Java로 배우는 디자인패턴 입문

Chapter 23. Interpreter

문법 규칙을 클래스로 표현한다.

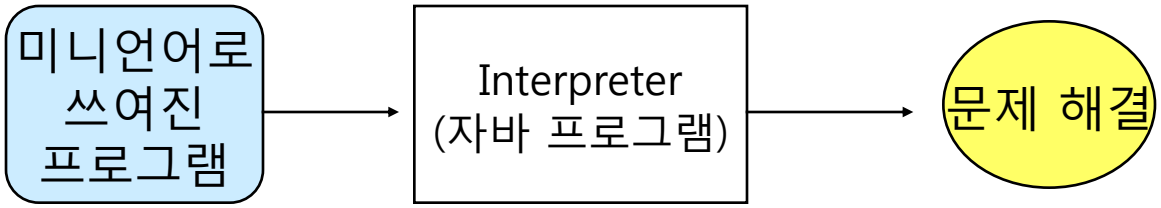
교재: 자바언어로배우는디자인패턴입문(개정판)/YukiHiroshi저/김윤정역/영진닷컴

2012-1

덕성여자대학교 정보미디어대학

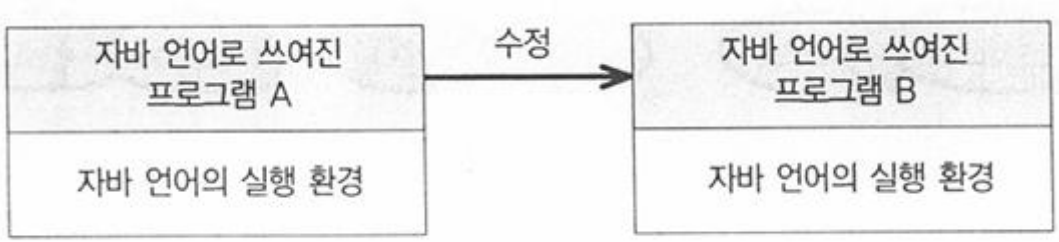
01. Interpreter 패턴

- 프로그램이 해결하려고 하는 문제를 간단한 '미니 언어'로 표현함
  - 구체적인 문제를 미니 언어로 쓰여진 '미니 프로그램'으로 표현한다.
  - 미니 프로그램을 자바 언어로 '통역'하는 역할을 하는 프로그램을 만든다.
  - 통역 프로그램은 미니 언어를 이해하고 미니 프로그램을 해석 및 실행한다.
  - 해결해야 할 문제에 변화가 생겼을 때, 프로그램을 고치지 않고 미니 프로그램을 고쳐서 해결한다.

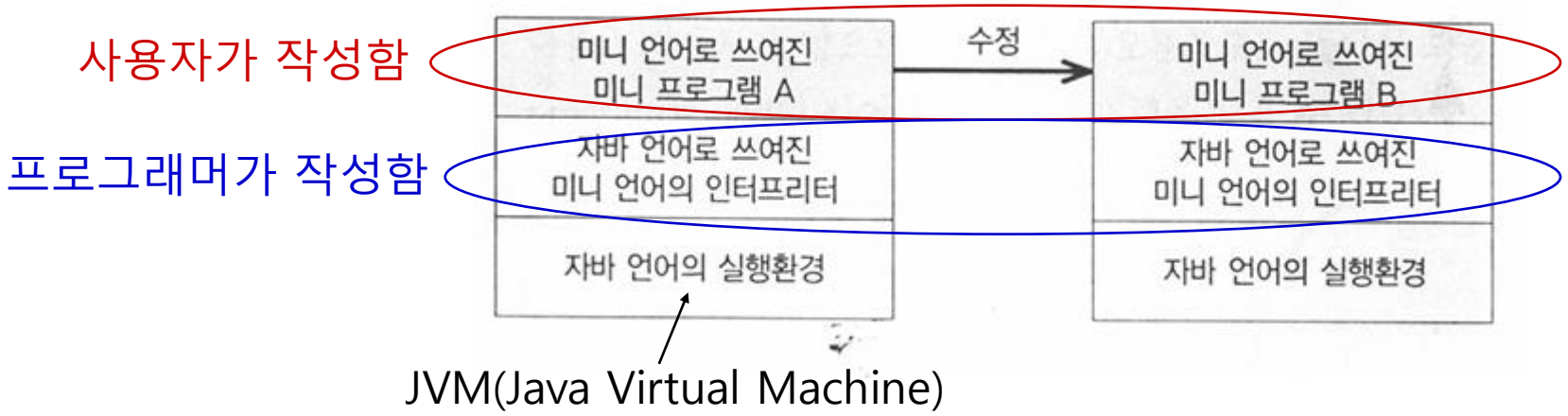


01. Interpreter 패턴

- 그림 23-1 과 23-2
  - Interpreter 패턴 적용 시, 해결하고자 하는 문제에 변화가 생겼을 때 미니 언어로 쓰여진 프로그램만 수정하면 된다.
  - Interpreter 패턴 사용 안 하는 경우

