CS 153: Concepts of Compiler Design

August 31 Class Meeting

Department of Computer Science
San Jose State University

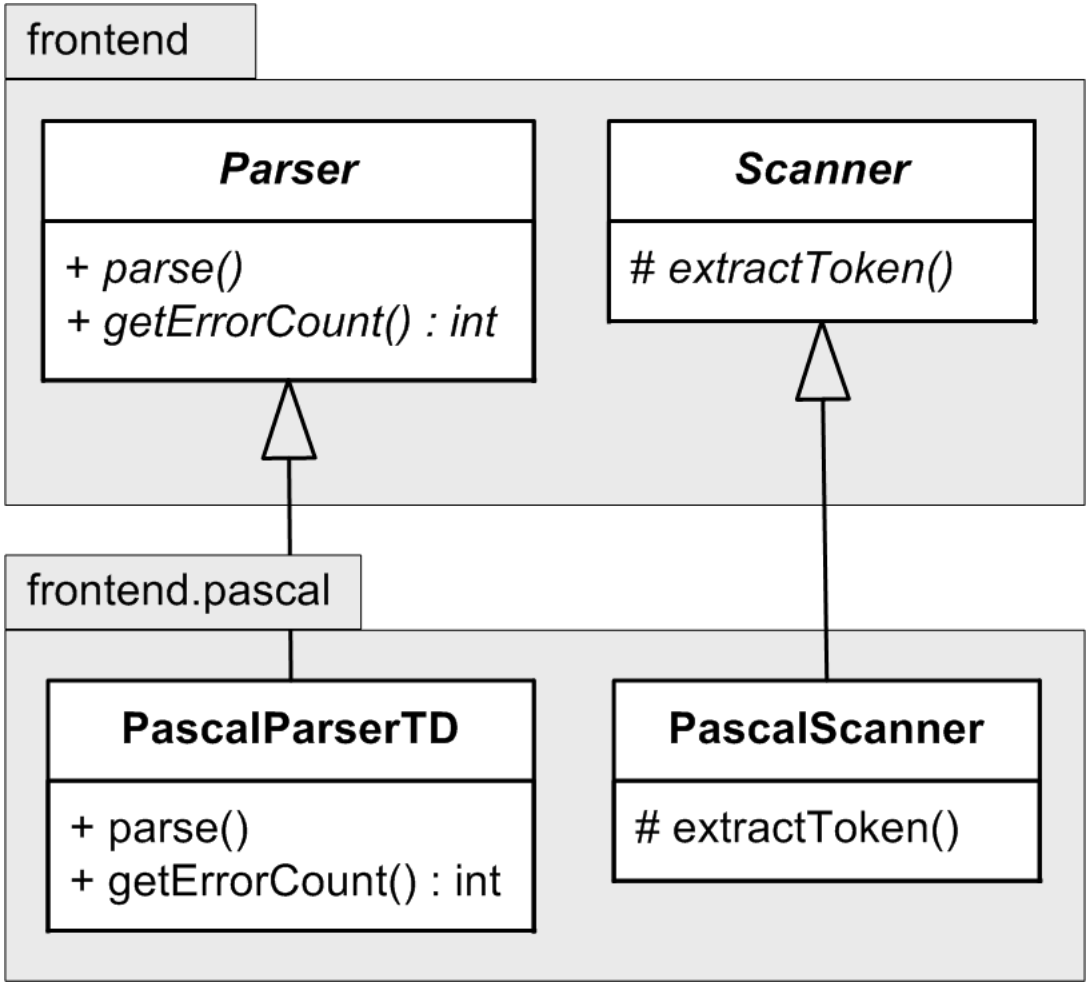
Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

An Apt Quote?

Before I came here, I was confused about this subject.
Having listened to your lecture, I am still confused,
but on a higher level.

Enrico Fermi, physicist, 1901-1954

Pascal-Specific Front End Classes



- **`PascalParserTD`** is a subclass of **`Parser`** and implements the **`parse()`** and **`getErrorCount()`** methods for Pascal.
  - TD for "top down"
- **`PascalScanner`** is a subclass of **`Scanner`** and implements the **`extractToken()`** method for Pascal.

