CMPE 152: Compiler Design

August 31 Class Meeting

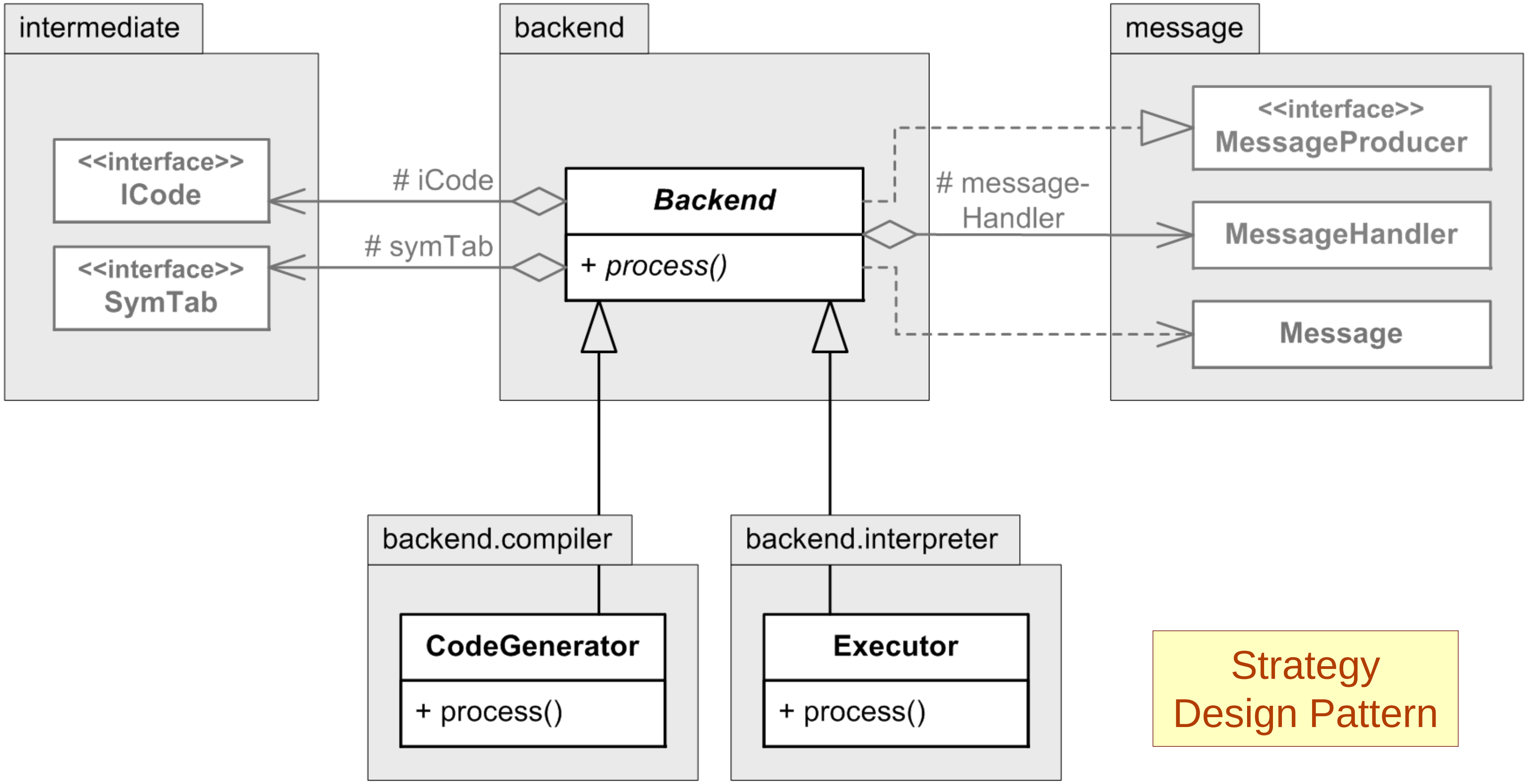
Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

San José State
UNIVERSITY

Initial Back End Subclasses

- The **`CodeGenerator`** and **`Executor`** subclasses will only be (do-nothing) stubs for now.