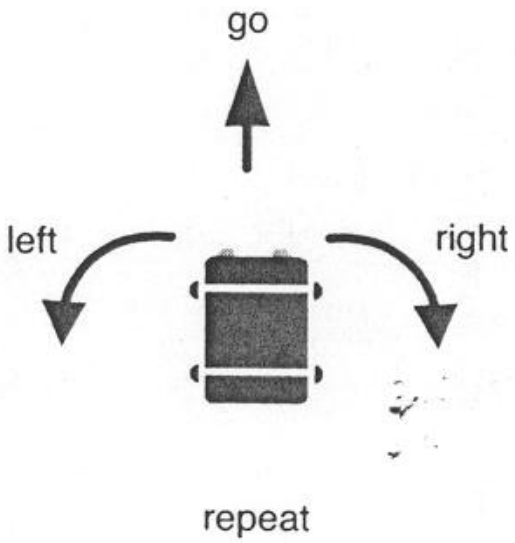


  - Interpreter 패턴을 사용하는 경우



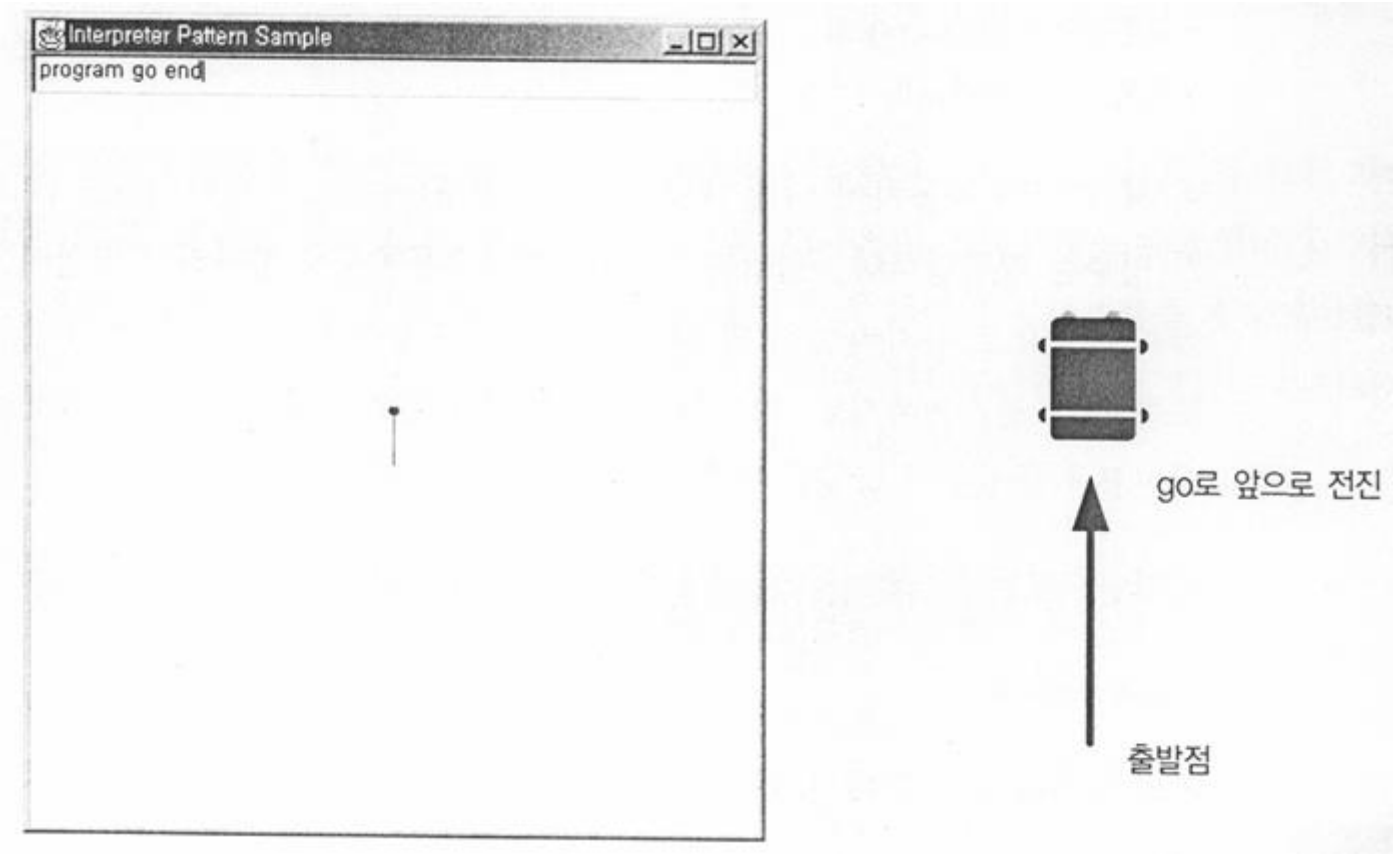
02. 미니 언어

- "미니 언어"로 작성된 명령
  - 예: 무선 조정기로 자동차 움직이기
    - 자동차에 내릴 수 있는 명령
      - 앞으로 1미터 전진(go) / 우회전(right) / 좌회전(left)
      - 반복(repeat)
      - 좌/우회전은, 제자리에서 회전하는 것으로 가정함(좌향좌, 우향우)



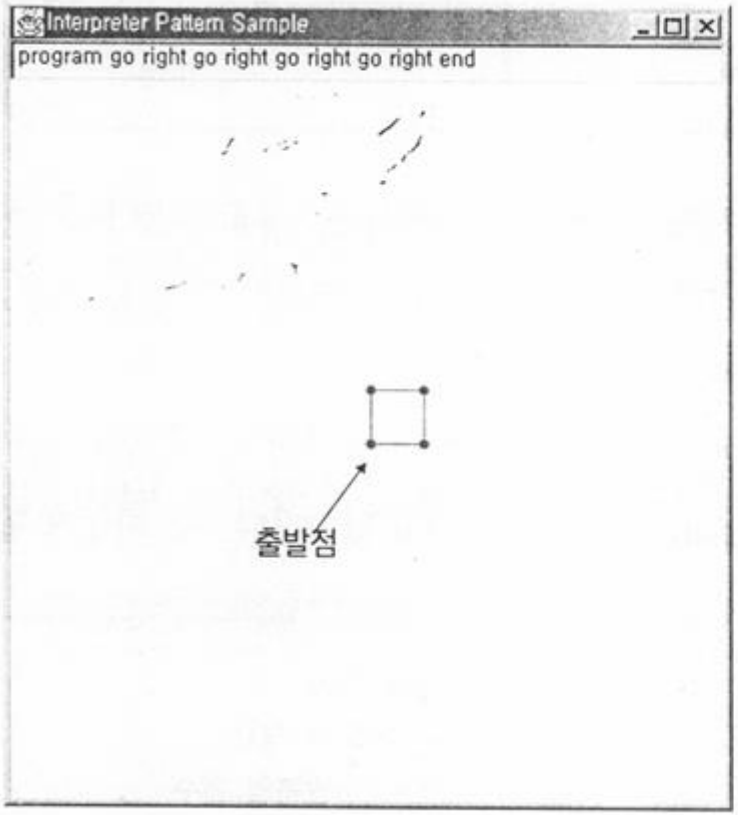
02. 미니 언어

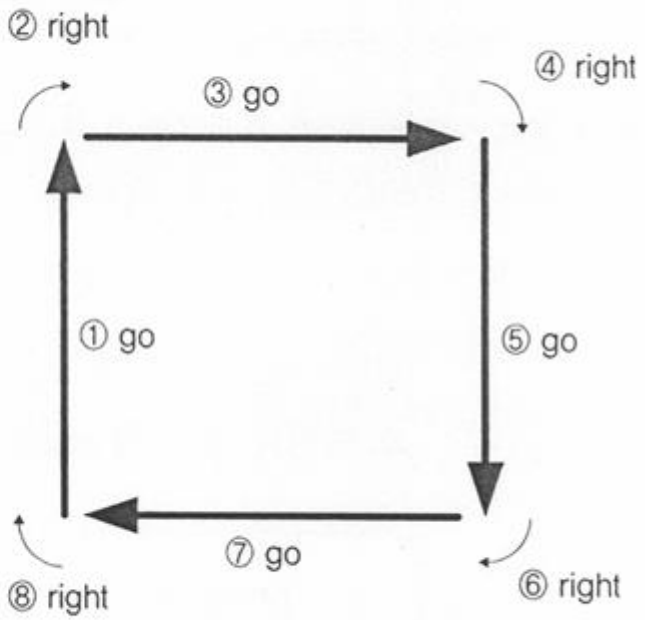
- 미니 언어로 작성된 미니 프로그램의 예
  - program go end



02. 미니 언어

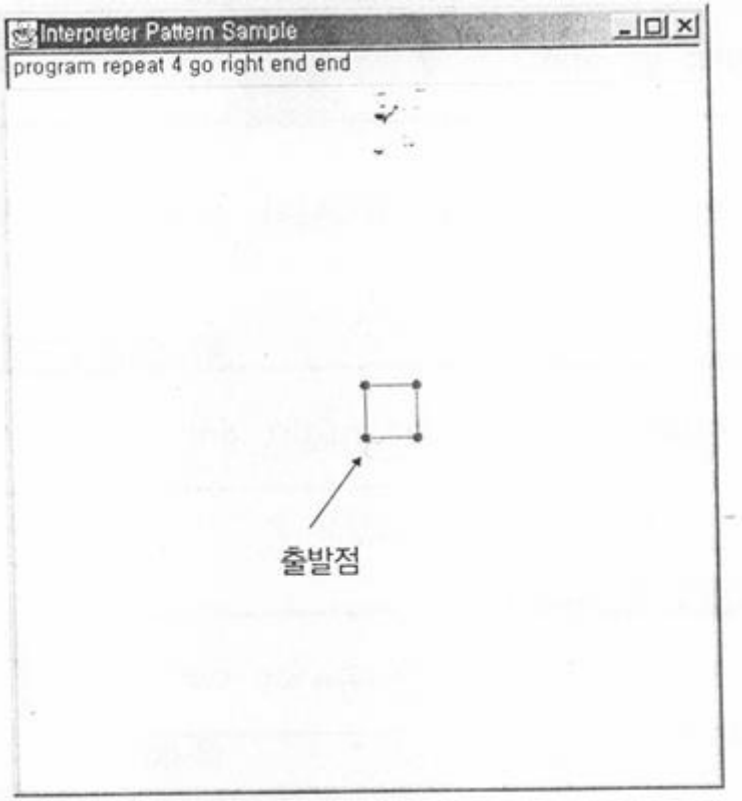
- 미니 프로그램의 예 (계속)
  - program go right go right go right go right end





