OOPS
ABSTRACTION &
ENCAPSULATION
#2
54:01

Tuesday
13-05-2025
May

DAY - 2
LECTURE - 2

Hardwork always pay off

Date
Page

OOPs Real-World Examples | OOPs Pillars

→ History of Programming

Machine language

Use Binary Code (0/1) to interact with CPU
→ 0011 11001

Drawbacks:
→ All code will be in 0/1 which is prone to error
→ Tedious (Difficult to write)
→ Scalable X Can't think to make system

Assembly language

→ Enchament better than Machine
→ Use mnemonics means english Keyword
→ Understandable but tightly couple with hardware

Example: MOV A, 61H

Drawbacks:
→ Prone to Error
→ Scalable (If hardware change then code change)
→ Tedious little bit

Procedural

Introduce everything expect OOPs
→ functions
→ loops
→ Blocks (if else, Switch)

→ Do everything but expect making large scale application
→ like a recipe book
→ Do this
→ then do this

object-Oriented Programming

→ Covers the drawbacks of Procedural
→ Real-world Modelling
→ Data Security
→ Scalable/Reusable

Real-World Example To Understand OOPS

Imagine a zomato, Uber, OlA clone.

These solve real-world problem. If we want to solve real world problem then we have to understand real world.

In real world everything is object and Object are interacting with each other.

Objects => Interact

Example: Seeing a laptop video on ~~youtube~~

→ Using mic for conversation

Problem will solve if we use real world in object in programming

Object is classified on two things

- Characteristics (unique identifier to recognize object)
- Behaviour (Methods or function they perform)

Example :-

Car → object

Characteristics
- Engine
- Brand
- Model
- Wheels

Behaviour
- start()
- stop()
- gearshift()
- accelerate()
- brake()

If we want to understand car in a programming terms then we use Class

```
Class Car {

    // Code

}
```

← Blueprint to make Car

Example:

```
Car *myCar =
        new Car()
```

→ What if we don't have OOPs?

Car → Brand
↳ Model
↳ IsEngineOn

→ start()
→ stop()
→ gearshift()

```
String brand;
String model;
bool isEngine On;

Start() {          stop() {
  // Code            // code
}                  }
```

Here we represent car using procedural

Owner owns the Car.

we don't have car, this entire code represent car

Like car we declare thing like that we have to declare for owner

Owner → String name;

Behaviour → Drive()

```
void drive(
    brand, model)
{
    Start();
    gear shift();
    accerlate();
}
```

If we want to show owner owns the car first we have to make owner behavior and characteristics and same for car and then we have to pass some parameters in drive method and call some methods in that

The above thing is only for one car but if we want to introduce another car then we have to agains make variables and functions

Procedural Programming is not highly scalable and reusable and don't have real-world modelling. ~~we~~ In previous code we can't say that it handle owners owns the car

→ In OOPs it will be easy

→ Car

```
Class Car {

    String brand;
    String model;
    bool isEngineOn;

    void start() { }
    void stop() { }

}
```

Through this we can create many car

```
Class Owner {

    Car car;
    String name

    void drive() {
        car.start();
        car.stop();
    }

}
```

