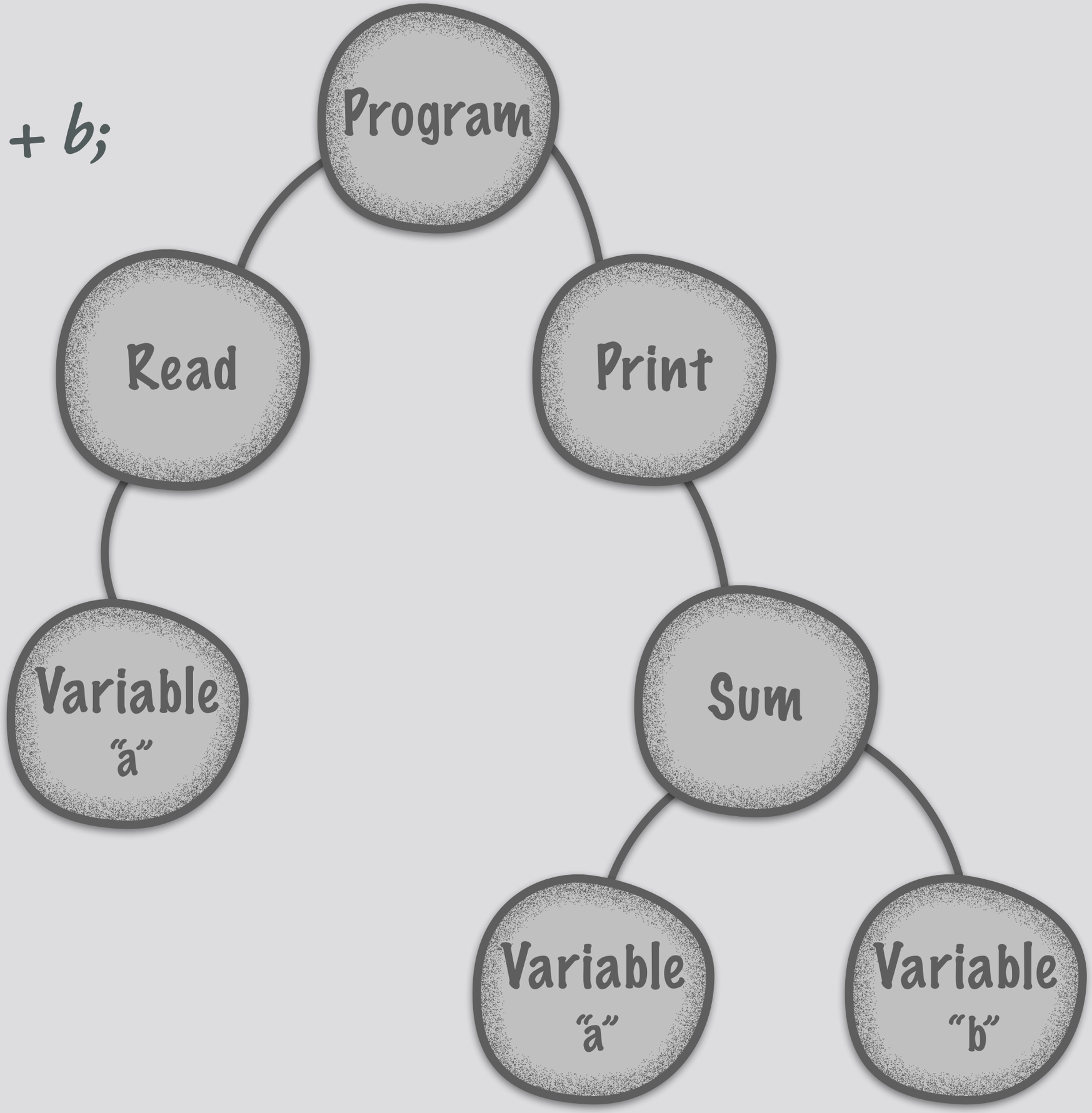
Introduction

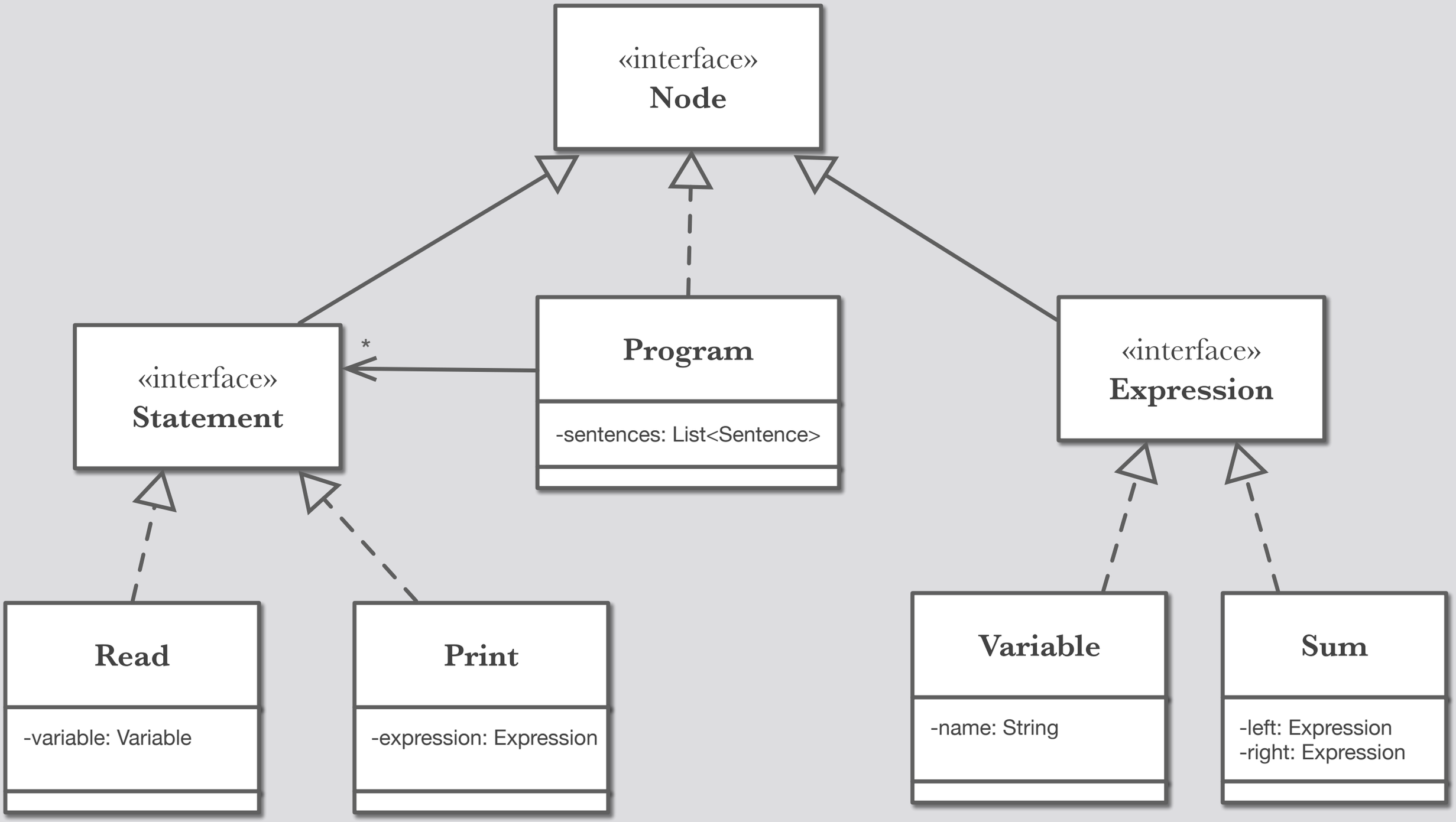
We are creating a compiler for programs written in some programming language.

We want to be able to model—to represent in memory, as objects—programs like this:

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

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




«interface»
Node
«interface»
Statement
*
Program
-sentences: List<Sentence>
«interface»
Expression
Read
-variable: Variable
Print
-expression: Expression
Variable
-name: String
Sum
-left: Expression
-right: Expression

```
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;
}
```

Now, several operations have to be implemented.

Pretty printing the program

Semantic analysis

Type checking

Code generation

Doc generation

And those that come in a future.

Who should be the responsible for implementing those operations?

Who should be the responsible for implementing those operations?

Two possibilities:

a) Decentralised implementation

The nodes themselves

b) Centralised implementation

Each operation in its own class

The nodes themselves

Interpreter pattern

(What we did in the laboratory session 2)

Each operation will be distributed among the different nodes.

In other words, each node will have a method for each operation.

Print.java

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

Sum.java

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

Print.java

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

Sum.java

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

Problems of this approach?

Solution

Each operation in its own class

With a method for dealing with every node.