02. 미니 언어

- 미니 프로그램의 예 (계속)
  - program repeat 4 go right end end

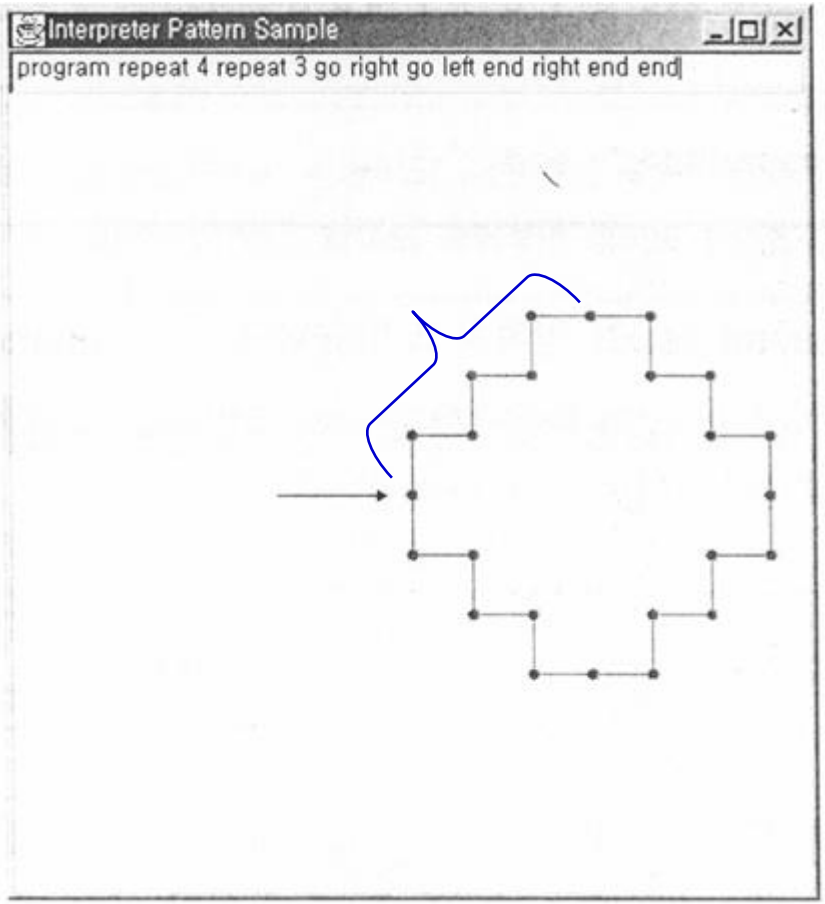


02. 미니 언어

- 미니 프로그램의 예 (계속)
  - program repeat 4 repeat 3 go right go left end right end end

```
program
  repeat
    4
    repeat
      3
      go right
      go left
    end
    right
  end
end
```

을 3번 반복



02. 미니 언어

- 미니 언어의 문법
  - BNF(Backus-Naur Form 또는 Backus Normal Form)의 확장으로 문법을 표기(정의)함

start symbol

```
<program> : : = program <command list>
<command list> : : = <command> * end
<command> : : = <repeat command> | <primitive command>
<repeat command> : : = repeat <number> <command list>
<primitive command> : : = go | right | left
```

02. 미니 언어

미니 언어의 문법 (계속)

- <program> ::= program <command list>
  - '<program> 이란, program 이라는 단어 뒤에 <command list>가 이어진 것' 이는 정의를 나타낸다.

- <command list> ::= <command>* end
  - '<command list>는, <command>가 0개 이상 반복된 후 end라는 단어가 온 것' 이라는 뜻
  - * : 앞의 것을 0번 이상 반복한다는 의미

- <command> ::= <repeat command> | <primitive command>
  - '<command>란, <repeat command> 또는 <primitive command> 둘 중 하나'라는 뜻
  - | : '또는' 이라는 의미

02. 미니 언어

❑ 미니 언어의 문법(계속)

- <repeat command> ::= repeat <number><command list>
  - '<repeat command>란, repeat 라는 단어 뒤에 반복횟수 <number>가 이어지고, 다시 <command list>가 이어진 것'이라는 뜻
  - 그런데, <command list>는 이미 정의되어 있다.
    - <command list> 정의 안에 <command>가 사용되고,
    - <command> 정의 안에 <repeat command>가 사용되고,
    - <repeat command> 정의 안에 <command list>가 사용된다.

    => 즉, <command list> 정의 시, <command list>가 다시 등장한다.
  - '재귀적인 정의' : 어떤 것을 정의하는데 자기 자신이 등장하는 경우
    - 재귀적인 메소드: 메소드가 자기 자신을 호출함.

02. 미니 언어

- 미니 언어의 문법(계속)
  - <primitive command> ::= go | right | left
    - '<primitive command>란, go 또는 right 또는 left 이다'라는 뜻
  - <number>는, 숫자로 이루어진 자연수를 나타냄

02. 미니 언어

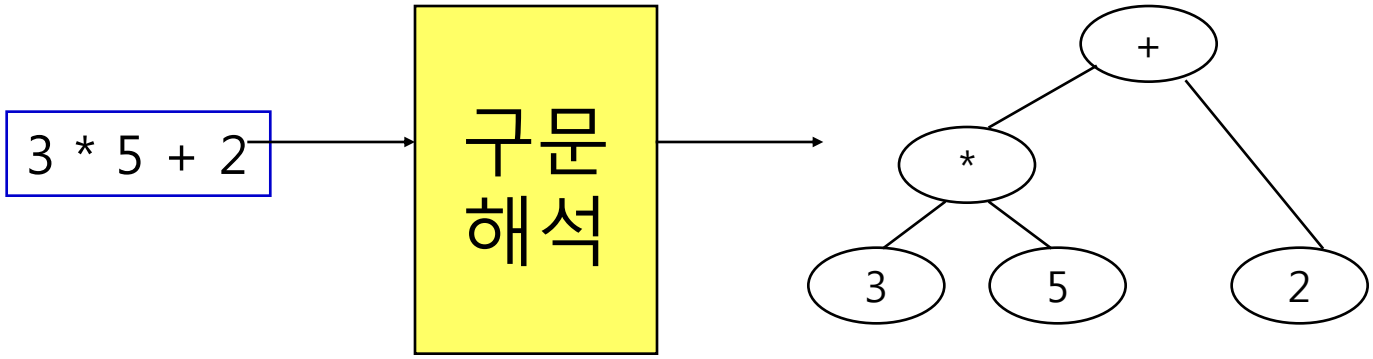
terminal expression과 non-terminal expression

- terminal expression
  - 문법 규칙에서, 더 이상 전개되지 않는 expression
  - 문법 규칙의 종착점을 의미함
  - 예: go, right, left, repeat 등

- non-terminal expression
  - 문법 규칙에서, 계속해서 다시 전개되는 expression
  - 예: <program> 또는 <command>

03. 예제 프로그램

- 2절에 정의된 미니 언어를 해석하는 프로그램
  - 문자열로 이루어진 미니 프로그램을 분해해서, 각 부분이 어떤 구조로 되어 있는지 해석한다.
  - 이를, 구문 해석(syntax analysis)이라고 한다.
  - 결과로 구문 트리(syntax tree)가 만들어 진다.
  - 예: 3 * 5 + 2의 구문 해석