→ Pass the car as object

Interaction

Owner

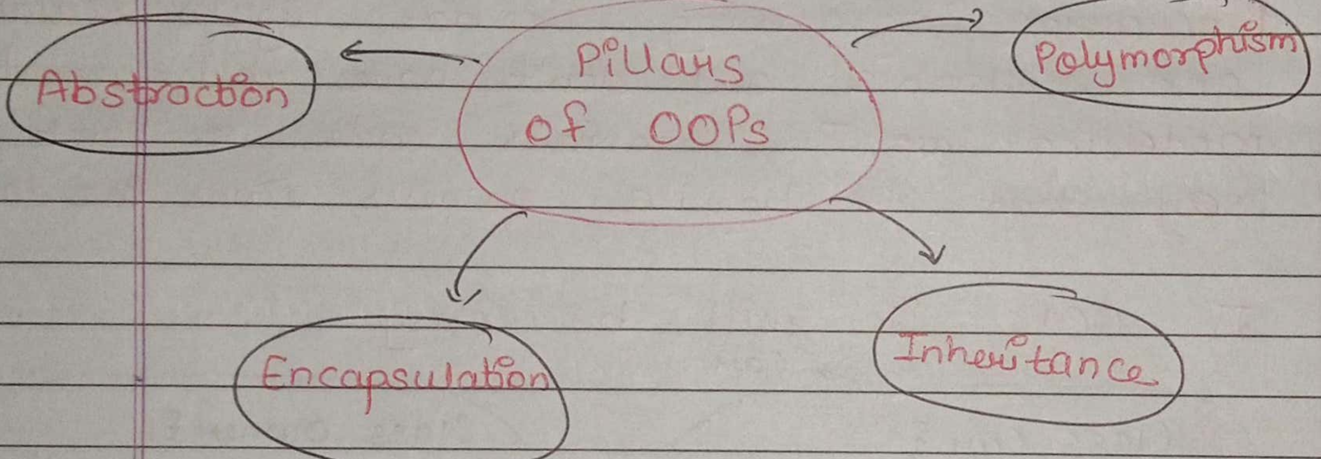
Car ← Owner

Owner owns the car

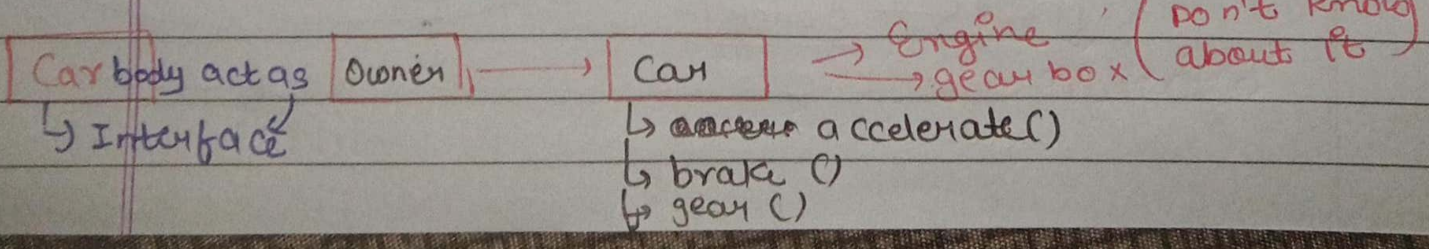
In simple terms, Imagine OOPs as a real-world which is interacting with many objects. Interaction is done via passing one object as a parameter or one object calling methods of another objects.

→ Pillars of OOPS

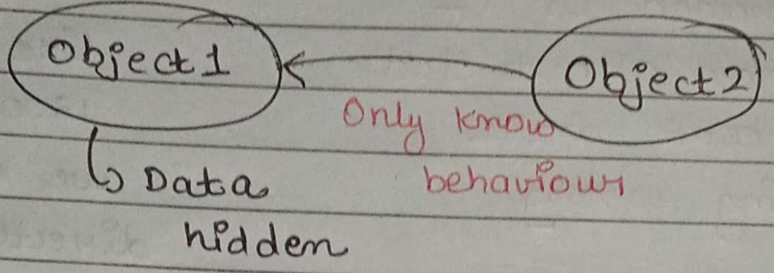
Representing a object is not enough. In real-life object perform actions so we have to map that also



i) Abstraction: Imagine you sit in a car, start the car, give accelrate, change gear but you doesn't know about the engine, gear box. Because car (how it work) provide a interface or acts as a interface you don't need unnecessary information



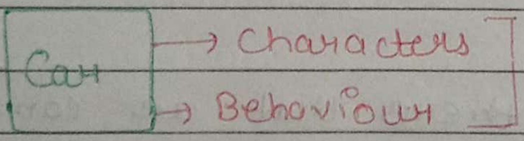
In real life object exist in ~~accelerated~~ abstracted form. Some data are hiddens. Another object don't know all information about other object.



Deffination : Abstraction hides unnecessary details from a client and showcase only what is necessary.

Example : High level Programming language like C++, Java we only use if(), for(), while but it will get converted into machine code. We don't have to know about how if() will be converted into binary.

ii) Encapsulation : (like a capsule)



