

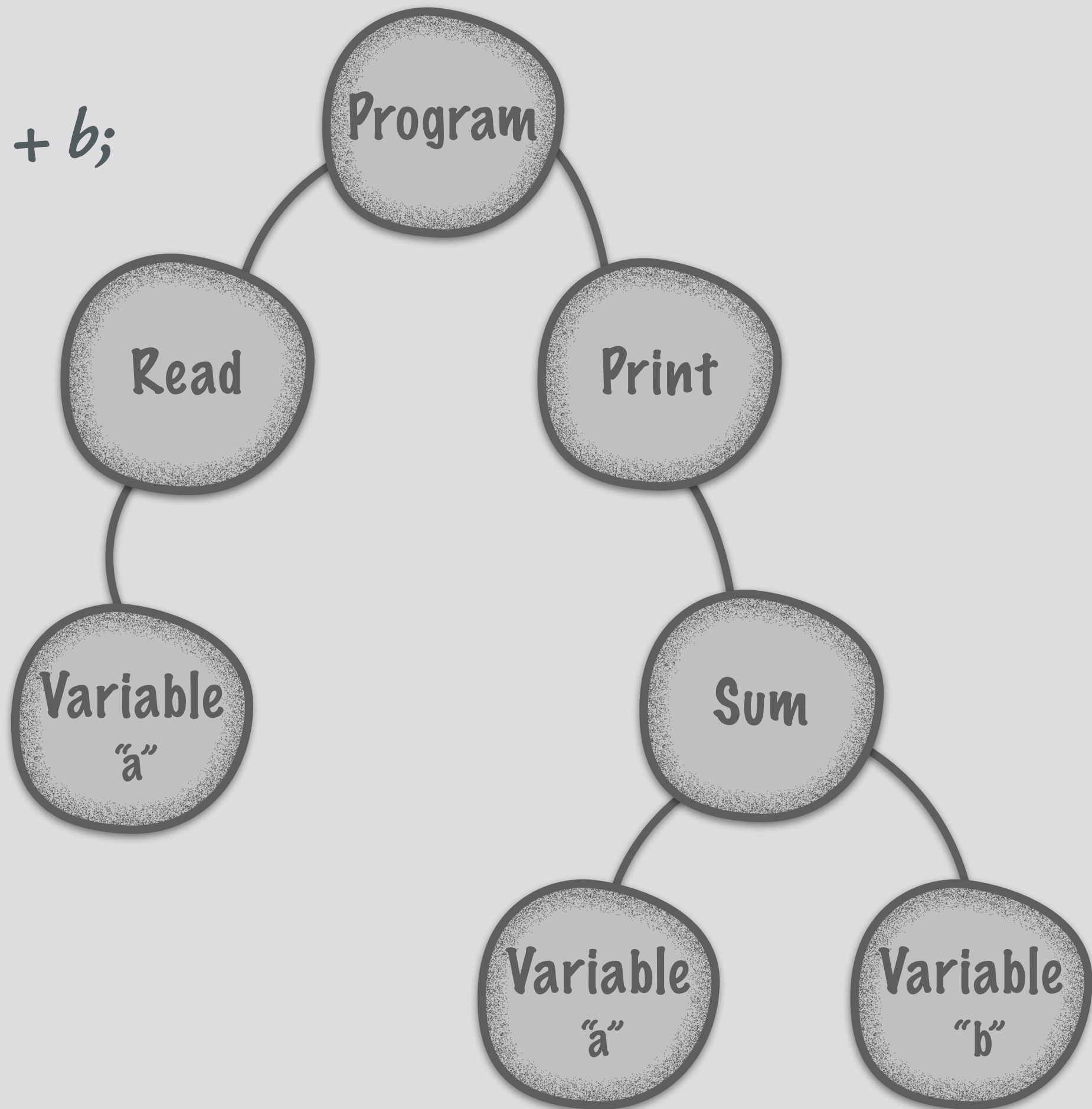
Introduction

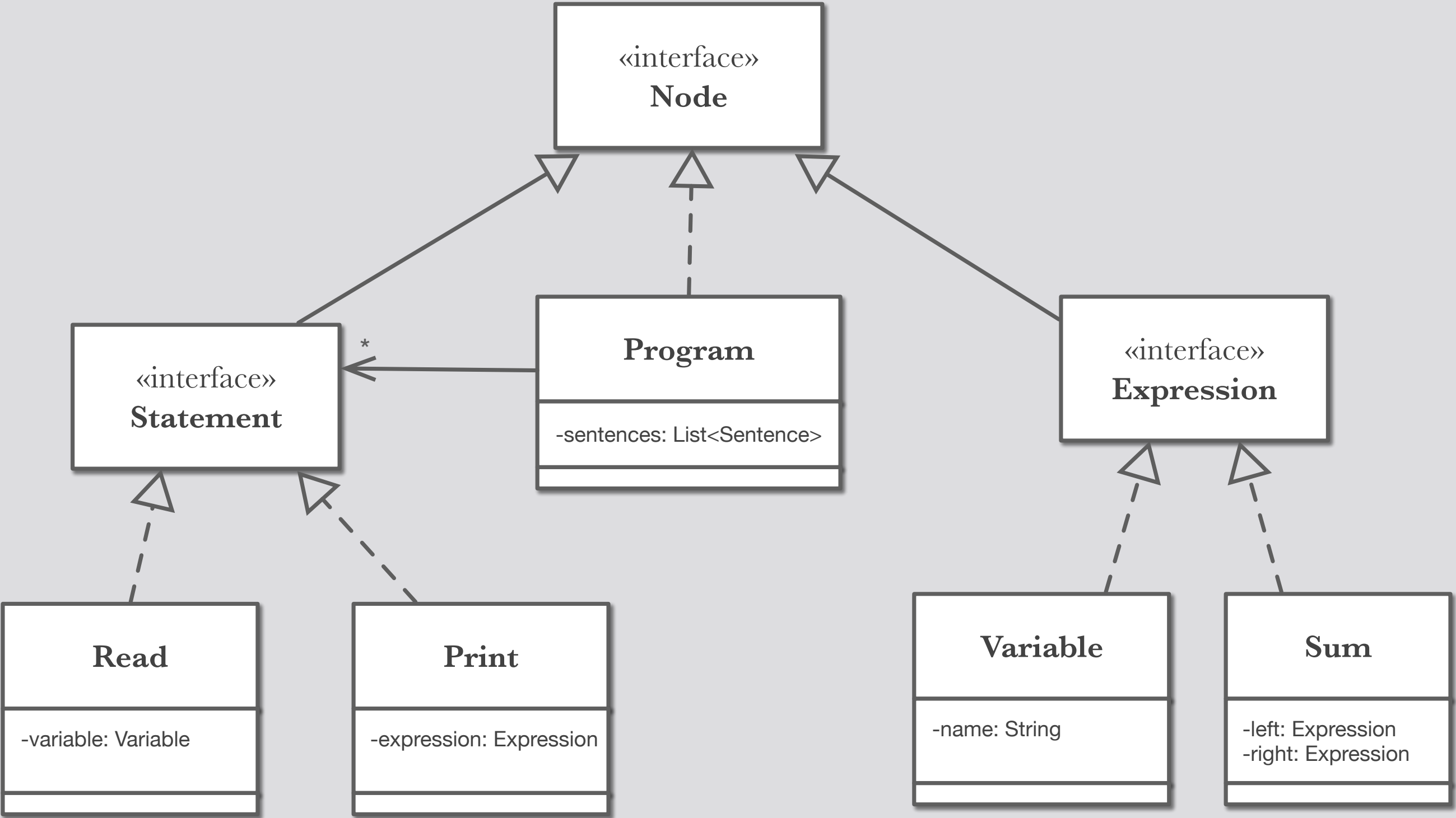
Estamos creando un intérprete para programas escritos en un lenguaje de programación.

Queremos poder modelar (representar en memoria, como objetos) programas como este:

```
read a;  
print a + b;
```

*read a;
print a + b;*







```
interface Node { }

class Program implements Node {
    List<Sentence> sentences;
}

interface Statement extends Node { }

class Read implements Statement {
    Variable variable;
}

interface Expression extends Node { }

class Sum implements Expression {
    Expression left, right;
}

class Variable implements Expression {
    String name;
}
```

Ahora tenemos que implementar varias operaciones

Imprimir el programa con formato

Análisis semántico

Comprobación de errores

Generación de código

Generación de documentación

Y las que vengan en un futuro.

¿Quién debería ser el responsable de realizar dichas operaciones?

¿Quién debería ser el responsable de realizar dichas operaciones?

Dos posibilidades:

a) Implementación descentralizada

Que sean los propios nodos

b) Implementación centralizada

Cada operación en su propia clase

En los propios nodos

Patrón Interpreter

(Lo que hicimos en la práctica 2)

Cada operación se distribuye en las distintas clases de los nodos del árbol.

Es decir, cada nodo tendrá un método para cada operación.



Print.java

```
public class Print implements Statement {  
    public void prettyPrint() { ... }  
    public void typeCheck() { ... }  
    public void generateCode() { ... }  
    // ...  
}
```



Sum.java

```
public class Sum implements Expression {  
    public void prettyPrint() { ... }  
    public void typeCheck() { ... }  
    public void generateCode() { ... }  
    // ...  
}
```



Print.java

```
public class Print implements Statement {  
    public void prettyPrint() { ... }  
    public void typeCheck() { ... }  
    public void generateCode() { ... }  
    // ...  
}
```



Sum.java

```
public class Sum implements Expression {  
    public void prettyPrint() { ... }  
    public void typeCheck() { ... }  
    public void generateCode() { ... }  
    // ...  
}
```

¿Qué problemas tiene
este enfoque?

