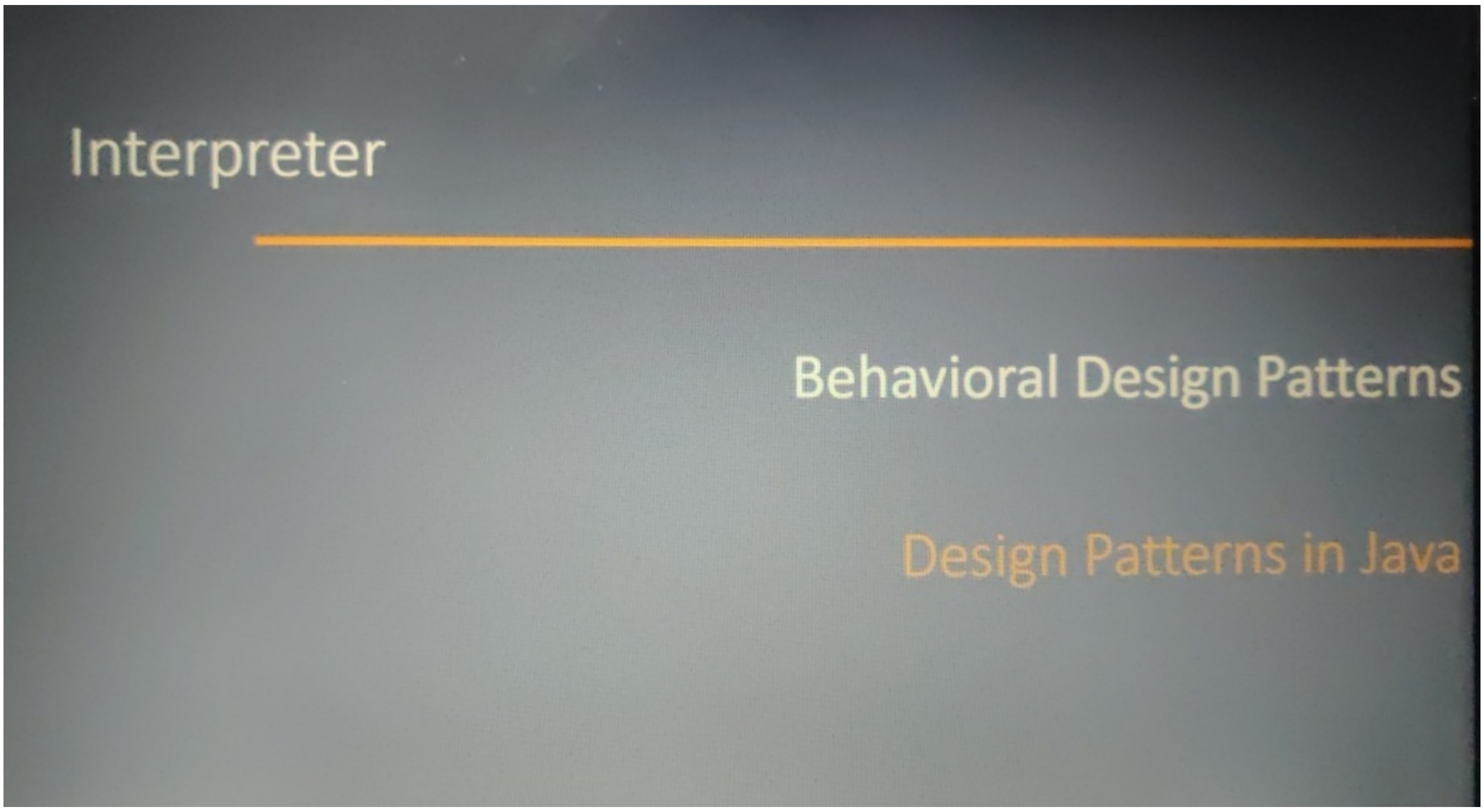
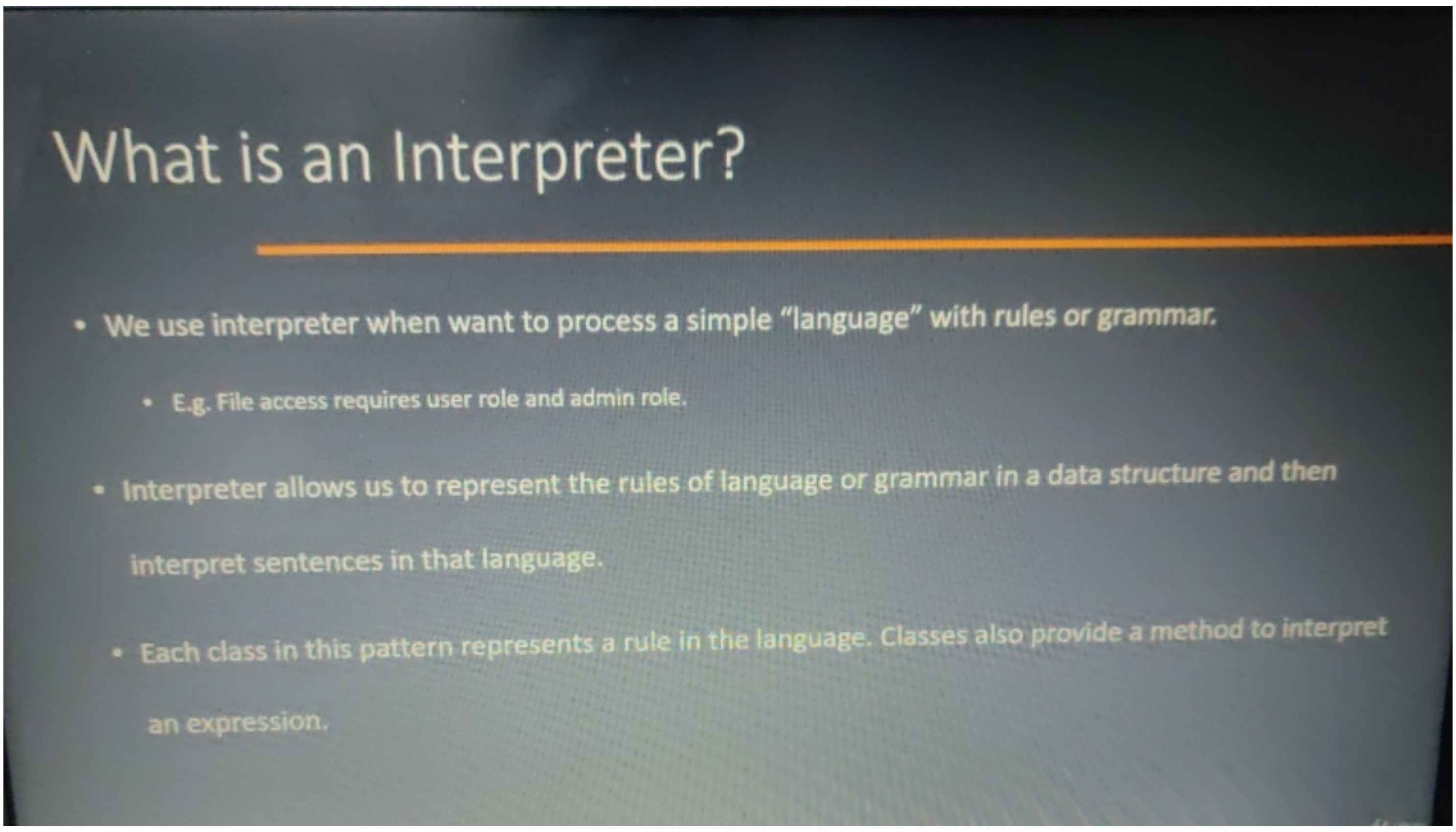


