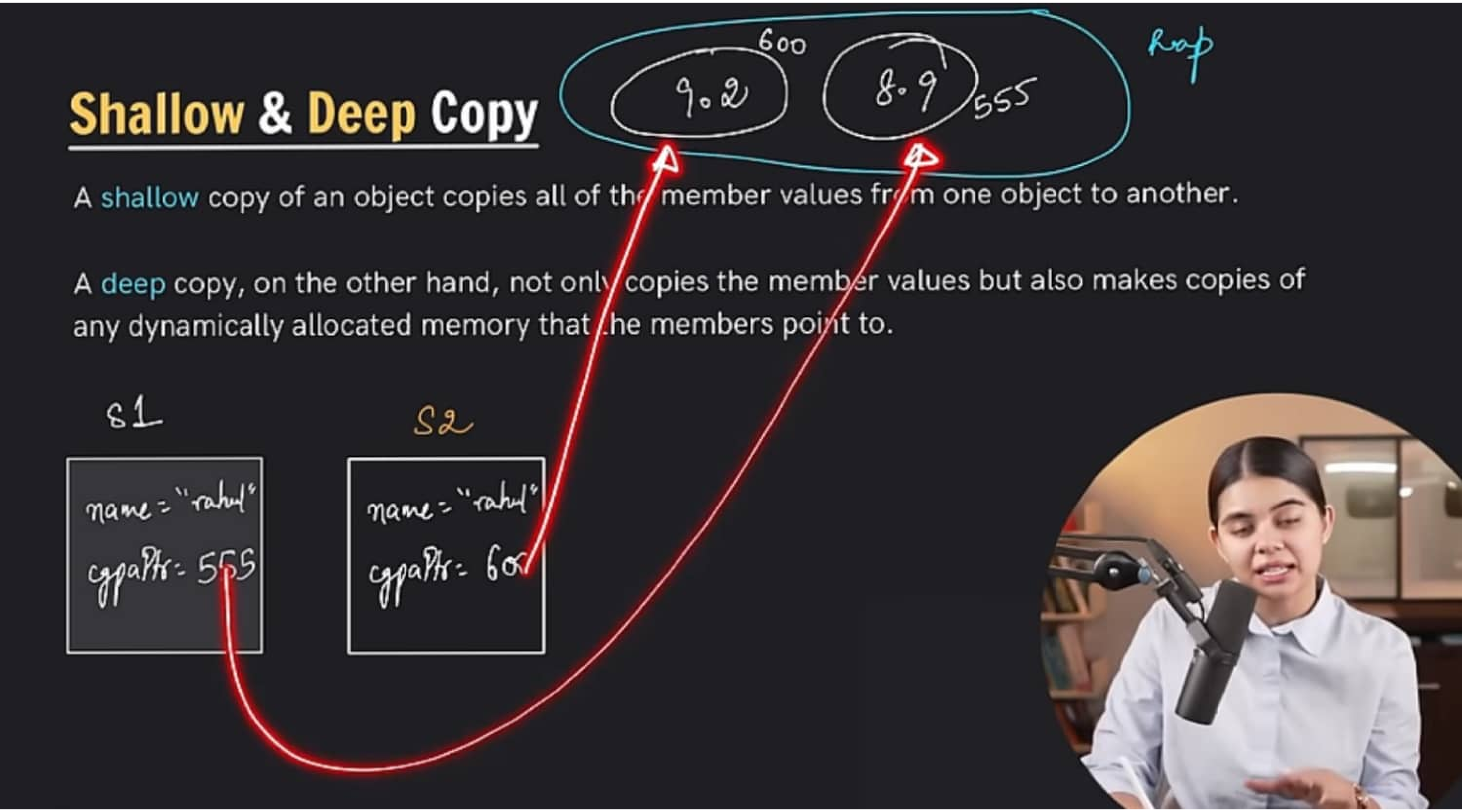
CamScanner



Shallow & Deep Copy

A shallow copy of an object copies all of the memb[illegible]rom one object to another.

A deep copy, on the other hand, not only copies the member values but also makes copies of any dynamically allocated memory that the members point to.



Shallow & Deep Copy

A shallow copy of an object copies all of the member values from one object to another.

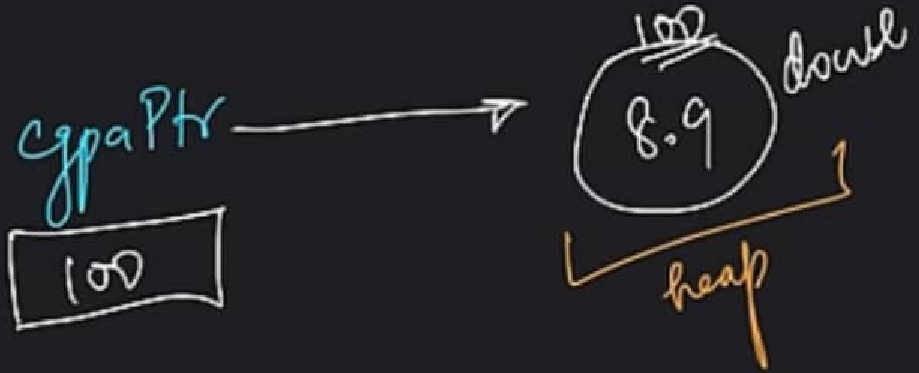
A deep copy, on the other hand, not only copies the member values but also makes copies of any dynamically allocated memory that the members point to.

Shallow & Deep Copy

Dynamic Memory Allocation

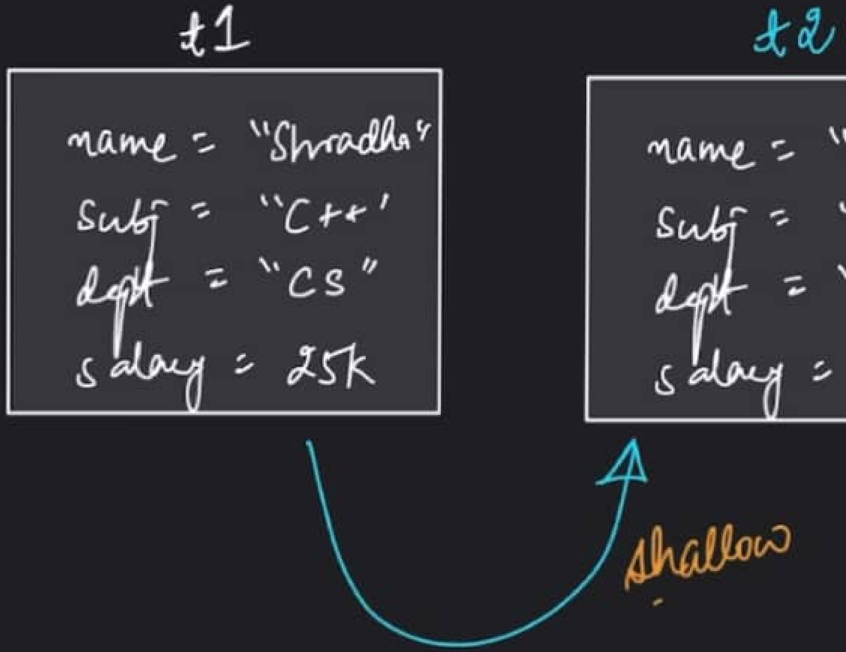
A shallow copy of an object copies all of the member values from one object to another.

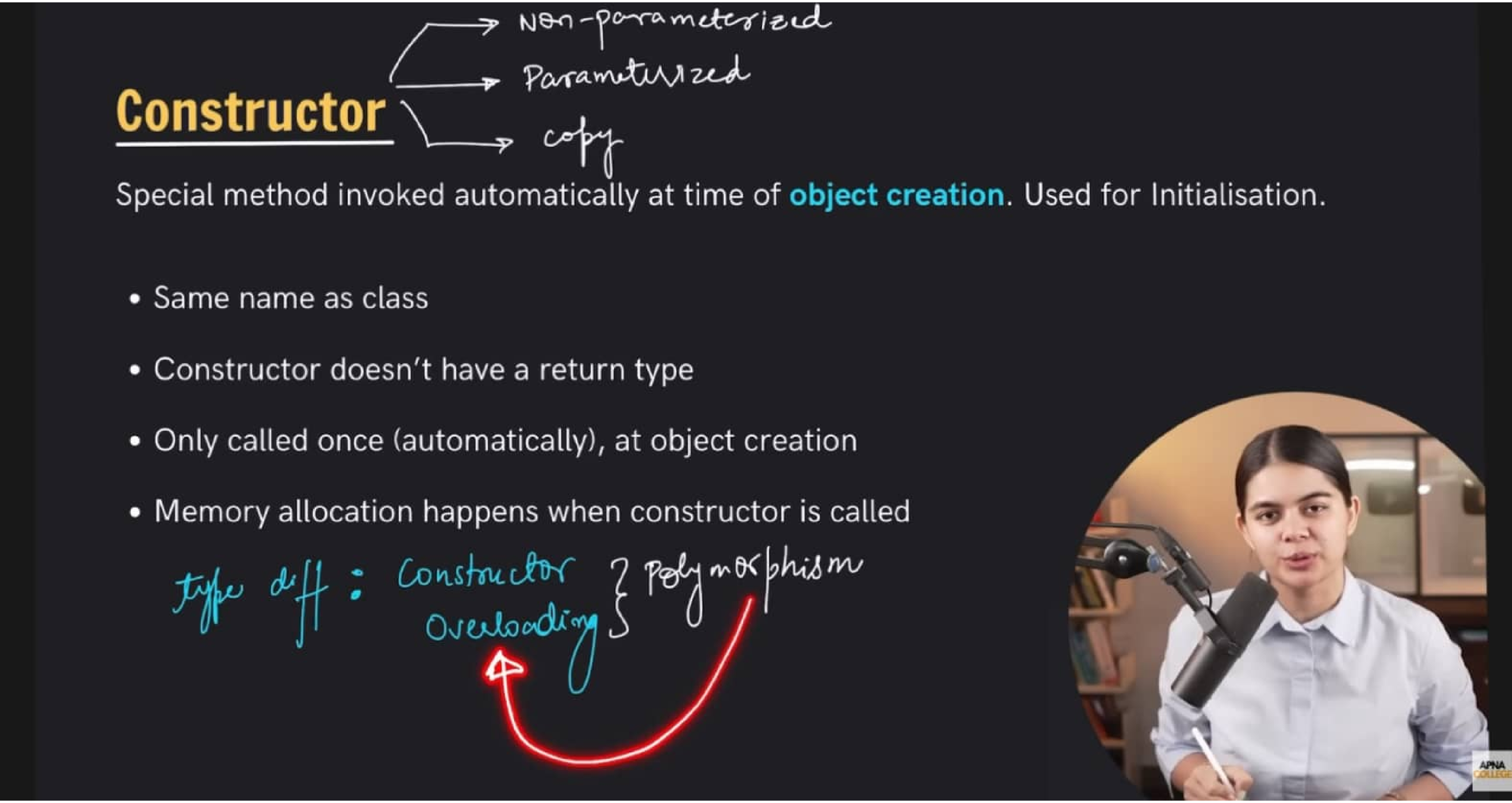
A deep copy, on the other hand, not only copies the member values but also makes copies of any dynamically allocated memory that the members point to.