Solución

Cada operación en su propia clase

Con un método para cada tipo de nodo

Facilita añadir nuevas operaciones.

¿Cómo lo implementamos?

¿Cómo implementar cada operación con el enfoque centralizado?

Dos posibilidades:

- a) Recorrido recursivo
- b) Patrón Visitor



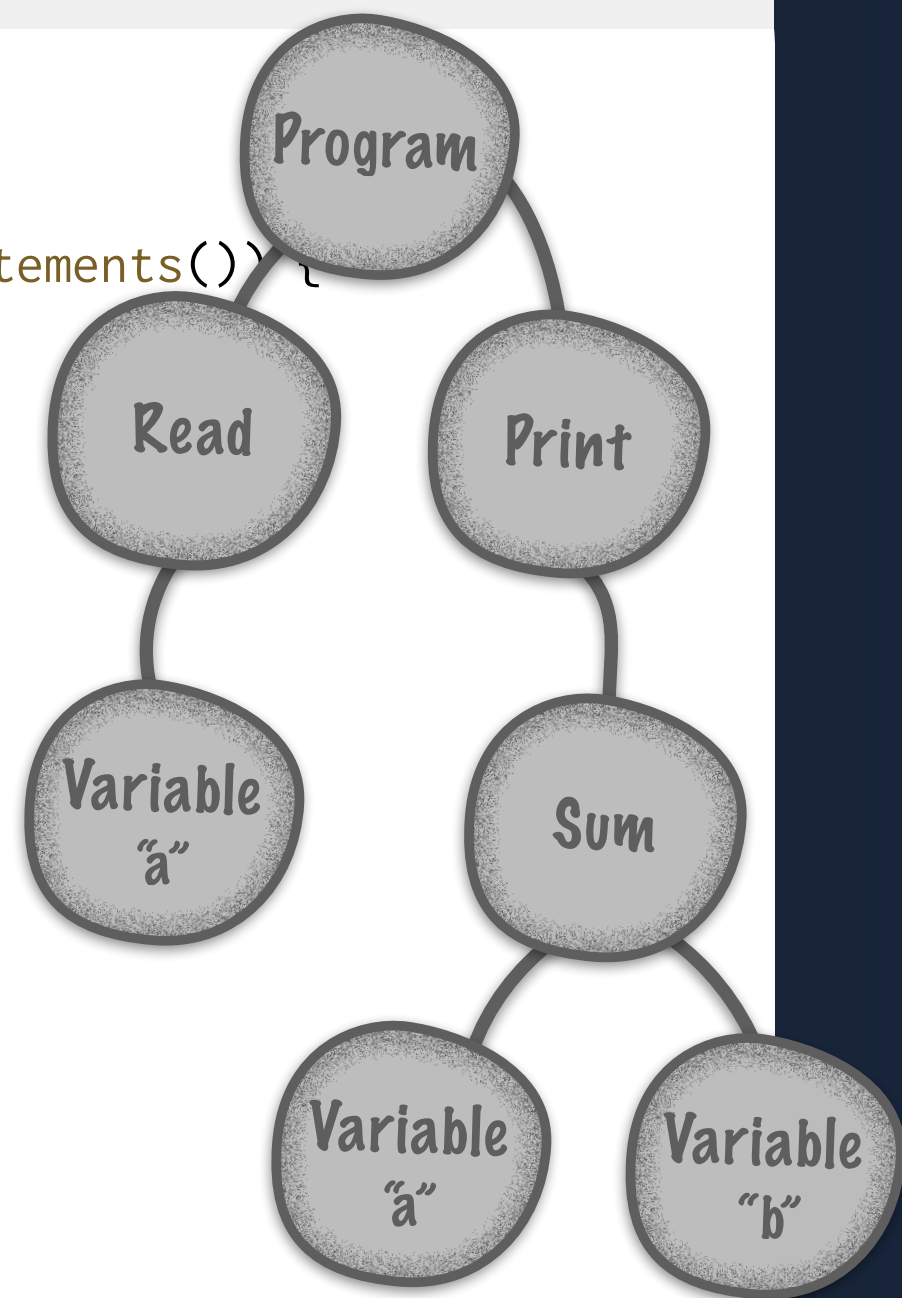
RecursivePrintTest.java

```
public static void main(String[] args) {  
    // Build the Abstract Syntax Tree  
    Program program = new Program();  
    // ...  
  
    RecursivePrint print = new RecursivePrint();  
    print.visit(program);  
}
```