Interpreter

Behavioral Design Patterns

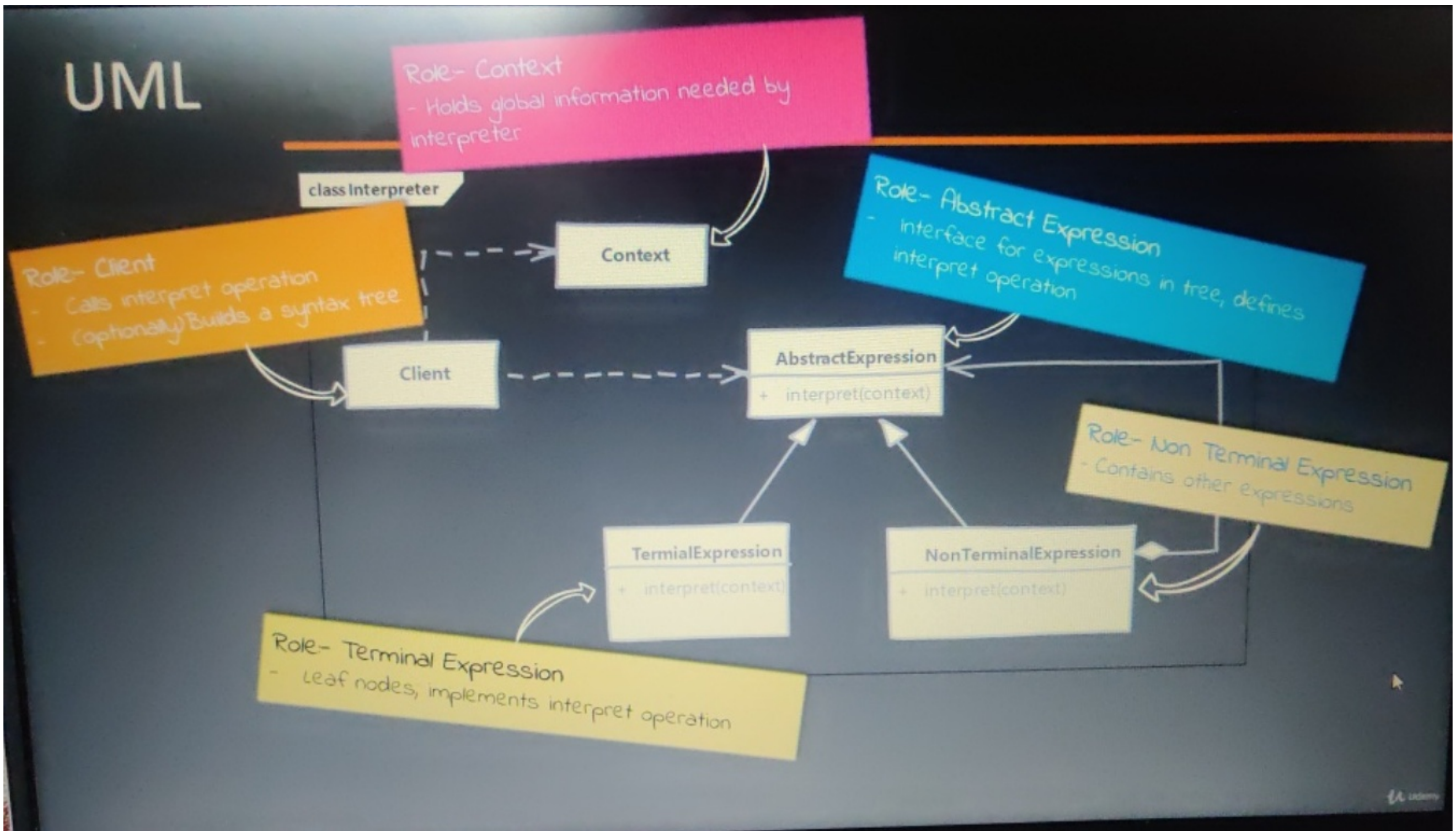
Design Patterns in Java



What is an Interpreter?

- We use interpreter when want to process a simple "language" with rules or grammar.
  - E.g. File access requires user role and admin role.
- Interpreter allows us to represent the rules of language or grammar in a data structure and then interpret sentences in that language.
  - Each class in this pattern represents a rule in the language. Classes also provide a method to interpret an expression.