중요 용어

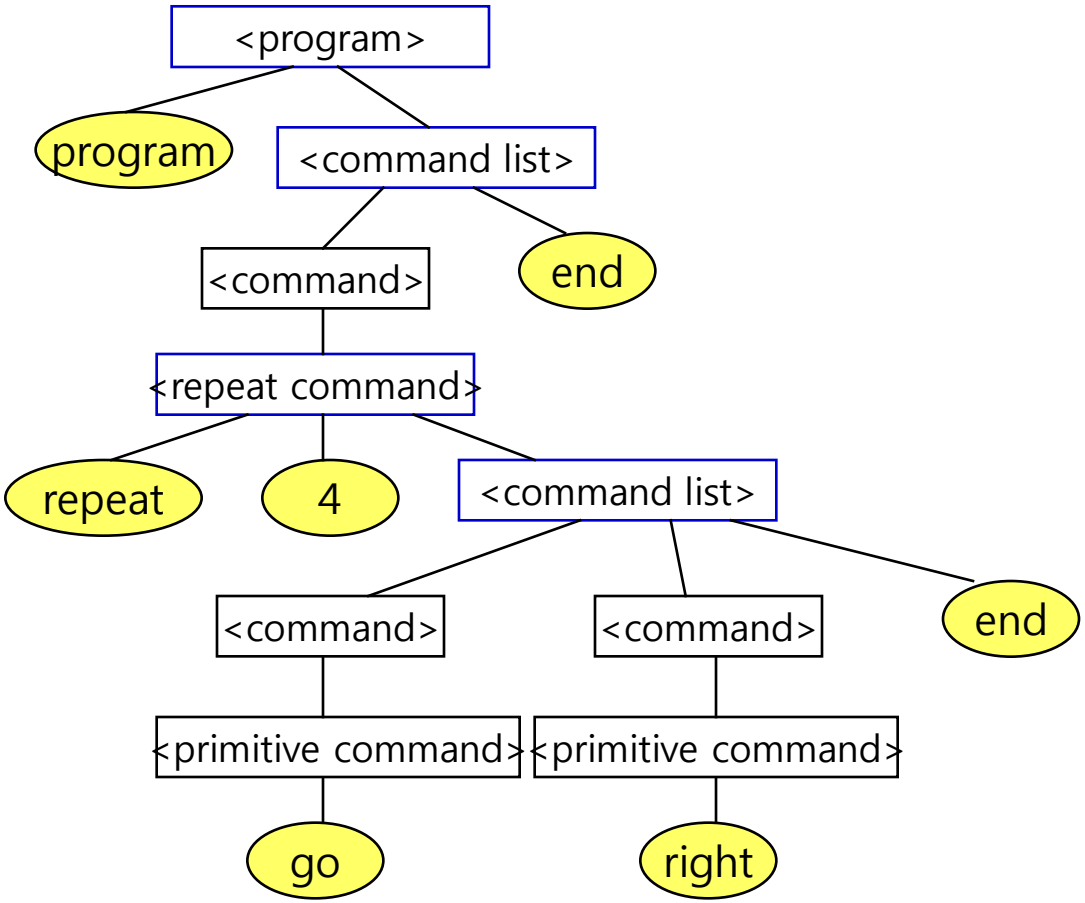
- 구문 해석 시 처리의 기본 단위를 '토큰(token)'이라고 한다.
  - 예를 들면, 3 + 5 는 세 개의 토큰으로 나누어진다.

- 입력 문자열을 토큰으로 분리시키는 작업을 '어구 해석(lexical analysis)'라고 한다.

- 토큰들부터 구문 트리를 만드는 작업을 '구문 해석(parse 또는 syntax analysis)'라고 한다.

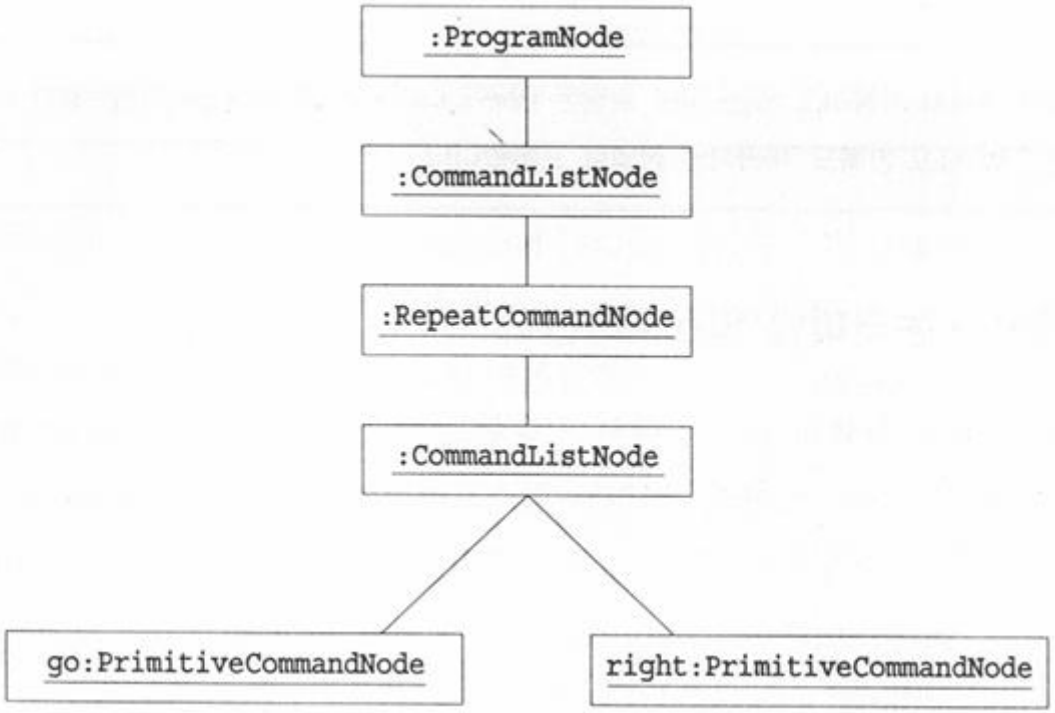
03. 예제 프로그램

- 2절에서 정의된 미니 언어를 해석하는 프로그램
  - 예: "program repeat 4 go right end end" 라는 미니 프로그램이 주어지면, 구문 해석을 통해 다음과 같은 구문 트리(syntax tree)가 만들어 진다.



03. 예제 프로그램

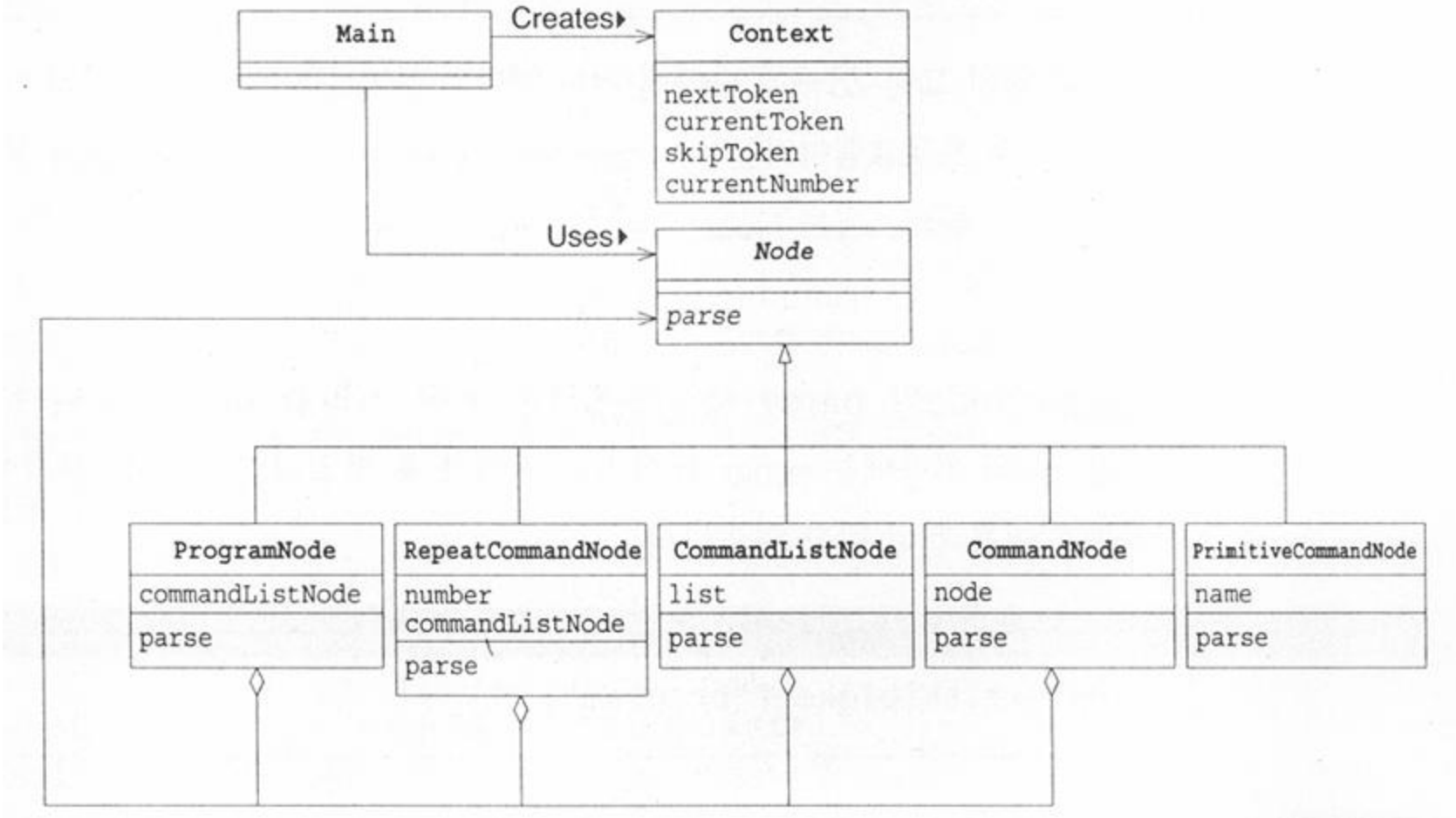
- 2절에서 정의된 미니 언어를 해석하는 프로그램
  - 예제 프로그램은, "program repeat 4 go right end end" 를 해석하여 다음과 같은 구조를 메모리 상에 만든다.