Strategy Design Pattern

The Pascal Parser Class

- The initial version of method **`parse()`** does hardly anything, but it forces the scanner into action and serves our purpose of doing end-to-end testing.

```
public void parse()
    throws Exception
{
    Token token;
    long startTime = System.currentTimeMillis();

    while (!((token = nextToken()) instanceof EofToken)) {}

    // Send the parser summary message.
    float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
    sendMessage(new Message(PARSER_SUMMARY,
                            new Number[] {token.getLineNumber(),
                                          getErrorCount(),
                                          elapsedTime}));
}
```

What does this **`while`** loop do?

The Pascal Scanner Class

- The initial version of method **`extractToken()`** doesn't do much either, other than create and return either a default token or the EOF token.

```
protected Token extractToken()
    throws Exception
{
    Token token;
    char currentChar = currentChar();

    // Construct the next token. The current character determines the
    // token type.
    if (currentChar == EOF) {
        token = new EofToken(source);
    }
    else {
        token = new Token(source);
    }

    return token;
}
```

Remember that the **`Scanner`** method **`nextToken()`** calls the abstract method **`extractToken()`**.

Here, the **`Scanner`** subclass **`PascalScanner`** implements method **`extractToken()`**.

The Token Class

- The **`Token`** class's default **`extract()`** method extracts just one character from the source.
  - This method will be overridden by the various token subclasses.
  - It serves our purpose of doing end-to-end testing.

```
protected void extract()
    throws Exception
{
    text = Character.toString(currentChar());
    value = null;

    nextChar();  // consume current character
}
```

The Token Class, *cont'd*

- A character (or a token) is "consumed" after it has been read and processed, and the next one is about to be read.
- If you forget to consume, you will loop forever on the same character or token.

A Front End Factory Class

- A language-specific parser goes together with a scanner for the same language.
- But we don't want the framework classes to be tied to a specific language. Framework classes should be language-independent.
- We use a factory class to create a matching parser-scanner pair.

Factory Method Design Pattern

A Front End Factory Class, *cont'd*

- **Good:**

"Coding to the interface."

```
Parser parser =
    FrontendFactory.createParser( … );
```

  - Arguments to the **`createParser()`** method enable it to create and return a parser bound to an appropriate scanner.
  - Variable **`parser`** doesn't have to know what kind of parser subclass the factory created.
  - Once again, the idea is to maintain loose coupling.