UML



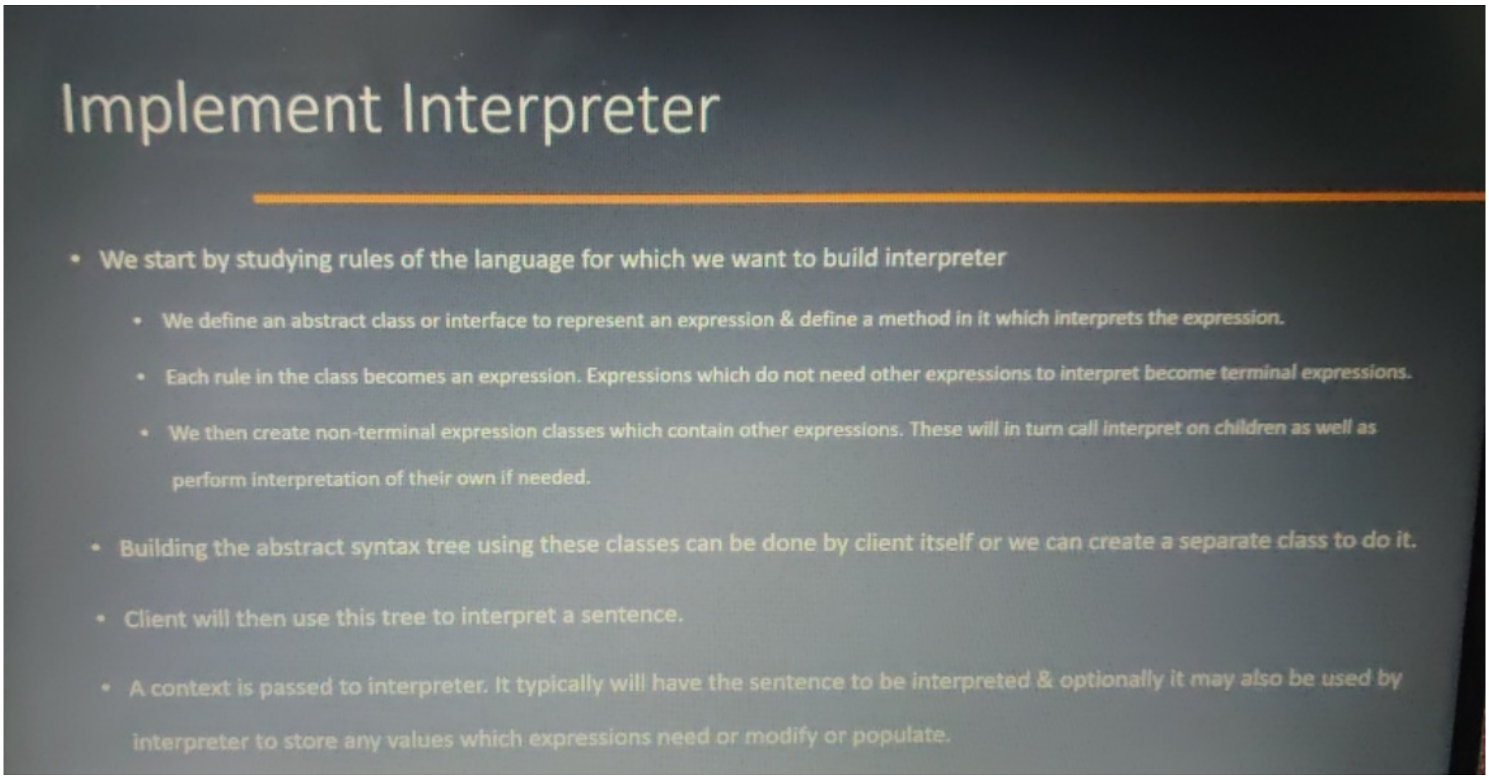
Role- Context
- Holds global information needed by interpreter

Role- Client
- Calls interpret operation
- (optionally) Builds a syntax tree

Role- Abstract Expression
- interface for expressions in tree, defines interpret operation

Role- Non Terminal Expression
- Contains other expressions

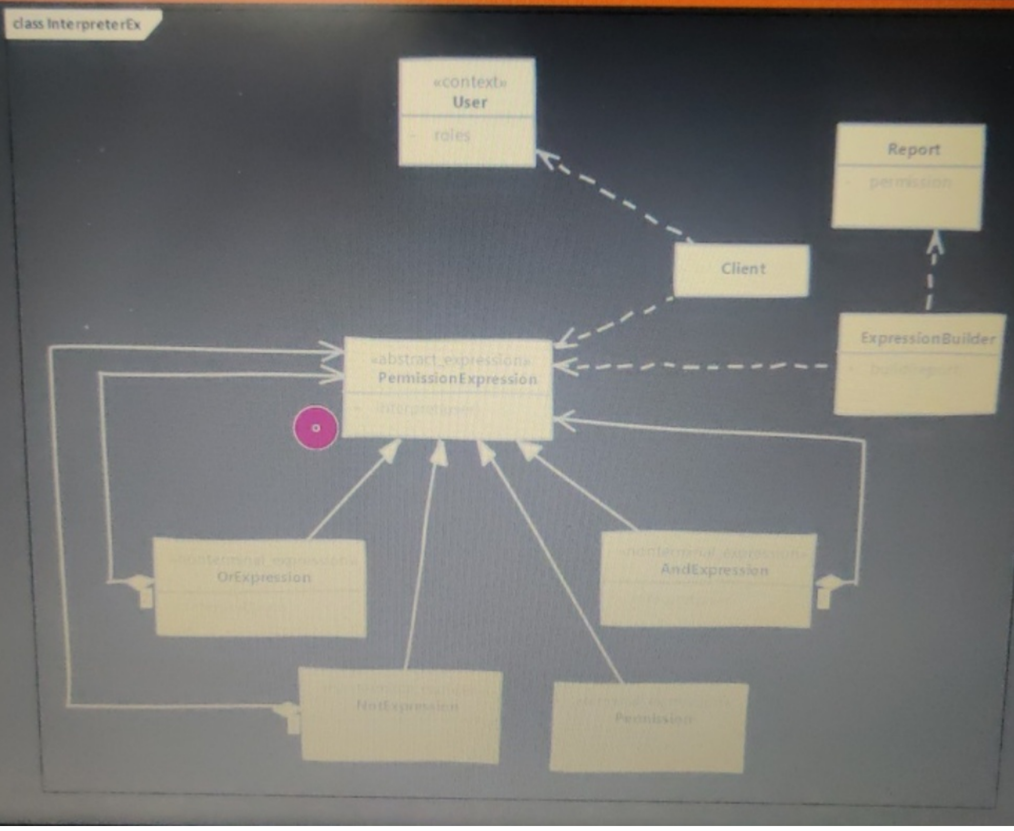
Role- Terminal Expression
- Leaf nodes, implements interpret operation

