# i219 Software Design Methodology
## 9. Dynamic modeling 2

Kazuhiro Ogata (JAIST)



# Outline of lecture

- Another simple calculator
- Use case, state diagram & class diagram of simple calculator
- Simple calculator in Java
- State diagram of virtual machine
- Virtual machine in Java
- Class diagram, object diagram & sequence diagram of parse tree
- Parse tree in Java
- Parsing expression

# Another simple calculator

Let us design another simple calculator:
✓ it uses a virtual machine that has a set of commands (instructions),
✓ it generates a list of such commands (instructions) from a parse tree, and
✓ it calculates an expression in string by first converting it into a list of tokens (*tl*), making a parse tree (*pt*) from *tl*, generating a list of commands (*cl*) from *pt*, and making the virtual machine execute *cl*.



# Use case of simple calculator (1)

**In Expression in String**

Main Success Scenario:
1. User inputs an exp in string, making *sc* (source code) available
2. System converts it into a token list *tl*
3. System makes a parse tree *pt* from *tl*

Extensions:
3a: There is a syntax error
.1: System lets user know it, and makes *pt* and *vm* (virtual machine) unavailable;

**Compile**

Main Success Scenario:
1. User inputs command compile
2. System generates a command (instruction) list *cl* from *pt*
3. System makes *vm* that has *cl*

Extensions:
2a: *pt* is unavailable
.1: System lets user know it

# Use case of simple calculator (2)

**Run**

Main Success Scenario:
1. User inputs command run
2. System executes *vm*
3. System displays the result

Extensions:
2a: *vm* is unavailable
.1: System lets user know it
2b: runtime error such as "division by zero" occurs
.1: System lets user know it

**Show SC (Source Code)**

Main Success Scenario:
1. User asks system to display *sc*
2. System displays *sc*

**Show PT (Parse Tree)**

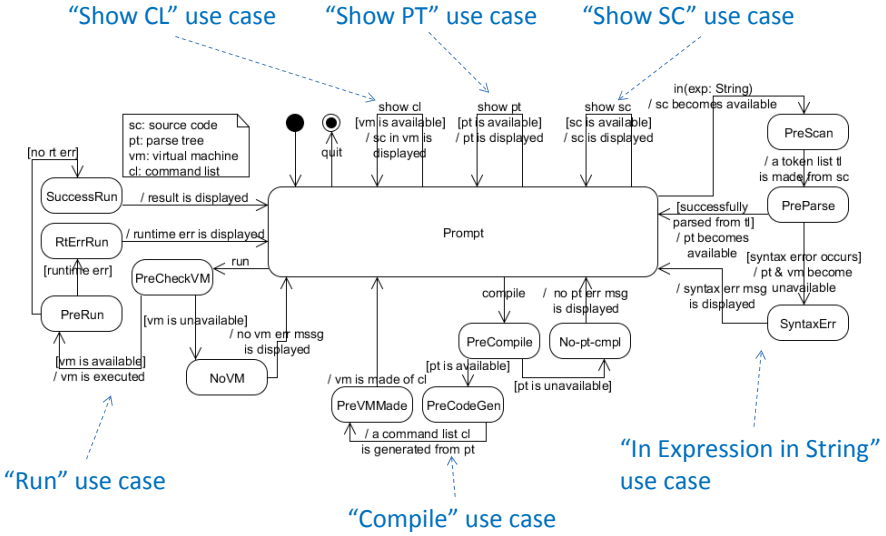
Main Success Scenario:
1. User asks system to display *pt*
2. System displays *pt*

**Show CL (Command List)**

Main Success Scenario:
1. User asks system to display *cl*
2. System displays *cl* in *vm*



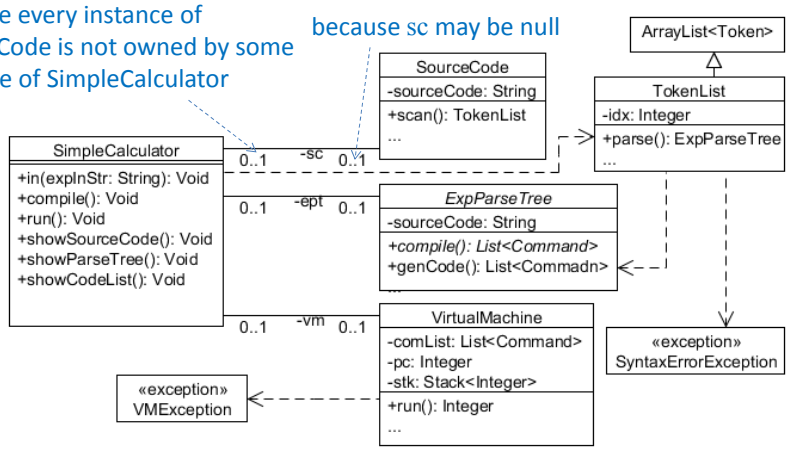
# State diagram of simple calculator



# Class diagram of simple calculator

Because every instance of SourceCode is not owned by some instance of SimpleCalculator