Strategy Design Pattern

The Code Generator Class

```
#include "../Backend.h"

namespace wci { namespace backend { namespace compiler {

using namespace std;
using namespace wci::backend;

class CodeGenerator : public Backend
{
public:
    /**
     * Process the intermediate code and the symbol table
     * created by the parser to generate object code.
     * Implementation of wci::backend::Backend.
     * @param icode the intermediate code.
     * @param symtab the symbol table.
     * @throw a string message if an error occurred.
     */
    void process(ICode *icode, SymTab *symtab) throw (string);
};

}}} // namespace wci::backend::compiler
```

The Code Generator Class, *cont'd*

- All the **`process()`** method does for now is send the **`COMPILER_SUMMARY`** message.
  - number of instructions generated (none for now)
  - code generation time (nearly no time at all for now)

```
#include <chrono>
#include "CodeGenerator.h"
#include "../Backend.h"
#include "../../message/Message.h"

namespace wci { namespace backend { namespace compiler {

using namespace std;
using namespace std::chrono;
using namespace wci::backend;
using namespace wci::message;
```

The Code Generator Class, *cont'd*

```
void CodeGenerator::process(ICode *icode, SymTab *symtab) throw (string)
{
    steady_clock::time_point start_time = steady_clock::now();
    int instruction_count = 0;

    // Send the compiler summary message.
    steady_clock::time_point end_time = steady_clock::now();
    double elapsed_time =
            duration_cast<duration<double>>(end_time - start_time).count();
    Message message(COMPILER_SUMMARY,
                    INSTRUCTION_COUNT, to_string(instruction_count),
                    ELAPSED_TIME, to_string(elapsed_time));
    send_message(message);
}

}}} // namespace wci::backend::compiler
```

The Executor Class

```
#include "../Backend.h"

namespace wci { namespace backend { namespace interpreter {

using namespace std;
using namespace wci::backend;

class Executor : public Backend
{
public:
    /**
     * Process the intermediate code and the symbol table
     * created by theparser to execute the program.
     * Implementation of wci::backend::Backend.
     * @param icode the intermediate code.
     * @param symtab the symbol table.
     * @throw a string message if an error occurred.
     */
    void process(ICode *icode, SymTab *symtab) throw (string);
};

}}} // namespace wci::backend::interpreter
```

The Executor Class, *cont'd*

- All the **process()** method does for now is send the **INTERPRETER_SUMMARY** message.
  - number of statements executed (none for now)
  - number of runtime errors (none for now)
  - execution time (nearly no time at all for now)

```
#include <chrono>
#include "Executor.h"
#include "../Backend.h"
#include "../../message/Message.h"

namespace wci { namespace backend { namespace interpreter {

using namespace std;
using namespace std::chrono;
using namespace wci::backend;
using namespace wci::message;
```

The Executor Class, *cont’d*

```
void Executor::process(ICode *icode, SymTab *symtab) throw (string)
{
    steady_clock::time_point start_time = steady_clock::now();
    int execution_count = 0;
    int runtime_errors = 0;

    // Send the interpreter summary message.
    steady_clock::time_point end_time = steady_clock::now();
    double elapsed_time =
            duration_cast<duration<double>>(end_time - start_time).count();
    Message message(INTERPRETER_SUMMARY,
                    EXECUTION_COUNT, to_string(execution_count),
                    ERROR_COUNT, to_string(runtime_errors),
                    ELAPSED_TIME, to_string(elapsed_time));
    send_message(message);
}

}}} // namespace wci::backend::interpreter
```

A Back End Factory Class

```
Backend *BackendFactory::create_backend(string operation) throw (string)
{
    if (operation == "compile")
    {
        return new CodeGenerator();
    }
    else if (operation == "execute")
    {
        return new Executor();
    }
    else
    {
        throw new string("Backend factory: Invalid operation '" +
                         operation + "'");
    }
}
```

End-to-End: Program Listings

- Here's the heart of the main **Pascal** class's constructor:

```
source = new Source(input);
source->add_message_listener(this);

parser = FrontendFactory::create_parser("Pascal", "top-down", source);
parser->add_message_listener(this);
parser->parse();

source->close();

symtab = parser->get_symtab();
icode = parser->get_icode();

backend = BackendFactory::create_backend(operation);
backend->add_message_listener(this);
backend->process(icode, symtab);
```

The front end parser creates the intermediate code and the symbol table of the intermediate tier.

