# Overview:

* The Interpreter design pattern provides a way to evaluate a language grammar or expression.
* The formal definition is “Give a language; define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language”.
  + We define a grammatical representation or a language and provide an interpreter to deal with that grammar.
* Involves implementing an expression interface, which tells how to interpret a particular context.
* The interpreter pattern requires some knowledge of formal grammars.

# Grammars:

* A grammar is a way to represent valid sentences in a language.
* It defines a language using special symbols and syntax.
* It is also define by specifying a number of rules:
  + Each rule specifying how one of a selection of sequences of atoms and symbols can replace a single symbol.
* A grammar consists of the following:
  + A set of variables or non-terminal symbols.
  + A set of Terminal symbols.
  + S is a special variable called the Start symbol.
  + P represents Production rules for Terminals and Non-terminals.

# Grammar Example:

({S,A,B} , {a,b} , S, {S -> AB, A -> a, B ->b})

* S,A and B are Non-terminal symbols (variables)
* a and b are Terminal symbols (actual characters)
* S is the Start symbol.
* Productions, P: S -> AB, A -> a, B -> b.
  + The production rules are applied in any order, until a string that contains neither the start symbol nor designated non-terminal symbols is produced.
* A single rule is applied to a string by replacing one occurrence of the production rule’s left-hand side in the string by that production rule’s right-hand side.
  + The language formed by the grammar consist of all distinct string that can be generated in this manner.
    - Any particular sequence of production rules on the start symbol yields a distinct string in the language.

# Back to the interpreter:

* When you need to implement a simple language you can use the interpreter pattern:
  + Describes ow to define a grammar for simple languages, represent sentences in the language, and interpret these sentences.
* To represent the language, you use a class to represent each rule in the language.
* Searching for strings match a pattern is a common problem.
  + Regular expressions are standard language for specifying patterns of strings.
  + Search algorithms could interpret a regular expression that specifies a set of strings to match.

# Examples:

* A language translator who translates a language for us:
  + Google translator where the input can be in any language and we can get the output interpreted in another language.
* We can consider music notes as grammar and musicians as our interpreters.
* A Java compiler interprets the source code into byte code:
  + Byte code is understandable by JVM (Java Virtual Machine).
* Widely used to interpret statements in a language as abstract syntax trees.
* Used in SQL parsing and a symbol-processing engine.
* Java.util.Pattern and subclasses of java.text.Format are some of the examples of interpreter pattern used in JDK.

# Advantages and drawbacks:

* Easy to implement if each grammar rule is represented by a class.
  + Allows you to easily change or extend the language.
  + By adding methods to the class structure, you can add new behaviors beyond interpretation.
* The pattern makes it easier to evaluate an expression in a new way
  + You can support pretty printing or type checking an expression by defining a new operation on the expression classes.
* One drawback is that the number of grammar rules is large, it is harder to maintain the code:
  + The interpreter pattern defines at least one class for every rule in the grammar.
  + In these cases, a parser/compiler generator may be more appropriate.
* Requires a lot of error checking and a lot of expressions and code to evaluate.

# When to use this pattern?

* Use the interpreter pattern when there is a language to interpret and when the language is simple, (grammar does not have many rules).
  + Should be able to represent statements in the language as abstract syntax trees.
* Appropriate when simplicity is more important than efficiency.
  + Most efficient interpreters are usually not implemented by interpreting pares trees directly but by first translating them into another form.
    - Regular expressions are often transformed into state machines.
* Used for scripting and programming languages.

# Implementation:

* **AbstractExpression:**
  + Declares an abstract interpreter operation that is common to all nodes in the abstract syntax tree.
* **TerminalExpression:**
  + Implements an Interpret operation associated with terminal symbols in the grammar.
  + An instance is required for every terminal symbol in a sentence.
* **NonTerminalExpression:**
  + One such class is required for every rule R::= R1R2 … in the grammar.
  + Maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn
  + Implements an interpreter operation for non-terminal symbols in the grammar.
    - Interpret typically calls itself recursively on the variables representing R1 through Rn.
* **Context:**
  + Contains information that is global to the interpreter.
* **Client:**
  + Builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines.
    - The abstract syntax tree is assembled from instances of the NonTerminalExpression and TerminalExpression classes.
  + Invokes the interpret operation.

# Implementation issues:

* The interpreter pattern does not explain how to create an abstract syntax tree:
  + Does not address parsing.
  + Will need to be created by a table-driven parser, by a hand-crafted parser, or directly by the client.
* You do not have to define the Interpret operation in the expression classes.
* Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol by using the visitor pattern.
* Terminal nodes generally do not store information about their position in the abstract syntax tree.