It makes easy adding new operations.

How to implement it?

How to implement every operation with the centralised approach?

Two possibilities:

a) Recursive traverse

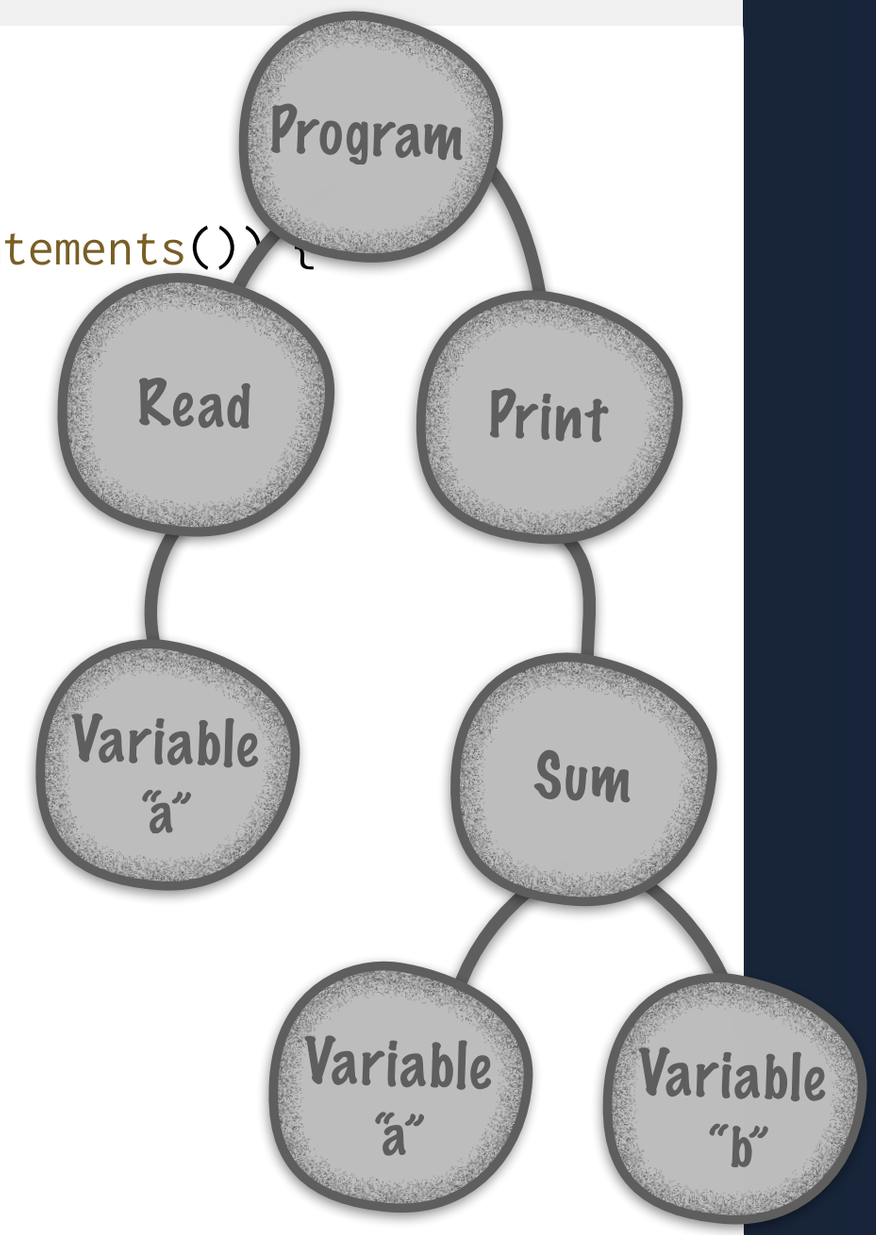
b) The Visitor pattern

RecursivePrintTest.java

```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

```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(((Suma) node).right);
  } else if (node instanceof Variable) {
      System.out.println(((Variable) node).name());
  }
}
```



read a;
print a + b;

Problems of that implementation

All code is in a single method.

It is true that it could be improved extracting the code for each conditional branch to a private method.

But, even so, there would be needed the ‘main’ conditional logic and instanceofs in **visit(Node)** for deciding which method to invoke.

We do not have type safety.

This is what we would like

IdealPrint.java

```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());
  }
}
```

The problem is

It does not compile!

Why not?

The Problem

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

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

```java
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?
```

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

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