RecursivePrint.java

```
public void visit(Node node) {  
    if (node instanceof Program) {  
        for (Statement statement : ((Program) node).statements()) {  
            visit(statement);  
        }  
    } else if (node instanceof Print) {  
        System.out.print("print ");  
        visit(((Print) node).expression());  
        System.out.println(";");  
    } else if (node instanceof Read) {  
        System.out.print("read ");  
        visit(((Read) node).variable());  
        System.out.println(";");  
    } else if (node instanceof Sum) {  
        visit(((Sum) node).left());  
        System.out.print(" + ");  
        visit(((Sum) node).right());  
    } else if (node instanceof Variable) {  
        System.out.println(((Variable) node).name());  
    }  
}
```



*read a;
print a + b;*

Problemas de dicha implementación

Todo el código está en un único método.

Aunque es cierto que se podría mejorar extrayendo el código de cada rama condicional a un método privado.

Pero, aun así, sería necesaria la lógica condicional global y los `instanceof` en `visit(Node)` para decidir a qué método llamar.

No hay comprobación estática de tipos.

Queríamos tener esto



```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");  
    }  
  
    public void visit(Read read) {  
        System.out.print("read ");  
        visit(read.variable());  
        System.out.println(";");  
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());  
    }  
  
    public void visit(Variable variable) {  
        System.out.println(variable.name());  
    }  
}
```

El problema es que

¡No compila!

¿Por qué?

El problema



```
interface Figure {  
    // ...  
}
```


```
class Circle implements Figure {  
    // ...  
}
```



```
void draw(Figure figure) {  
    System.out.println("I'm a figure!");  
}
```


```
void draw(Circle circle) {  
    System.out.println("I'm a circle!");  
}
```

```
Figure circle = new Circle();  
draw(circle); // What will be printed?
```

```
interface Figure {  
    // ...  
}
```

```
class Circle implements Figure {  
    // ...  
}
```



```
void draw(Figure figure) {  
    System.out.println("I'm a figure!");  
}
```

```
void draw(Circle circle) {  
    System.out.println("I'm a circle!");  
}
```

```
Figure circle = new Circle();  
draw(circle); // What will be printed?
```