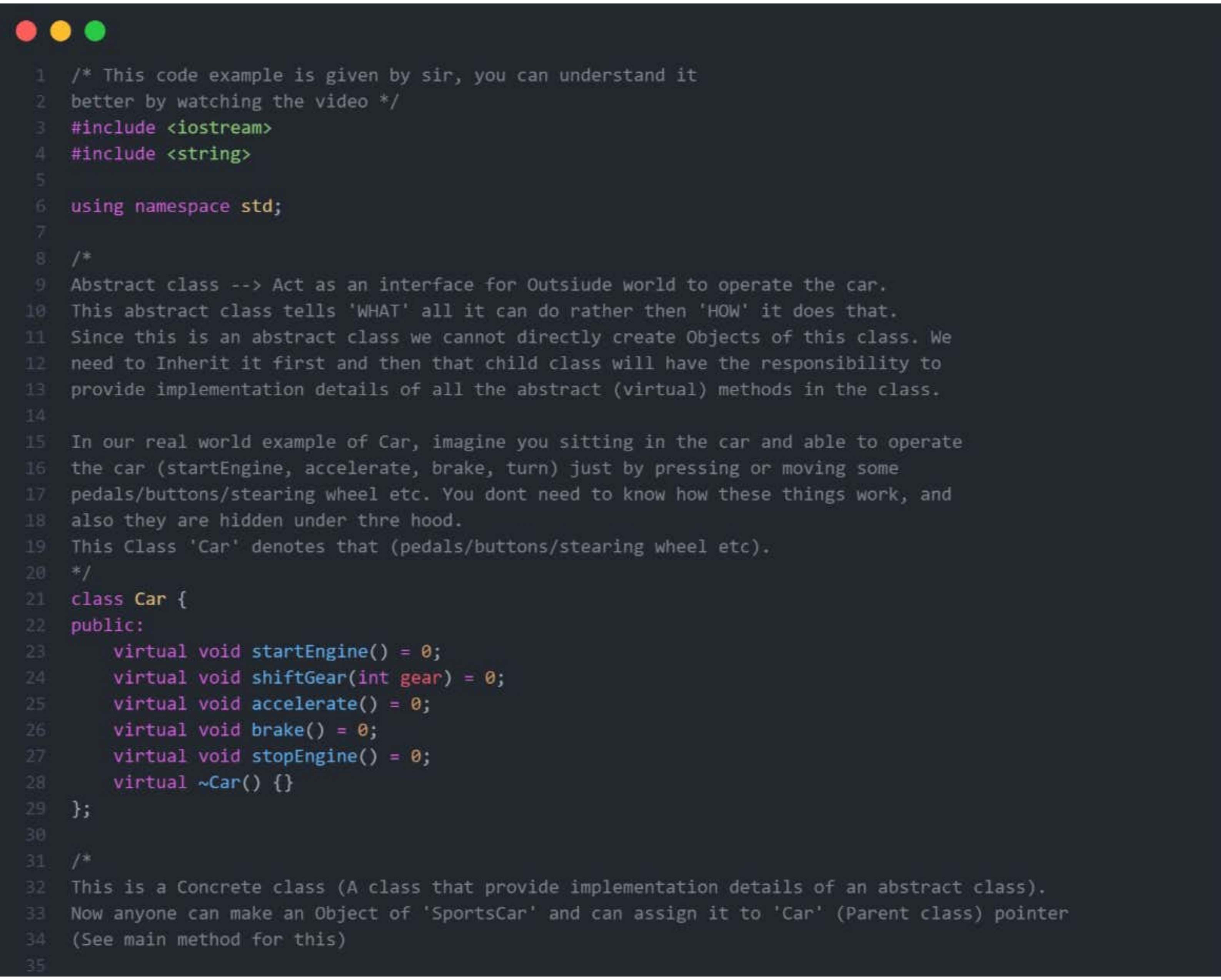
All are characters & behaviours of object combine in a box which we call as class

```
Class Car {
      //variables → Character
      //Methods → Behaviour
}
```

