In object oriented programming, we group related variables and functions into a unit called object.

Eg: car is an object (variables like make, model, color, and functions like start(), stop(), move())

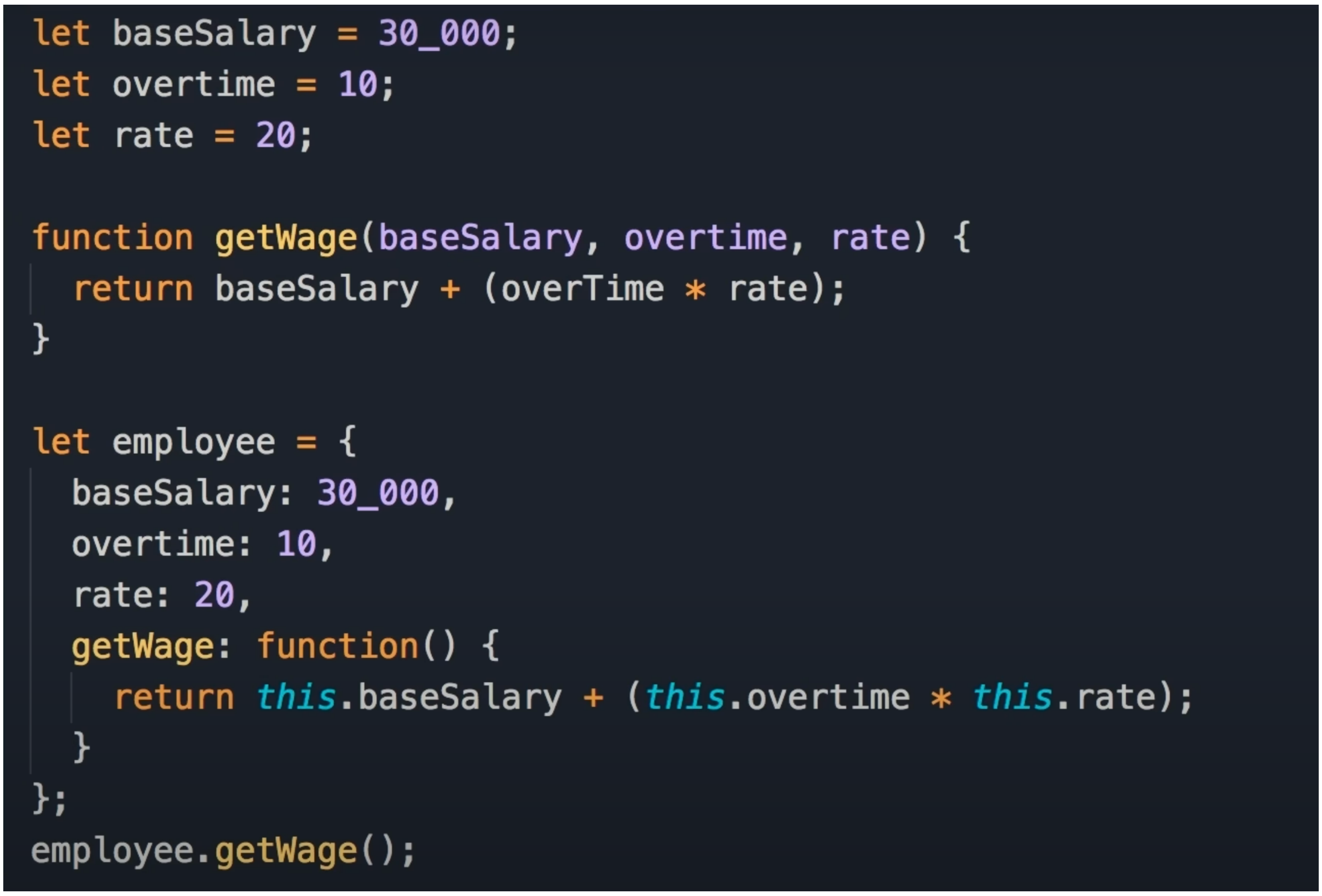
Eg: Browser has a localStorage object which has properties like setItem() and removeItem()

In **Encapsulation** we group related variables and functions into a unit called object.

First part is **procedural**: variables on one side and functions on the other side.

