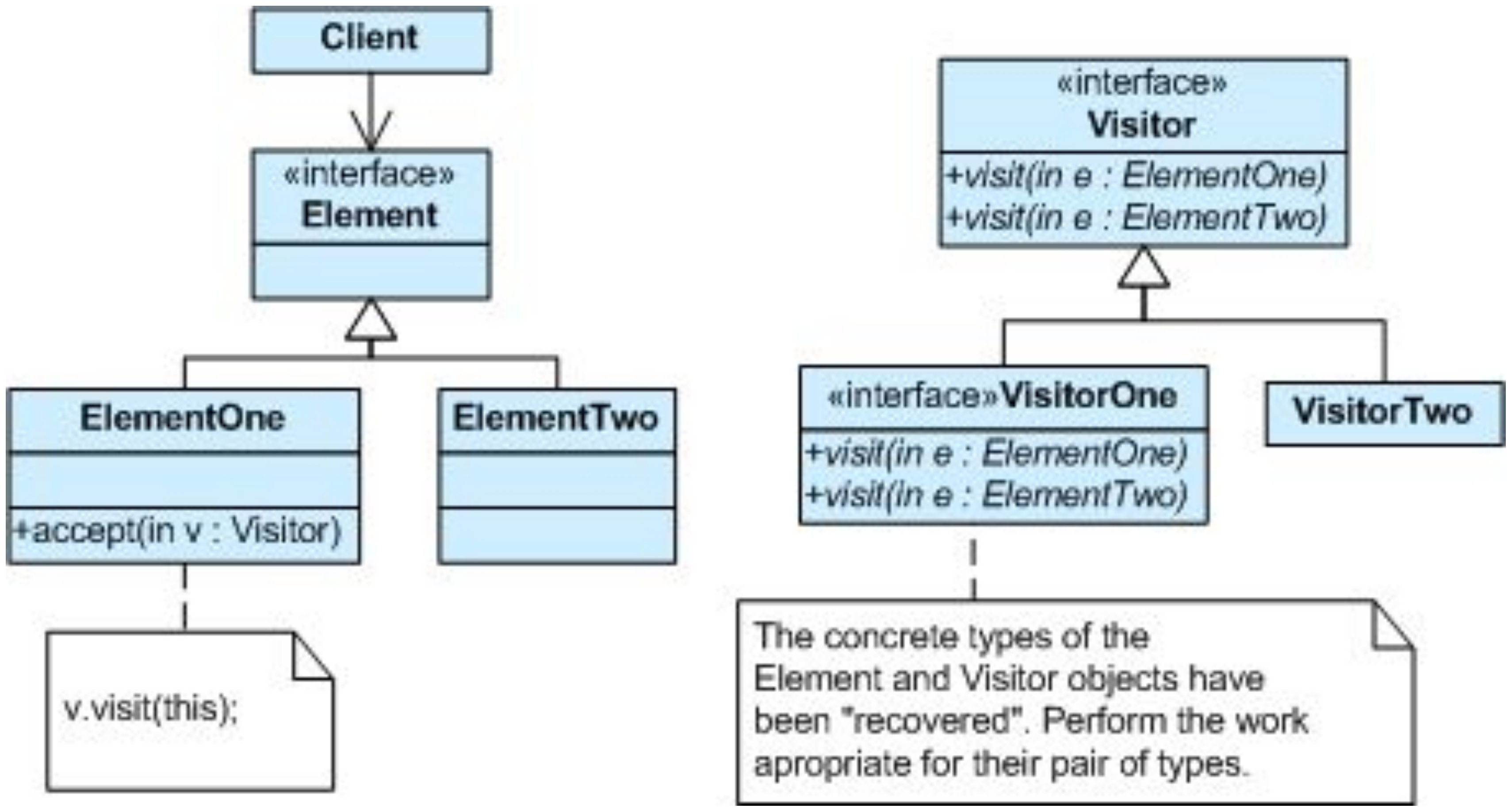
Visitor

Intent

- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Structure


Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
ElementTwo
v.visit(this);
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface»VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
VisitorTwo
The concrete types of the Element and Visitor objects have been "recovered". Perform the work apropriate for their pair of types.