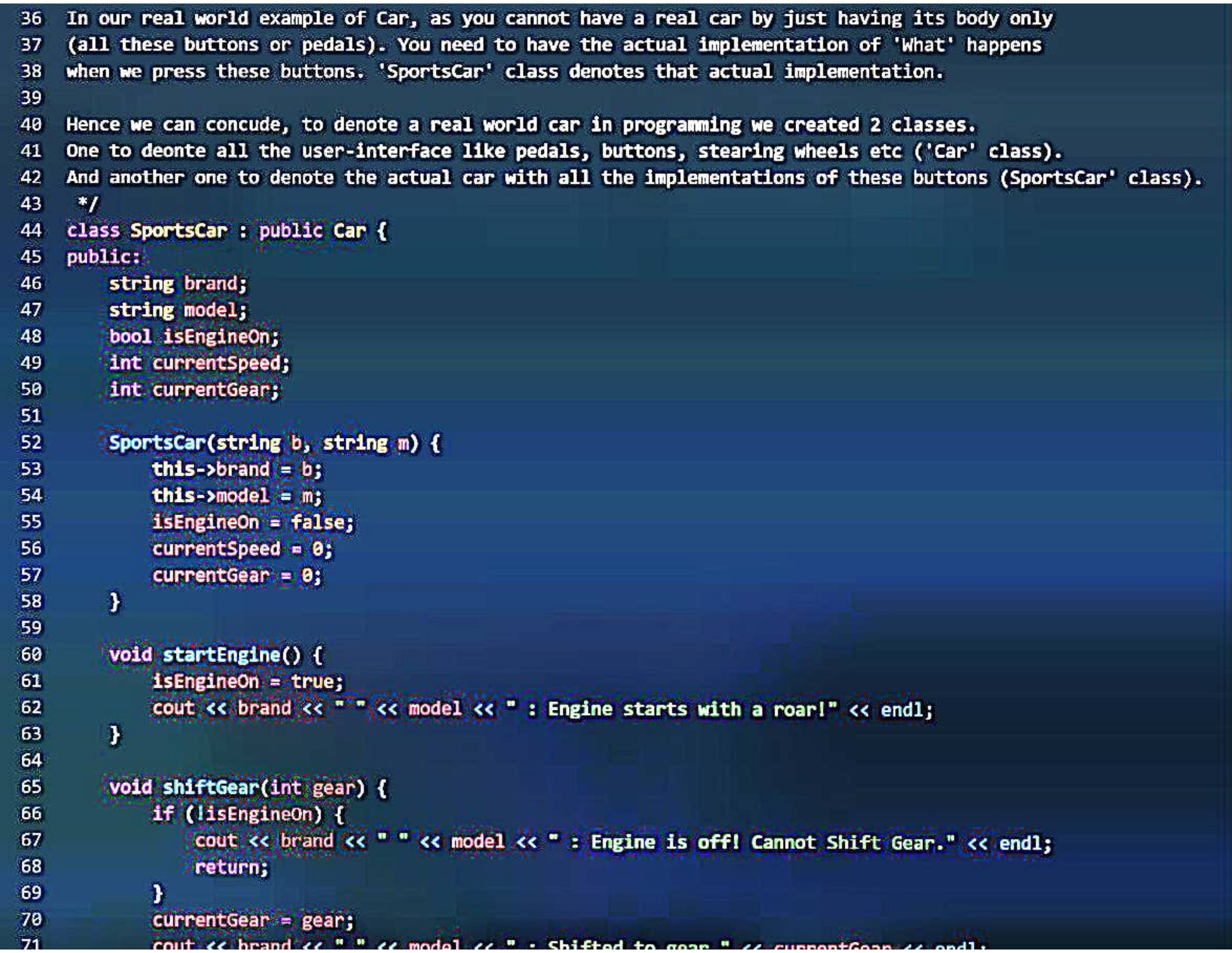
Important Point - Data Security

This is different from data hiding (Abstraction

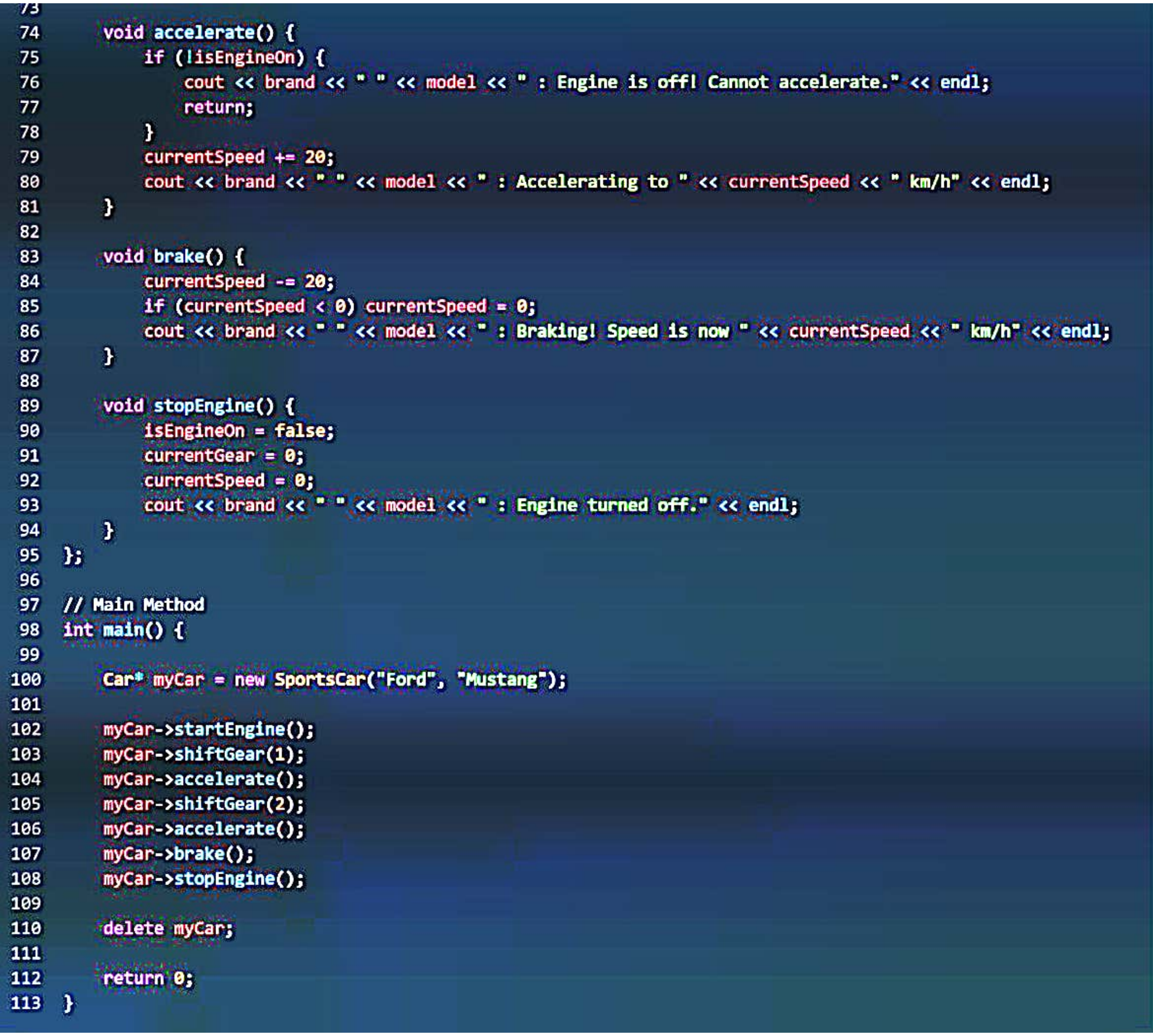
There is a Difference between Data hiding and Data security



```cpp
/* This code example is given by sir, you can understand it
better by watching the video */
#include <iostream>
#include <string>

using namespace std;

/*
Abstract class --> Act as an interface for Outsiude world to operate the car.
This abstract class tells 'WHAT' all it can do rather then 'HOW' it does that.
Since this is an abstract class we cannot directly create Objects of this class. We
need to Inherit it first and then that child class will have the responsibility to
provide implementation details of all the abstract (virtual) methods in the class.

In our real world example of Car, imagine you sitting in the car and able to operate
the car (startEngine, accelerate, brake, turn) just by pressing or moving some
pedals/buttons/stearing wheel etc. You dont need to know how these things work, and
also they are hidden under thre hood.
This Class 'Car' denotes that (pedals/buttons/stearing wheel etc).
*/
class Car {
public:
    virtual void startEngine() = 0;
    virtual void shiftGear(int gear) = 0;
    virtual void accelerate() = 0;
    virtual void brake() = 0;
    virtual void stopEngine() = 0;
    virtual ~Car() {}
};

/*
This is a Concrete class (A class that provide implementation details of an abstract class).
Now anyone can make an Object of 'SportsCar' and can assign it to 'Car' (Parent class) pointer
(See main method for this)