Second part is we have employee object with variables and functions and then a method .getWage()

Advantage: We don't have so many parameters as all of them are modelled as properties of the employee object. The fewer the number of parameters, the easier it is to use and maintain the function.



```
let baseSalary = 30_000;
let overtime = 10;
let rate = 20;

function getWage(baseSalary, overtime, rate) {
  return baseSalary + (overTime * rate);
}

let employee = {
  baseSalary: 30_000,
  overtime: 10,
  rate: 20,
  getWage: function() {
    return this.baseSalary + (this.overtime * this.rate);
  }
};
employee.getWage();
```

Abstraction:

Data Abstraction is providing only the required details and hides the implementation from the world. It can be achieved in Python by using interfaces and abstract classes.

Think of a DVD player as an object this DVD player has a complex

logic board on the inside and a few buttons on the outside that you interact

with you simply press the play button and you don't care what happens on the

inside all that complexity is hidden from you this is abstraction in practice

we can use the same technique in our ABSTRACTION objects so we can hide some of the properties and methods from the outside and this gives us a couple of benefits first is that we'll make the interface of those objects simpler using an understanding an object with a few properties and methods is easier than an

object with several properties and methods the second benefit is that it

helps us reduce the impact of change let's imagine that tomorrow we change

these inner or private methods but none of these changes will leak to the outside because we

don't have any code that touches these methods outside of their containing object

we may delete a method or change its parameters but none of these changes will impact the rest of the applications code so with abstraction we reduce the impact of

change.

Inheritance:

It is a mechanism that allows us to eliminate redundant code.

All these elements have a few things in common they should have properties like

hidden and inner HTML and metals like click and focus instead of redefining

all these properties and methods for every type of HTML element we can define

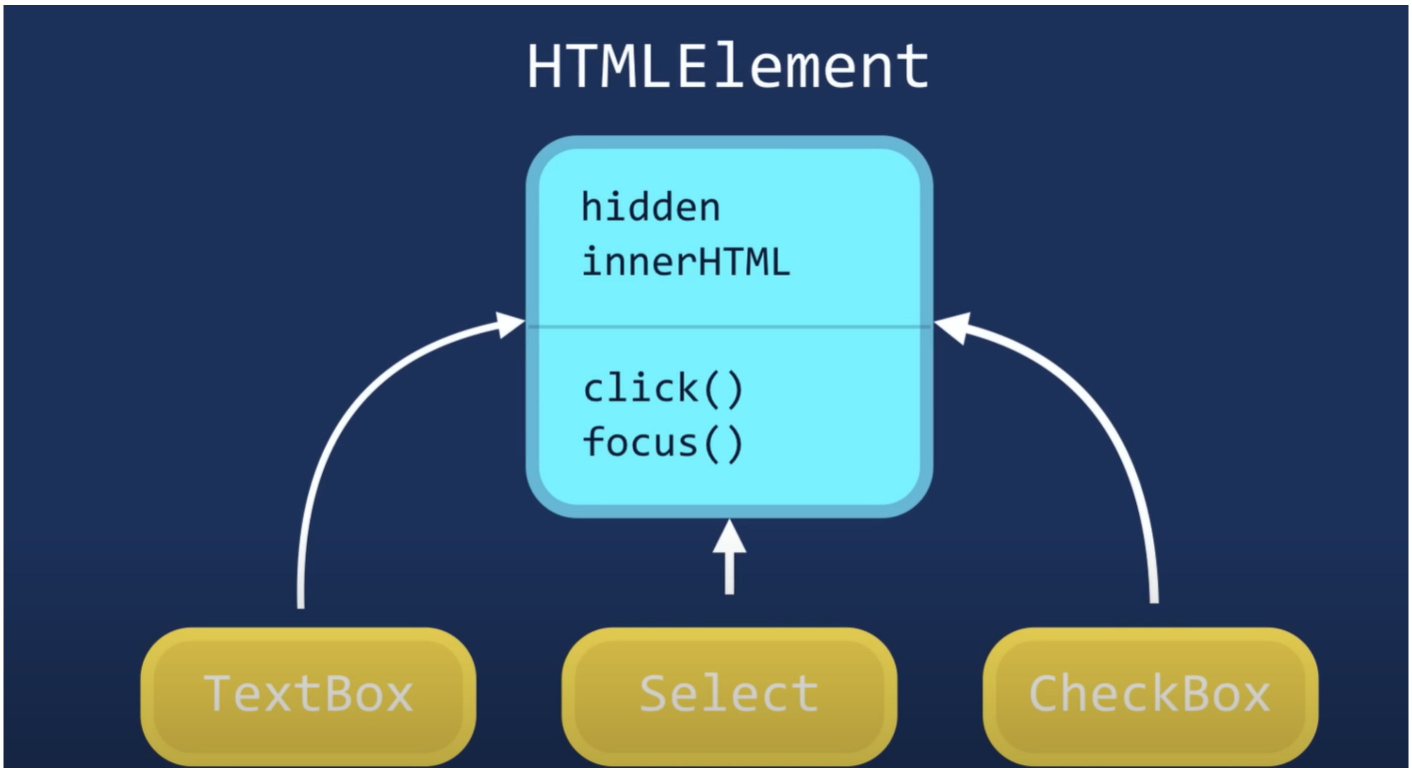
them once in a generic object call it HTML element and have other objects