```java
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?
```

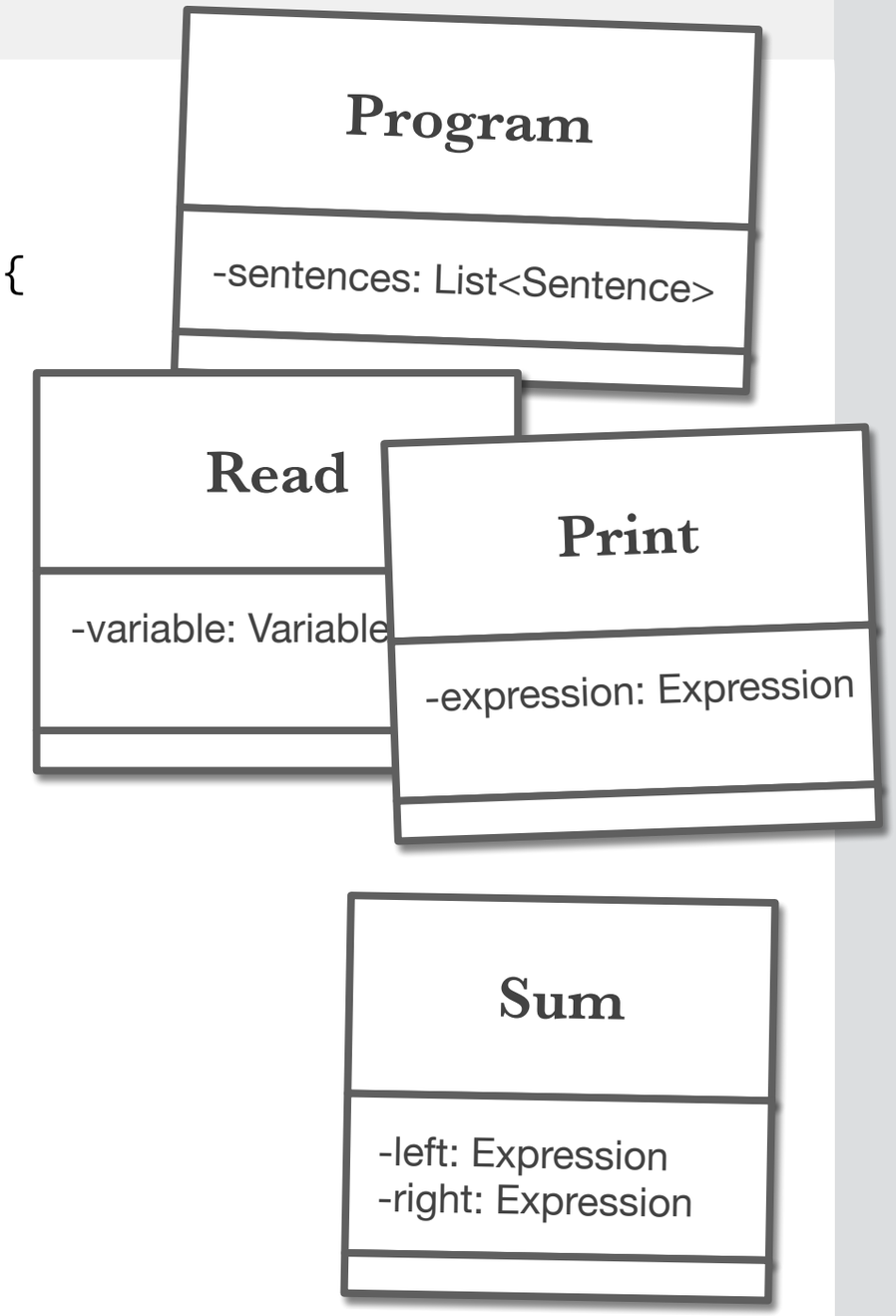
Let us go back to our example.

IdealPrint.java

```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

```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());
  }
}
```