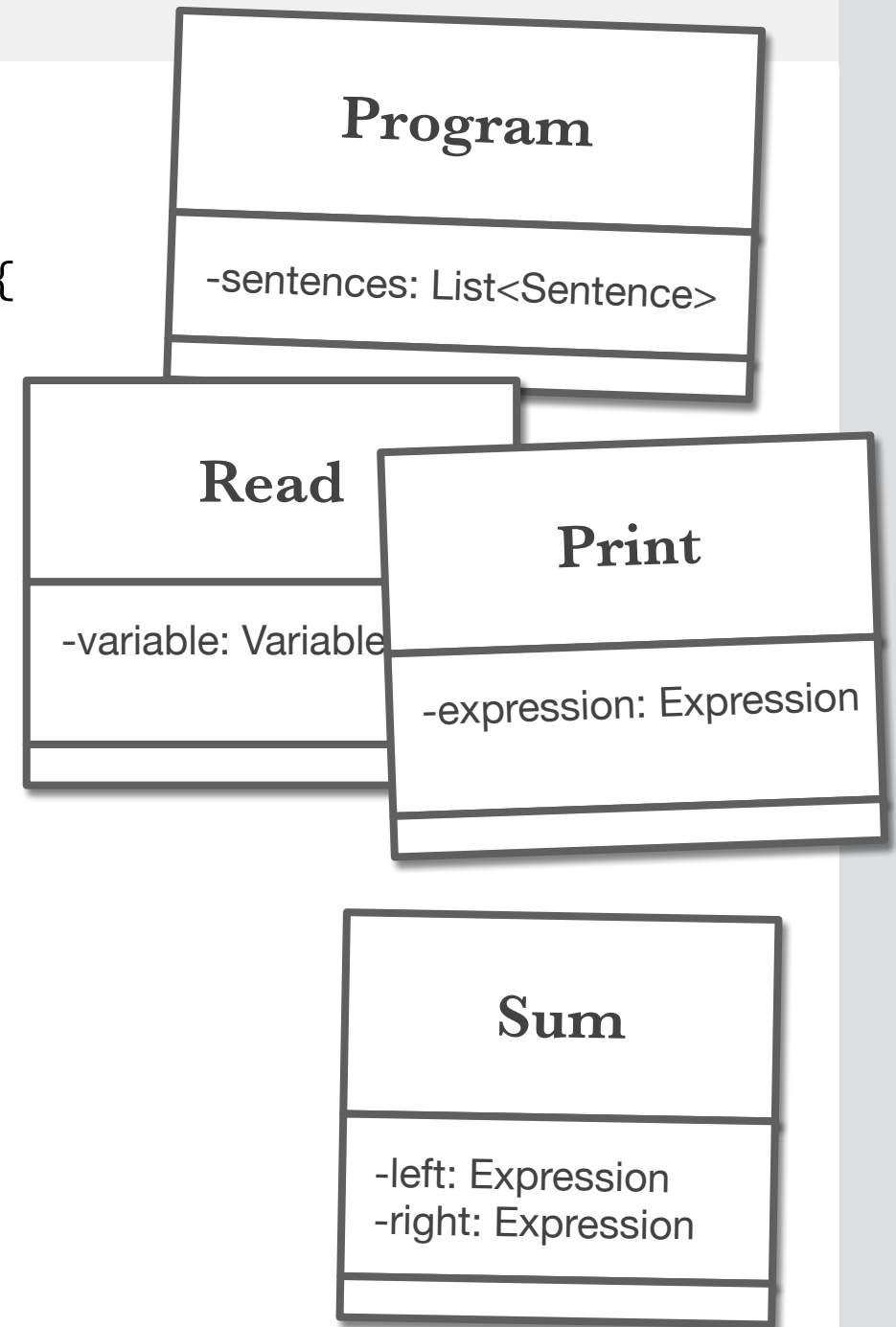

IdealPrint.java

```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statement()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");  
    }  
  
    public void visit(Read read) {  
        System.out.print("read ");  
        visit(read.variable());  
        System.out.println(";");  
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());  
    }  
  
    public void visit(Variable variable) {  
        System.out.println(variable.name());  
    }  
}
```



IdealPrint.java

```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");  
    }  
  
    public void visit(Read read) {  
        System.out.print("read ");  
        visit(read.variable());  
        System.out.println(";");  
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());  
    }  
  
    public void visit(Variable variable) {  
        System.out.println(variable.name());  
    }  
}
```



*read a;
print a + b;*



IdealPrint.java

```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");  
    }  
  
    public void visit(Read read) {  
        System.out.print("read ");  
        visit(read.variable());  
        System.out.println(";");  
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());  
    }  
  
    public void visit(Variable variable) {  
        System.out.println(variable.name());  
    }  
}
```

Program

-sentences: List<Sentence>

Read

-variable: Variable

Print

-expression: Expression

Sum

-left: Expression
-right: Expression

¿Qué método
"visit" buscan?