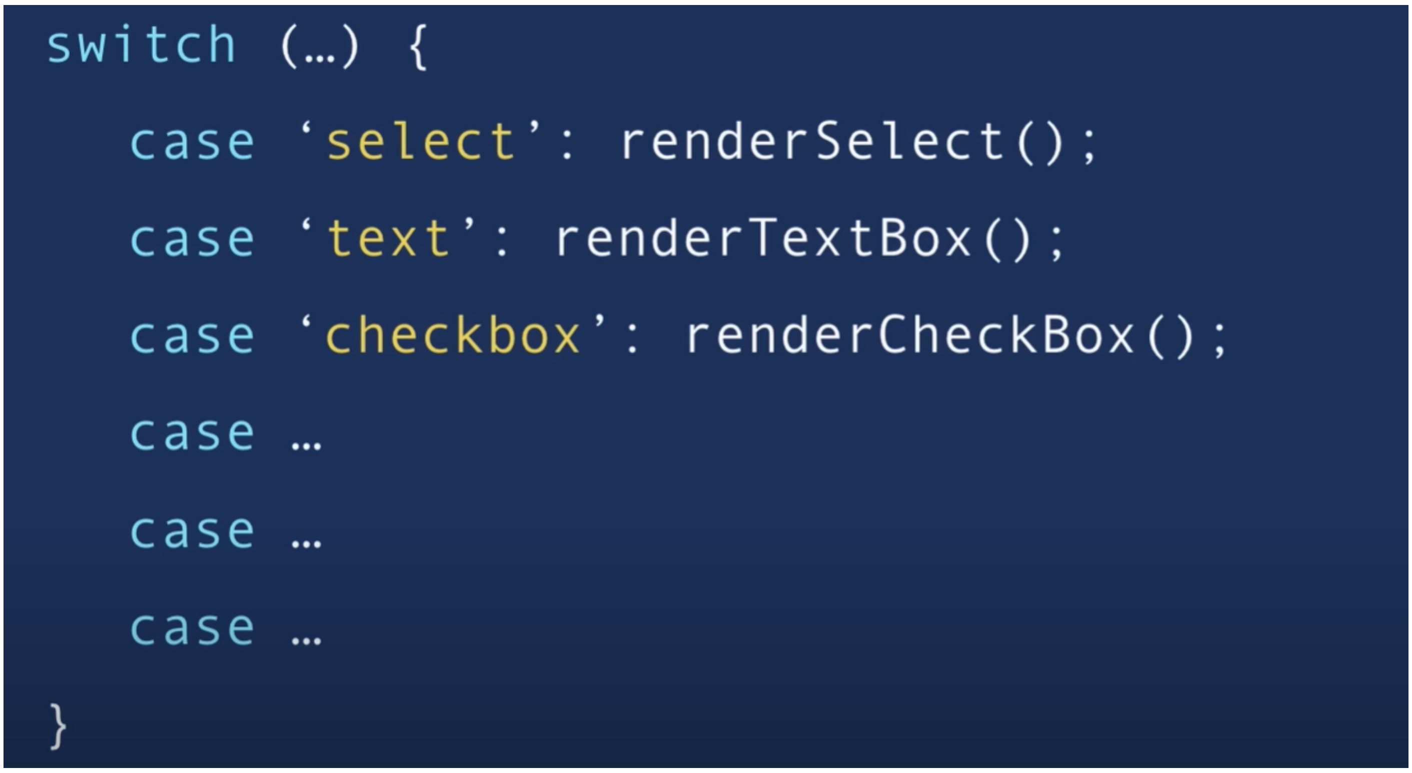
inherit these properties and methods so inheritance helps us eliminate redundant