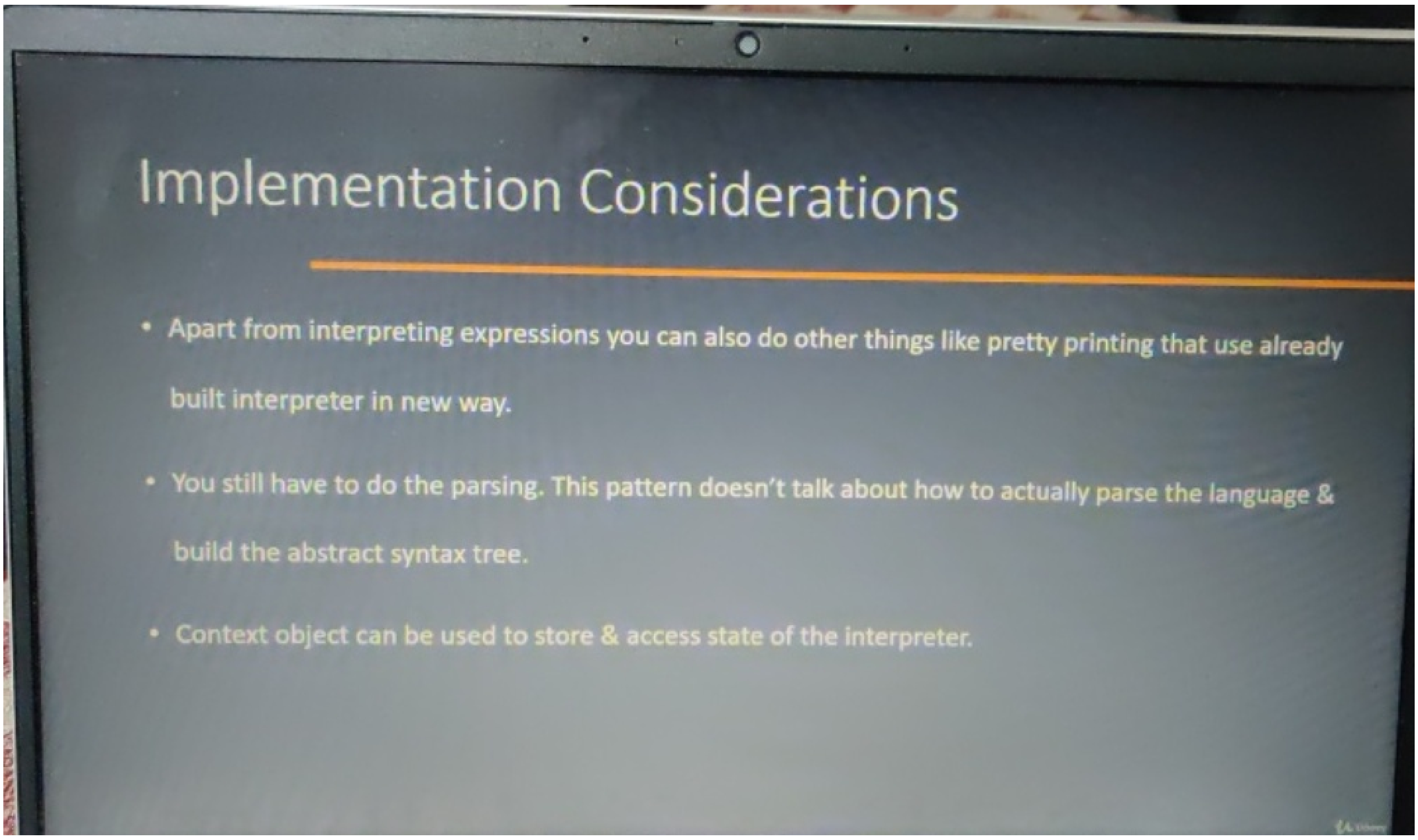


Implement Interpreter

- We start by studying rules of the language for which we want to build interpreter
  - We define an abstract class or interface to represent an expression & define a method in it which interprets the expression.
  - Each rule in the class becomes an expression. Expressions which do not need other expressions to interpret become terminal expressions.
  - We then create non-terminal expression classes which contain other expressions. These will in turn call interpret on children as well as perform interpretation of their own if needed.
- Building the abstract syntax tree using these classes can be done by client itself or we can create a separate class to do it.
- Client will then use this tree to interpret a sentence.
- A context is passed to interpreter. It typically will have the sentence to be interpreted & optionally it may also be used by interpreter to store any values which expressions need or modify or populate.