03. 예제 프로그램

이름	해설
Node	구문 트리의 '노드'가 되는 클래스
ProgramNode	〈program〉에 대응하는 클래스
CommandListNode	〈command list〉에 대응하는 클래스
CommandNode	〈command〉에 대응하는 클래스
RepeatCommandNode	〈repeat command〉에 대응하는 클래스
PrimitiveCommandNode	〈primitive command〉에 대응하는 클래스
Context	구문해석을 위한 전후 관계를 나타내는 클래스
ParseException	구문해석 중의 예외 클래스
Main	동작 테스트용 클래스

03. 예제 프로그램


Main
Creates
Context
nextToken
currentToken
skipToken
currentNumber
Uses
Node
parse
ProgramNode
commandListNode
parse
RepeatCommandNode
number
commandListNode
parse
CommandListNode
list
parse
CommandNode
node
parse
PrimitiveCommandNode
name
parse

03. 예제 프로그램

프로그램 작동 방식

- 예: program go right go left end 인 경우...
  - Context 객체가 구문 분석할 문자열(미니 프로그램)을 가지고 있다.
  - 먼저, ProgramNode의 parse( )는,
    - "program"이라는 단어를 확인한 후에,
    - CommandListNode에게 나머지 부분을 parse 하도록 한다.
  - CommandListNode의 parse( )는,
    - "end"라는 단어가 나올 때까지 CommandNode에게 parse( )를 시킨다.
  - CommandNode의 parse( )는,
    - 현재 토큰이 "repeat"라는 단어이면, RepeatCommandNode에게 parse( )를 부탁하고,
    - 그렇지 않으면, PrimitiveCommandNode에게 parse( )를 부탁한다.
  - PrimitiveCommandNode의 parse( )는,
    - "go"나 "right"나 "left"가 아니면, 예외를 발생시킨다.

03. 예제 프로그램

- Context 클래스
  - 구문 해석을 위해 필요한 메소드를 제공한다.
    - 미니 프로그램을 유지하면서, 필요할 때마다 토큰으로 잘라서 (lexical analysis) 그 토큰을 반환하는 일을 함

이름	해설
nextToken	다음의 토큰을 얻습니다(다음의 토큰으로 나아갑니다).
currentToken	현재의 토큰을 얻습니다(다음의 토큰으로는 나아가지 않습니다).
skipToken	현재의 토큰을 체크하고 나서 다음의 토큰을 얻습니다(다음의 토큰으로 나아갑니다).
currentNumber	현재의 토큰을 수치로서 얻습니다(다음의 토큰으로 나아가지 않습니다).

03. 예제 프로그램

- Context 클래스(계속)
  - java.util.StringTokenizer 를 이용한다.
    - 주어진 문자열을 토큰으로 분할해 주는 클래스
    - 구분 문자(delimiter): 스페이스(' '), 탭('₩t'), 뉴라인('₩n'), 캐리지 리턴('₩r'), 폼피드('₩f')

이름	해설
nextToken	다음의 토큰을 얻습니다(다음의 토큰으로 나아갑니다).
hasMoreTokens	다음의 토큰이 있는지 없는지 조사합니다.

03. 예제 프로그램

- Context 클래스(계속)
  - skipToken(String token)
    - 현재 토큰과 입력인자 token을 비교해서,
      - 같으면 다음 토큰으로 진행하고
      - 다르면, 예외를 발생시킨다.
  - currentNumber( )
    - 현재 토큰 문자열을 정수로 바꾸어주는 메소드

03. 예제 프로그램

- Node 클래스
  - 구문 트리의 각 부분(노드)를 구성하는 대표하는 최상위 클래스
  - 추상 메소드 parse(Context)
    - '구문 해석'이라는 처리를 실행하기 위한 메소드
    - 입력 인자 Context는, 현재 구문 해석을 실행하고 있는 '상황'을 나타내는 클래스
      - 입력 문장 중에서 현재 어느 토큰까지 해석이 진행되었는지를 알고 있다.
    - 구문 해석 중에 에러가 발생하면, ParseException을 던진다.

03. 예제 프로그램

- ProgramNode 클래스
  - <program> ::= program <command list>

    에서 <program>을 나타냄
  - Node commandListNode 필드
    - 자신의 뒤에 이어지는 <command list>에 대응하는 구조(노드)를 보관함

03. 예제 프로그램

- ProgramNode 클래스(계속)
  - parse( )
    - Context 객체에게 'program' 이라는 토큰이 나오는지 확인한다.
      - 확인 후 현재 토큰의 위치를 다음 토큰으로 이동시킨다.
    - 다음으로, <command list>에 대응하는 CommandListNode의 인스턴스를 생성하고 그 인스턴스의 parse( )를 호출한다.
  - toString()
    - 이 노드를 표현하는 문자열을 표시하는 메소드

03. 예제 프로그램

- CommandListNode 클래스
  - <command list> ::= <command>* end
    에서 <command list>을 나타냄
  - Vector list 필드
    - 0번 이상 반복출현하는 <command>를 보관함
    - <command>에 대응하는 CommandNode 클래스의 인스턴스를 넣어둠
  - parse( )
    - 남은 토큰이 없다면(null) => 'end'가 빠졌다는 예외를 발생시킴.
    - 현재 토큰이 'end'라면 => 'end' 뒤로 이동한 후 while 루프를 break함.
    - 현재 토큰이 'end'가 아니라면 => 이 때는 <command>를 의미하므로, CommandNode의 인스턴스를 만들어서 그것의 parse( ) 호출한 후, list에 추가한다.

03. 예제 프로그램

- CommandNode 클래스
  - <command> ::= <repeat command> | <primitive command>
    에서 <command>를 나타냄
  - node 필드
    - RepeatCommandNode 또는 PrimitiveCommandNode의 인스턴스를 넣어두는 변수
  - parse( )
    - 현재 토큰이 "repeat" 이면, RepeatCommandNode 인스턴스를 생성한 후 그것의 parse( )를 호출한다.
    - 그렇지 않으면, PrimitiveCommandNode 인스턴스를 생성한 후 그것의 parse( )를 호출한다.