Copy Constructor

Special Constructor (default) used to copy properties of one object into another.





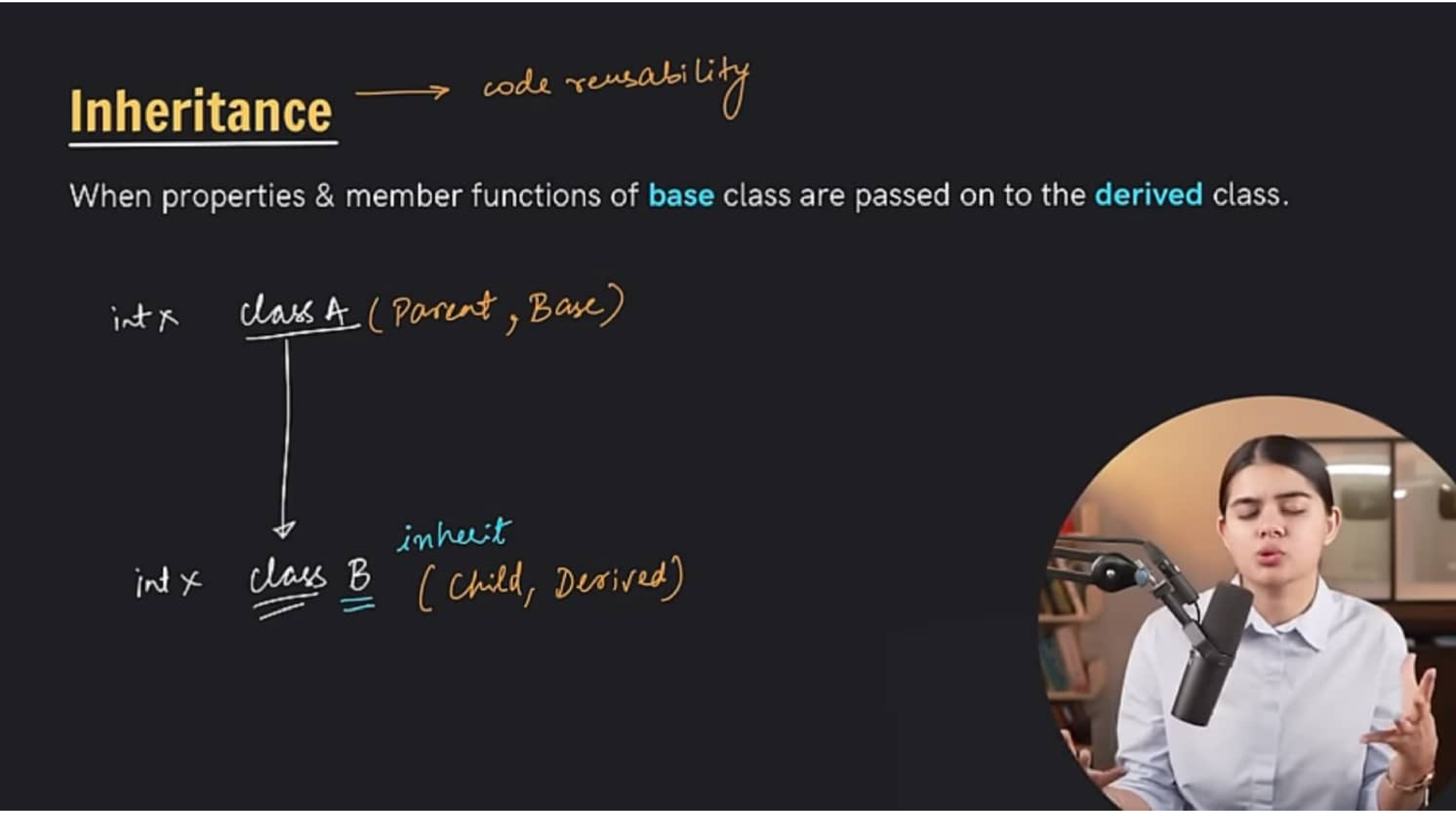
Constructor

→ Non-parameterized
→ Parametrized
→ copy

Special method invoked automatically at time of **object creation**. Used for Initialisation.

- Same name as class
- Constructor doesn't have a return type
- Only called once (automatically), at object creation
- Memory allocation happens when constructor is called

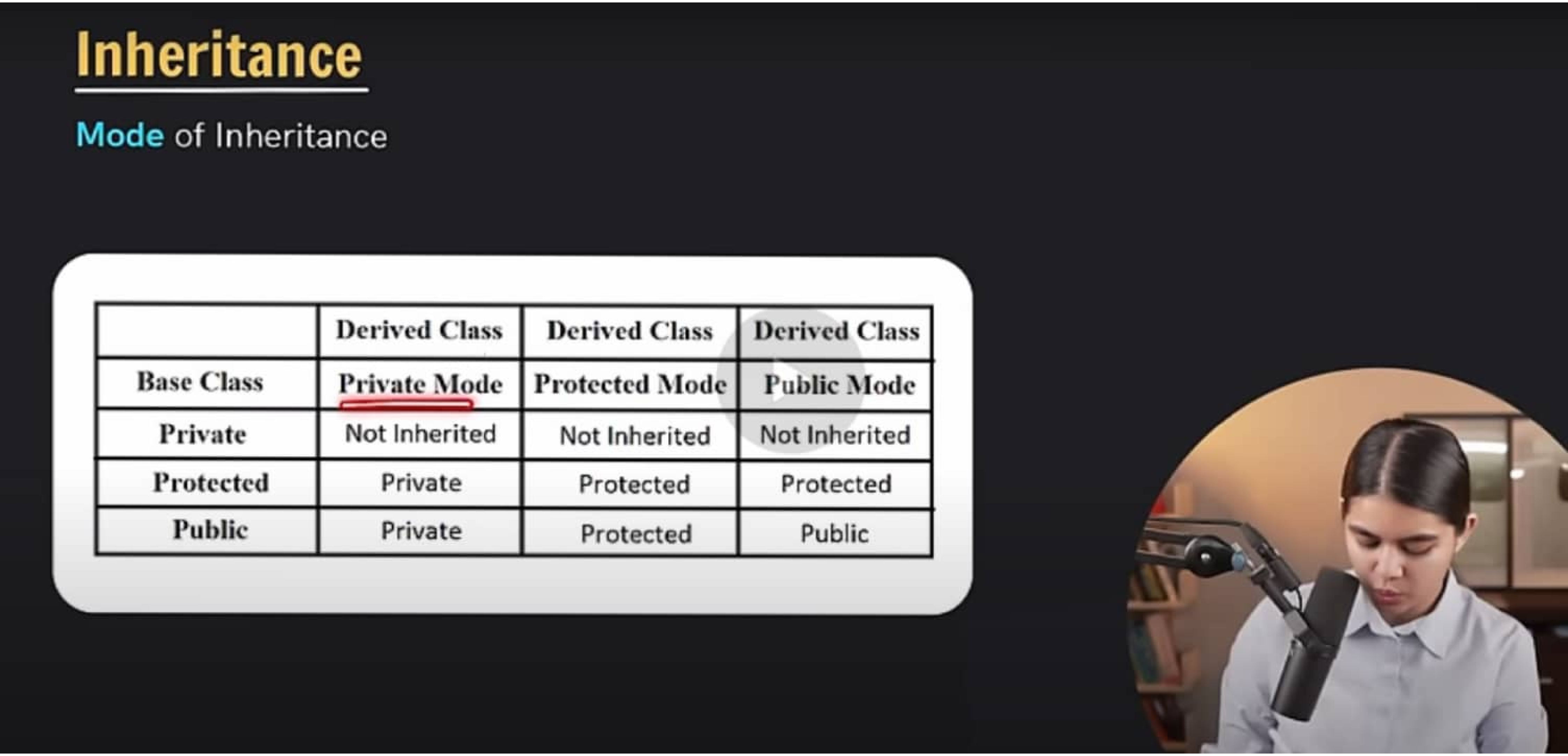
type diff : Constructor Overloading } Polymorphism



Inheritance $\longrightarrow$ code reusability

When properties & member functions of **base** class are passed on to the **derived** class.

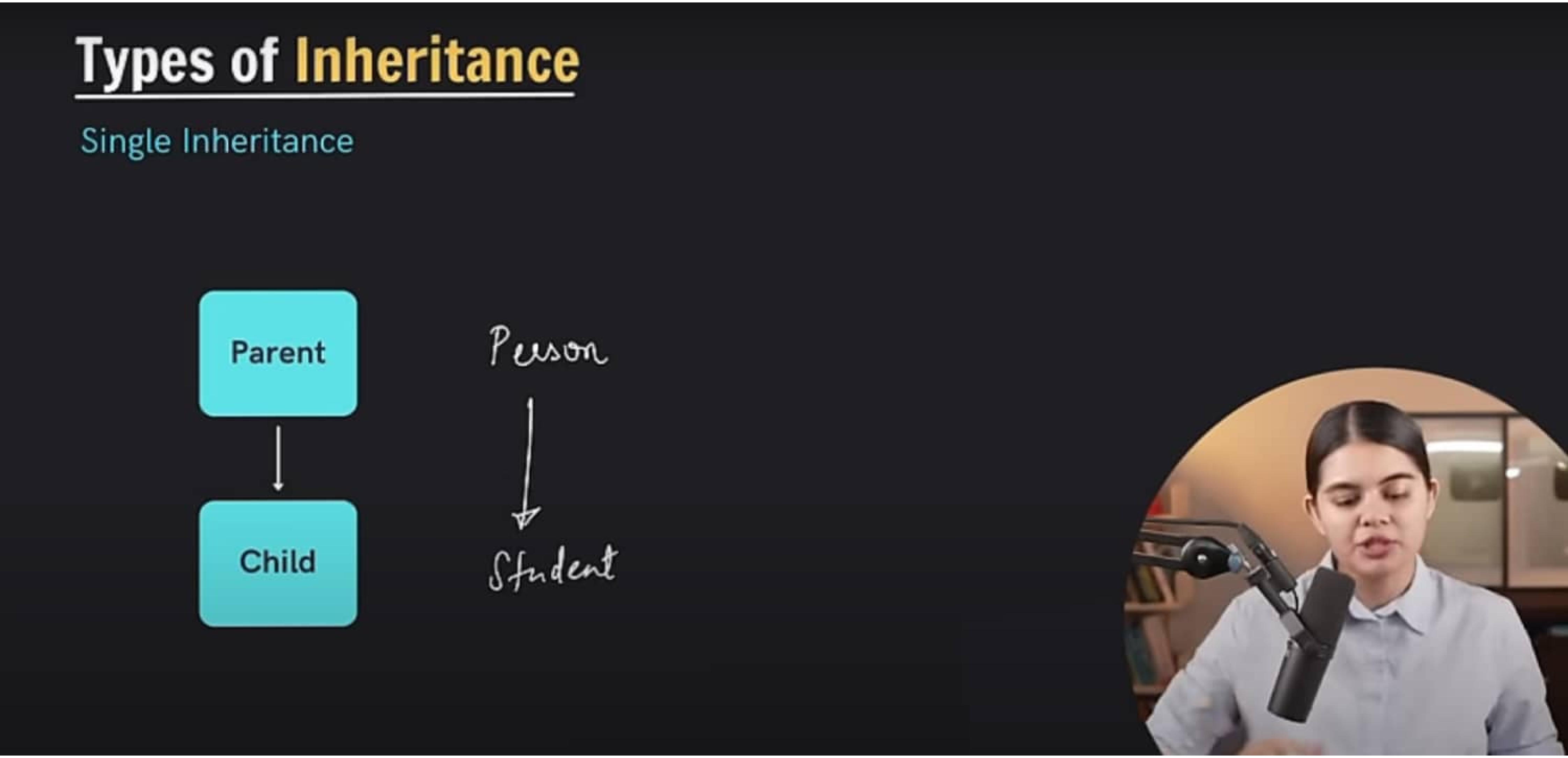
$$\begin{tikzcd}
\text{int x} & \text{class A} \arrow[d] & \text{(Parent, Base)} \\
\text{int x} & \text{class B} & \overset{\text{inherit}}{\text{(Child, Derived)}}
\end{tikzcd}$$