because sc may be null




three attributes sc, ept & vm are expressed as associations

note that ExpParseTree is an abstract class, and then an object to which ept refers is of a (concrete) class (such as AddParseTree) that extends ExpParseTree



# Simple calculator in Java (1)

```
public class SimpleCalculator {
  private SourceCode sc;
  private ExpParseTree ept;
  private VirtualMachine vm;
  public SimpleCalculator() { sc = null;  ept = null;  vm = null; }
  public void in(String expInStr) {...}
  public void compile() {...}
  public void run() {...}
  public void showSourceCode() {...}
  public void showParseTree() {...}
  public void showCodeList() {
    if (vm == null) { System.err.println("No compiled!"); }
    else { System.out.println(vm.getComList()); }  }
}
```

explained later

similar to the body of showCodeList

# Simple calculator in Java (2)

```
public void in(String expInStr) {
  sc = new SourceCode(expInStr);
  TokenList tl = sc.scan();
  try {
    ept = tl.parse();
    vm = null;
    System.out.println("Successfully loaded.");
  } catch (SyntaxErrorException e) {
    ept = null;
    vm = null;
    System.err.println(e.getMessage());
  }
}
```

a token list tl is made by lexically analyzing sc

a parse tree ept is made by parsing tl; a SyntaxErrorException may be thrown

a SyntexErrorException is caught; a syntax error message is displayed and the control goes back to the caller



# Simple calculator in Java (3)

```
public void compile() {
  if (ept == null) {
    System.err.println("No program loaded!");
  } else {
    List<Command> cl = ept.genCode();
    vm = new VirtualMachine(cl);
    System.out.println("Successfully compiled.");
  }
}
```

a list cl of commands (instructions) is generated from a parse tree ept

a virtual machine vm is made of the command list cl

# Simple calculator in Java (4)

```
public void run() {
  if (vm == null) {
    System.err.println("No compiled!");
  } else {
    try {
      Stack<Integer> stk = new Stack<Integer>();
      vm.reset(0,stk);
      int x = vm.run();
      System.out.println(x);
    } catch (VMException e) {
      System.err.println(e.getMessage());
    }
  }
}
```

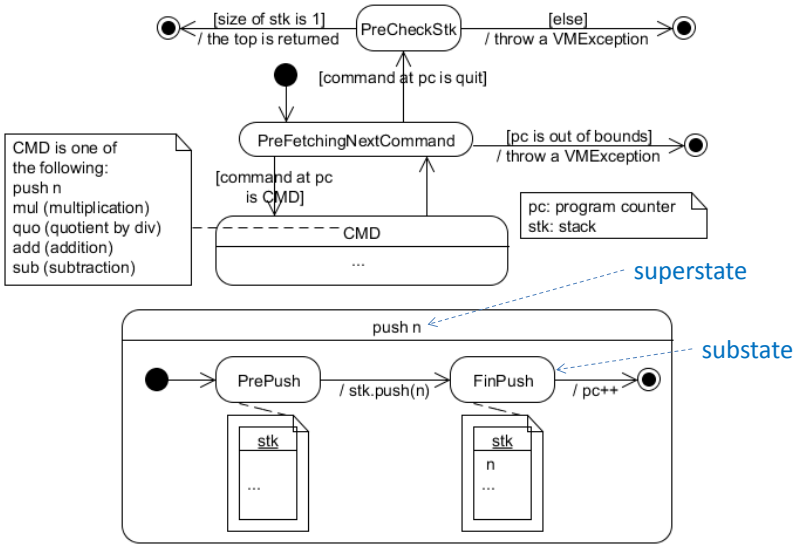
vm is set to the program counter 0 and the empty stack stk