The back end processes the intermediate code and the symbol table .

Listening to Messages

- Class `Pascal` implements the `MessageListener` interface.

Demo

Listening to Messages, *cont'd*

```
const string Pascal::SOURCE_LINE_FORMAT = "%03d %s\n";

void Pascal::message_received(Message& message)
{
    MessageType type = message.get_type();

    switch (type)
    {
        case SOURCE_LINE:
        {
            string line_number = message[LINE_NUMBER];
            string line_text = message[LINE_TEXT];

            printf(SOURCE_LINE_FORMAT.c_str(),
                   stoi(line_number), line_text.c_str());
            break;
        }

        case PARSER_SUMMARY: ...
        case INTERPRETER_SUMMARY: ...
        case COMPILER_SUMMARY: ...

        default: break;
    }
}
```

Demo

Is it Really Worth All this Trouble?

- Major software engineering challenges:
  - Managing **change**.
  - Managing **complexity**.

- To help manage change,
  use the open-closed principle.
  - Close the code for modification.
    Open the code for extension.
  - **Closed:** The language-independent framework classes.
  - **Open:** The language-specific subclasses.

Is it Really Worth All this Trouble? *cont'd*

- Techniques to help manage complexity:
  - Partitioning
  - Loose coupling
  - Incremental development
    - Always build upon working code.
- Good object-oriented design with design patterns.

Source Files from the Book

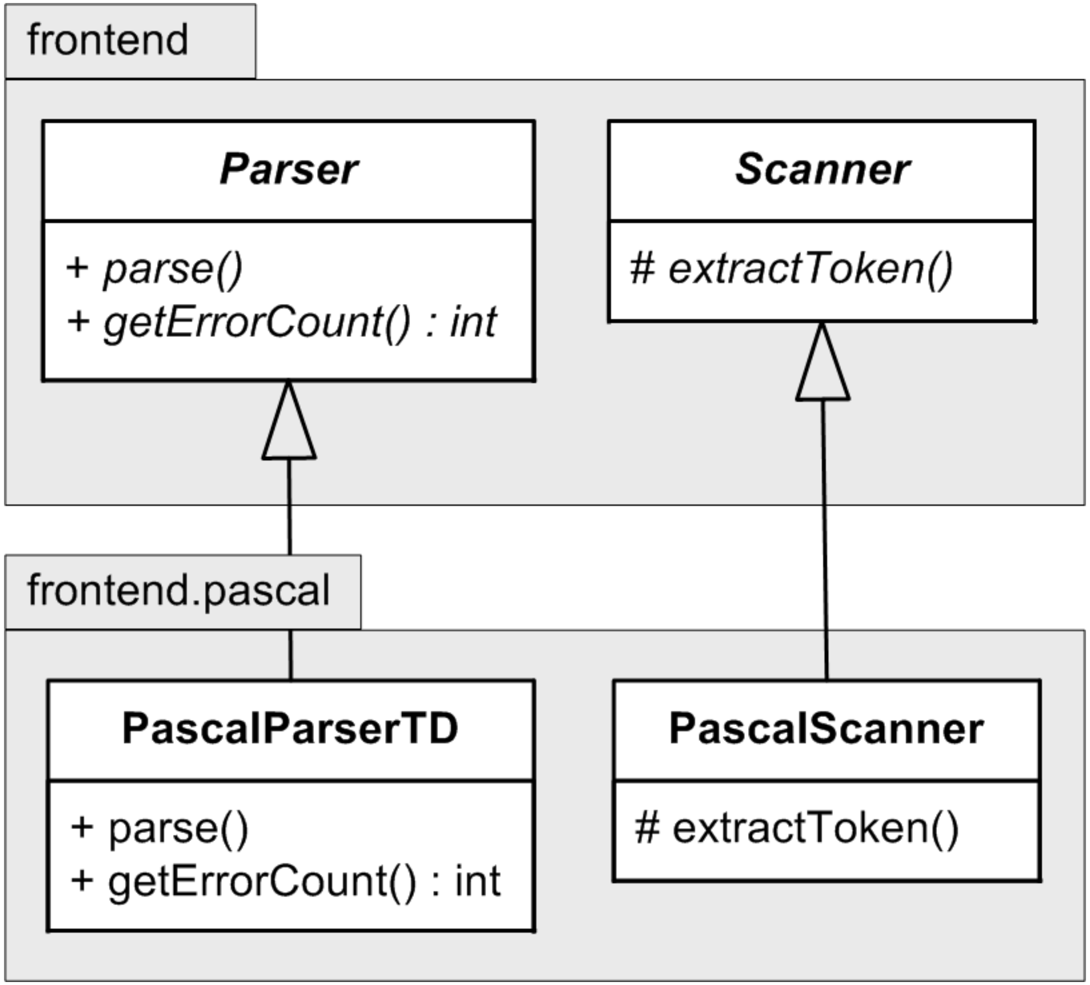
- Download the Java source code from each chapter of the book: http://www.cs.sjsu.edu/~mak/CMPE152/sources/