03. 예제 프로그램

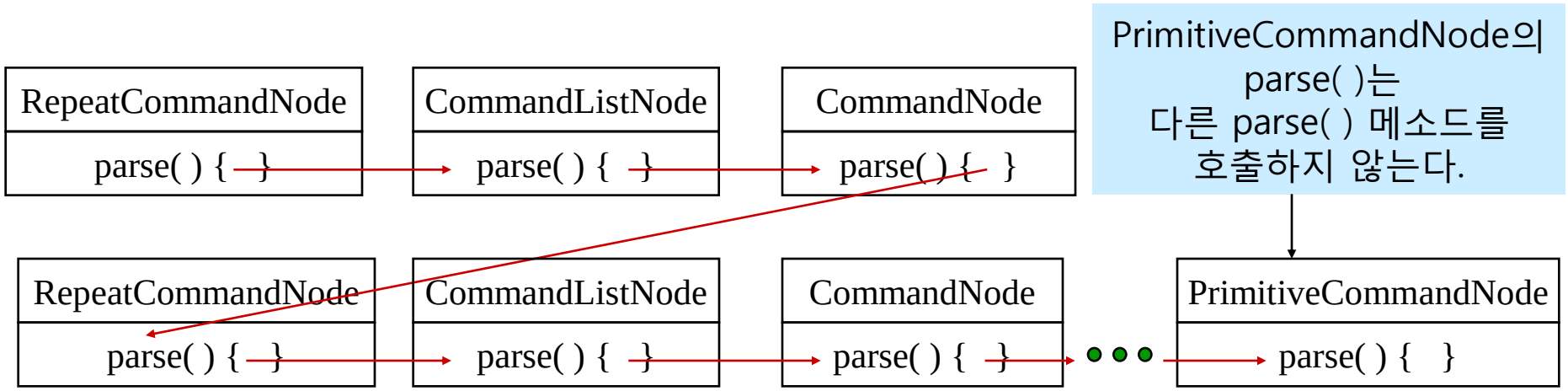
- RepeatCommandNode 클래스
  - <repeat command> ::= repeat <number> <command list>
    에서 <repeat command>를 나타냄
  - number 필드: <number> 부분이 저장됨
  - commandListNode 필드: <command list> 부분이 저장됨

03. 예제 프로그램

- RepeatCommandNode 클래스 (계속)

  <repeat command> ::= repeat <number> <command list>

  - parse( )
    - 재귀성을 가진다.
      - <repeat command>가 또 다른 <repeat command>를 가지는 경우
      - 예: program repeat 4 repeat 3 go right go left end right end end



03. 예제 프로그램

- PrimitiveCommandNode 클래스
  - <primitive command> ::= go | right | left
    에서 < primitive command >를 나타냄
  - parse( )
    - 다른 parse( ) 메소드를 호출하지 않는다.
    - 현재 토큰을 얻은 후, 다음 토큰으로 이동시킨다.
    - 얻은 토큰이 "go", "right", "left" 중에 하나가 아니면, 예외를 발생시킨다.

03. 예제 프로그램

- ParseException 클래스
  - 구문 해석 중에 발생한 예외를 위한 클래스
  - Exception을 클래스를 상속했으므로
    - 소스 코드 컴파일 시에 예외 처리를 해 주었는지 자바 컴파일러가 확인하고
    - 예외 처리를 안 했으면 컴파일 에러가 발생한다.
  - 비교: RuntimeException을 상속한 경우에는
    - 예외 처리를 해 주었는지 컴파일 시에 검사하지는 않는다.

03. 예제 프로그램

- Main 클래스
  - 미니언어의 인터프리터를 작동시키기 위한 클래스
  - program.txt 파일의 내용을 한 줄씩 읽어서 구문 해석을 시킨다.
  - 그리고 나서, 그 결과를 문자열로 화면에 출력한다.

03. 예제 프로그램

```
text = "program end"     ←미니 프로그램의 내용
node = [program [] ]     ←구문해석의 결과
text = "program go end"
node = [program [go] ]
text = "program go right go right go right go right end"
node = [program [go, right, go, right, go, right, go, right] ]
text = "program repeat 4 go right end end"
node = [program [[repeat 4 [ go, right ]]]]
text = "program repeat 4 repeat 3 go right go left end right end end"
node = [program [[repeat 4 [ repeat 3 [go, right, go, left ]], right]]]
```

04. 등장 역할

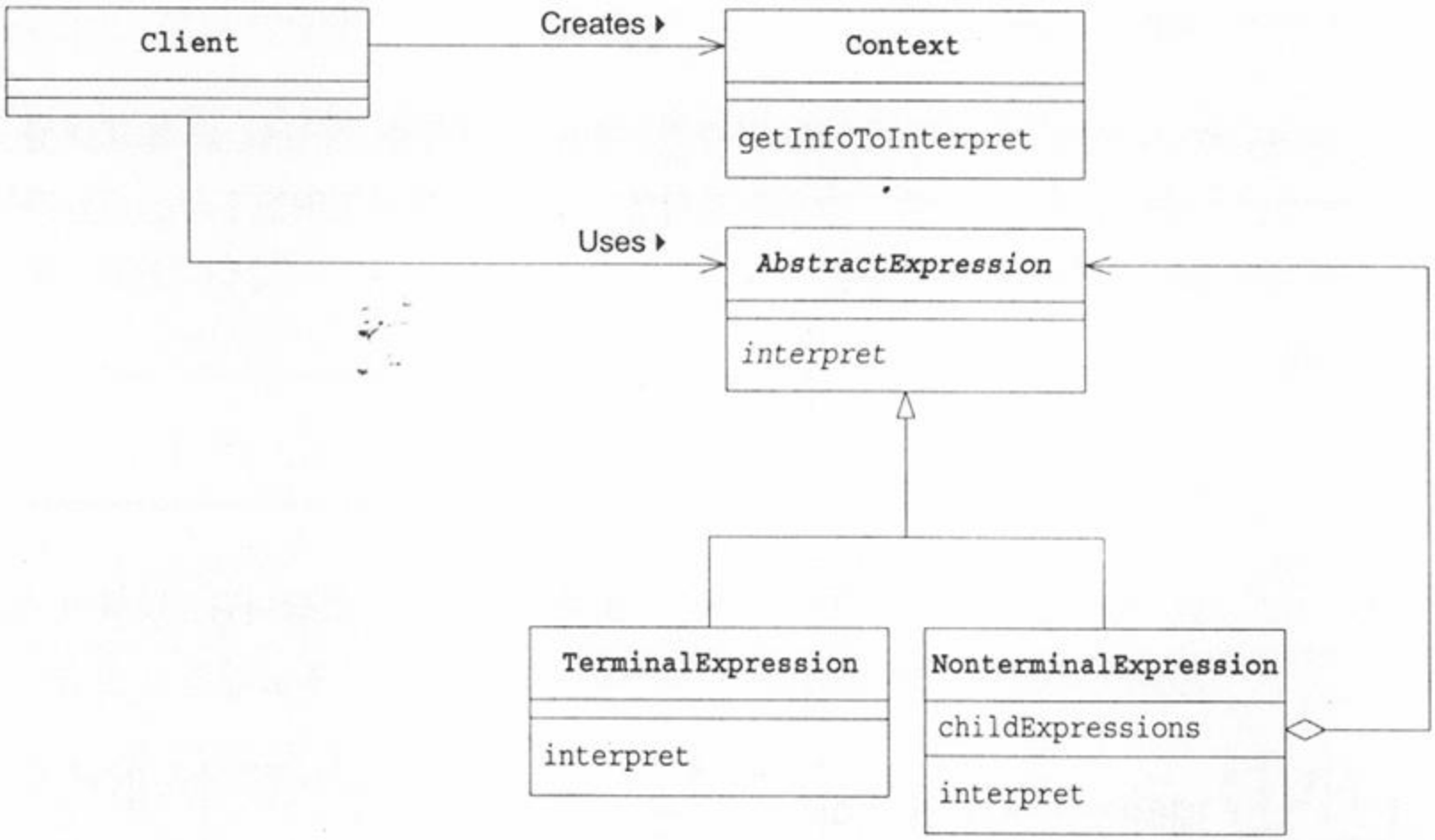
- AbstractExpression(추상적인 수식) 역할
  - 구문 트리의 노드가 가지는 공통적인 인터페이스(API)를 정하는 역할
  - 예제에서는 Node 클래스가 해당됨
- TerminalExpression(종착점 수식)의 역할
  - BNF의 터미널 익스프레션에 대응하는 역할
  - 예제에서는 PrimitiveCommandNode 클래스가 해당됨
- NonterminalExpression(논터미널 수식)의 역할
  - BNF의 논터미널 익스프레션에 대응하는 역할
  - ProgramNode, CommandNode, RepeatCommandNode, CommandListNode 가 해당됨