code

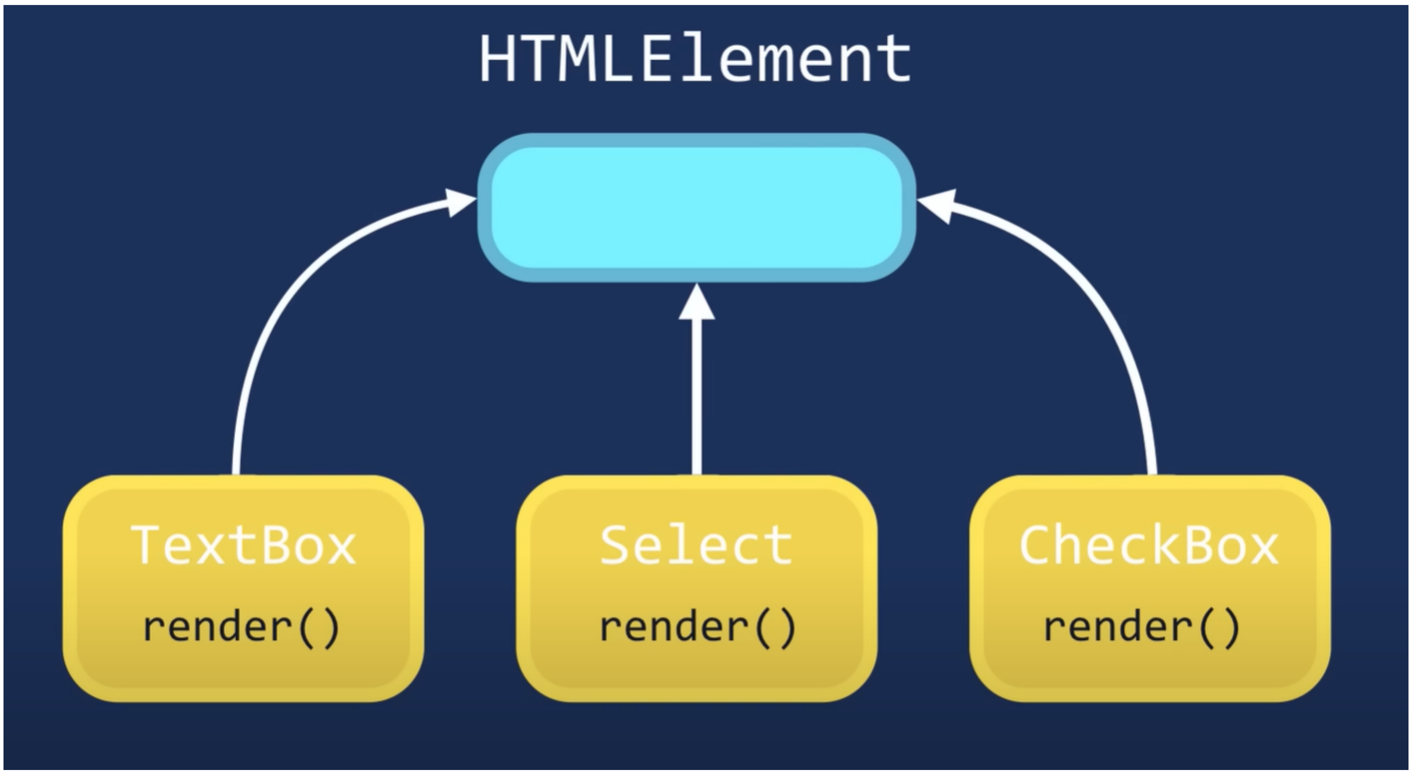
Polymorphism:

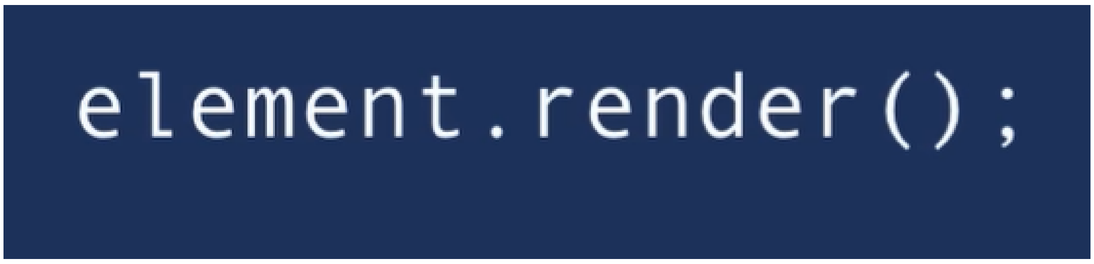
Objects should have the ability to be rendered on a page but the way each element is rendered is different from the others if you want to render multiple HTML elements in a procedural way our code would probably look like this but with object orientation we can implement a render method in each of these objects and the render method will behave differently depending on the type of the object you're referencing so we can get rid of this nasty switch and case and use one line of code.





```
switch (…) {
   case 'select': renderSelect();
   case 'text': renderTextBox();
   case 'checkbox': renderCheckBox();
   case …
   case …
   case …
}
```