vm is executed; a VMException may be thrown

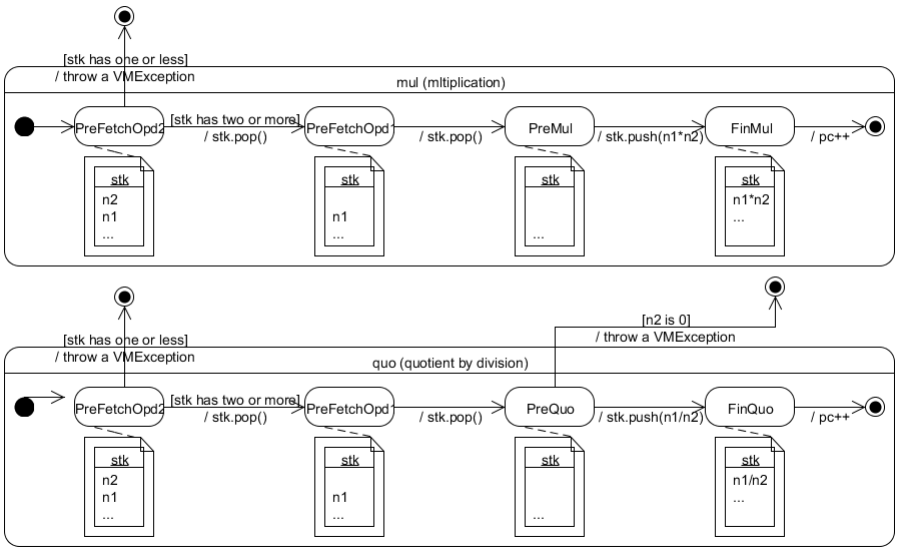
a VMException is caught



# State diagram of virtual machine (1)

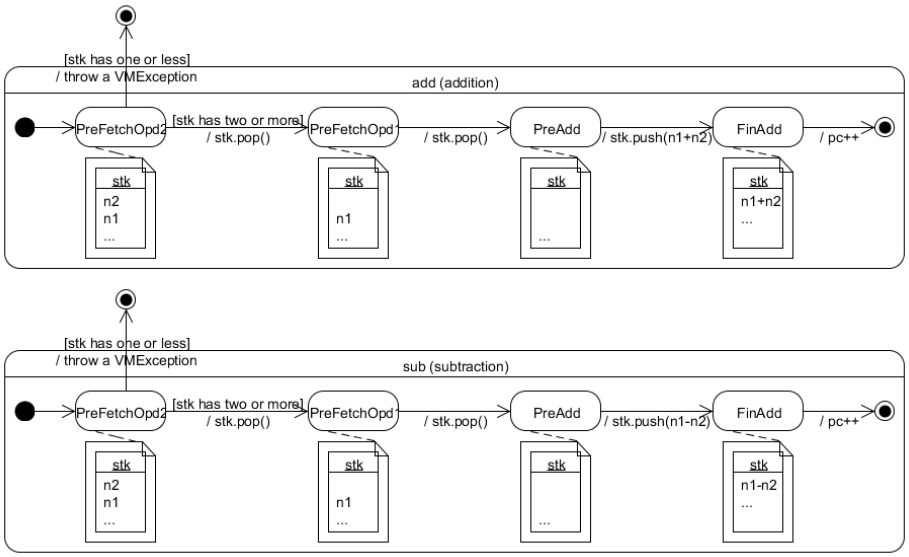


# State diagram of virtual machine (2)






# State diagram of virtual machine (3)



# Virtual machine in Java (1)

```
public class VirtualMachine {
  private List<Command> comList;  private int pc;  private Stack<Integer> stk;
  public VirtualMachine(List<Command> cl) { comList = cl; }
  public void reset(int pc, Stack<Integer> stk) { this.pc = pc; this.stk = stk; }
  public List<Command> getComList() { return comList; }
  public int run() throws VMException { …
    while (true) {
      if (pc < 0 || pc >= comList.size()) {
        throw new VMException(pc,comList.size()); }
      com = comList.get(pc);
      switch (com.getName()) {
      …
      default:
        throw new IllegalStateException("pc1: " + pc + "cl1: " + comList); }
    } } }
```