Why Visitors?

- The Visitor pattern is one among many design patterns aimed at making object-oriented systems more flexible
- The issue addressed by the Visitor pattern is the manipulation of composite objects.
- Without visitors, such manipulation runs into several problems as illustrated by considering an implementation of integer lists, written in Java

Integer lists, written in Java (Without Generics)

```
interface List {}
class Nil implements List {}
class Cons implements List {
   int head;
   List tail;
   }
```

What happens when we write a program which computes the sum of all components of a given List object?

First Attempt: Instanceof and Type Casts

```
List l;
// The List-object we are working on.
int sum = 0;
// Contains the sum after the loop.
boolean proceed = true;
while (proceed) {
   if (l instanceof Nil)
     proceed = false;
   else if (l instanceof Cons) {
     sum = sum + ((Cons) l).head; // Type cast!
     l = ((Cons) l).tail;         // Type cast!
   }
}
```

What are the problems here?

What are the problems here?

- Type Casts?
  - We want static (compile time) type checking
- Flexible?
  - Probably not well illustrated with this example

Second Attempt: Dedicated Methods

```
interface List {
   int sum();
}
class Nil implements List {
   public int sum() { return 0; }
}
class Cons implements List {
   int head;
   List tail;
   public int sum() {
       return head + tail.sum();
   }
 }
```

Tradeoffs

- Can compute the sum of all components of a given List-object l by writing l.sum().
- Advantage: type casts and instanceof operations have disappeared, and that the code can be written in a systematic way.
- Disadvantage: Every time we want to perform a new operation on List-objects, say, compute the product of all integer parts, then new dedicated methods have to be written for all the classes, and the classes must be recompiled

Third Attempt: The Visitor Pattern.

```
interface List {
   void accept(Visitor v);
}
class Nil implements List {
   public void accept(Visitor v) {  v.visitNil(this); }
}
class Cons implements List {
  int head;
    List tail;
    public void accept(Visitor v) {
  v.visitCons(this);
   } }
```

Second Part of Visitor Idea

```
interface Visitor {
  void visitNil(Nil x);
  void visitCons(Cons x);
}


class SumVisitor implements Visitor {
    int sum = 0;
    public void visitNil(Nil x) {}
    public void visitCons(Cons x){
        sum = sum + x.head;
        x.tail.accept(this); } }
```