```



```
In our real world example of Car, as you cannot have a real car by just having its body only
(all these buttons or pedals). You need to have the actual implementation of 'What' happens
when we press these buttons. 'SportsCar' class denotes that actual implementation.

Hence we can concude, to denote a real world car in programming we created 2 classes.
One to deonte all the user-interface like pedals, buttons, stearing wheels etc ('Car' class).
And another one to denote the actual car with all the implementations of these buttons (SportsCar' class).
 */
class SportsCar : public Car {
public:
    string brand;
    string model;
    bool isEngineOn;
    int currentSpeed;
    int currentGear;

    SportsCar(string b, string m) {
        this->brand = b;
        this->model = m;
        isEngineOn = false;
        currentSpeed = 0;
        currentGear = 0;
    }

    void startEngine() {
        isEngineOn = true;
        cout << brand << " " << model << " : Engine starts with a roar!" << endl;
    }

    void shiftGear(int gear) {
        if (!isEngineOn) {
            cout << brand << " " << model << " : Engine is off! Cannot Shift Gear." << endl;
            return;
        }
        currentGear = gear;
        cout << brand << " " << model << " : Shifted to gear " << currentGear << endl;
```



```cpp
    void accelerate() {
        if (!isEngineOn) {
            cout << brand << " " << model << " : Engine is off! Cannot accelerate." << endl;
            return;
        }
        currentSpeed += 20;
        cout << brand << " " << model << " : Accelerating to " << currentSpeed << " km/h" << endl;
    }

    void brake() {
        currentSpeed -= 20;
        if (currentSpeed < 0) currentSpeed = 0;
        cout << brand << " " << model << " : Braking! Speed is now " << currentSpeed << " km/h" << endl;
    }

    void stopEngine() {
        isEngineOn = false;
        currentGear = 0;
        currentSpeed = 0;
        cout << brand << " " << model << " : Engine turned off." << endl;
    }
};

// Main Method
int main() {

    Car* myCar = new SportsCar("Ford", "Mustang");

    myCar->startEngine();
    myCar->shiftGear(1);
    myCar->accelerate();
    myCar->shiftGear(2);
    myCar->accelerate();
    myCar->brake();
    myCar->stopEngine();

    delete myCar;

    return 0;
}
```

Abstraction
(Data hiding)

Mean don't need to know unnecessary information or if we know about that it doesn't make anything wrong

Example:- If we want to know about engine we can learn that or about working of gear behind the scene.

Encapsulation
(Data security)

Car
↳ odometer

like car complete 15000 km. Can we sit and directly change it to 25000 km, no.