- You will not survive this course if you use a simple text editor like Notepad to view and edit the Java code.
  - The complete Pascal interpreter in Chapter 12 contains over 120 classes.

Integrated Development Environment (IDE)

- You can use either Eclipse or NetBeans.
  - Eclipse is preferred because later you will be able to use an ANTLR plug-in.
- Learn how to create projects, edit source files, single-step execution, set breakpoints, examine variables, read stack dumps, etc.

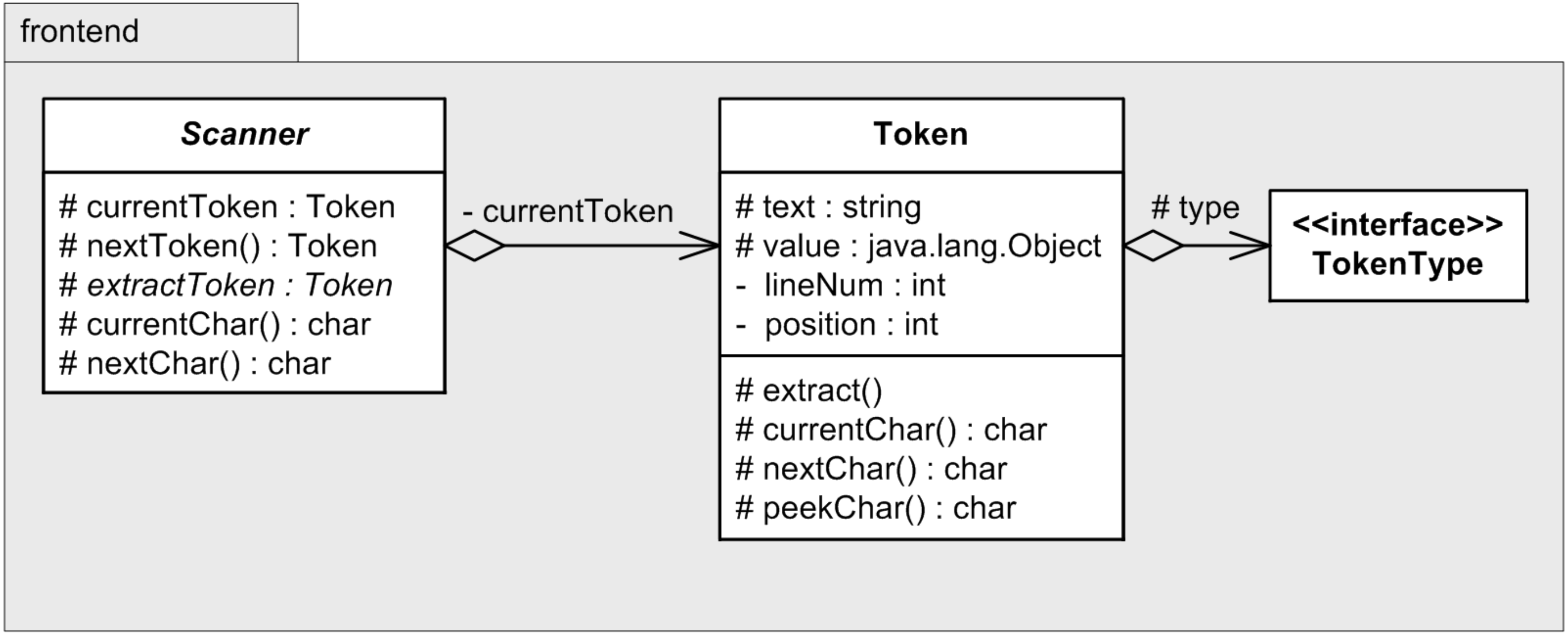
Pascal-Specific Front End Classes



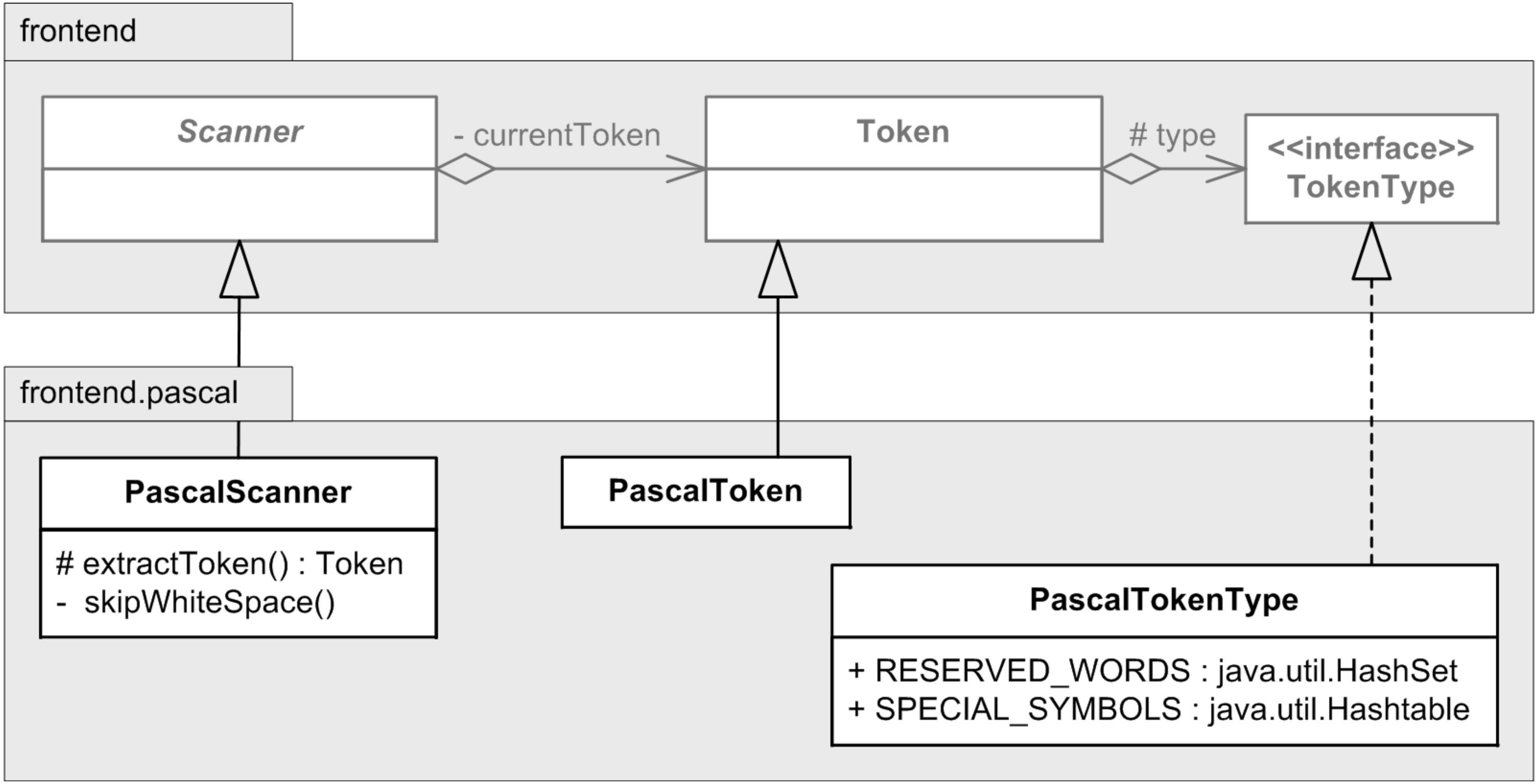
The Payoff

- Now that we have …
  - Source language-independent framework classes
  - Pascal-specific subclasses
    - Mostly just placeholders for now
  - An end-to-end test (the program listing generator)
- … we can work on the individual components
  - Without worrying (too much) about breaking the rest of the code.