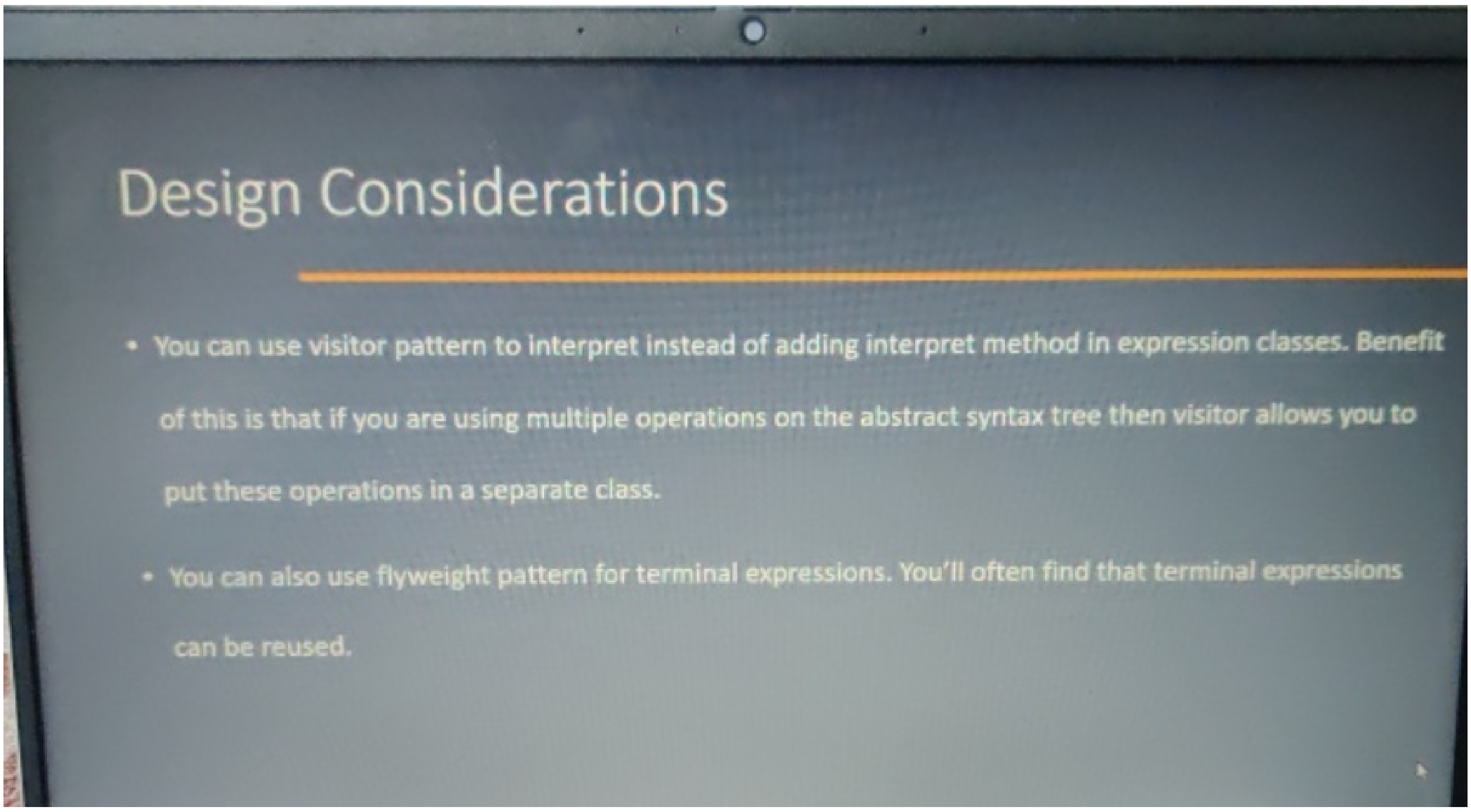
ample: UML


class InterpreterEx
«context»
User
- roles
Report
- permission
Client
ExpressionBuilder
«abstract_expression»
PermissionExpression
+ interpret(user)
«nonterminal_expression»
OrExpression
«nonterminal_expression»
AndExpression
NotExpression
Permission