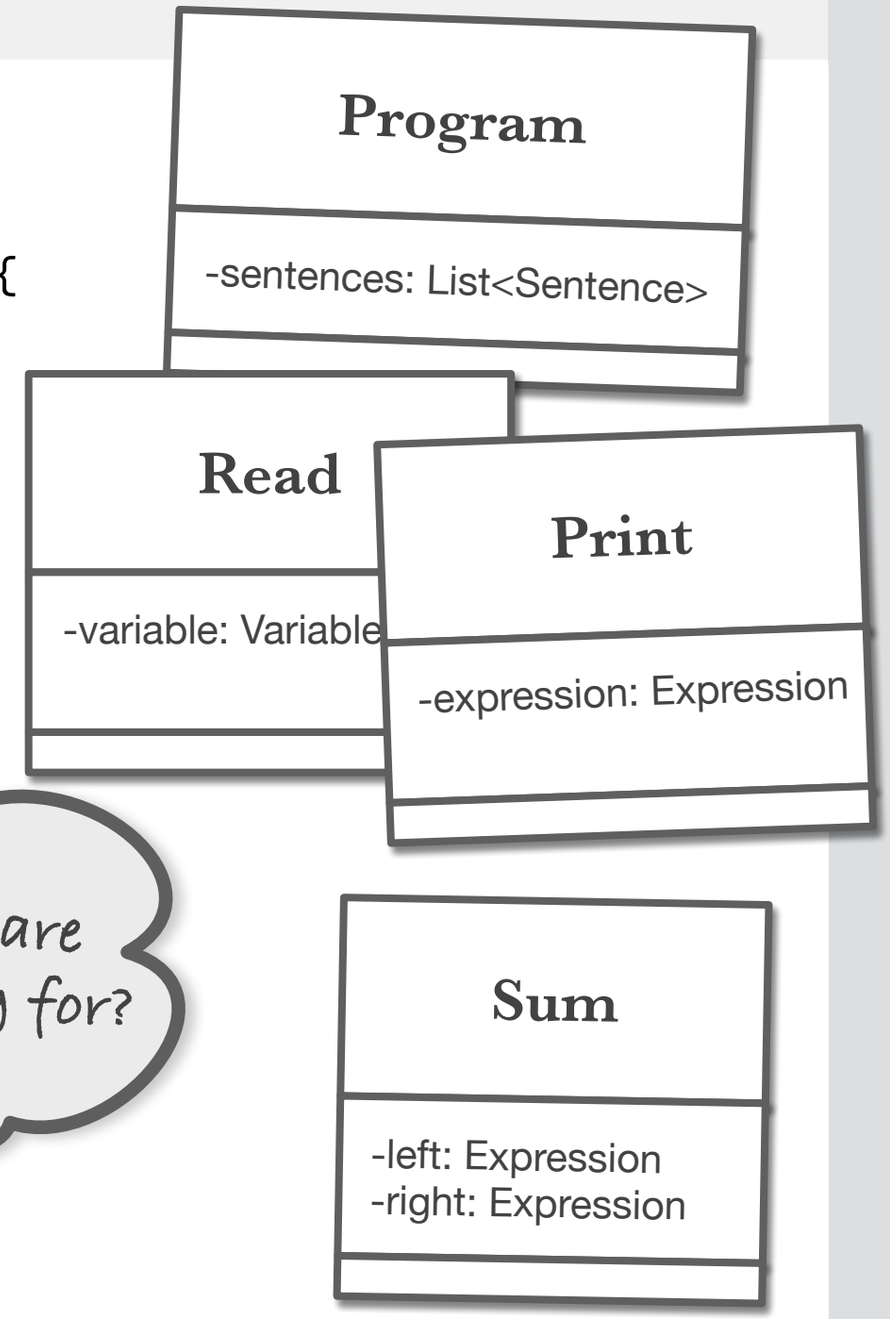


read a;
print a + b;

IdealPrint.java

```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());
  }
}
```

