

The Visitor Design Pattern

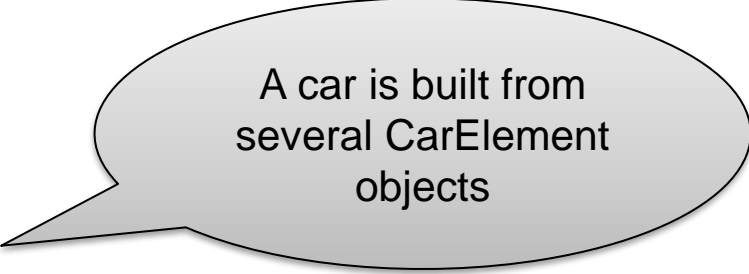
Let's start with a simple example

```
public abstract class CarElement {  
    public abstract int getPrice();  
}
```

```
class Engine extends CarElement {  
    private int hp;  
  
    public Engine(int hp) { this.hp=hp; }  
  
    public int getHP() { return hp; }  
    @Override  
    public int getPrice() { return hp*100; }  
}
```

```
class Car {  
    private Engine engine=new Engine(90);  
    private Wheel[] wheels=new Wheel[] {  
        new Wheel(), new Wheel(), new Wheel(), new Wheel()  
    };  
}
```

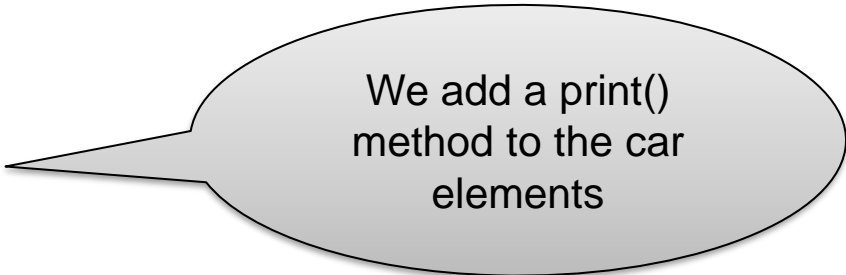
```
class Wheel extends CarElement {  
    @Override  
    public int getPrice() { return 100; }  
}
```



A car is built from
several CarElement
objects

Print a description of the car

```
public abstract class CarElement {  
    public abstract int getPrice();  
    public abstract void print();  
}
```



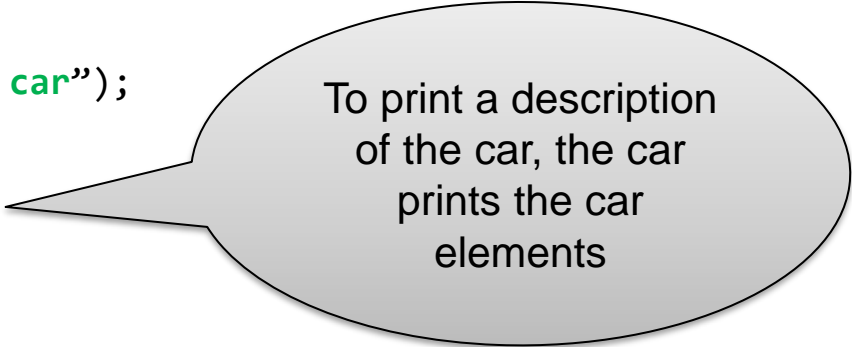
We add a print()
method to the car
elements

```
class Engine extends CarElement {  
    private int hp;  
  
    public Engine(int hp) { this.hp=hp; }  
  
    public int getHP() { return hp; }  
    @Override  
    public int getPrice() { return hp*100; }  
    @Override  
    public void print() { System.out.println("Engine with "+hp+" hp"); }  
}
```

```
class Wheel extends CarElement {  
    @Override  
    public int getPrice() { return 100; }  
    @Override  
    public void print() { System.out.println("A wheel"); }  
}
```

Print a description of the car (2)

```
class Car {  
    private Engine engine=new Engine(90);  
    private Wheel[] wheels=new Wheel[] {  
        new Wheel(), new Wheel(),  
        new Wheel(), new Wheel()  
    };  
  
    public void print() {  
        System.out.println("A car");  
        engine.print();  
        wheels[0].print();  
        wheels[1].print();  
        wheels[2].print();  
        wheels[3].print();  
    }  
}
```



To print a description
of the car, the car
prints the car
elements

- Okay, that works. No problem here. But it's a little bit annoying that we have to modify the Car and CarElement classes to be able to print them...