Inheritance

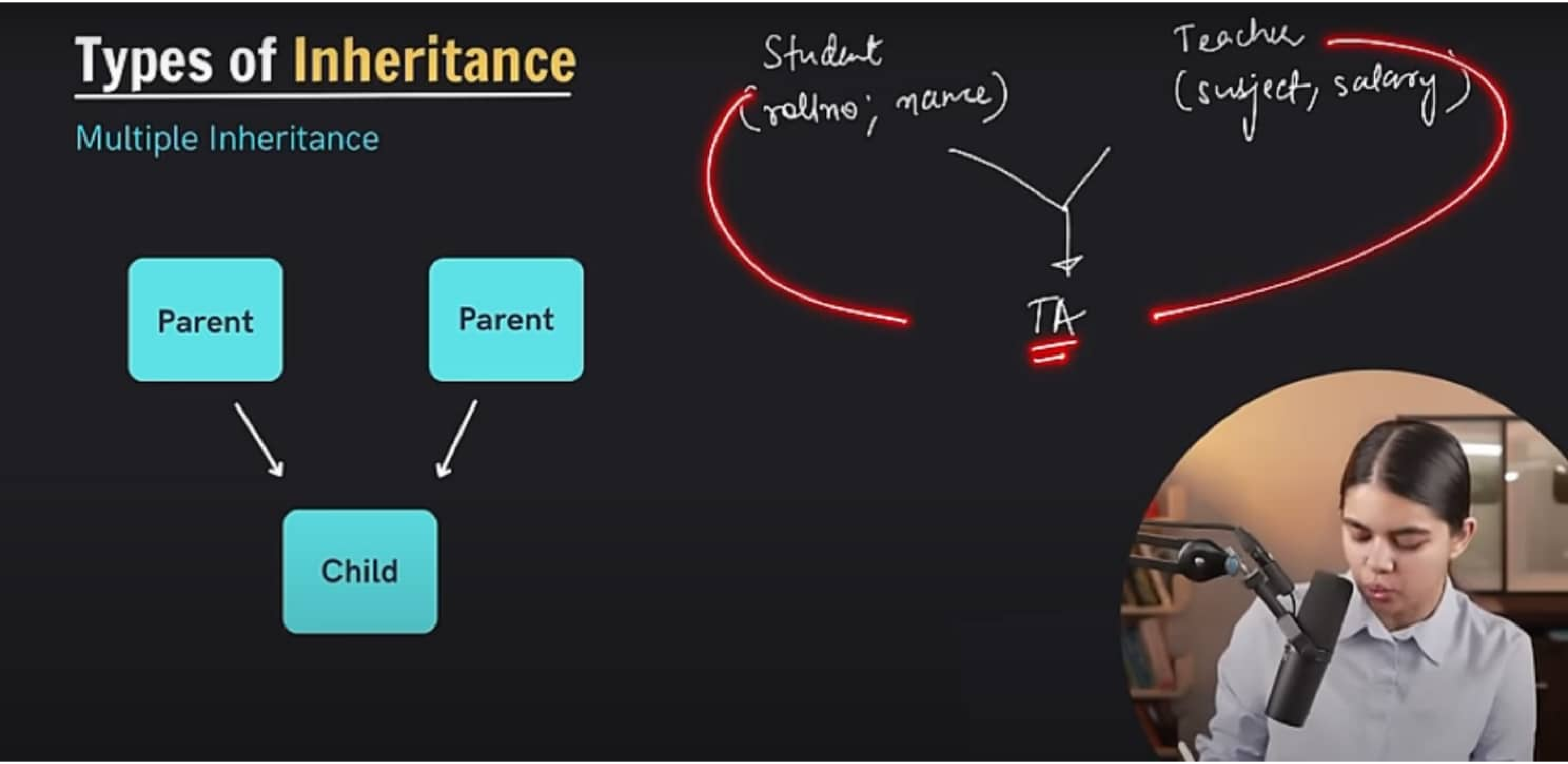
Mode of Inheritance

	Derived Class	Derived Class	Derived Class
Base Class	Private Mode	Protected Mode	Public Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public



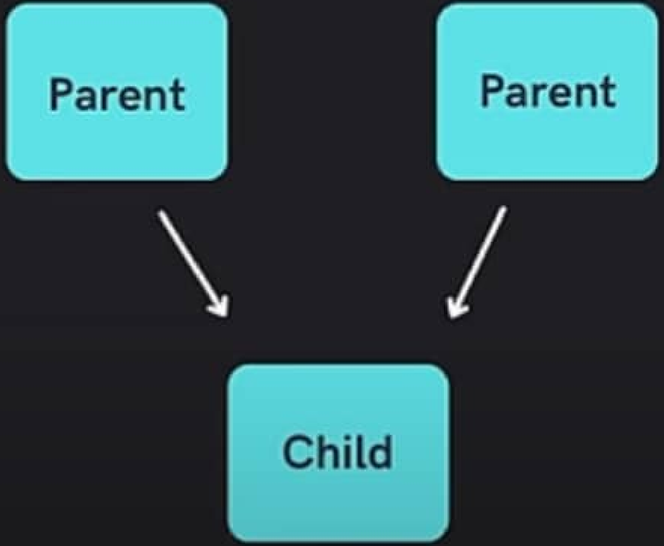
Types of Inheritance

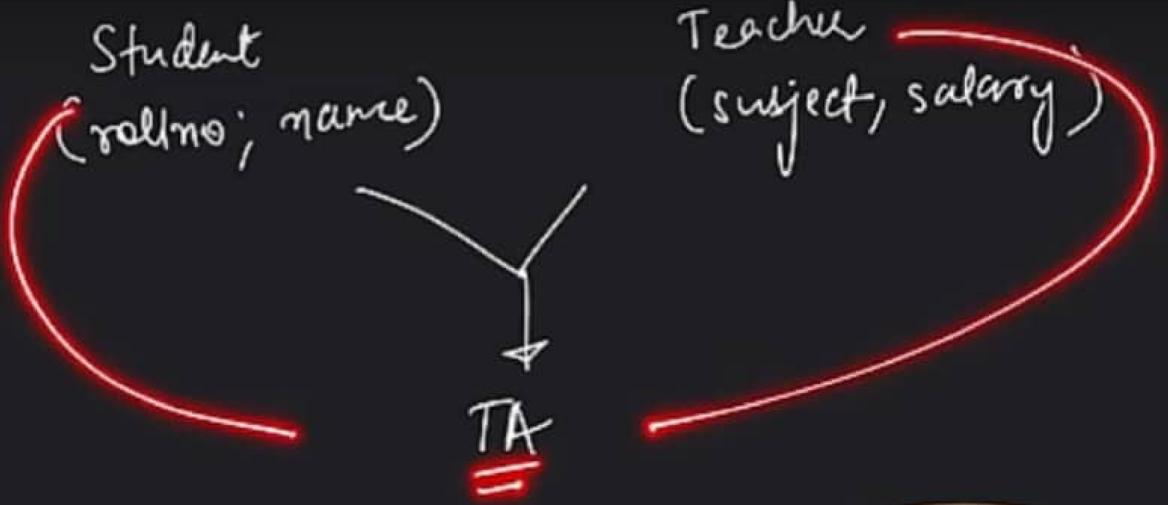
Single Inheritance



Types of Inheritance

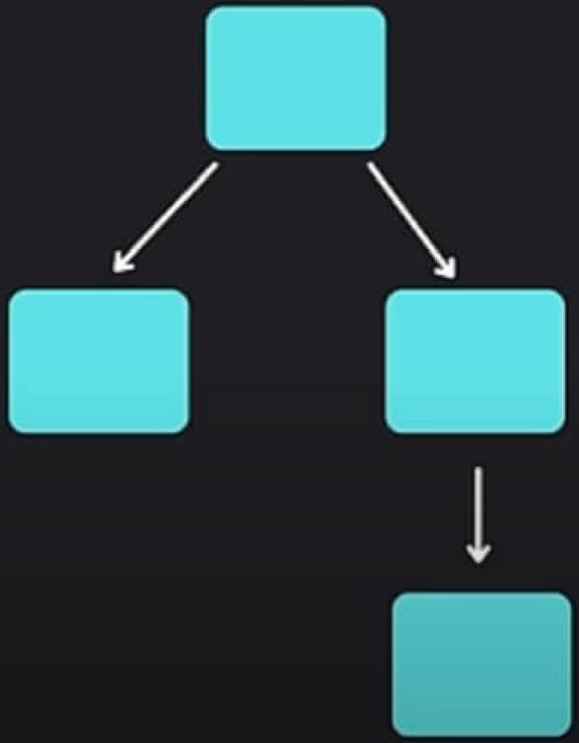
Multiple Inheritance





Types of Inheritance

Hybrid Inheritance

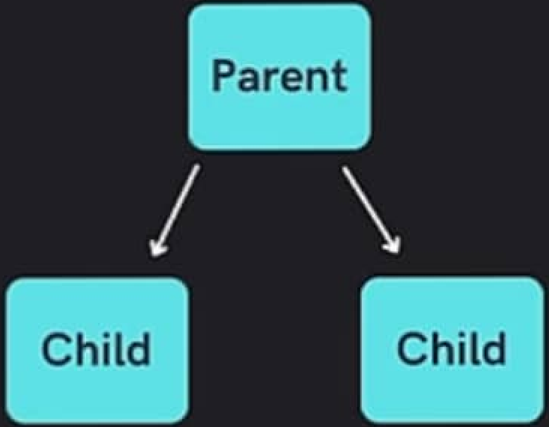


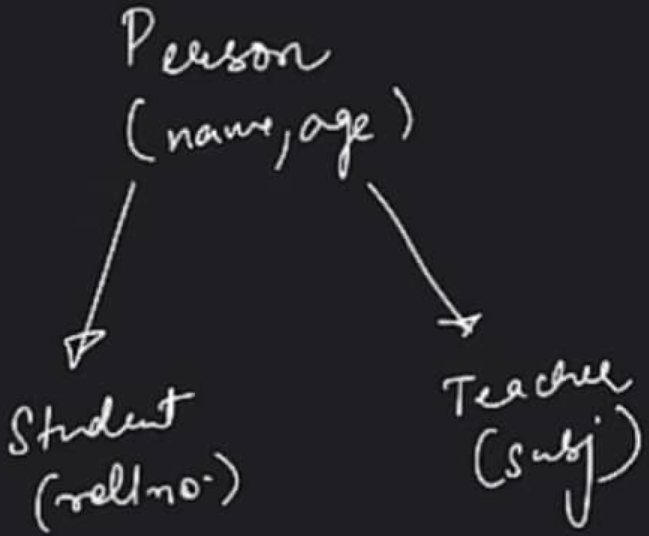
$$\begin{tikzcd}
\text{Student} \arrow[d] \arrow[dr] & \text{Teacher} \arrow[d] \\
\text{GradStudent} & \text{TA}
\end{tikzcd}$$