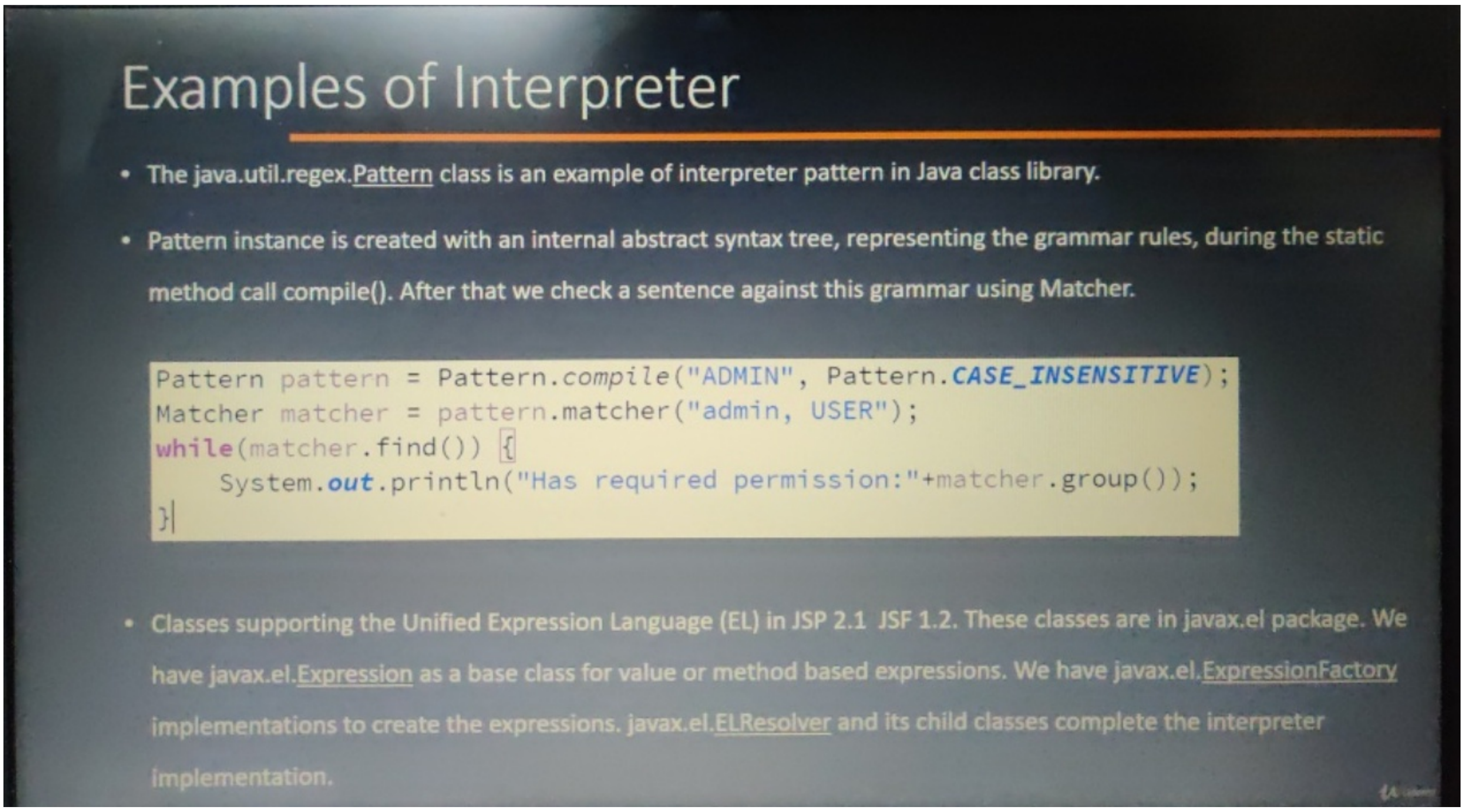
Implementation Considerations

- Apart from interpreting expressions you can also do other things like pretty printing that use already built interpreter in new way.
- You still have to do the parsing. This pattern doesn't talk about how to actually parse the language & build the abstract syntax tree.
- Context object can be used to store & access state of the interpreter.



Design Considerations

- You can use visitor pattern to interpret instead of adding interpret method in expression classes. Benefit of this is that if you are using multiple operations on the abstract syntax tree then visitor allows you to put these operations in a separate class.
- You can also use flyweight pattern for terminal expressions. You'll often find that terminal expressions can be reused.

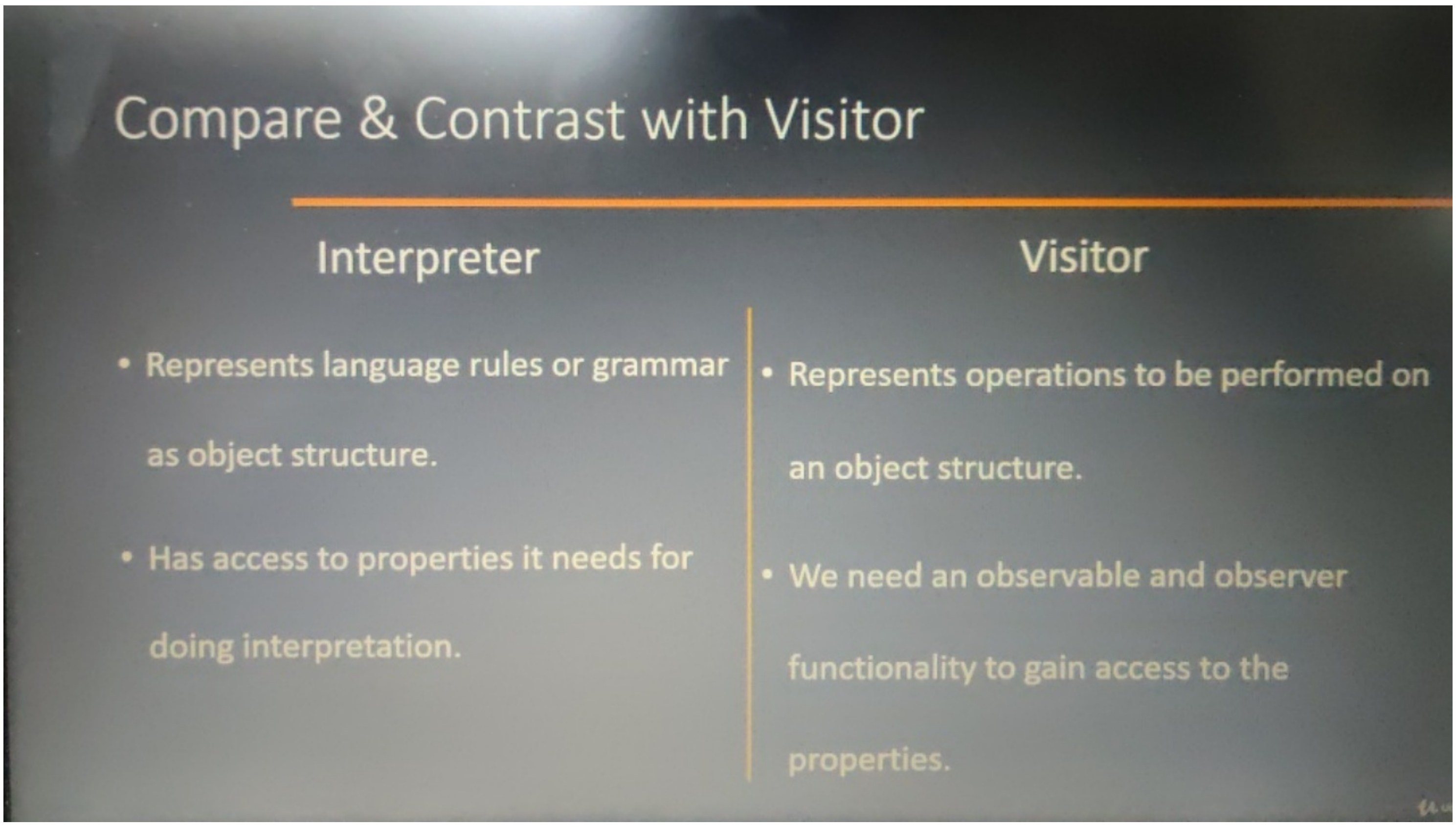


Examples of Interpreter

- The java.util.regex.Pattern class is an example of interpreter pattern in Java class library.
- Pattern instance is created with an internal abstract syntax tree, representing the grammar rules, during the static method call compile(). After that we check a sentence against this grammar using Matcher.