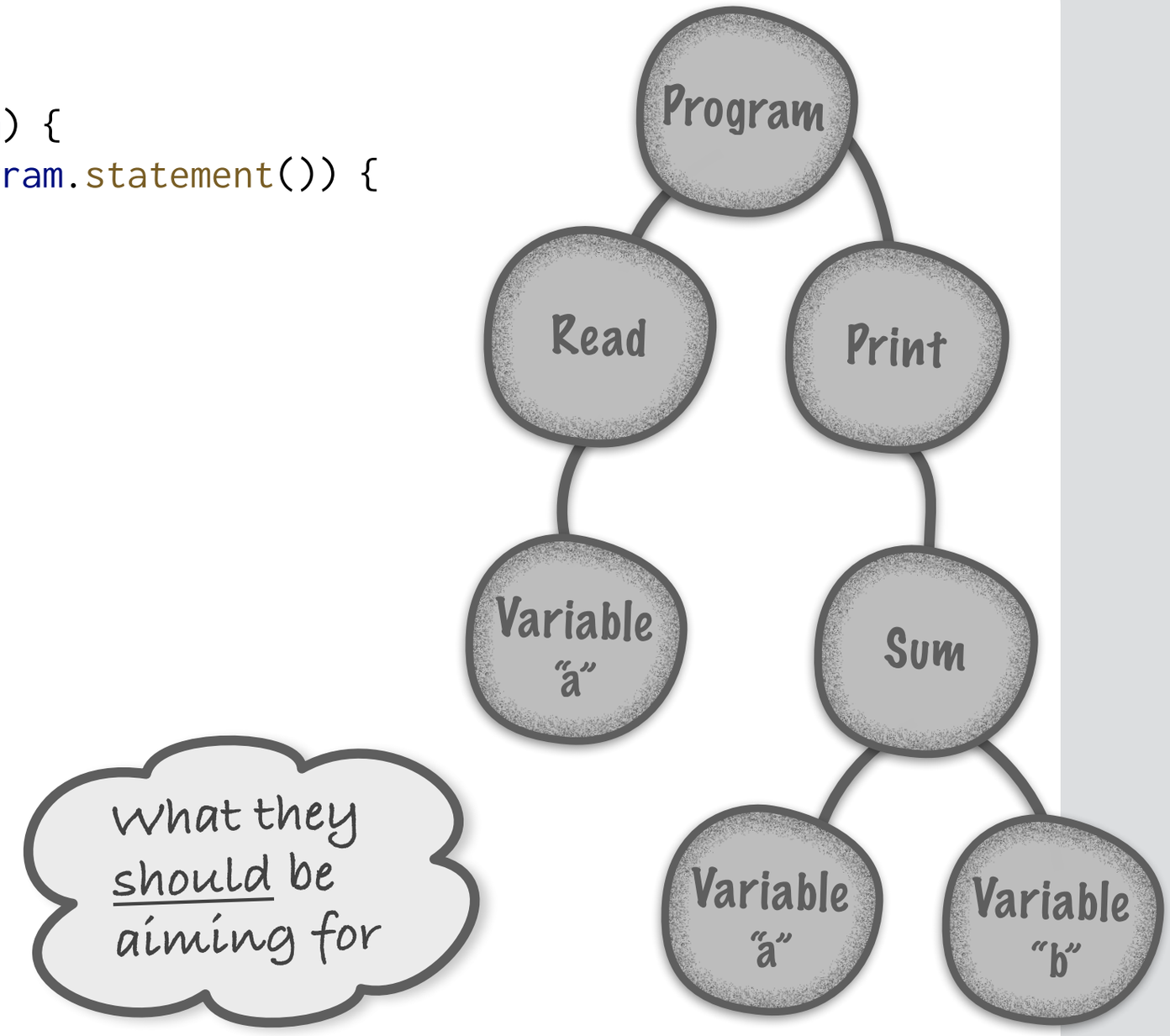
What 'visit' are they looking for?



read a;
print a + b;

IdealPrint.java

```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());
  }
}
```