Summary

- Each accept method takes a visitor as argument.
- The interface Visitor has a header for each of the basic classes.
- We can now compute and print the sum of all components of a given List-object l by writing

```
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Summary Continued

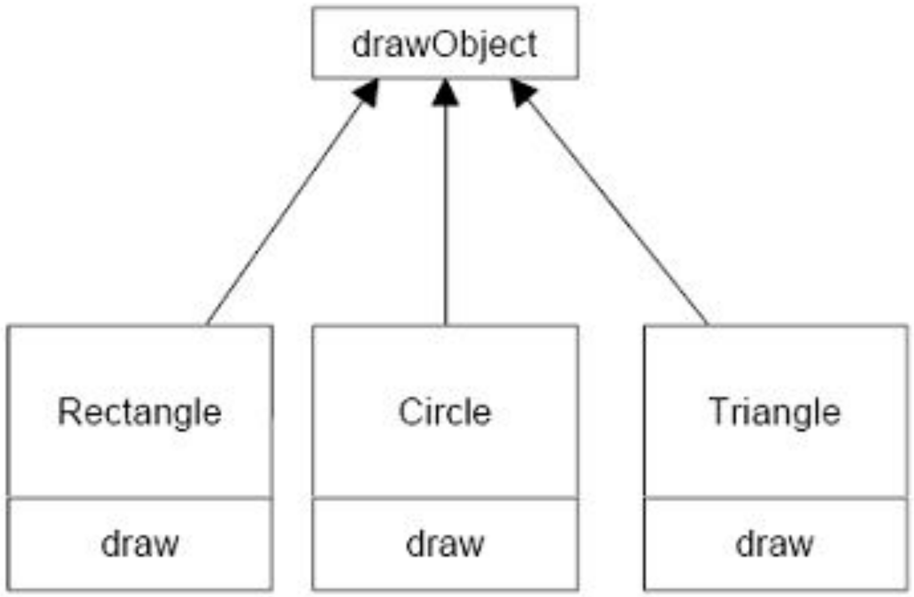
- The advantage is that one can write code that manipulates objects of existing classes without recompiling those classes.
- The price is that all objects must have an accept method.
- In summary, the Visitor pattern combines the advantages of the two other approaches

Summary Table

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

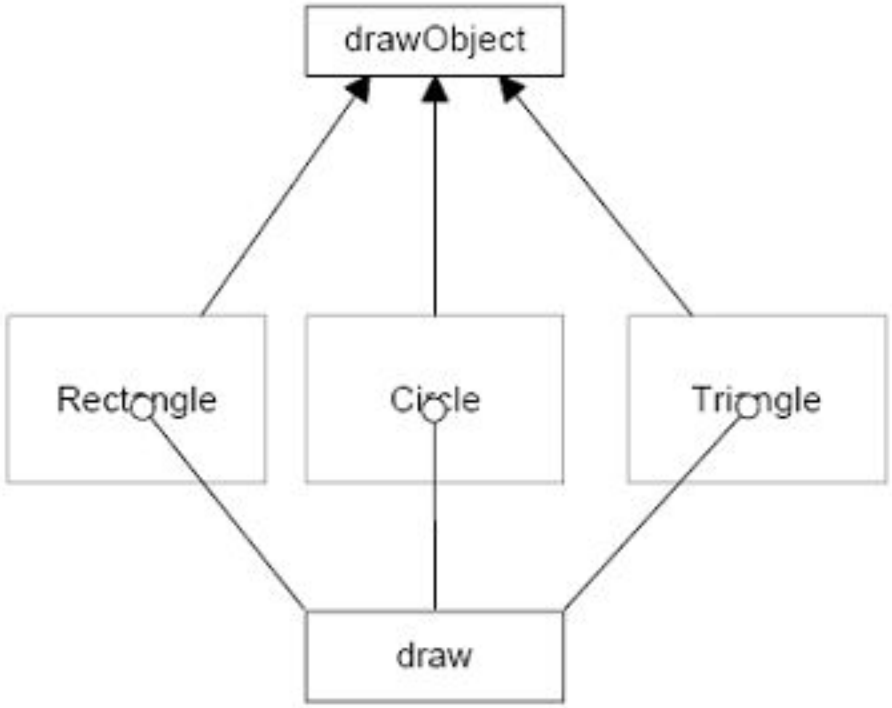
An example

- Suppose each of a number of drawing object classes has similar code for drawing itself.
- The drawing methods may be different, but they probably all use underlying utility functions that we might have to duplicate in each class.