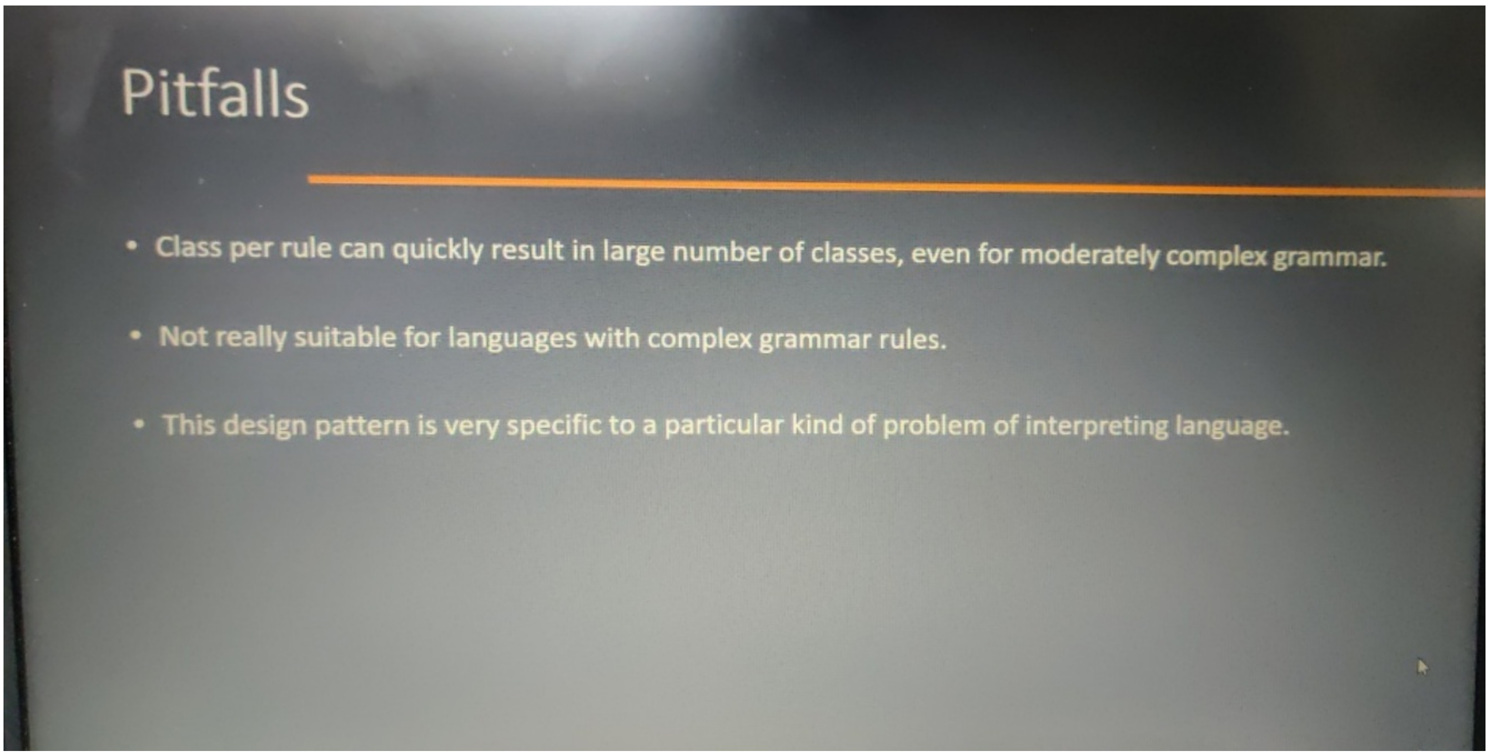
```
Pattern pattern = Pattern.compile("ADMIN", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher("admin, USER");
while(matcher.find()) {
    System.out.println("Has required permission:"+matcher.group());
}
```

- Classes supporting the Unified Expression Language (EL) in JSP 2.1 JSF 1.2. These classes are in javax.el package. We have javax.el.Expression as a base class for value or method based expressions. We have javax.el.ExpressionFactory implementations to create the expressions. javax.el.ELResolver and its child classes complete the interpreter implementation.