Front End Framework Classes



Pascal-Specific Subclasses



PascalTokenType

- Each token is an enumerated value.

```
enum class PascalTokenType
{
    // Reserved words.
    AND, ARRAY, BEGIN, CASE, CONST, DIV, DO, DOWNTO, ELSE, END,
    FILE, FOR, FUNCTION, GOTO, IF, IN, LABEL, MOD, NIL, NOT,
    OF, OR, PACKED, PROCEDURE, PROGRAM, RECORD, REPEAT, SET,
    THEN, TO, TYPE, UNTIL, VAR, WHILE, WITH,

    // Special symbols.
    PLUS, MINUS, STAR, SLASH, COLON_EQUALS,
    DOT, COMMA, SEMICOLON, COLON, QUOTE,
    EQUALS, NOT_EQUALS, LESS_THAN, LESS_EQUALS,
    GREATER_EQUALS, GREATER_THAN, LEFT_PAREN, RIGHT_PAREN,
    LEFT_BRACKET, RIGHT_BRACKET, LEFT_BRACE, RIGHT_BRACE,
    UP_ARROW, DOT_DOT,

    IDENTIFIER, INTEGER, REAL, STRING,
    ERROR, END_OF_FILE,
};
```

PascalTokenType, *cont’d*

- The static set **RESERVED_WORDS** contains all of Pascal’s reserved word strings.

```
vector<string> rw_strings =
{
    "AND", "ARRAY", "BEGIN", "CASE", "CONST", "DIV", "DO", "DOWNTO",
    "ELSE", "END", "FILE", "FOR", "FUNCTION", "GOTO", "IF", "IN",
    "LABEL", "MOD", "NIL", "NOT", "OF", "OR", "PACKED", "PROCEDURE",
    "PROGRAM", "RECORD", "REPEAT", "SET", "THEN", "TO", "TYPE",
    "UNTIL", "VAR", "WHILE", "WITH"
};
```

```
vector<PascalTokenType> rw_keys =
{
    PascalTokenType::AND,
    PascalTokenType::ARRAY,
    PascalTokenType::BEGIN,
    PascalTokenType::CASE,
    ...
};
```

PascalTokenType, *cont’d*

```
for (int i = 0; i < rw_strings.size(); i++)
{
    RESERVED_WORDS[rw_strings[i]] = rw_keys[i];
}
```

PascalTokenType, *cont'd*

- We can test whether a token is a reserved word or an identifier:

```
// Is it a reserved word or an identifier?
string upper_case(text);
transform(upper_case.begin(), upper_case.end(),
          upper_case.begin(), ::toupper);
if (PascalToken::RESERVED_WORDS.find(upper_case)
        != PascalToken::RESERVED_WORDS.end())
{
    // Reserved word.
    type = (TokenType) PascalToken::RESERVED_WORDS[upper_case];
    value = new DataValue(upper_case);
}
else
{
    // Identifier.
    type = (TokenType) PT_IDENTIFIER;
}
```

PascalTokenType, *cont'd*

- Static hash table **SPECIAL_SYMBOLS** contains all of Pascal's special symbols.
  - Each entry's key is the string, such as **"<"** , **"="** , **"<=”**
  - Each entry's value is the corresponding enumerated value.

```
vector<string> ss_strings =
{
    "+", "-", "*", "/", ":=", ".", ",", ";", ":", "'", "=", "<>",
    "<", "<=", ">=", ">", "(", ")", "[", "]", "{", "}",  "^", ".."
};

vector<PascalTokenType> ss_keys =
{
    PascalTokenType::PLUS,
    PascalTokenType::MINUS,
    PascalTokenType::STAR,
    PascalTokenType::SLASH,
    ...
}
```