- Further, a set of closely related functions is scattered throughout a number of different classes.



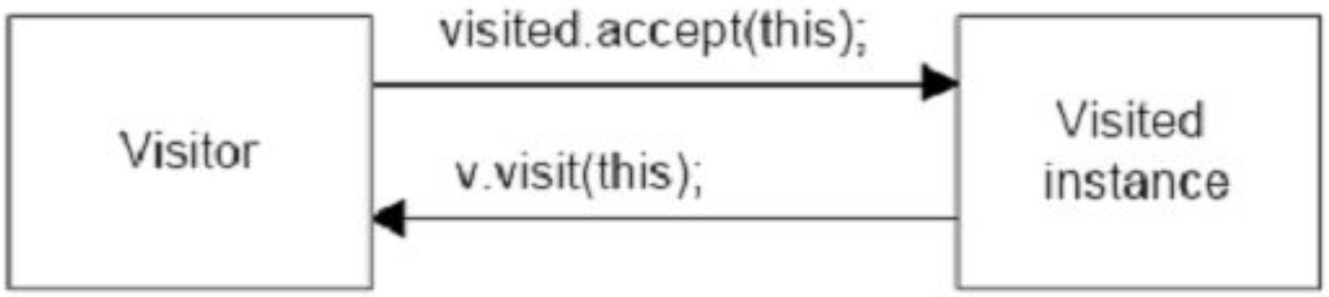
The Visitor Solution

- Instead, we write a Visitor class which contains all the related *draw* methods and have it visit each of the objects in succession.



But what does “Visiting” mean?

- There is only one way that an outside class can gain access to another class
  - by calling its public methods.
- In the Visitor case, visiting each class means that you are calling a method already installed for this purpose, called *accept*.
- The *accept* method has one argument: the instance of the visitor, and in return, it calls the *visit* method of the Visitor, passing itself as an argument.



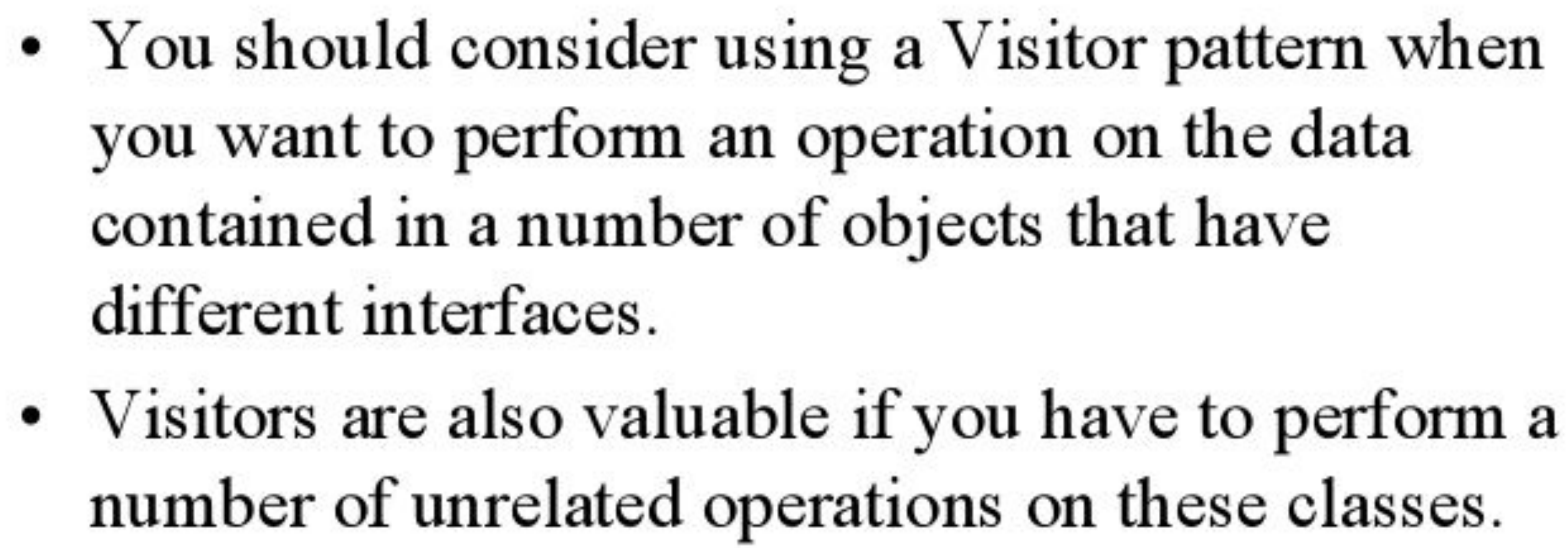
In terms of code…..

- Every object that you want to visit must have the following method:

```
public void accept(Visitor v)
    {
        v.visit(this);          //call visitor method
    }
```

- In this way, the Visitor object receives a reference to each of the instances, one by one, and can then call its public methods to obtain data, perform calculations, generate reports, or just draw the object on the screen.

Use a Visitor when…..

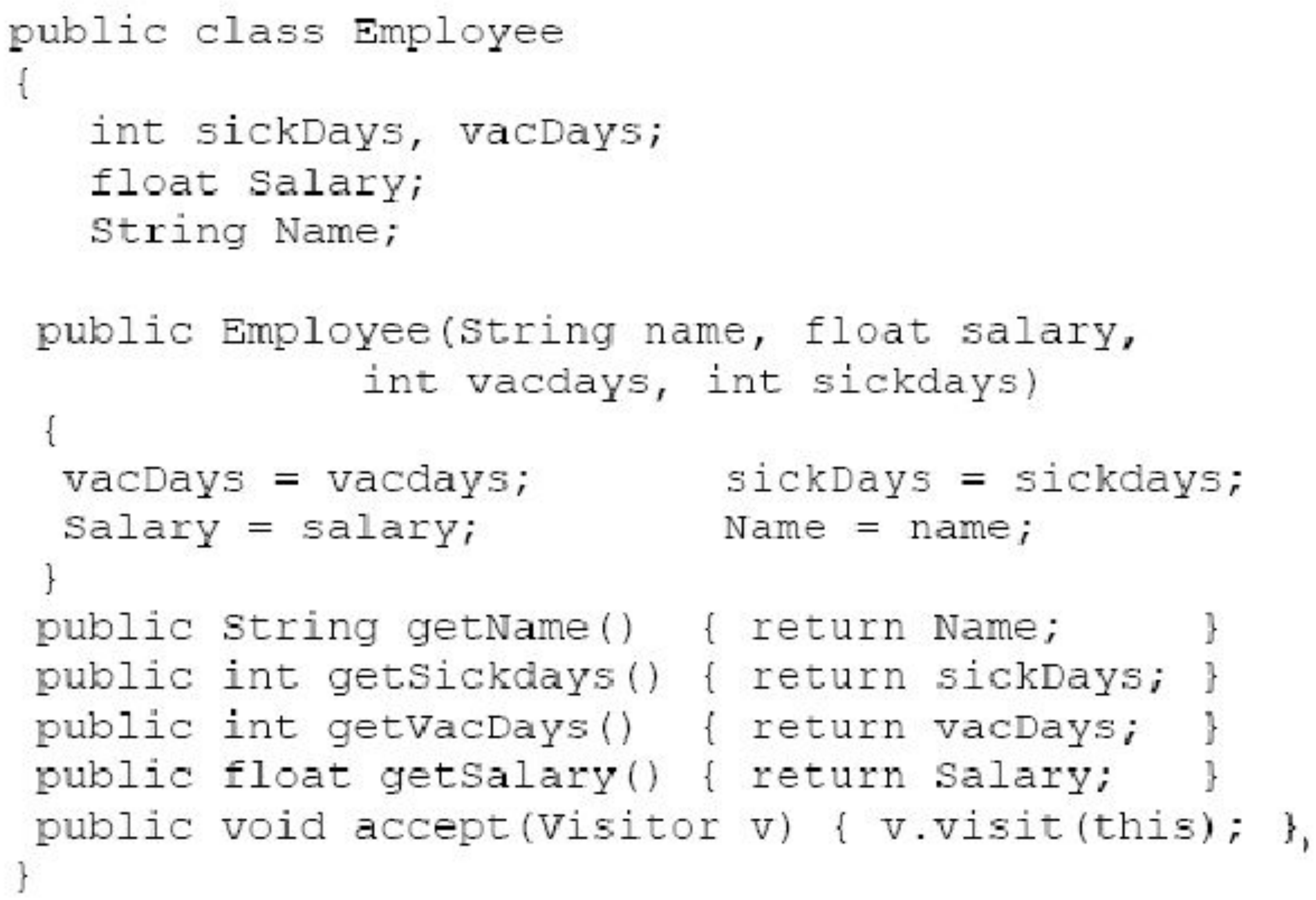


- You should consider using a Visitor pattern when you want to perform an operation on the data contained in a number of objects that have different interfaces.
- Visitors are also valuable if you have to perform a number of unrelated operations on these classes.

Another Example

We have a simple Employee object which maintains a record of the employee’s name, salary, vacation taken and number of sick days taken.

The class