We can do ~~accelerate~~
accelerate () ↑↑
brake () ↑↑
odometer ↑ ↑

Encapsulation:

```
Class Car {

    //variables → Character
    // Methods  → Behaviour

}
```

Some variables are there that we can't access outside the car class so we learn about Access Modifiers

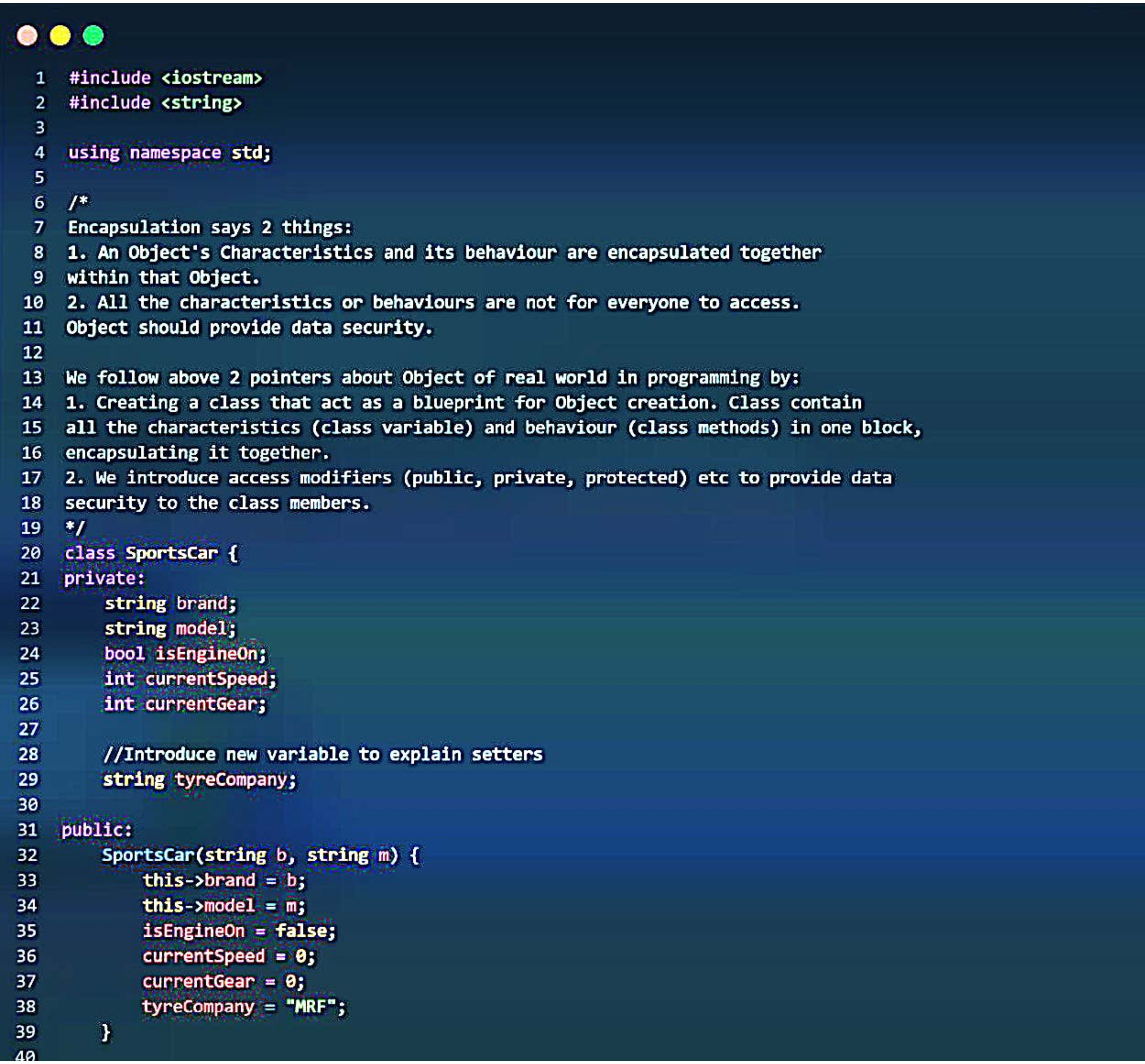
- Public
- Private
- Protected

Public: Anyone can access

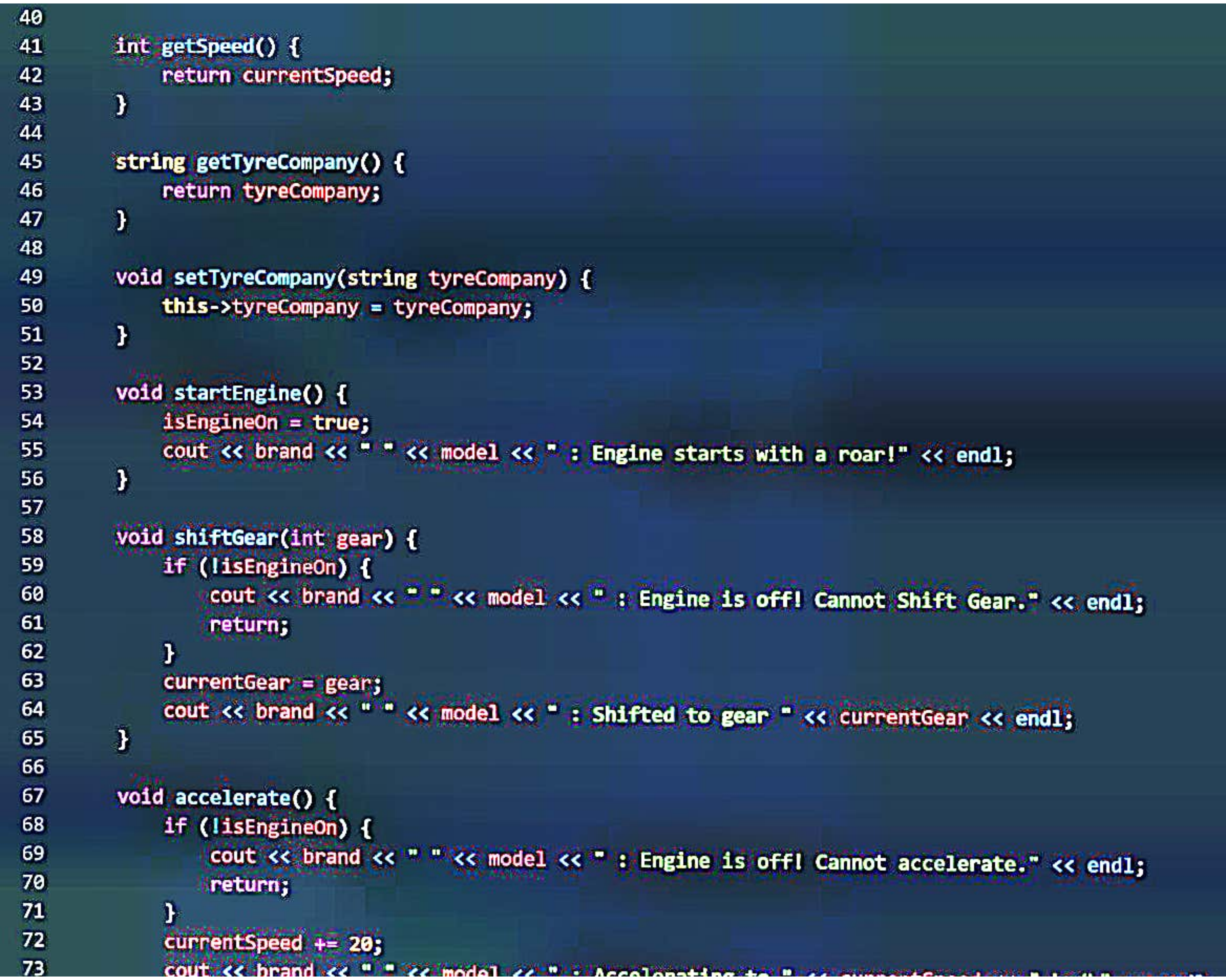
Example: AC of car : we can change temp by ourself

Private: Only child class can access
We declare variables as private and use the concept of getters and setters.