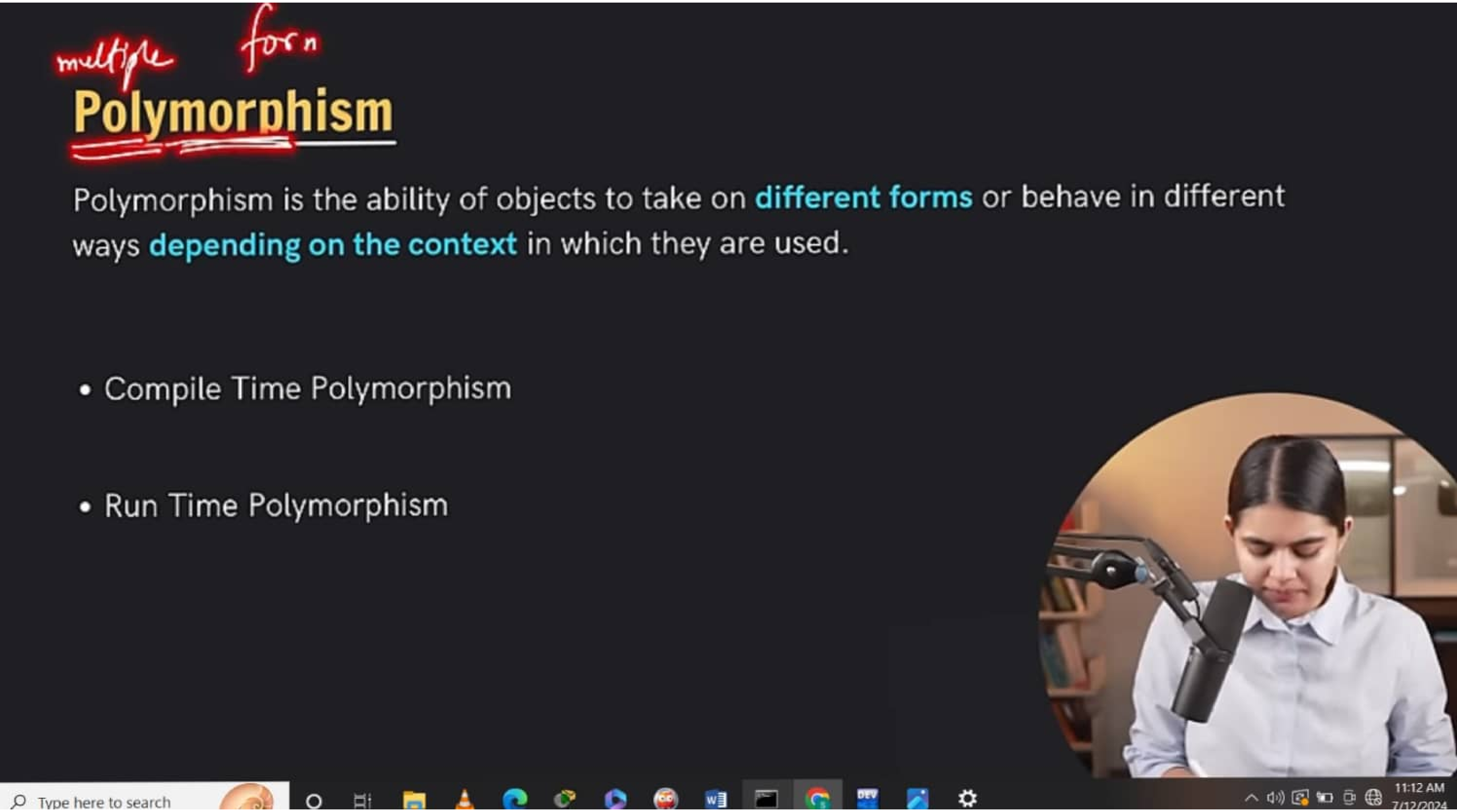
$\Big\}$ hybrid

Types of Inheritance

Hierarchial Inheritance





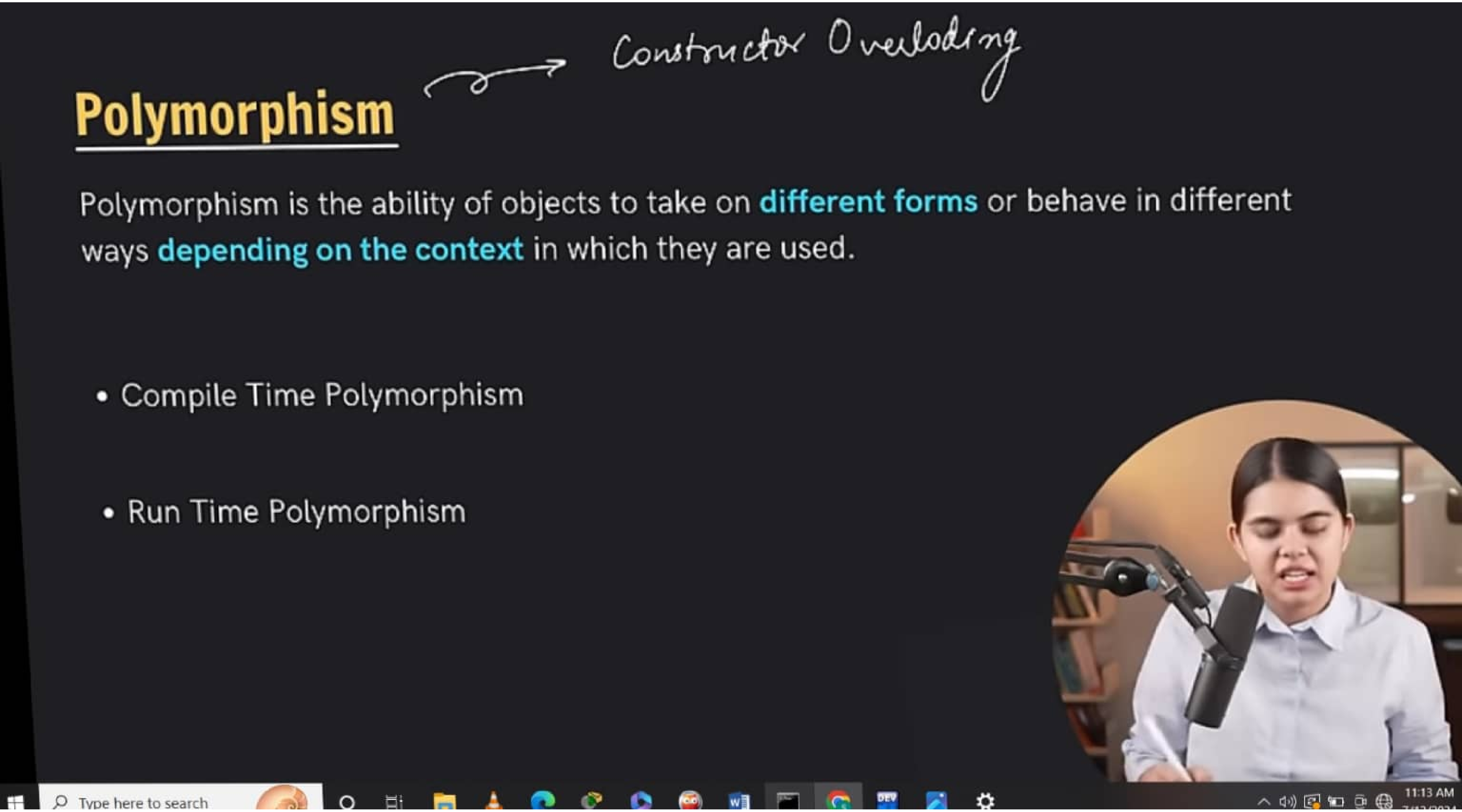


multiple form

Polymorphism

Polymorphism is the ability of objects to take on **different forms** or behave in different ways **depending on the context** in which they are used.