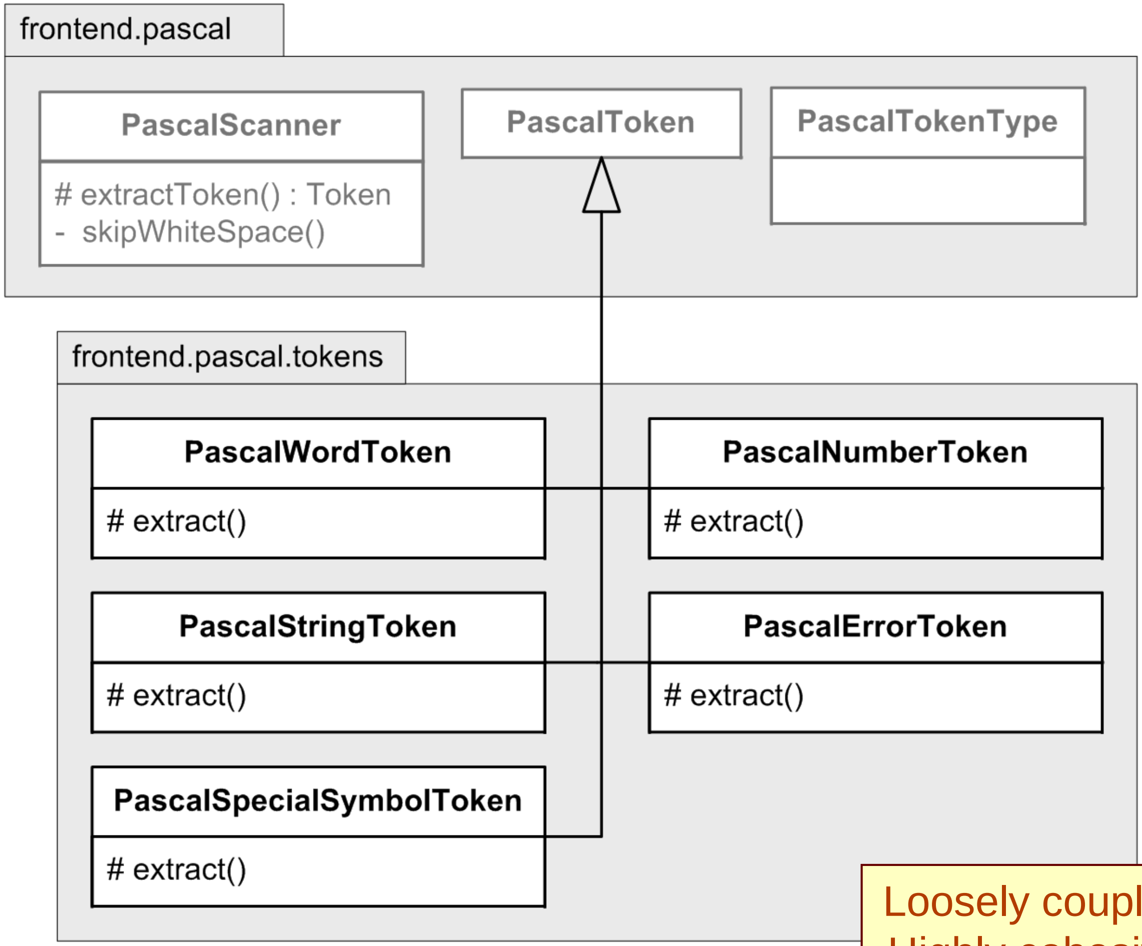
PascalTokenType, *cont'd*

```
for (int i = 0; i < ss_strings.size(); i++)
{
    SPECIAL_SYMBOLS[ss_strings[i]] = ss_keys[i];
}
```

- We can test whether a token is a special symbol:

```
if (PascalToken::SPECIAL_SYMBOLS.find(string_ch)
       != PascalToken::SPECIAL_SYMBOLS.end())
{
    token = new PascalSpecialSymbolToken(source);
}
```

Pascal-Specific Token Classes



Loosely coupled.
Highly cohesive.

- Each class **PascalWordToken**, **PascalNumberToken**, **PascalStringToken**, **PascalSpecialSymbolToken**, and **PascalErrorToken** is is a subclass of class **PascalToken**.
  - **PascalToken** is a subclass of class **Token**.
- Each Pascal token subclass overrides the default **extract()** method of class **Token**.
  - The default method could only create single-character tokens.