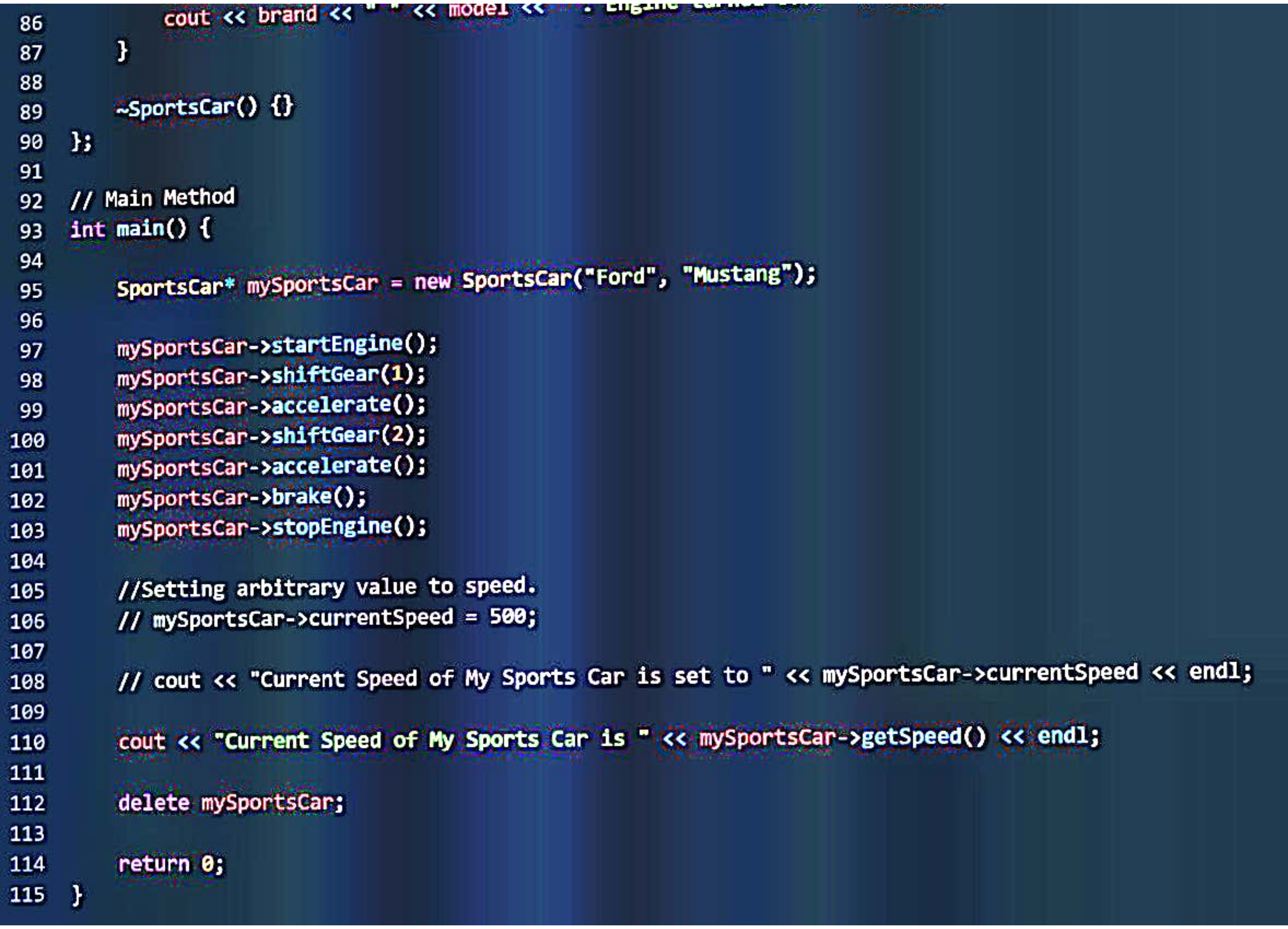
→ Because sometime we want user can see the value and before setting the user's value we can do some validations.



```cpp
#include <iostream>
#include <string>

using namespace std;

/*
Encapsulation says 2 things:
1. An Object's Characteristics and its behaviour are encapsulated together
within that Object.
2. All the characteristics or behaviours are not for everyone to access.
Object should provide data security.

We follow above 2 pointers about Object of real world in programming by:
1. Creating a class that act as a blueprint for Object creation. Class contain
all the characteristics (class variable) and behaviour (class methods) in one block,
encapsulating it together.
2. We introduce access modifiers (public, private, protected) etc to provide data
security to the class members.
*/
class SportsCar {
private:
    string brand;
    string model;
    bool isEngineOn;
    int currentSpeed;
    int currentGear;

    //Introduce new variable to explain setters
    string tyreCompany;

public:
    SportsCar(string b, string m) {
        this->brand = b;
        this->model = m;
        isEngineOn = false;
        currentSpeed = 0;
        currentGear = 0;
        tyreCompany = "MRF";
    }

```



```cpp

    int getSpeed() {
        return currentSpeed;
    }

    string getTyreCompany() {
        return tyreCompany;
    }

    void setTyreCompany(string tyreCompany) {
        this->tyreCompany = tyreCompany;
    }

    void startEngine() {
        isEngineOn = true;
        cout << brand << " " << model << " : Engine starts with a roar!" << endl;
    }

    void shiftGear(int gear) {
        if (!isEngineOn) {
            cout << brand << " " << model << " : Engine is off! Cannot Shift Gear." << endl;
            return;
        }
        currentGear = gear;
        cout << brand << " " << model << " : Shifted to gear " << currentGear << endl;
    }

    void accelerate() {
        if (!isEngineOn) {
            cout << brand << " " << model << " : Engine is off! Cannot accelerate." << endl;
            return;
        }
        currentSpeed += 20;
        cout << brand << " " << model << " : Accelerating to " << currentSpeed <<
```



```cpp
        cout << brand << " " << model << ": Engine turned
    }

    ~SportsCar() {}
};

// Main Method
int main() {

    SportsCar* mySportsCar = new SportsCar("Ford", "Mustang");

    mySportsCar->startEngine();
    mySportsCar->shiftGear(1);
    mySportsCar->accelerate();
    mySportsCar->shiftGear(2);
    mySportsCar->accelerate();
    mySportsCar->brake();
    mySportsCar->stopEngine();

    //Setting arbitrary value to speed.
    // mySportsCar->currentSpeed = 500;

    // cout << "Current Speed of My Sports Car is set to " << mySportsCar->currentSpeed << endl;

    cout << "Current Speed of My Sports Car is " << mySportsCar->getSpeed() << endl;

    delete mySportsCar;

    return 0;
}
```