- Compile Time Polymorphism
- Run Time Polymorphism

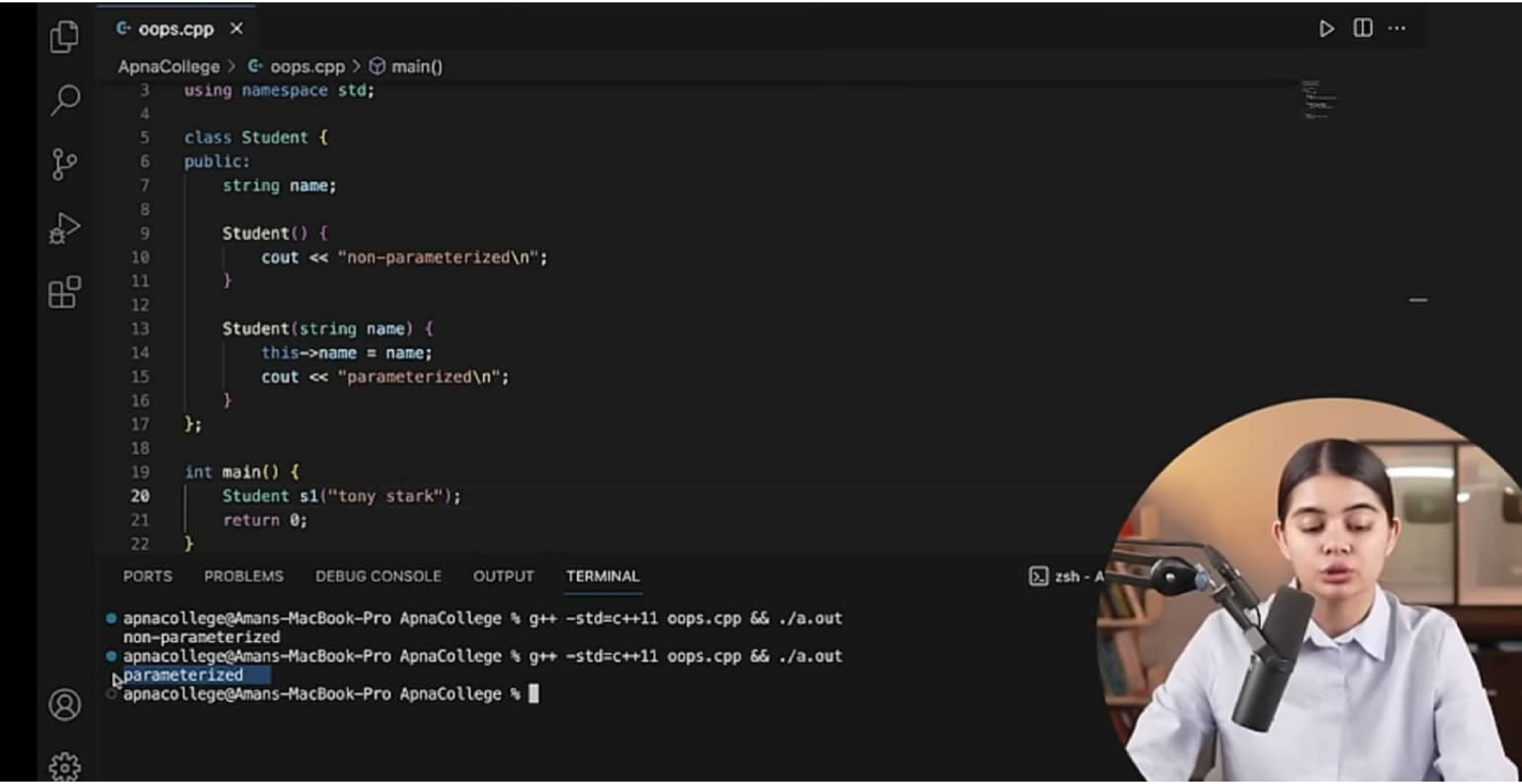


Polymorphism

$\longrightarrow$ Constructor Overloding

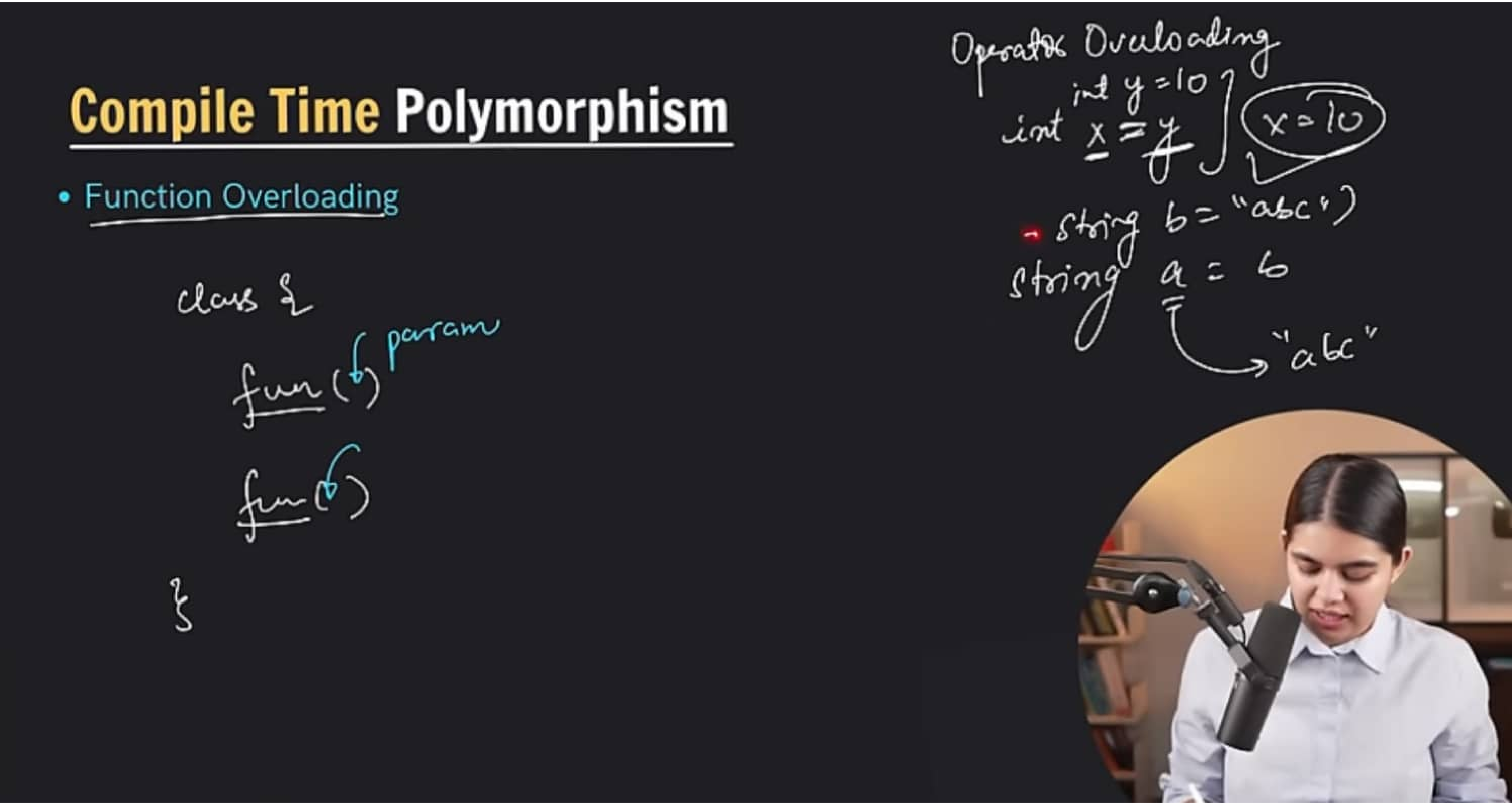
Polymorphism is the ability of objects to take on **different forms** or behave in different ways **depending on the context** in which they are used.

- Compile Time Polymorphism
- Run Time Polymorphism



```cpp
using namespace std;

class Student {
public:
    string name;

    Student() {
        cout << "non-parameterized\n";
    }

    Student(string name) {
        this->name = name;
        cout << "parameterized\n";
    }
};

int main() {
    Student s1("tony stark");
    return 0;
}
```

```
apnacollege@Amans-MacBook-Pro ApnaCollege % g++ -std=c++11 oops.cpp && ./a.out
non-parameterized
apnacollege@Amans-MacBook-Pro ApnaCollege % g++ -std=c++11 oops.cpp && ./a.out
parameterized
apnacollege@Amans-MacBook-Pro ApnaCollege %
```