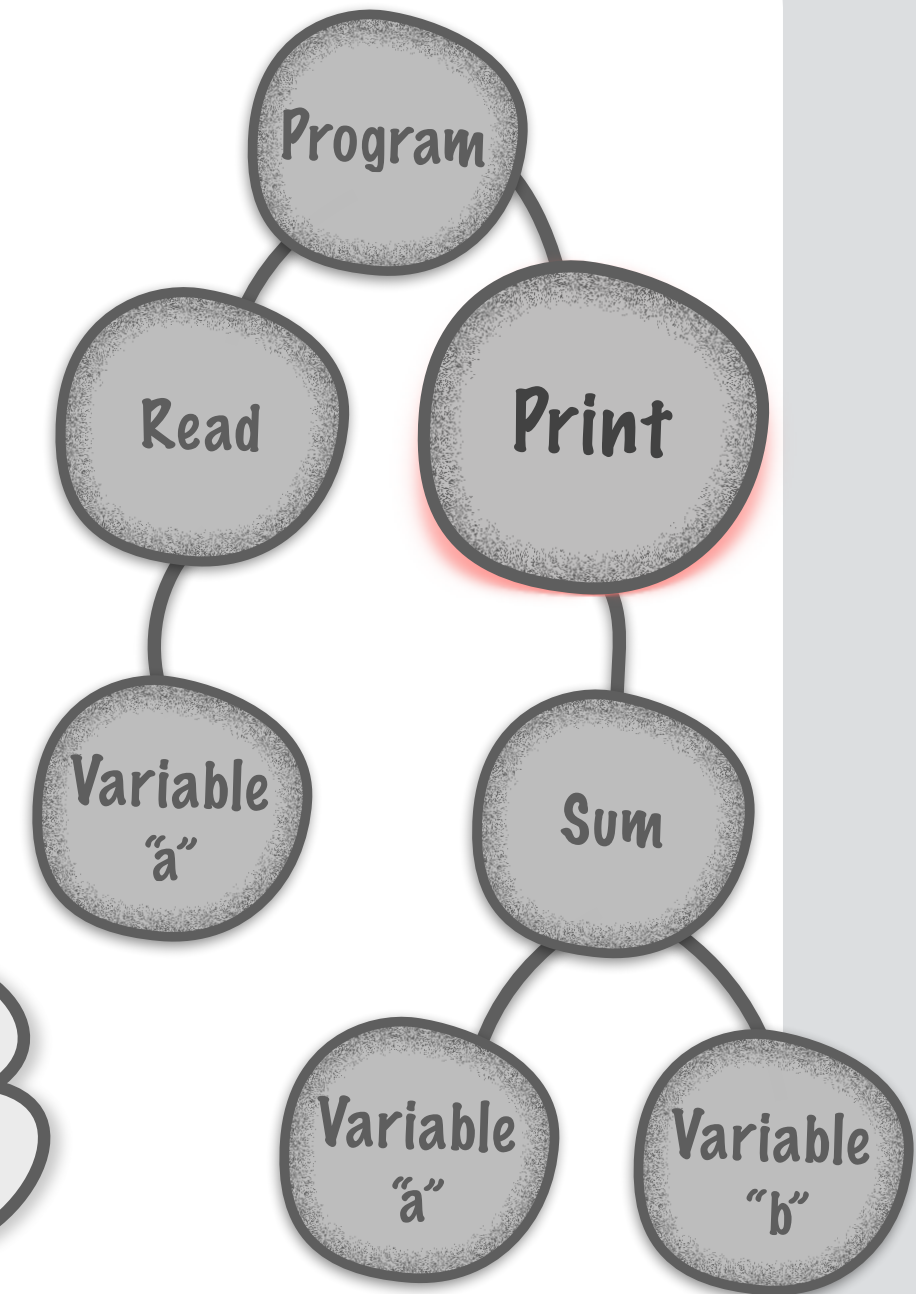
*read a;
print a + b;*



IdealPrint.java

```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statement()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");  
    }  
  
    public void visit(Read read) {  
        System.out.print("read ");  
        visit(read.variable());  
        System.out.println(";");  
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());  
    }  
  
    public void visit(Variable variable) {  
        System.out.println(variable.name());  
    }  
}
```