Calculating the price of a car

```
class Car {
    private Engine engine=new Engine(90);
    private Wheel[] wheels=new Wheel[] {
        new Wheel(), new Wheel(),
        new Wheel(), new Wheel()
    };

    public void printCar() {
        engine.print();
        wheels[0].print(); wheels[1].print(); wheels[2].print(); wheels[3].print();
    }

    public int getCarPrice() {
        return engine.getPrice()+wheels[0].getPrice()+wheels[1].getPrice()
            +wheels[2].getPrice()+wheels[3].getPrice();
    }
}
```

- Again, that works. And again, it's a little bit annoying that we have to modify the Car class to add this new functionality
- Is it really necessary that the Car class should know how to print a car and how to calculate the price of a car? Do we have to modify the classes everytime we want a new functionality?


Making everything public?

- We could move the printCar() and getCarPrice() methods into a different class. But that requires that the fields of the car are public:

```
class Car {  
    public Engine engine=new Engine(90);  
    public Wheel[] wheels=new Wheel[] {  
        new Wheel(), new Wheel(),  
        new Wheel(), new Wheel()  
    };  
}
```

```
class CarTools {  
    public void printCar(Car car) {  
        car.engine.print();  
        car.wheels[0].print(); car.wheels[1].print();  
        car.wheels[2].print(); car.wheels[3].print();  
    }  
}
```

```
public int getCarPrice(Car car) {  
    return car.engine.getPrice()+car.wheels[0].getPrice()+...  
}  
}
```



Not nice!
Now, everybody can
see how a car object
is implemented
internally.

The Visitor Design Pattern

- In the Visitor design pattern, complex data structures (like the Car class) allow “visitors” to visit their elements.

```
public interface Visitor {  
    public void visit(Wheel wheel);  
    public void visit(Engine engine);  
    public void visit(Car car);  
}
```

- Visitors can do work (like printing, calculating the price,...) that we don't want to put into the Car class. Here is a visitor printing the car description:

```
public class PrintVisitor implements Visitor {  
    @Override  
    public void visit(Wheel wheel) { System.out.println("A wheel"); }  
  
    @Override  
    public void visit(Engine engine) {  
        System.out.println("Engine with "+engine.getHP()+" hp");  
    }  
  
    @Override  
    public void visit(Car car) { System.out.println("A car"); }  
}
```

The Visitor Design Pattern (2)

- Every object visited by the visitor decides what to do with the visitor:

```
public interface Visitable {  
    public void accept(Visitor visitor);  
}
```

```
class Wheel extends CarElement implements Visitable {  
    @Override  
    public int getPrice() { return 100; }  
    @Override  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}
```

A wheel does not know what a visitor does. It just “accepts” the visitor.

This will print “A wheel”

```
class Engine extends CarElement implements Visitable {  
    private int hp;  
    public Engine(int hp) { this.hp=hp; }  
    public int getHP() { return hp; }  
    @Override  
    public int getPrice() { return hp*100; }  
    @Override  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}
```

This will print “Engine with 90 hp”

The Visitor Design Pattern (3)

```
class Car implements Visitable {  
    private Engine engine=new Engine(90);  
    private Wheel[] wheels=new Wheel[] {  
        new Wheel(), new Wheel(),  
        new Wheel(), new Wheel()  
    };  
};
```

@Override

```
public void accept(Visitor visitor) {  
    visitor.visit(this);  
    engine.accept(visitor);  
    wheels[0].accept(visitor);  
    wheels[1].accept(visitor);  
    wheels[2].accept(visitor);  
    wheels[3].accept(visitor);  
}  
}
```

This will print "A car"

Here, we send the visitor to the car elements

- We can now print the car description for a car object:

```
car.accept(new PrintVisitor());
```

- Note:

- The Visitor does not need to know how a car is structured internally
- The visited car does not need to know how to print a description

Another visitor for price calculation

- We can implement the price calculation also as a visitor:

```
public class PriceVisitor implements Visitor {
    int totalPrice=0;

    @Override
    public void visit(Wheel wheel) { totalPrice+=wheel.getPrice(); }
    @Override
    public void visit(Engine engine) { totalPrice+=engine.getPrice(); }
    @Override
    public void visit(Car car) { }
}
```

- And we can use the PriceVisitor to calculate the price of a car:
car.accept(new PriceVisitor());
- Again, the visitor does not need to know how a car is structured internally. It doesn't even know how many wheels a car has!

Visitors: Summary

- Visitors are a way to separate the structure of an object from the code working on that object
 - Visited object = the object containing the data of interest
 - Visitor = the code working on the data
- In our examples, we can
 - modify the structure of the car without having to modify the PrintVisitor or the PriceVisitor
 - We could add more wheels (6 instead of 4)
 - We could store the wheels in a list instead of an array
 - ...
 - create new visitors without having to modify the Car and CarElement classes