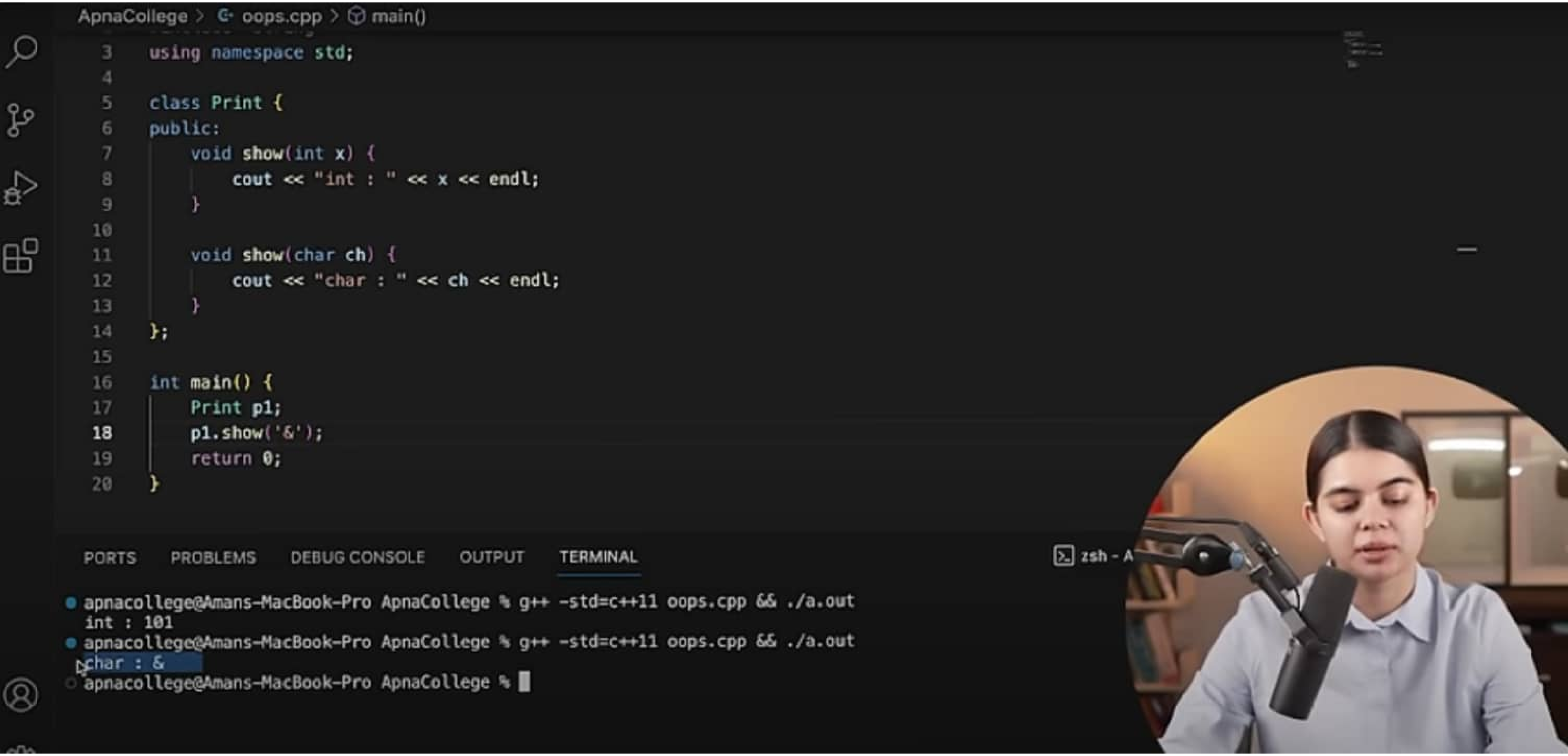
Compile Time Polymorphism

- Function Overloading

```
class {
    fun(↓) param

    fun(↓)
}
```

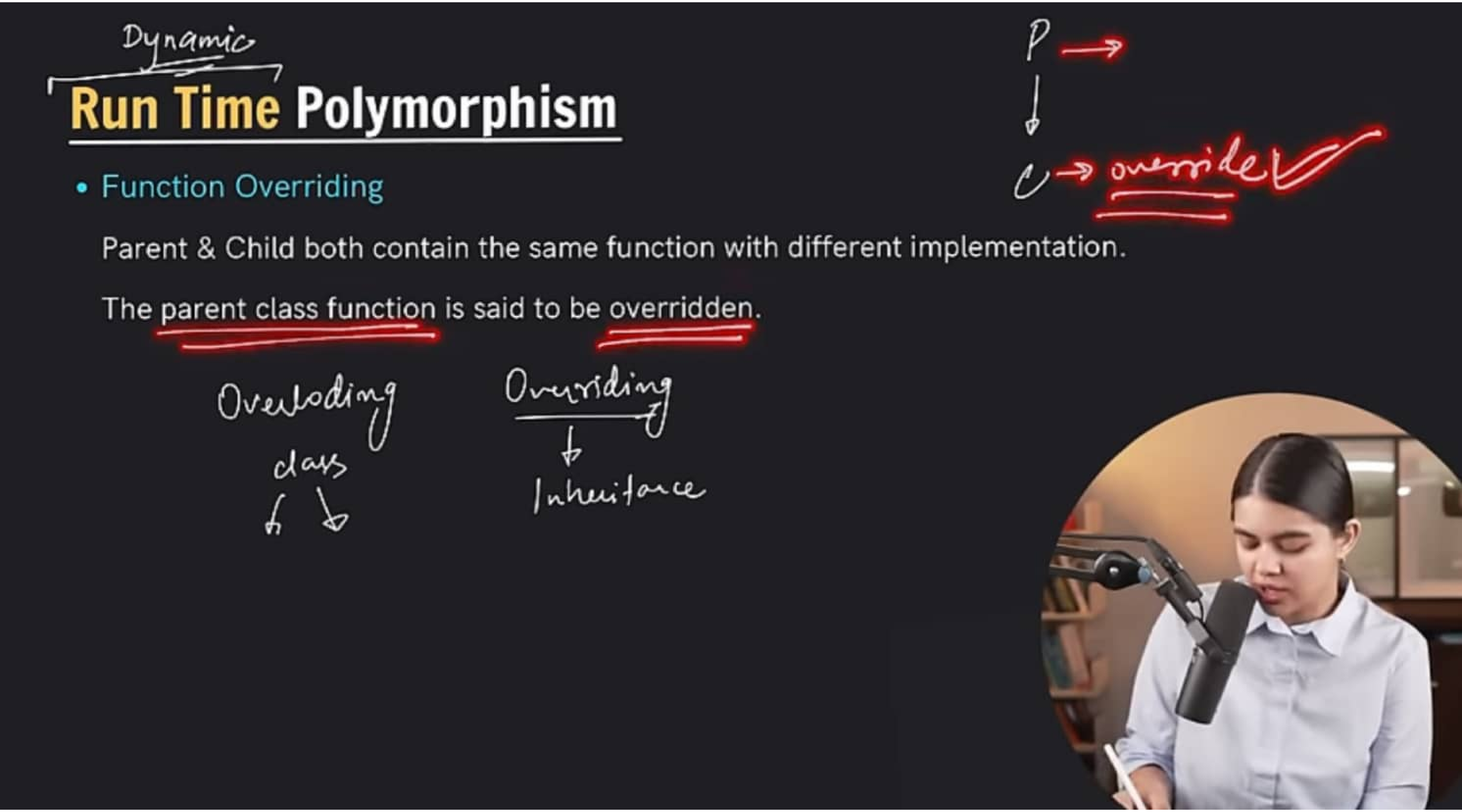
Operator Overloading

```
     int y = 10 ]
int  x = y      ]   x = 10

string b = "abc" )
string a = b
       a → "abc"
```



```cpp
using namespace std;

class Print {
public:
    void show(int x) {
        cout << "int : " << x << endl;
    }

    void show(char ch) {
        cout << "char : " << ch << endl;
    }
};

int main() {
    Print p1;
    p1.show('&');
    return 0;
}
```

```
apnacollege@Amans-MacBook-Pro ApnaCollege % g++ -std=c++11 oops.cpp && ./a.out
int : 101
apnacollege@Amans-MacBook-Pro ApnaCollege % g++ -std=c++11 oops.cpp && ./a.out
char : &
apnacollege@Amans-MacBook-Pro ApnaCollege % 
```



Dynamic

Run Time Polymorphism

$$\begin{tikzcd} P \arrow[r] \arrow[d] & {} \\ C \arrow[r] & \text{override} ✓ \end{tikzcd}$$

- Function Overriding

Parent & Child both contain the same function with different implementation.

The parent class function is said to be overridden.