A Front End Factory Class, *cont'd*

- **Good:**

```
Parser parser =
    FrontendFactory.createParser( … );
```

- **Bad:**

```
PascalParserTD parser =
    new PascalParserTD( … )
```

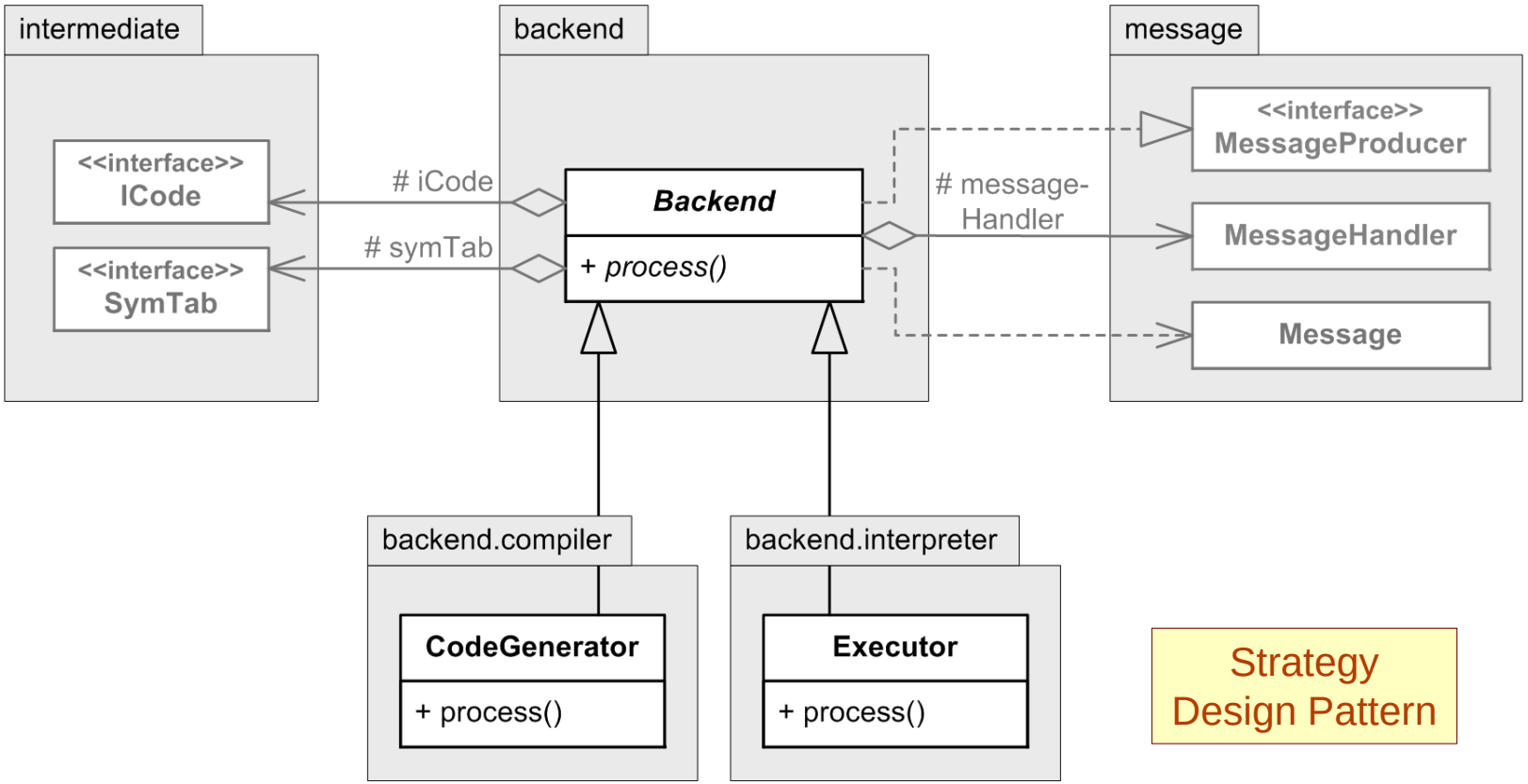
  - Why is this bad?
  - Now variable **`parser`** is tied to a specific language.

A Front End Factory Class, *cont'd*

```
public static Parser createParser(String language, String type,
                                  Source source)
    throws Exception
{
    if (language.equalsIgnoreCase("Pascal") &&
        type.equalsIgnoreCase("top-down"))
    {
        Scanner scanner = new PascalScanner(source);
        return new PascalParserTD(scanner);
    }
    else if (!language.equalsIgnoreCase("Pascal")) {
        throw new Exception("Parser factory: Invalid language '" +
                            language + "'");
    }
    else {
        throw new Exception("Parser factory: Invalid type '" +
                            type + "'");
    }
}
```

Initial Back End Subclasses

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



The Code Generator Class

- 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)

```
public void process(ICode iCode, SymTab symTab)
    throws Exception
{
    long startTime = System.currentTimeMillis();
    float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
    int instructionCount = 0;

    // Send the compiler summary message.
    sendMessage(new Message(COMPILER_SUMMARY,
                            new Number[] {instructionCount,
                                          elapsedTime}));
}
```

The Executor Class

- 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)

```
public void process(ICode iCode, SymTab symTab)
    throws Exception
{
    long startTime = System.currentTimeMillis();
    float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
    int executionCount = 0;
    int runtimeErrors = 0;

    // Send the interpreter summary message.
    sendMessage(new Message(INTERPRETER_SUMMARY,
                            new Number[] {executionCount,
                                          runtimeErrors,
                                          elapsedTime}));
}
```

A Back End Factory Class

```
public static Backend createBackend(String operation)
     throws Exception
{
    if (operation.equalsIgnoreCase("compile") {
        return new CodeGenerator();
    }
    else if (operation.equalsIgnoreCase("execute")) {
        return new Executor();
    }
    else {
        throw new Exception("Backend factory: Invalid operation '" +
                            operation + "'");
    }
}
```

