# OOPS

* **ENCAPSULATION:**
Wrapping of data & methods.
All the data can be accessed through member functions.

Difference b/w Data abstraction & encapsulation:
Encapsulation is a method to bind the data in a single entity or unit along with a method to protect info from outside.
Abstraction is hiding unwanted info.

Objects that help to perform abstraction are encapsulated

* **ABSTRACTION:**
Hiding data that is not necessary

* **INHERITANCE:**
Capability of one class to inherit capabilities from another class.

* **Polymorphism:**
Ability for a message or data to be processed in more than one form.

# STATIC Vs DYNAMIC POLYMORPHISM

Overloading: Static

Differentiating different fns on the basis of argument list in the same class.

```
public class sum {
    public int sum(int x, int y)
    {
        return (x+y);
    }

    public int sum(int x, int y, int z)
    {
        return(x+y+z);
    }
    public double sum(double x, double y)
    {
        return(x+y);
    }
```

We cannot overload a fn just on the basis of return type

DYNAMIC : OVERRIDING

Same function name, parent class & sub class method.

```
class Parent{
        void show()
      {  output("a"); }
   }
class child extends parent{
          void show()
        {
          output("b");
        }
      }
Parent obj1 = new Parent();
  obj1.show(); //a

Parent obj2= new child ();
  obj2. show(); //b
```

```
interface Todo {
    title: string;
    description?: string;
}

const todo1 = {
    title: ~~string~~ "organize";
    extra: "meta data",
};

const updateTodo = (
  todo: Todo,
  fieldsToUpdate: Partial<Todo>
) => ({ ...todo, ...fieldsToUpdate });

const result1 = updateTodo(todo1, {
    description: "throw out trash";
});

const todo2 = {
    ...todo1,
    description: "clean up",
};

const updateRequiredTodo = (
    todo: Required<Todo>,
    fieldsToUpdate: Partial<Todo>
): Required<Todo> => ({ ...todo, ...fieldToUpdate
});
```

```
interface GenericAdd<AddType> {
  add: (x: AddType, y: AddType) => AddType;
}

class GenericNumber implements GenericAdd<number> {
  add(x: number, y: number) {
    return x + y;
  }
}

class GenericString implements GenericAdd<string> {
  add(x: string, y: string) {
    return x + y
  }
}

const genericNumber = new GenericNumber();
genericNumber.add(1.2);
```

# A LITTLE ABOUT INTERFACES---

Purpose of interfaces is to allow the computer to enforce these properties and to know that an object of TYPE T (whatever interface it is) must have fn x, y, z etc.

For ex. we have 3 classes : car, scooter & truck each has a fn start_engine(). how they are implemented are upto to each particular class but the fact that they must have a start_engine() is domain of the interface.

NO VARIABLE allowed
An interface is about actions that are allowed not about data / implementations
All members are public in interface.

## POLYMORPHISM:

Polymorphism is implemented by interface

```
var vehicle1 : vehicle = new car();
var vehicle2 : vehicle = new truck();
```

Interface : Data abstraction + polymorphism