Overloding

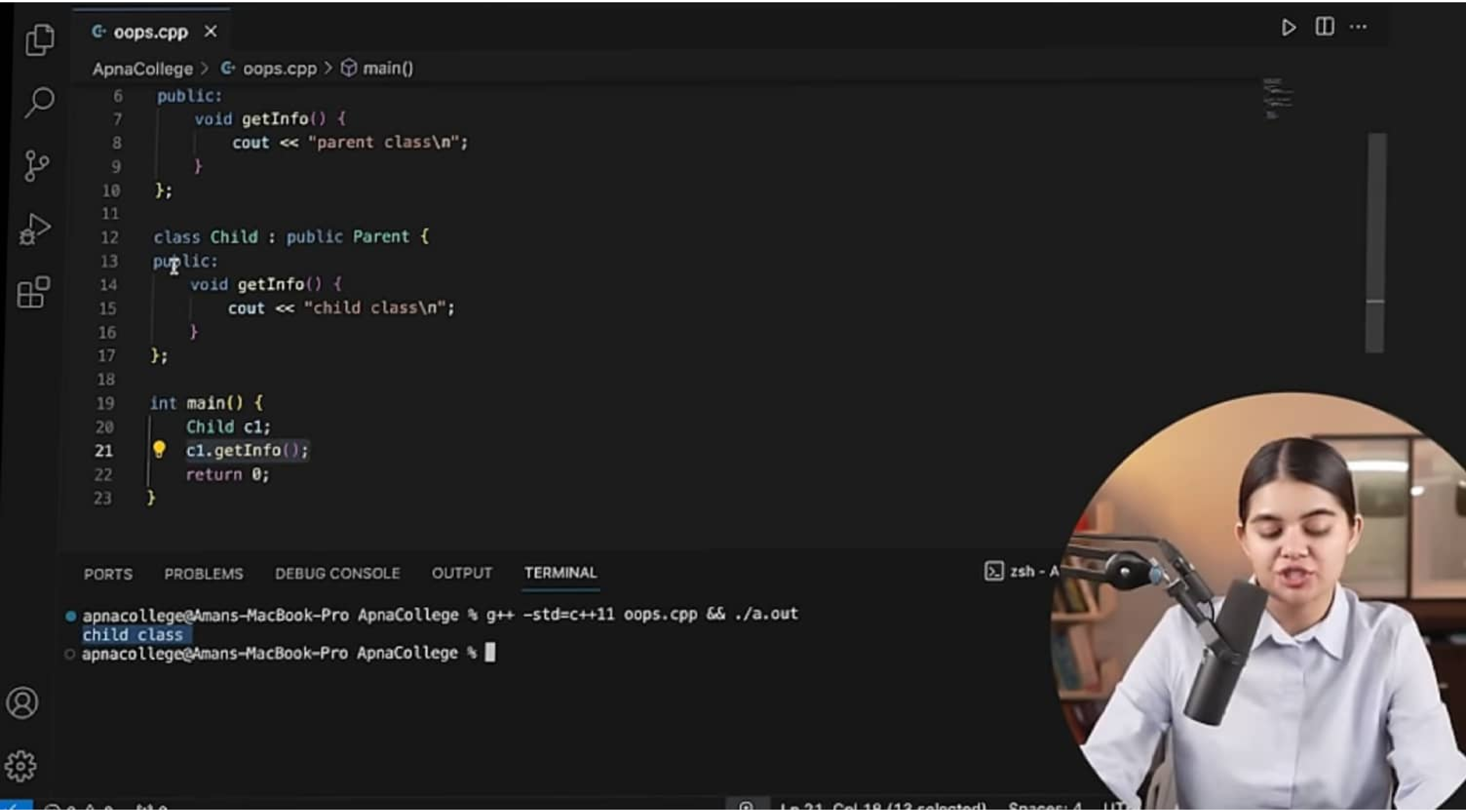
class

f f

Overriding

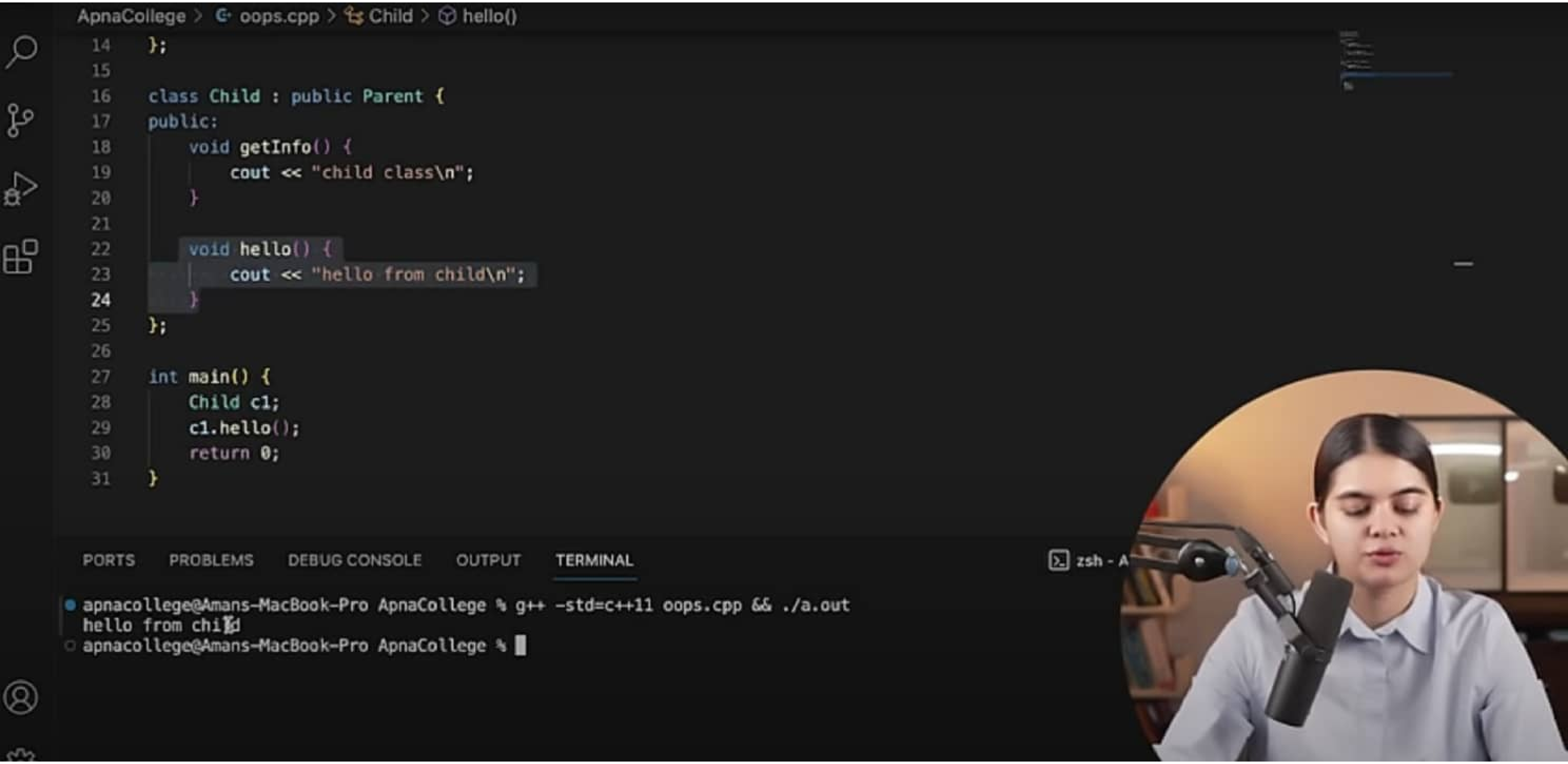
↓

Inheritance



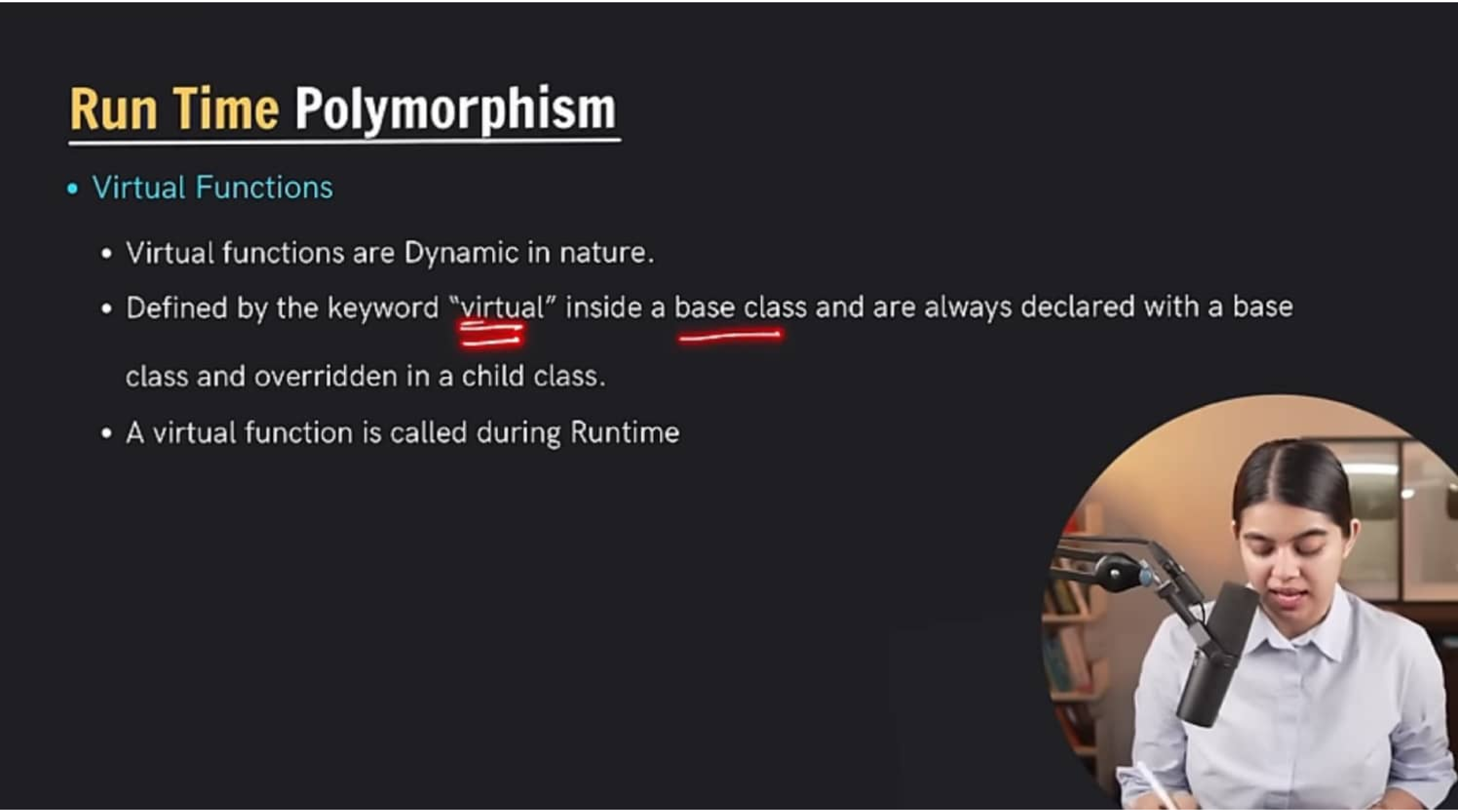
```cpp
public:
    void getInfo() {
        cout << "parent class\n";
    }
};

class Child : public Parent {
public:
    void getInfo() {
        cout << "child class\n";
    }
};

int main() {
    Child c1;
    c1.getInfo();
    return 0;
}
```

```
apnacollege@Amans-MacBook-Pro ApnaCollege % g++ -std=c++11 oops.cpp && ./a.out
child class
apnacollege@Amans-MacBook-Pro ApnaCollege %
```



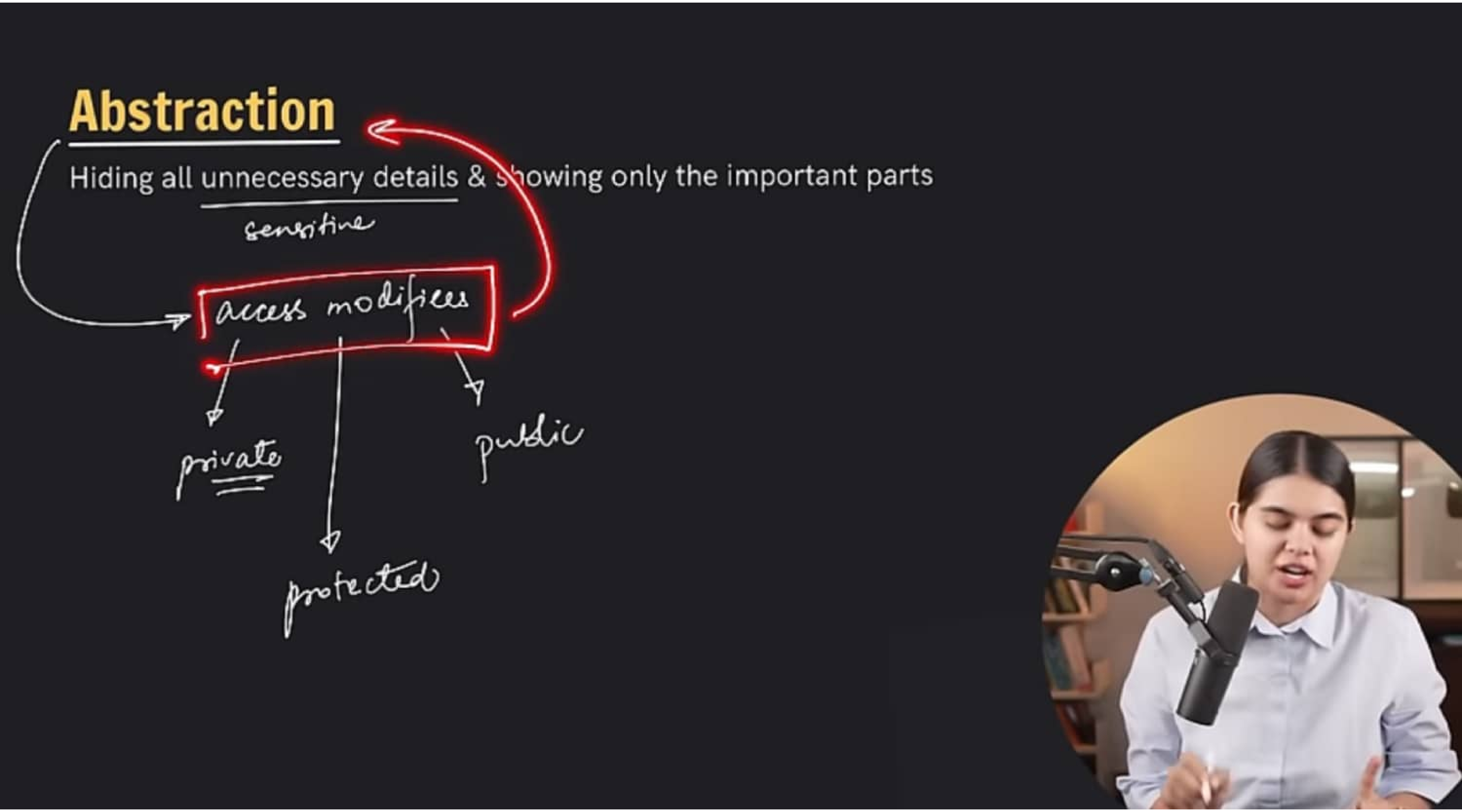
```cpp
};

class Child : public Parent {
public:
    void getInfo() {
        cout << "child class\n";
    }

    void hello() {
        cout << "hello from child\n";
    }
};

int main() {
    Child c1;
    c1.hello();
    return 0;
}
```

```
apnacollege@Amans-MacBook-Pro ApnaCollege % g++ -std=c++11 oops.cpp && ./a.out
hello from child
apnacollege@Amans-MacBook-Pro ApnaCollege %
```



Run Time Polymorphism

- Virtual Functions
  - Virtual functions are Dynamic in nature.
  - Defined by the keyword "virtual" inside a base class and are always declared with a base class and overridden in a child class.
  - A virtual function is called during Runtime