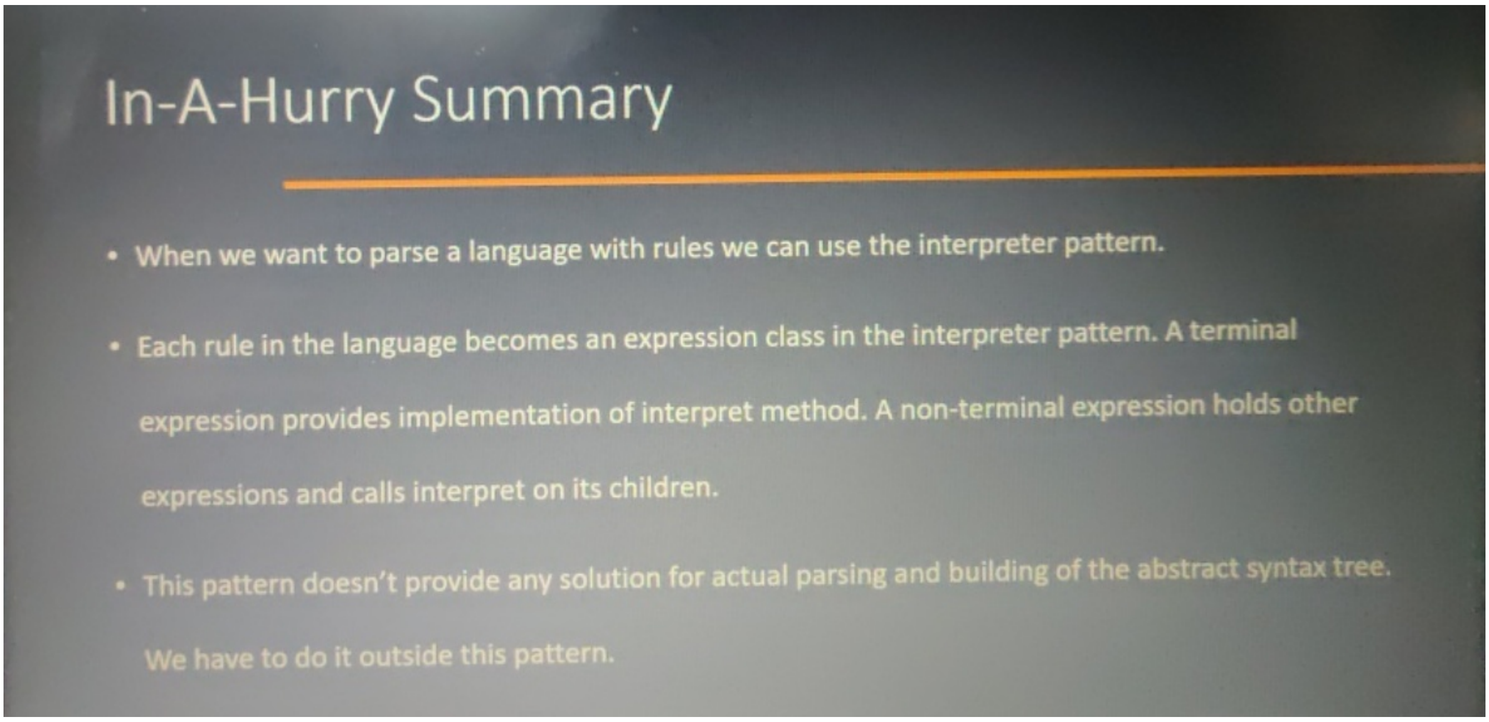
Compare & Contrast with Visitor

Interpreter	Visitor
• Represents language rules or grammar as object structure. • Has access to properties it needs for doing interpretation.	• Represents operations to be performed on an object structure. • We need an observable and observer functionality to gain access to the properties.



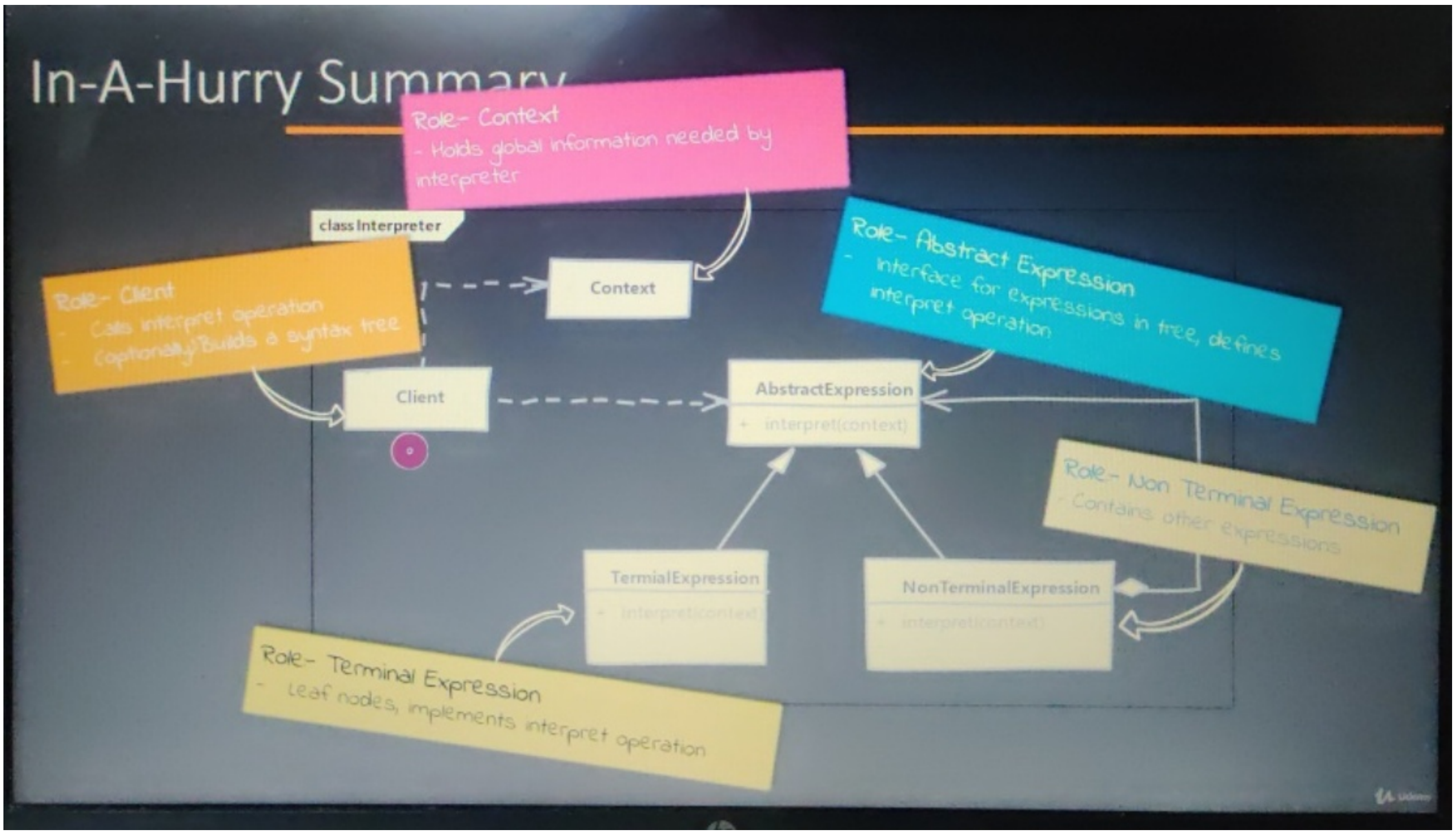
Pitfalls

- Class per rule can quickly result in large number of classes, even for moderately complex grammar.
- Not really suitable for languages with complex grammar rules.
- This design pattern is very specific to a particular kind of problem of interpreting language.



In-A-Hurry Summary

- When we want to parse a language with rules we can use the interpreter pattern.
- Each rule in the language becomes an expression class in the interpreter pattern. A terminal expression provides implementation of interpret method. A non-terminal expression holds other expressions and calls interpret on its children.
- This pattern doesn't provide any solution for actual parsing and building of the abstract syntax tree. We have to do it outside this pattern.



In-A-Hurry Summary

Role- Context
- Holds global information needed by interpreter

Role- Client
- Calls interpret operation
- (optionally) Builds a syntax tree

Role- Abstract Expression
- Interface for expressions in tree, defines interpret operation

Role- Non Terminal Expression
- Contains other expressions

Role- Terminal Expression
- Leaf nodes, implements interpret operation