Lo que
deberían
buscar



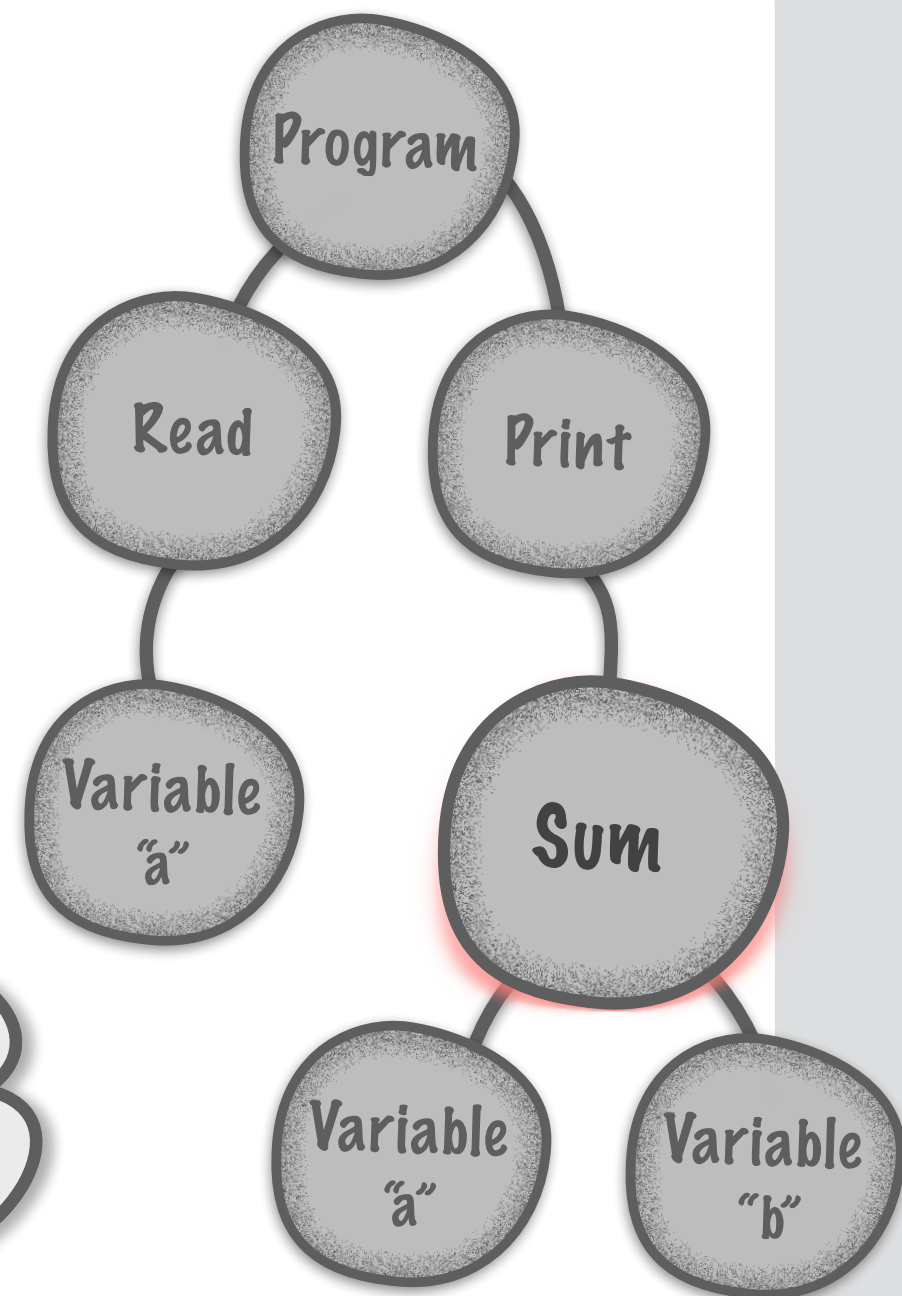
*read a;
print a + b;*



IdealPrint.java

```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statement()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");  
    }  
  
    public void visit(Read read) {  
        System.out.print("read ");  
        visit(read.variable());  
        System.out.println(";");  
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());  
    }  
  
    public void visit(Variable variable) {  
        System.out.println(variable.name());  
    }  
}
```

Lo que
deberían
buscar



*read a;
print a + b;*

Patrón Visitor



Para tener seguridad de tipos

Una interfaz Visitor con un método “visit” para cada nodo



Visitor.java

```
public interface Visitor {  
    void visitProgram(Program program);  
    void visitPrint(Print print);  
    void visitRead(Read read);  
    void visitSum(Sum sum);  
    void visitVariable(Variable variable);  
}
```

2

Para decidir a qué método llamar

Un método “accept” en cada
clase de nodo



```
public interface Node {  
    void accept(Visitor visitor);  
}  
  
public class Print implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visitProgram(this);  
    }  
}  
  
public class Read implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visitRead(this);  
    }  
}  
  
// ...
```



Una clase de visitor para cada operación

Dentro de los métodos **visit**, en vez de visitar directamente sus nodos hijos, debemos hacerlo llamando a su correspondiente método **accept**.



PrintVisitor.java

```
public class PrintVisitor implements Visitor {  
    public void visitProgram(Program program) {  
        for (Statement statement : program.statements()) {  
            statement.accept(this);  
        }  
    }  
  
    public void visitPrint(Print print) {  
        System.out.print("print ");  
        print.expression().accept(this);  
        System.out.println(";");  
    }  
  
    public void visitRead(Read read) {  
        System.out.print("read ");  
        read.variable().accept(this);  
        System.out.println(";");  
    }  
  
    // ...  
}
```

No llamar nunca a `visit` directamente desde un método `visit`. En vez de eso, llamar al `accept` de cada hijo.



PrintVisitor.java

```
public class PrintVisitor implements Visitor {  
    public void visitProgram(Program program) {  
        for (Statement statement : program.statements()) {  
            statement.accept(this);  
        }  
    }  
  
    public void visitPrint(Print print) {  
        System.out.print("print ");  
        print.expression().accept(this);  
        System.out.println(";");  
    }  
  
    public void visitRead(Read read) {  
        System.out.print("read ");  
        read.variable().accept(this);  
        System.out.println(";");  
    }  
  
    // ...  
}
```

No llamar nunca a visit directamente desde un método visit. En vez de eso, llamar al accept de cada hijo.



Visitar el árbol



PrintVisitorTest.java

```
public static void main(String[] args) {  
    // Build the Abstract Syntax Tree  
    Program program = new Program();  
    // ...  
  
    Visitor print = new PrintVisitor();  
    print.visit(program);  
}
```

¿Hace falta que cada “visit” se llame diferente?



Visitor.java

```
public interface Visitor {  
    void visitProgram(Program program);  
    void visitPrint(Print print);  
    void visitRead(Read read);  
    void visitSum(Sum sum);  
    void visitVariable(Variable variable);  
}
```



Visitor.java

```
public interface Visitor {  
    void visitProgram(Program program);  
    void visitPrint(Print print);  
    void visitRead(Read read);  
    void visitSum(Sum sum);  
    void visitVariable(Variable variable);  
}
```