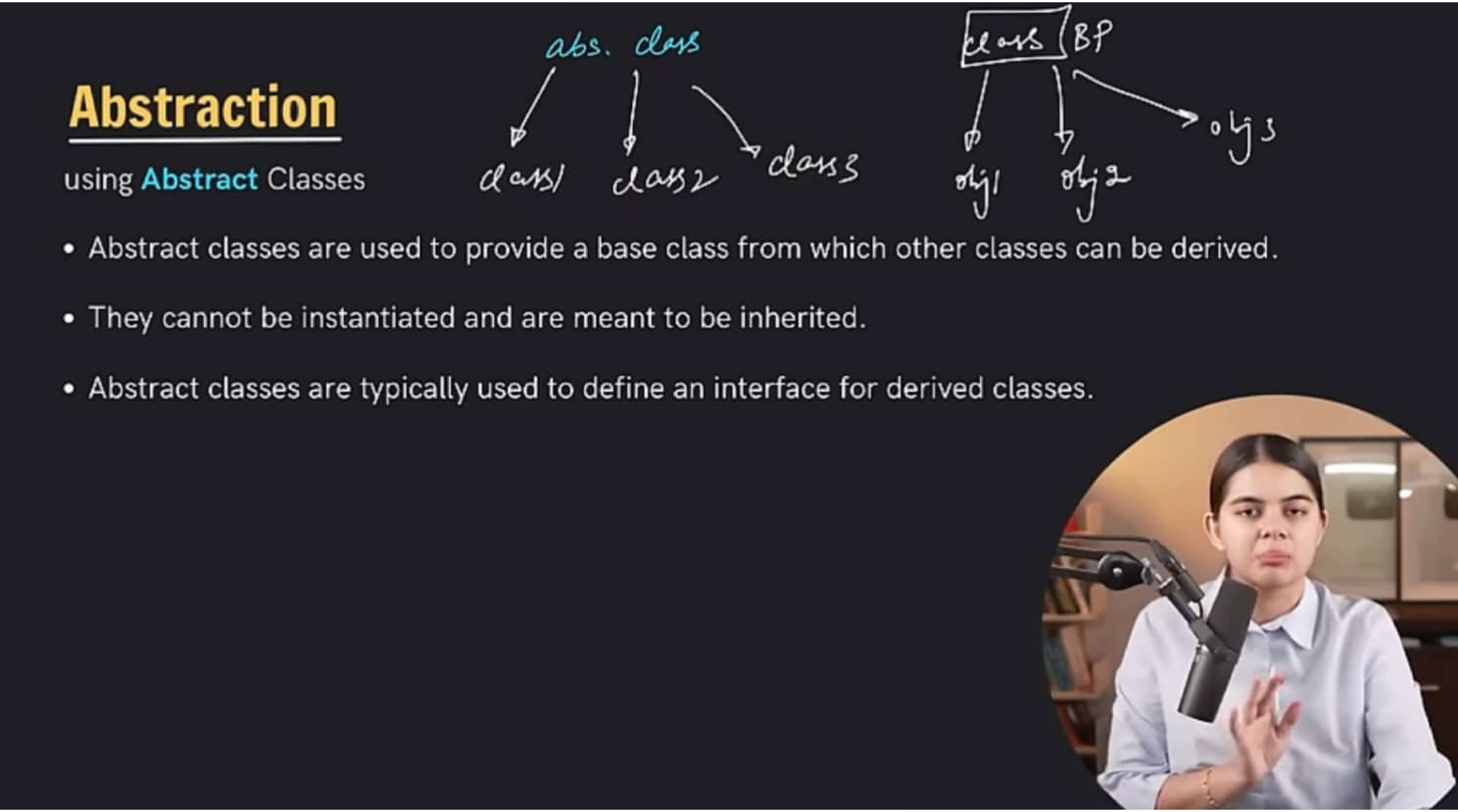


Abstraction

Hiding all <u>unnecessary</u> details & showing only the important parts

sensitive

$$\begin{tikzcd}
& \boxed{\text{access modifiers}} \arrow[dl] \arrow[d] \arrow[dr] & \\
\underline{\underline{\text{private}}} & \text{protected} & \text{public}
\end{tikzcd}$$



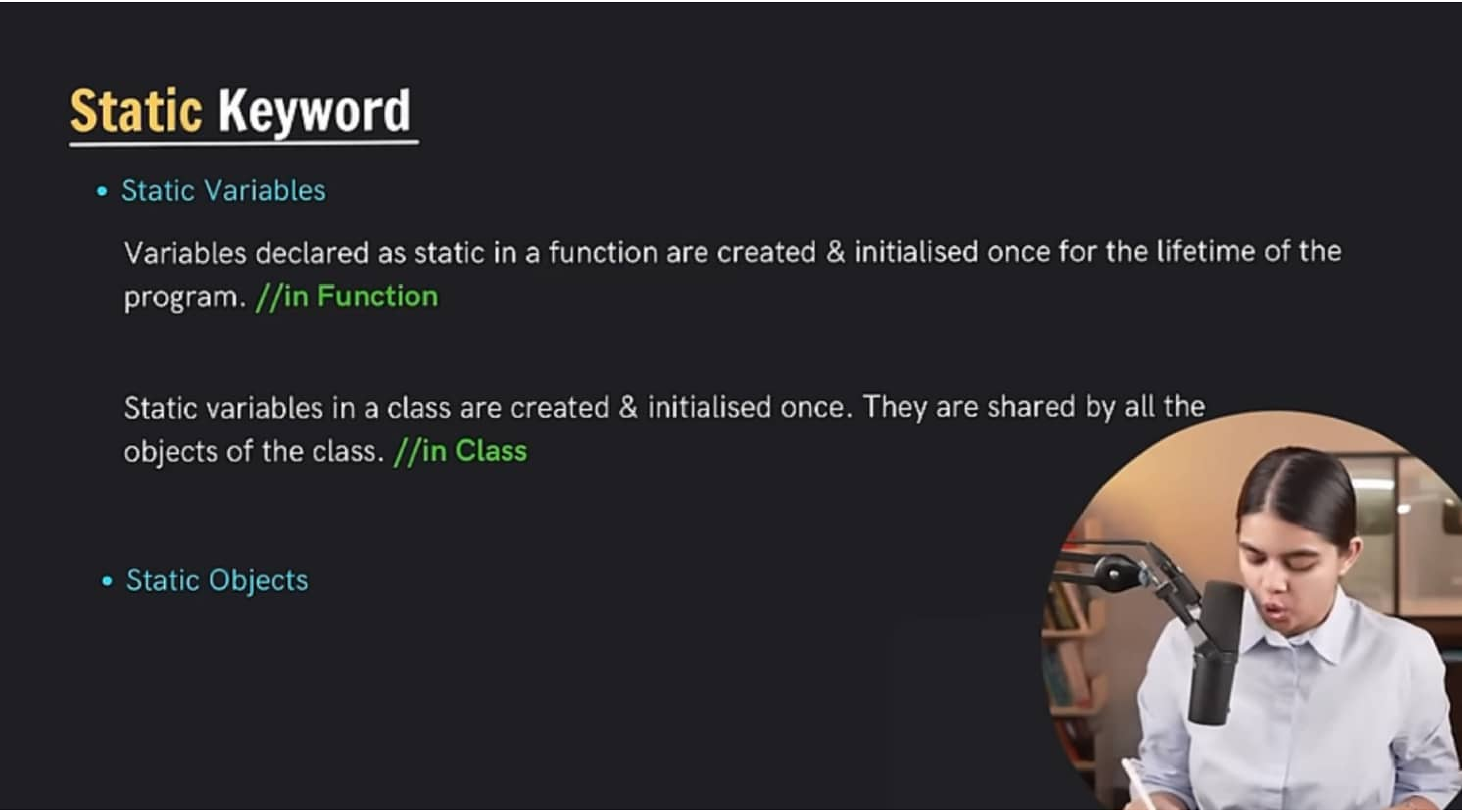
Abstraction

using **Abstract** Classes

$$\begin{tikzcd} & \text{abs. class} \arrow[dl] \arrow[d] \arrow[dr] & \\ \text{class1} & \text{class2} & \text{class3} \end{tikzcd}$$

$$\begin{tikzcd} & \boxed{\text{class}}\ \text{BP} \arrow[dl] \arrow[d] \arrow[dr] & \\ \text{obj1} & \text{obj2} & \text{obj3} \end{tikzcd}$$

- Abstract classes are used to provide a base class from which other classes can be derived.
- They cannot be instantiated and are meant to be inherited.
- Abstract classes are typically used to define an interface for derived classes.



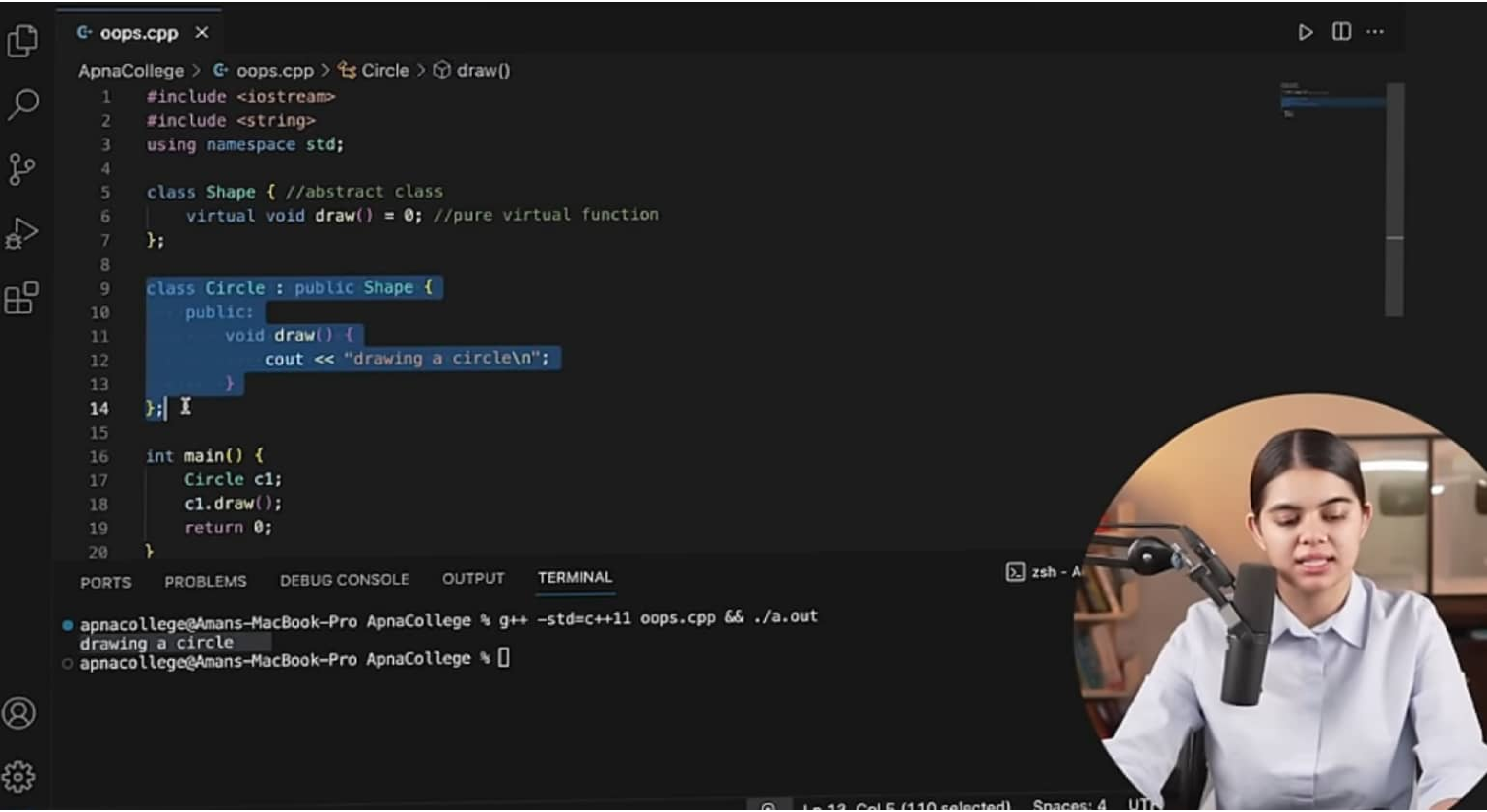
Static Keyword

- Static Variables

  Variables declared as static in a function are created & initialised once for the lifetime of the program. //in Function

  Static variables in a class are created & initialised once. They are shared by all the objects of the class. //in Class

- Static Objects



oops.cpp
ApnaCollege > oops.cpp > Circle > draw()
PORTS
PROBLEMS
DEBUG CONSOLE
OUTPUT
TERMINAL
zsh - A
Ln 13, Col 5 (110 selected)
Spaces: 4
UTF

```cpp
#include <iostream>
#include <string>
using namespace std;

class Shape { //abstract class
    virtual void draw() = 0; //pure virtual function
};

class Circle : public Shape {
    public:
        void draw() {
            cout << "drawing a circle\n";
        }
};

int main() {
    Circle c1;
    c1.draw();
    return 0;
}
```

```
apnacollege@Amans-MacBook-Pro ApnaCollege % g++ -std=c++11 oops.cpp && ./a.out
drawing a circle
apnacollege@Amans-MacBook-Pro ApnaCollege % 
```