04. 등장 역할

- Context(문맥, 전후관계)의 역할
  - 인터프리터가 구문 해석을 수행하도록 정보를 제공하는 역할
  - 예제에서는 Context 클래스가 해당됨

- Client(의뢰자)의 역할
  - 구문트리를 조리하기 위해, TerminalExpression이나 NonterminalExpression을 호출하는 역할
  - 예제에서는 Main 클래스가 해당됨

04. 등장 역할


Client
Creates
Context
getInfoToInterpret
Uses
AbstractExpression
interpret
TerminalExpression
interpret
NonterminalExpression
childExpressions
interpret

05. 사고를 넓혀주는 힌트

- 그 밖의 미니 언어
  - 정규 표현/정규식(regular expression)
    - describes a pattern or sequence of characters
    - 문자들의 패턴이나 조합을 기술하는 표현식
    - 예: raining & (dogs | cats) *
      - raining 뒤에 dogs나 cats가 0번 이상 반복되는 패턴을 기술한 식
      - 이 정규식에 맞는 문장
        - raining
        - raining dogs dogs cats
        - raining cats dogs
        - raining cats dogs cats dogs
  - 검색용 수식
    - 검색 시, 단어의 조합을 표현하기 위한 언어에 사용 가능하다.
    - 예: garlic and not onions
      - garlic은 포함하지만, onions는 포함하지 않는 검색식을 나타냄.

06. 관련 패턴

- Composite 패턴
- Flyweight 패턴
- Visitor 패턴

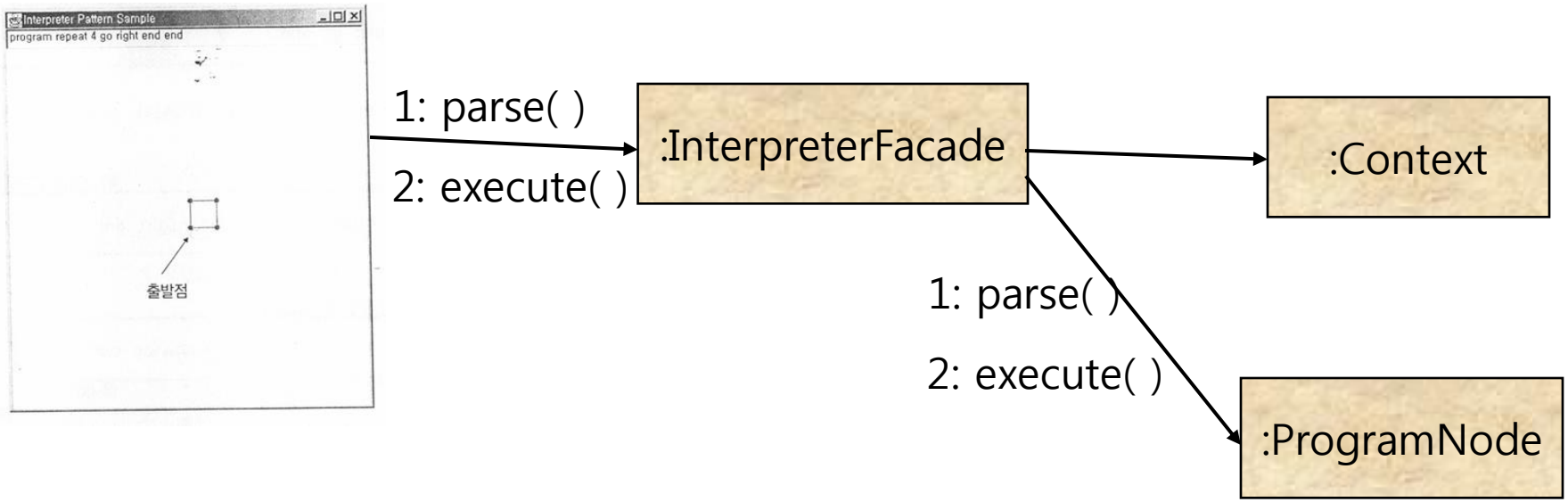
07. 관련 패턴

- 미니 언어를 사용해서 문제를 해결하는 Interpreter 패턴

연습문제

- 23-1
  - 구문 해석 후, 기본 커맨드인 go, right, left 에 해당하는 일을 하는 실제로 수행하는 프로그램을 작성하기
  - 특징
    - GUI를 사용해서 기본 커맨드의 결과를 화면에 직접 그린다.
    - Facade 패턴을 이용해서, 인터프리터를 이용하기 쉽게 함
    - 기본 커맨드를 처리하는 클래스를 생성하는 공장 클래스인 ExecutorFactory를 작성했다. (4장의 Factory 패턴 이용)
      - Context 객체가 이 공장을 가지고 있다가 여러 가지 Executor 객체들을 생성한다.
    - 인터프리터 부분을 다른 패키지에 넣음으로써, GUI 부분과 분리했다.