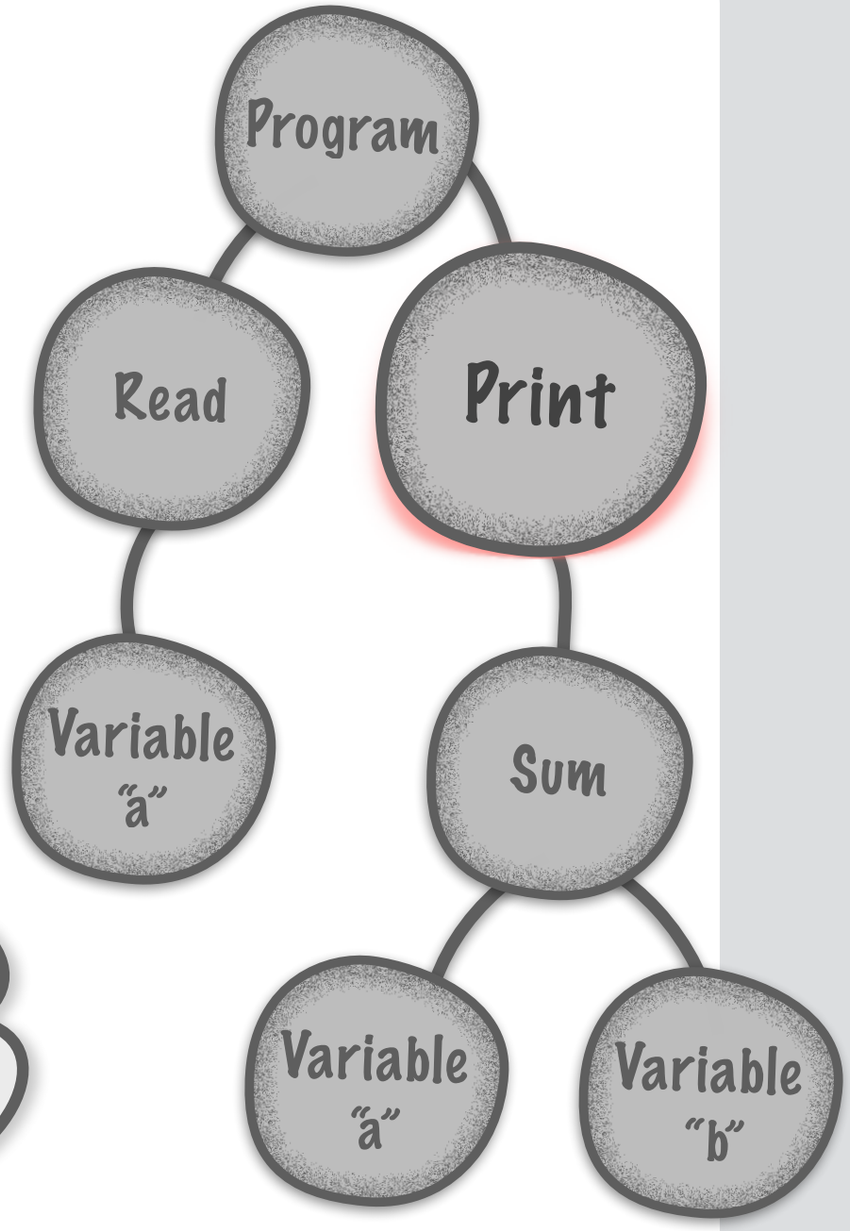


read a;
print a + b;

IdealPrint.java

```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());
  }
}
```