Visitor.java

```
public interface Visitor {  
    void visit(Program program);  
    void visit(Print print);  
    void visit(Read read);  
    void visit(Sum sum);  
    void visit(Variable variable);  
}
```



```
public interface Node {  
    void accept(Visitor visitor);  
}  
  
public class Print implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visitProgram(this);  
    }  
}  
  
public class Read implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visitRead(this);  
    }  
}  
  
// ...
```



```
public interface Node {  
    void accept(Visitor visitor);  
}  
  
public class Print implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visitProgram(this);  
    }  
}  
  
public class Read implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visitRead(this);  
    }  
}  
  
// ...
```



```
public interface Node {  
    void accept(Visitor visitor);  
}  
  
public class Print implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}  
  
public class Read implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}  
  
// ...
```


Qué tenemos hasta ahora



PrintVisitor.java

```
public class PrintVisitor implements Visitor {  
    public void visitProgram(Program program) {  
        for (Statement statement : program.statements()) {  
            statement.accept(this);  
        }  
    }  
  
    public void visitPrint(Print print) {  
        System.out.print("print ");  
        print.expression().accept(this);  
        System.out.println(";");  
    }  
  
    public void visitRead(Read read) {  
        System.out.print("read ");  
        read.variable().accept(this);  
        System.out.println(";");  
    }  
  
    // ...  
}
```



Visitor.java

```
public interface Visitor {  
    void visit(Program program);  
    void visit(Print print);  
    void visit(Read read);  
    void visit(Sum sum);  
    void visit(Variable variable);  
}
```

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

```
public class Print implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}
```

```
public class Read implements Statement {  
    // ...  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
}
```



PrintVisitor.java

```
public class PrintVisitor implements Visitor {  
    public void visitProgram(Program program) {  
        for (Statement statement : program.statements)  
            statement.accept(this);  
    }  
  
    public void visitPrint(Print print) {  
        System.out.print("print ");  
        print.expression().accept(this);  
        System.out.println(";");  
    }  
  
    public void visitRead(Read read) {  
        System.out.print("read ");  
        read.variable().accept(this);  
        System.out.println(";");  
    }  
  
    // ...  
}
```

Generalizando el patrón Visitor

El principal motivo para usar un visitor es poder añadir nuevas operaciones.

¿Qué ocurre si otro recorrido necesita pasar parámetros a algunos métodos “visit”, o que devuelvan algo?

Necesitamos generalizar la solución para que se pueda añadir cualquier nuevo visitante.



Visitor.java

```
public interface Visitor {  
    Object visit(Program program, Object param);  
    Object visit(Print print, Object param);  
    Object visit(Read read, Object param);  
    Object visit(Sum sum, Object param);  
    Object visit(Variable variable, Object param);  
}
```



```
public interface Node {
    Object accept(Visitor visitor, Object param);
}

public class Print implements Statement {
    // ...
    public Object accept(Visitor visitor, Object param) {
        return visitor.visit(this, param);
    }
}

public class Object implements Statement {
    // ...
    public Object accept(Visitor visitor, Object param) {
        return visitor.visit(this, param);
    }
}

// ...
```



PrintVisitor.java

```
public class PrintVisitor implements Visitor {

    public Object visitProgram(Program program, Object param) {
        for (Statement statement : program.statement()) {
            statement.accept(this, null);
        }
        return null;
    }

    public Object visitPrint(Print print, Object param) {
        System.out.print("print ");
        print.expression().accept(this, null);
        System.out.println(";");
        return null;
    }

    public Object visitRead(Read read, Object param) {
        System.out.print("read ");
        read.variable().accept(this, null);
        System.out.println(";");
        return null;
    }

    // ...

}
```


Resumen

Implementar el patrón Visitor

Estos pasos se harán solo una vez.

- 1) Crear la interfaz Visitor con un método para cada nodo.



Visitor.java

```
public interface Visitor {  
    Object visit(Program program, Object param);  
    Object visit(Print print, Object param);  
    // ...  
}
```

Implementar el patrón Visitor

Estos pasos se harán solo una vez.

2) Añadir un método accept a la interfaz raíz.



Node.java

```
public interface Node {  
    Object accept(Visitor visitor, Object param);  
    // Other (non visitor related) methods  
}
```

Implementar el patrón Visitor

Estos pasos se harán solo una vez.

3) Implementar el método accept en cada clase de nodo.

La implementación es **igual** en todos ellos (podemos **copiar y pegar**).



```
public Object accept(Visitor visitor, Object param) {  
    return visitor.visit(this, param);  
}
```

Añadir un visitante concreto

Uno por cada operación a realizar sobre la estructura.

Simplemente crearemos una nueva clase que implemente la interfaz Visitor.



SomeVisitor.java

```
public class SomeVisitor implements Visitor {  
    Object visit(Program program, Object param) { ... }  
    Object visit(Print print, Object param) { ... }  
    // ...  
}
```

¡No hace falta tocar los nodos!