If pc is out of bounds, a VMException is thrown

what will be done depends on the command (instruction) in comList at pc

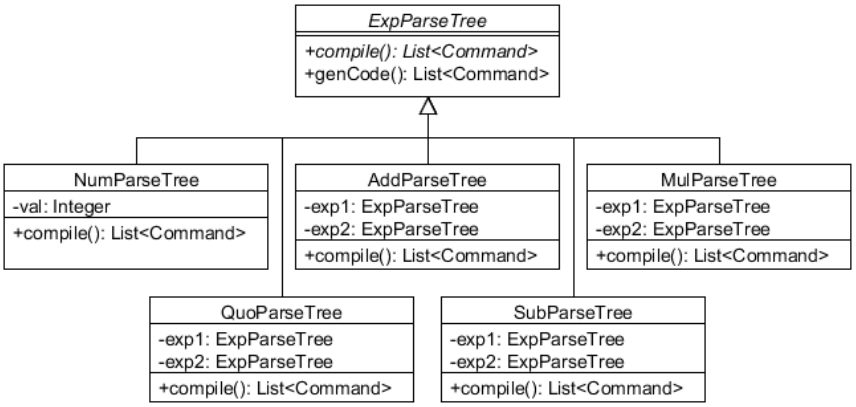
this can be used at runtime to check if all cases are covered



# Virtual machine in Java (2)

```
case PUSH: stk.push(com.getNum()); pc++; break;

case MUL: if (stk.size() < 2) { throw new VMException(stk); }
   x2 = stk.top(); stk.pop(); x1 = stk.top(); stk.pop();
   x = x1 * x2; stk.push(x); pc++; break;

case QUO: if (stk.size() < 2) { throw new VMException(stk); }
  x2 = stk.top(); stk.pop(); x1 = stk.top(); stk.pop();
  if (x2 == 0) { throw new VMException(); }
  x = x1 / x2; stk.push(x); pc++; break;

…

case QUIT: if (stk.size() != 1) { throw new VMException(stk,stk.size()); }
  x = stk.top(); stk.pop(); return x;
```

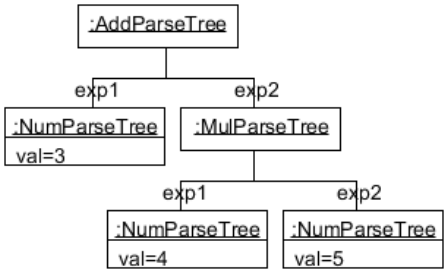
# Class diagram of parse tree





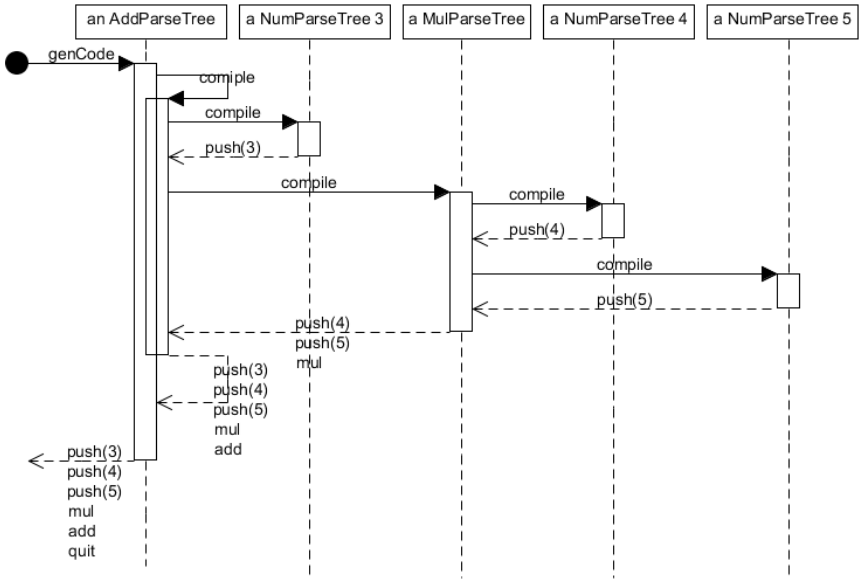

# Object diagram of parse tree

1. "3+4*5" is converted into the token list [NUM{num=3}, PLUS, NUM{num=4}, MUL, NUM{num=5}].
2. The parse tree represented by the following object diagram is made by sending parse() to the token list:




3. A list of commands (instructions) is generated by sending genCode() to the parse tree.

# Sequence diagram of parse tree






# Parse tree in Java (1)

```
public abstract class ExpParseTree {
   public abstract List<Command> compile();
   public List<Command> genCode() {
      List<Command> cl;
      cl = this.compile();
      cl.add(new Command(CommandName.QUIT));
      return cl; } }

public class NumParseTree extends ExpParseTree {
  private int val;
  public NumParseTree(int x) { val = x; }
  public List<Command> compile() {
    List<Command> cl = new ArrayList<Command>();
    cl.add(new Command(CommandName.PUSH,val));
    return cl; } }
```

abstract method

a list of commands (instructions) is generated by sending compile() to this

command (instruction) QUIT is put into the list of commands (instructions) at the end