End-to-End: Program Listings

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

```
source = new Source(new BufferedReader(new FileReader(filePath)));
source.addMessageListener(new SourceMessageListener());

parser = FrontendFactory.createParser("Pascal", "top-down", source);
parser.addMessageListener(new ParserMessageListener());

backend = BackendFactory.createBackend(operation);
backend.addMessageListener(new BackendMessageListener());

parser.parse();
iCode = parser.getICode();
symTab = parser.getSymTab();

backend.process(iCode, symTab);
source.close();
```

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`** has inner classes that implement the **`MessageListener`** interface.

```
private static final String SOURCE_LINE_FORMAT = "%03d %s";

private class SourceMessageListener implements MessageListener
{
    public void messageReceived(Message message)
    {
        MessageType type = message.getType();
        Object body[] = (Object []) message.getBody();

        switch (type) {

            case SOURCE_LINE: {
                int lineNumber = (Integer) body[0];
                String lineText = (String) body[1];

                System.out.println(String.format(SOURCE_LINE_FORMAT,
                                                 lineNumber, lineText));
                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.

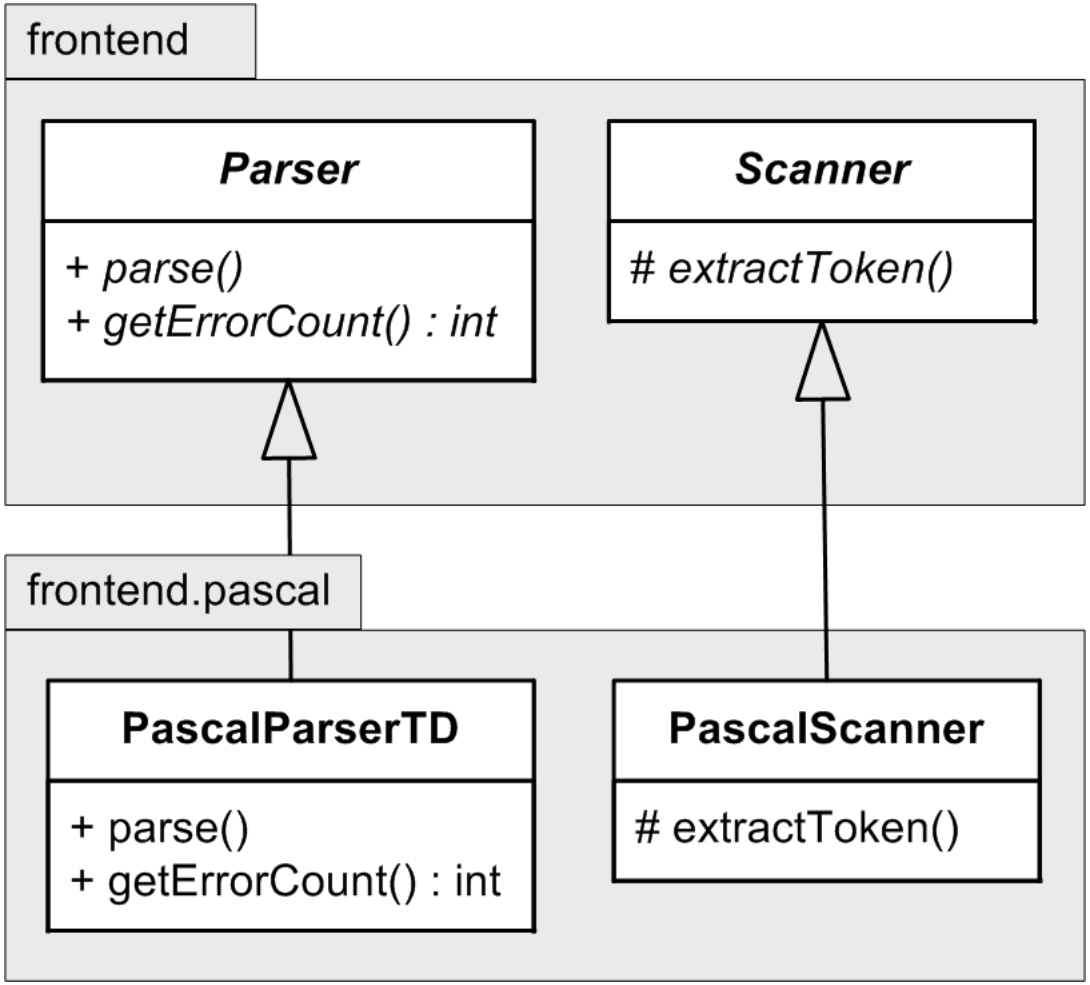
Source Files from the Book

- Download the Java source code from each chapter of the book: http://www.cs.sjsu.edu/~mak/CS153/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 127 classes and interfaces.

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.

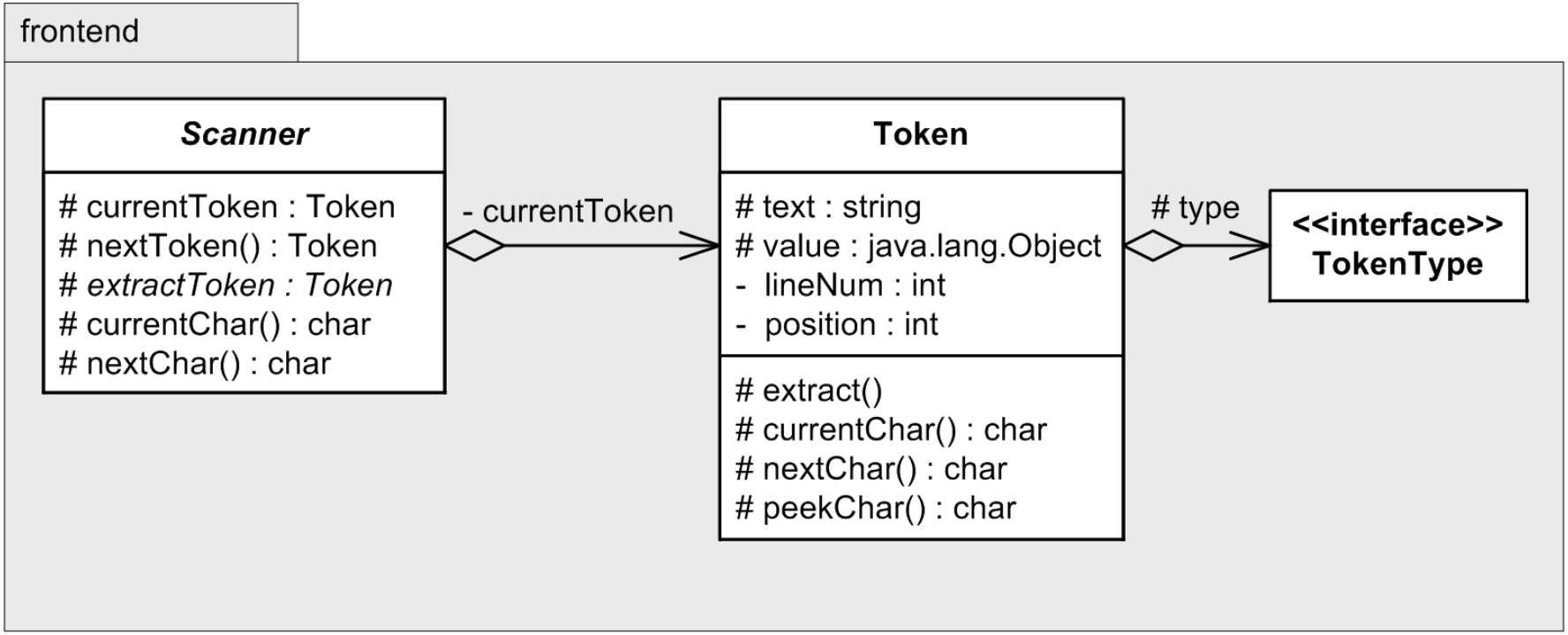
Pascal-Specific Front End Classes



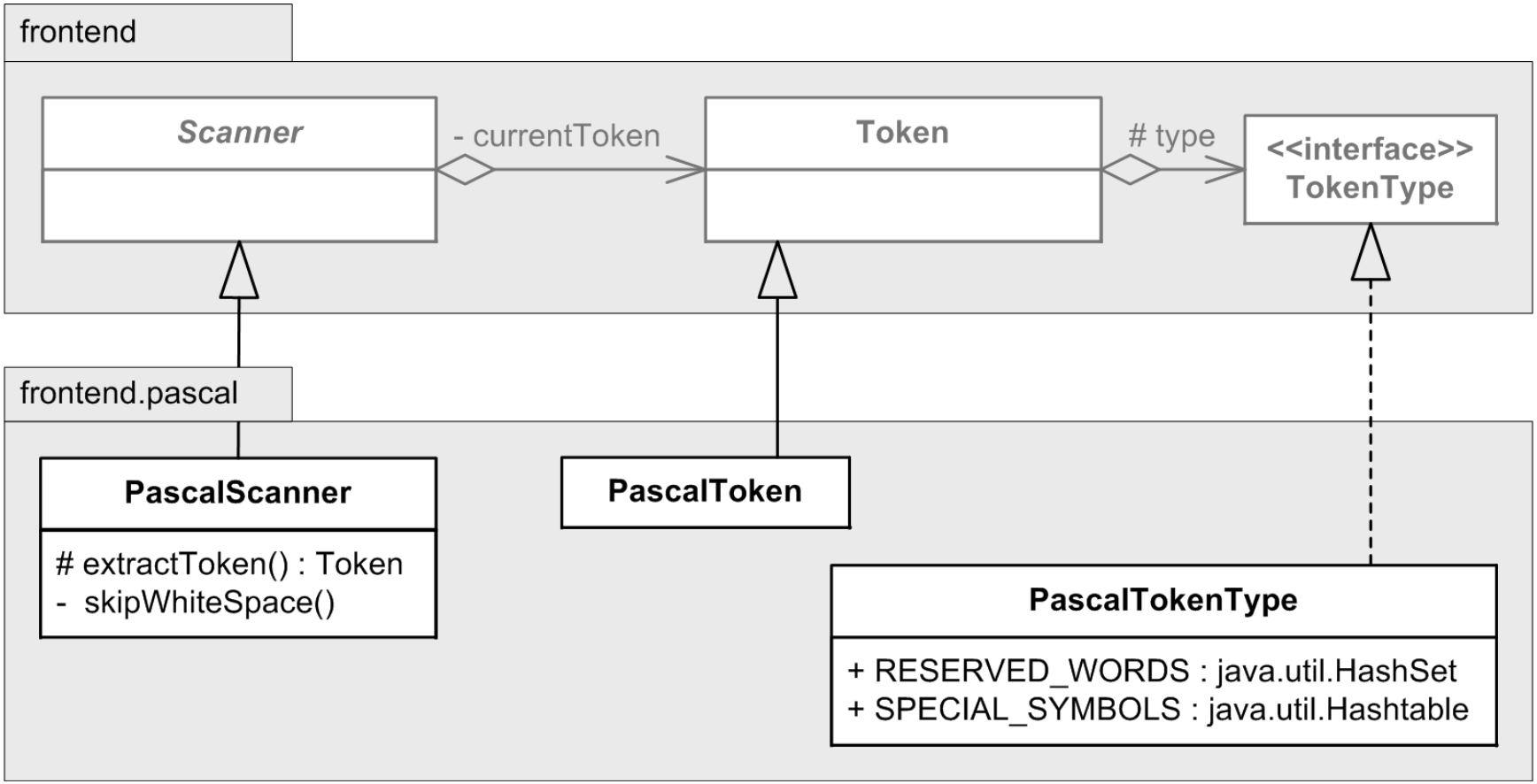
The Payoff

- 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.