```
public class Employee
{
   int sickDays, vacDays;
   float Salary;
   String Name;

 public Employee(String name, float salary,
             int vacdays, int sickdays)
 {
  vacDays = vacdays;         sickDays = sickdays;
  Salary = salary;           Name = name;
 }
 public String getName()   { return Name;      }
 public int getSickdays() { return sickDays; }
 public int getVacDays()   { return vacDays;   }
 public float getSalary() { return Salary;    }
 public void accept(Visitor v) { v.visit(this); },
}
```

Generating a Report

- Note that we have included the *accept* method in this class.
- Now let's suppose that we want to prepare a report of the number of vacation days that all employees have taken so far this year.
- We could just write some code in the client to sum the results of calls to each Employee's *getVacDays* function, or we could put this function into a Visitor.

The Visitor Abstract class

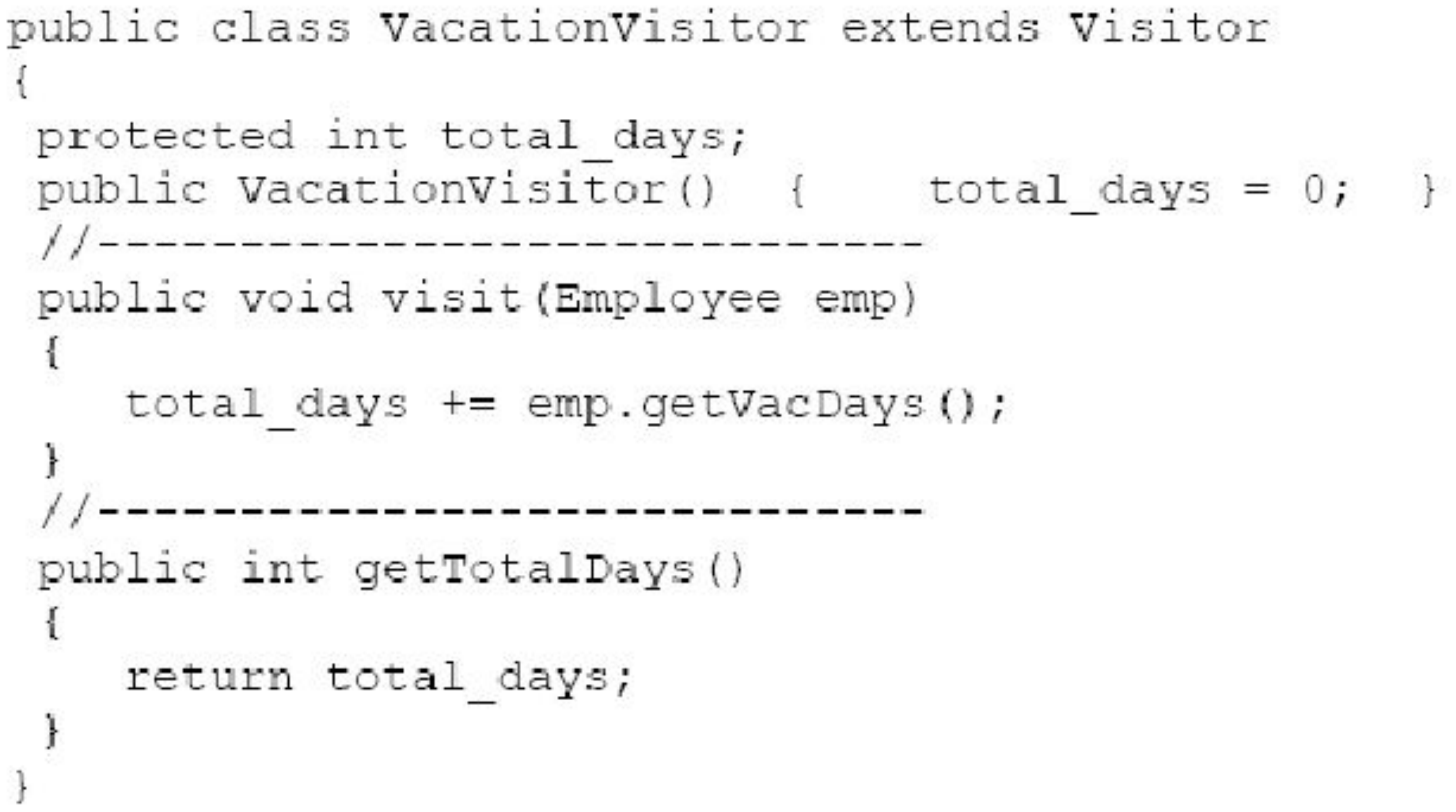
- Since Java is a strongly typed language, your base Visitor class needs to have a suitable abstract *visit* method for each kind of class in your program.
- In this simple example, we only have Employees, so our basic abstract Visitor class is just:

```
public abstract class Visitor
{
  public abstract void visit(Employee emp);
}
```

A Concrete Visitor

- Notice that there is no indication what the Visitor does with each class in either the client classes or the abstract Visitor class.
- We can in fact write a whole lot of visitors that do different things to the classes in our program.
- The Visitor we are going to write first just sums the vacation data for all our employees.

The Concrete Visitor code



```
public class VacationVisitor extends Visitor
{
 protected int total_days;
 public VacationVisitor()  {    total_days = 0;   }
 //-----------------------------
 public void visit(Employee emp)
 {
    total_days += emp.getVacDays();
 }
 //-----------------------------
 public int getTotalDays()
 {
    return total_days;
 }
}
```

The Main program

- Now, all we have to do to compute the total vacation taken is to go through a list of the employees and visit each of them, and then ask the Visitor for the total.

```
VacationVisitor vac = new VacationVisitor();
 for (int i = 0; i < employees.length; i++)
 {
  employees[i].accept(vac);
 }
 System.out.println(vac.getTotalDays());
```

The Steps

1. Move through a loop of all the Employees.
2. The Visitor calls each Employee's *accept* method.
3. That instance of Employee calls the Visitor's *visit* method.
4. The Visitor fetches the vacation days and adds them into the total.
5. The main program prints out the total when the loop is complete.

Benefits

- The visitor pattern is a great way to provide a flexible design for adding new visitors to extend existing functionality without changing existing code

Drawbacks

- If a new visitable object is added to the framework structure all the implemented visitors need to be modified.
- The separation of visitors and visitable is only in one sense: visitors depend of visitable objects while visitable are not dependent of visitors.
- Part of the dependency problems can be solved by using reflection with a performance cost.

Visitors and Iterators

- The iterator pattern and visitor pattern has the same benefit, they are used to traverse object structures. The main difference is that the iterator is intended to be used on collections. Usually collections contain objects of the same type. The visitor pattern can be used on complex structure such as hierarchical structures or composite structures. In this case the accept method of a complex object should call the accept method of all the child objects.
- Another difference is operation performed on the objects: In one case the visitor defines the operations that should be performed, while the iterator is used by the client to iterate through the objects form a collection and the operations are defined by the client itself.

Visitors and Composites

- The visitor pattern can be used in addition with the composite pattern. The object structure can be a composite structure. In this case in the implementation of the accept method of the composite object the accept methods of the component object has to be invoked.