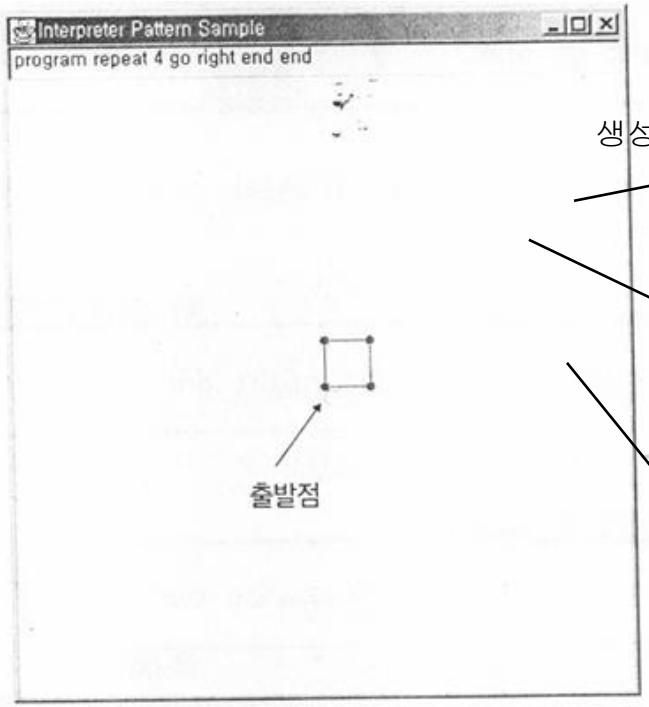
TurtleCanvas 객체

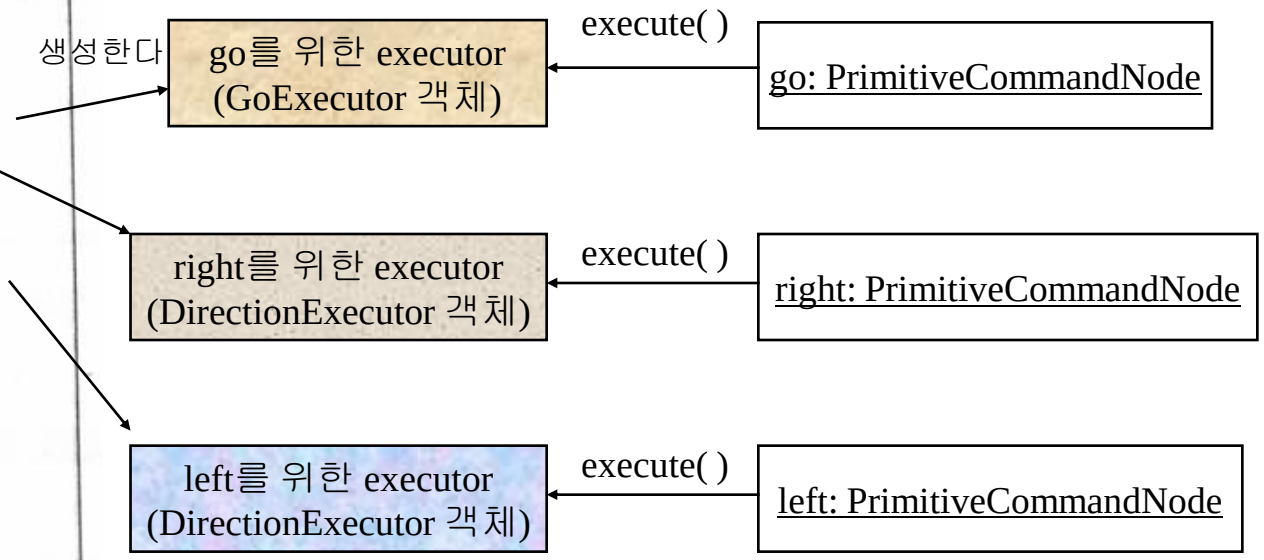


동작 방식

TurtleCanvas 객체
(ExecutorFactory 역할을 한다)

(1) 입력 텍스트의 파싱이 완료되면,
각 PrimitiveCommandNode 객체와
go/right/left 를 위한 executor 객체가 생성된다.





(2) 그 다음 Main의 paint( )에서 façade의 execute 실행 시 각 PrimitiveCommandNode 객체는, 해당 executor의 execute( ) 메소드를 호출한다.

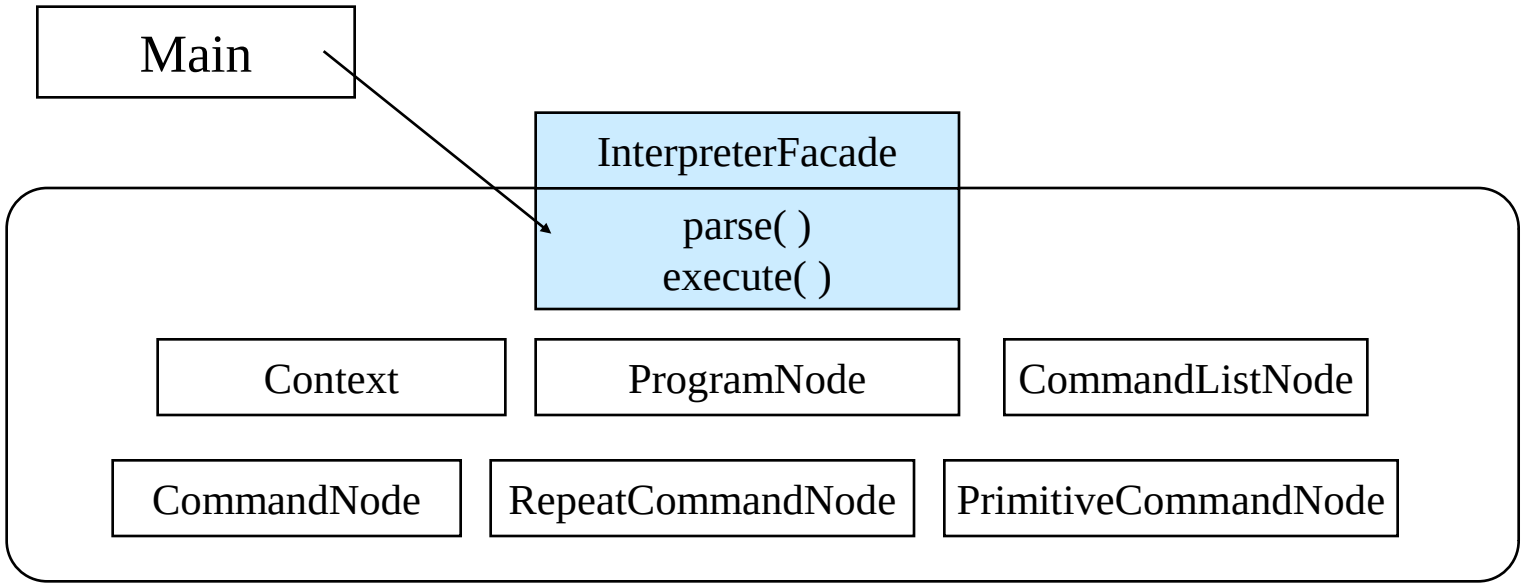
(3) 해당 executor의 execute( ) 메소드 안에서, 직선 그리기/우향우/좌향좌를 실행한다.

연습문제

- 23-1(계속)
  - 동작방식
    - 사용자가 GUI의 textfield에 미니 프로그램을 입력한 후 enter를 치면 => Main의 actionPerforamed( )가 실행된다.
    - Main의 actionPerforamed( ) 에서는, parseAndExecute( )를 호출한다.
    - parseAndExecute( )는, 미니 프로그램을 파싱한 후, repaint( )를 호출하여, 자동으로 paint( )가 실행되도록 한다.
    - paint( )에서는, InterpreterFacade의 execute( )를 호출한다.
    - InterpreterFacade의 execute( )는, 파싱 시에 생성된 ProgramNode의 execute( )을 호출한다.
    - ProgramNode의 execute( )은, CommandListNode의 execute( )을 호출한다.
    - 결국, PrimitiveNode의 execute( )이 호출이 되어서, 해당 Executor의 execute( )를 호출해서, 캔버스에 실제 자동차의 움직임을 그린다.

연습문제

- 23-1 (계속)
  - InterpreterFacade
    - 기존의 인터프리터 관련 클래스들을 사용하기 좋게 하기 위한 창구 역할을 한다.
  - Main의 main( )는,
    - InterpreterFacade 객체만을 이용해서 인터프리터를 사용한다.

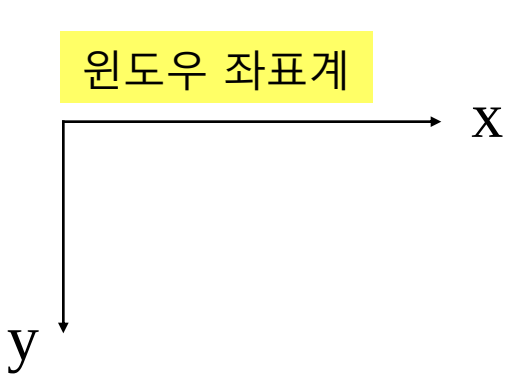


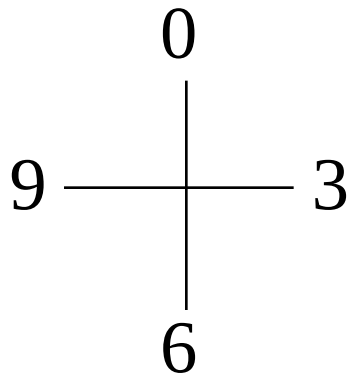
연습문제

- 23-1 (계속)
  - 구문 트리의 각 Node는,
    - execute( ) 메소드가 추가된 것만 다르고, 나머지는 예제 프로그램과 같다.
    - 단, PrimitiveCommandNode의 parse( ) 메소드에서, 파싱 시 해당 Executor를 생성하는 문장이 추가되었다.

연습문제

- 23-1(계속)
  - 해당 Executor는, ExecutorFactory를 구현한 TurtleCanvas가 생성한다.
    - createExecutor( ) 참조
      - "go"인 경우에는 GoExecutor를, "right"/"left" 인 경우에는 DirectionExecutor를 생성한다.
      - 이 메소드의 내용을 수정하면, 다른 일을 하는 Executor를 생성하도록 바꿀 수 있다. (Factory 패턴을 사용한 이유가 여기에 있다)
    - direction 변수: 0, 3, 6, 9 중 하나의 값을 가진다.





차가 위로 직진하려면, y 좌표를 줄인다.