PUSH{num=val} is generated

# Parse tree in Java (2)

```
public class AddParseTree extends ExpParseTree {
  private ExpParseTree ept1, ept2;
  public AddParseTree(ExpParseTree e1, ExpParseTree e2) {
    ept1 = e1; ept2 = e2; }
  public List<Command> compile() {
    List<Command> cl;
    cl = ept1.compile();
    cl.addAll(ept2.compile());
    cl.add(new Command(CommandName.ADD));
    return cl;
  }
}
```

a command list like the following is generated:

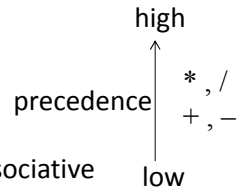



# Parsing expression (1)

The grammar for expressions is as follows:

$E ::= num \mid ( E ) \mid E * E \mid E / E \mid E + E \mid E - E$

$num ::= [0\text{-}9]+$




every operator is left associative

It is not straightforward, however, to make a parser that directly deals with this grammar, and then the grammar is modified as follows:

$F ::= num \mid ( E_2 )$
$E_1 ::= F\ E_{11}$
$E_{11} ::= \varepsilon \mid * F\ E_{11} \mid / F\ E_{11}$
$E_2 ::= E_1\ E_{22}$
$E_{22} ::= \varepsilon \mid + E_1\ E_{22} \mid - E_1\ E_{22}$
$E ::= E_2$

This grammar (LL(1)) can be dealt with by a predictive parser (a recursive descent parser that needs no backtracking) that can be straightforwardly implemented. A parser is implemented in class TokenList.

# Parsing expression (2)

```
public class TokenList extends ArrayList<Token> { private int idx;
  public TokenList() { idx = 0; } … }

public ExpParseTree parse() throws SyntaxErrorException { return E(); }
```

$F ::= num \mid ( E_2 )$

```
private ExpParseTree F() throws SyntaxErrorException { …
  if (idx == size()) { throw new SyntaxErrorException(…); }
  switch (t1.getName()) {
  case NUM: return new NumParseTree(t1.getNum());
  case LPAR: e = E2();
    if (idx == size()) { throw new SyntaxErrorException(…); }
    t2 = get(idx++);
    if (t2.getName() != TokenName.RPAR) { … /* syntax error */ }
    return e;
    default:  … /* syntax error */ } }
```



# Parsing expression (3)

$E_1 ::= F\ E_{11}$

```
private ExpParseTree E1() throws SyntaxErrorException {
  ExpParseTree e = F(); return E11(e);  }
```

$E_{11} ::= \varepsilon \mid * F\ E_{11} \mid / F\ E_{11}$

```
private ExpParseTree E11(ExpParseTree e) throws SyntaxErrorException {
  … if (idx == size()) { return e; }
  t1 = get(idx++);
  switch (t1.getName()) {
  case MUL: e1 = F(); e2 = new MulParseTree(e,e1); return E11(e2);
  case QUO: e1 = F(); e2 = new QuoParseTree(e,e1); return E11(e2);
  default: idx--; return e; }  }
```

# Parsing expression (4)

$E_2 ::= E_1\ E_{22}$

```
private ExpParseTree E2() throws SyntaxErrorException {
  ExpParseTree e = E1(); return E22(e);  }
```

$E_{22} ::= \varepsilon \mid + E_1\ E_{22} \mid - E_1\ E_{22}$

```
private ExpParseTree E22(ExpParseTree e) throws SyntaxErrorException {
  … if (idx == size()) { return e; }
  t1 = get(idx++);
  switch (t1.getName()) {
  case PLUS: e1 = E1(); e2 = new AddParseTree(e,e1); return E22(e2);
  case MINUS: e1 = E1(); e2 = new SubParseTree(e,e1); return E22(e2);
  default: idx--; return e; }  }
```



# Parsing expression (5)

$E ::= E_2$

```
private ExpParseTree E() throws SyntaxErrorException {
  ExpParseTree ept = E2();
  if (idx != size()) {
    throw new SyntaxErrorException("One exp in one line!"
                                   + "; something following one exp!");
  }
  return ept;
}
```

if two or more expressions are written in one line such as “3+4 5*5”, then a SyntaxErrorException is thrown

# Summary

- Another simple calculator
- Use case, state diagram & class diagram of simple calculator
- Simple calculator in Java
- State diagram of virtual machine
- Virtual machine in Java
- Class diagram, object diagram & sequence diagram of parse tree
- Parse tree in Java
- Parsing expression