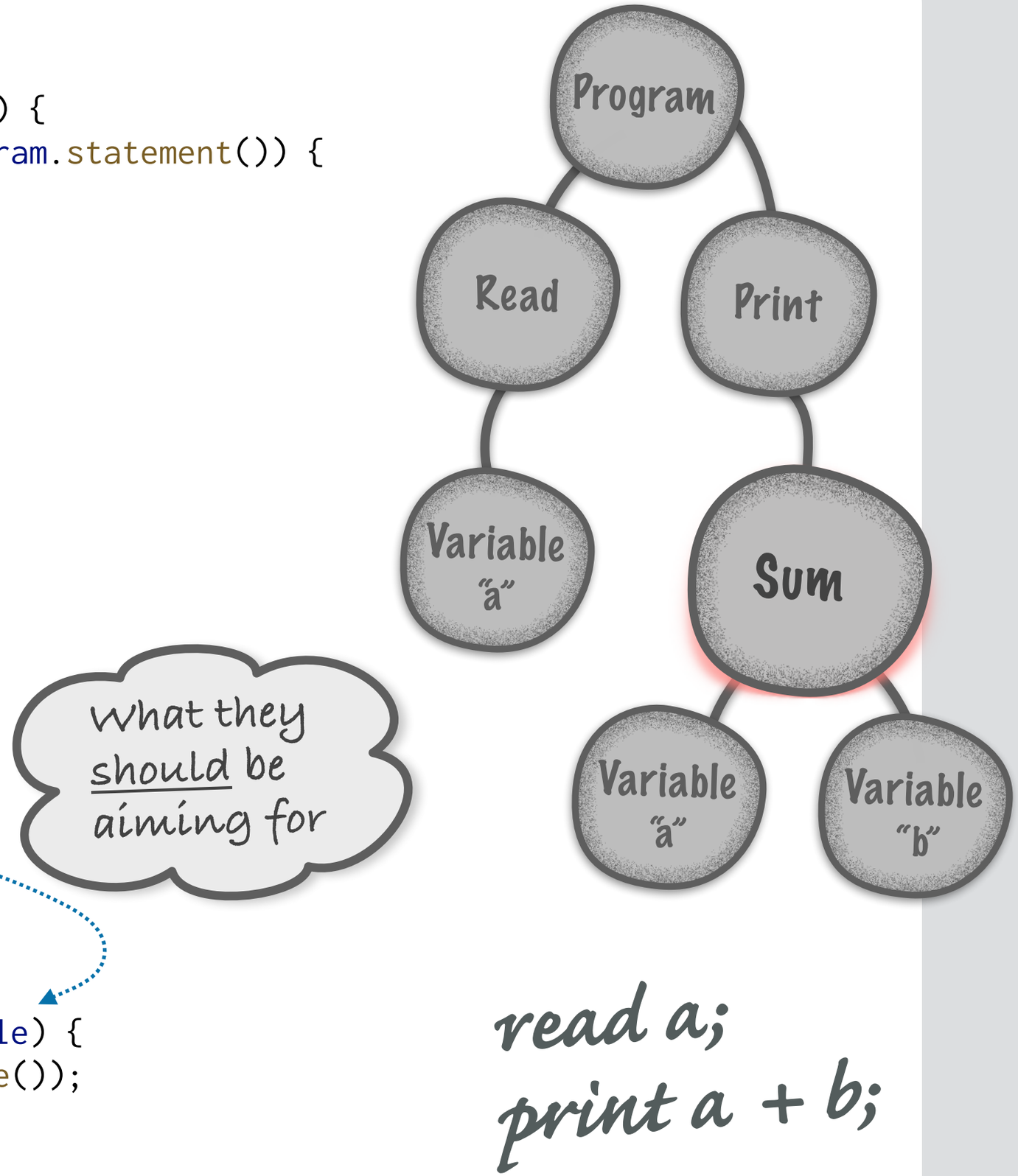
What they should be aiming for



read a;
print a + b;

IdealPrint.java

```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());
  }
}
```



Visitor Pattern

1

To provide type safety

An interface Visitor with a ‘visit’ method for every node

Visitor.java

```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

To decide what visit method to call

An ‘accept’ method in every node class

```java
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);
  }
}

// ...
```

3

A visitor class for every operation

Inside **visit** methods, instead of directly visiting a node's children, you must call their corresponding **accept**.

PrintVisitor.java

```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(";");
  }

  // ...

}
```

Never call visit directly from within a visit method. Use the accept method of the node's children instead

PrintVisitor.java

```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(";");
  }

  // ...

}
```

Never call visit directly from within a visit method. Use the accept method of the node's children instead.

4

Visiting the tree

PrintVisitorTest.java

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

  Visitor print = new PrintVisitor();
  print.visit(program);
}
```

Is it necessary for each ‘visit’ method to have a distinct name?

Visitor.java

```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

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

```java
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);
  }
}

// ...
```

```java
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);
  }
}

// ...
```

```java
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);
  }
}

// ...
```

What we have so far

PrintVisitor.java

```java
public class PrintVisitor implements Visitor {

  public void visit(Program program) {
    for (Statement statement : program.statements()) {
      statement.accept(this);
    }
  }

  public void visit(Print print) {
    System.out.print("print ");
    print.expression().accept(this);
    System.out.println(";");
  }

  public void visit(Read read) {
    System.out.print("read ");
    read.variable().accept(this);
    System.out.println(";");
  }

  // ...

}
```

Visitor.java

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

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

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

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

PrintVisitor.java

```java
public class PrintVisitor implements Visitor {

  public void visit(Program program) {
    for (Statement statement : program.statement
      statement.accept(this);
    }
  }

  public void visit(Print print) {
    System.out.print("print ");
    print.expression().accept(this);
    System.out.println(";");
  }

  public void visit(Read read) {
    System.out.print("read ");
    read.variable().accept(this);
    System.out.println(";");
  }

  // ...
```

Generalising the Visitor pattern

The main reason to use a Visitor is to be able to add easily new operations.

What happens if another visitor needs to pass arguments to some visit methods, or if they need to return a value?

We need to generalise the solution to accommodate any new visitor.

Visitor.java

```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);
}
```

```java
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

```java
public class PrintVisitor implements Visitor {

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

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

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

  // ...

}
```

Summary

Implementing the Visitor pattern

This steps are made just once.

1) Define a Visitor interface with a visit method for each node.

Visitor.java

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

Implementing the Visitor pattern

This steps are made just once.

2) Add an accept method to the root interface.

Node.java

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

Implementing the Visitor pattern

This steps are made just once.

3) Implement the accept method in every node class.

The implementation is **the same** in all of them (it can be **copied and pasted**).

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

Implementing a new visitor

A new visitor for every operation over the node structure

We simply create a new class that implements the Visitor interface.

SomeVisitor.java

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

The nodes remain unchanged!