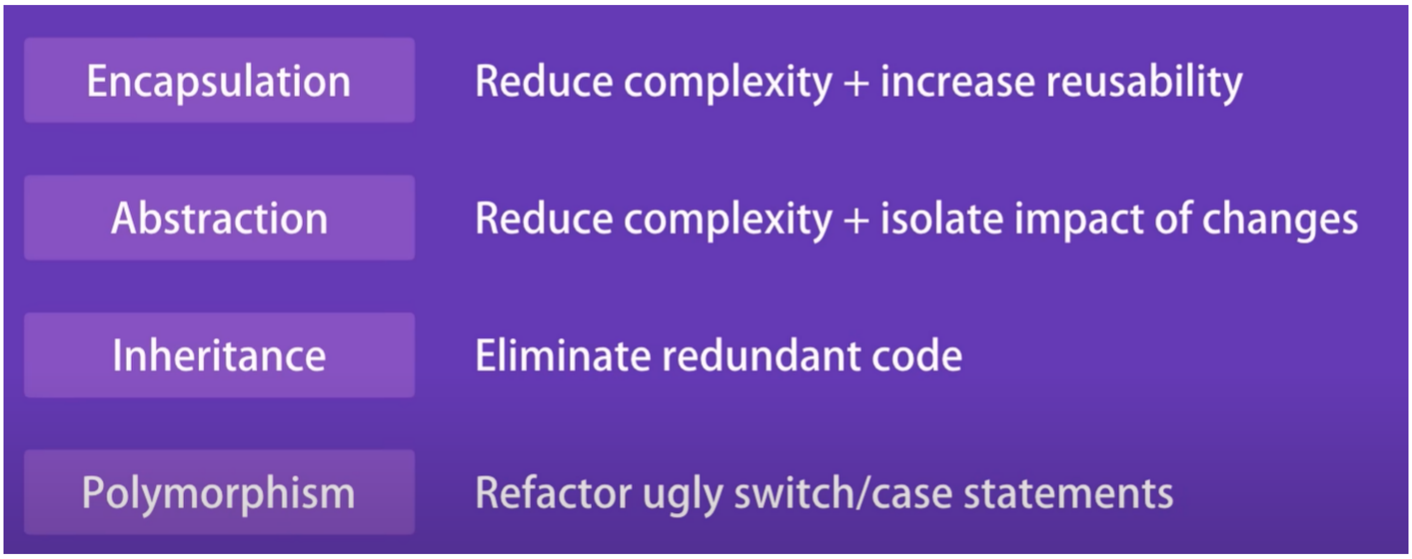


```
element.render();
```

Polymorphism means the ability to take multiple forms. So, for instance, if the parent class has a method named ABC then the child class also can have a method with the same name ABC having its own parameters and variables. Python allows polymorphism.

python

```python
# Polymorphism in Python allows objects of different
# types to be treated as objects of a common superclass.
# This means that different classes can define methods
# with the same name, and these methods can be called in
# a uniform manner regardless of the specific type of object.

class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

class Duck(Animal):
    def make_sound(self):
        return "Quack!"

# Function that demonstrates polymorphism
def animal_sound(animal):
    return animal.make_sound()

# Creating instances of different classes
dog = Dog()
cat = Cat()
duck = Duck()

# Calling the same function with different types of objects
print(animal_sound(dog))  # Output: Woof!
print(animal_sound(cat))  # Output: Meow!
print(animal_sound(duck))  # Output: Quack!
```



Encapsulation	Reduce complexity + increase reusability
Abstraction	Reduce complexity + isolate impact of changes
Inheritance	Eliminate redundant code
Polymorphism	Refactor ugly switch/case statements