```
public enum PascalTokenType implements TokenType
{
    // 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 in lower case: "**and**" , "**array**" , "**begin**" , etc.

```
// Set of lower-cased Pascal reserved word text strings.
public static HashSet<String> RESERVED_WORDS =
                                            new HashSet<String>();
static {
    PascalTokenType values[] = PascalTokenType.values();
    for (int i = AND.ordinal(); i <= WITH.ordinal(); ++i) {
        RESERVED_WORDS.add(values[i].getText().toLowerCase());
    }
}
```

- We can test whether a token is a reserved word:

```
if (RESERVED_WORDS.contains(text.toLowerCase())) …
```

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.

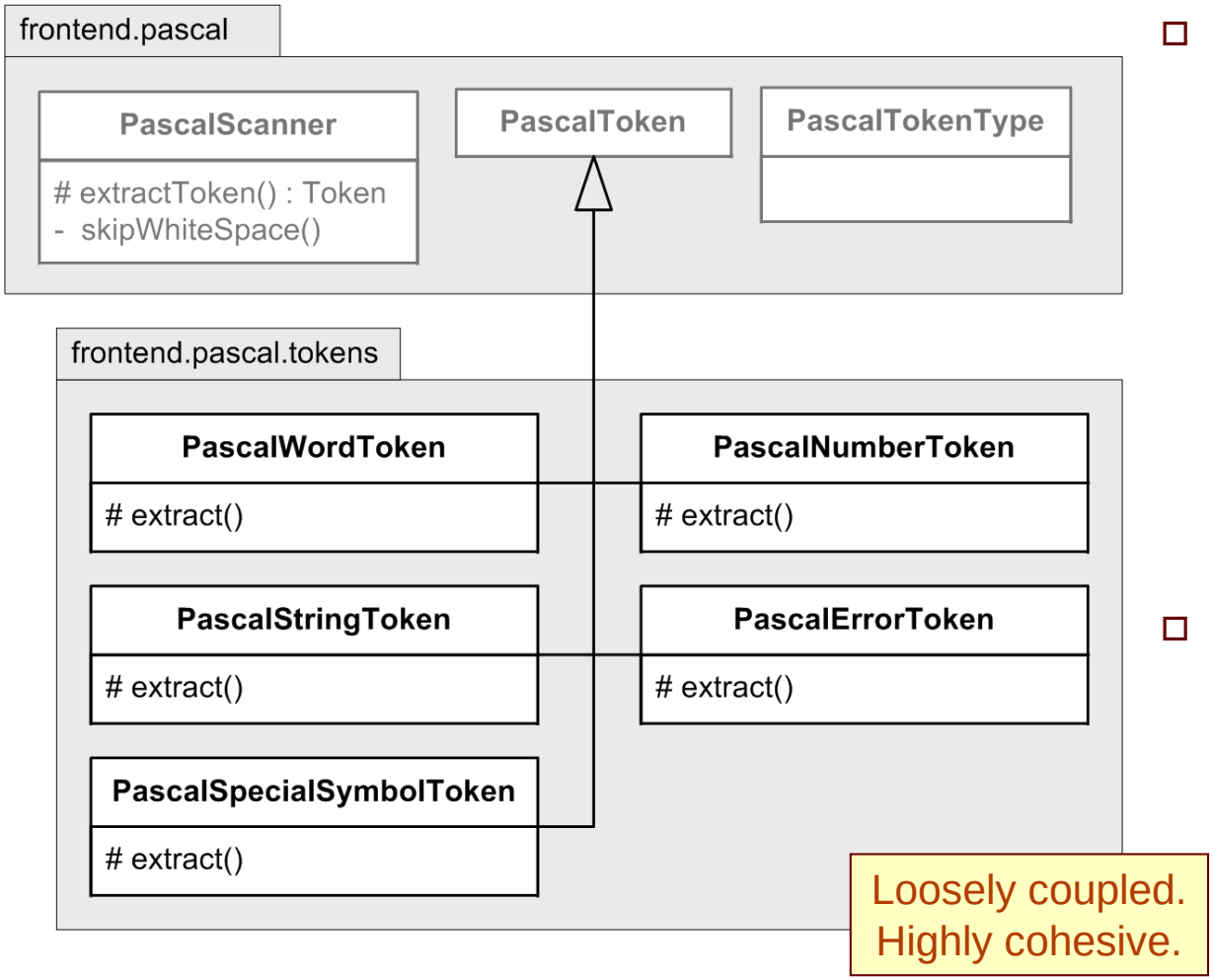
```
// Hash table of Pascal special symbols.
// Each special symbol's text is the key to its Pascal token type.
public static Hashtable<String, PascalTokenType> SPECIAL_SYMBOLS =
    new Hashtable<String, PascalTokenType>();
static {
    PascalTokenType values[] = PascalTokenType.values();
    for (int i = PLUS.ordinal(); i <= DOT_DOT.ordinal(); ++i) {
        SPECIAL_SYMBOLS.put(values[i].getText(), values[i]);
    }
}
```

PascalTokenType, *cont'd*

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

```
if (PascalTokenType.SPECIAL_SYMBOLS
      .containsKey(Character.toString(currentChar))) …
```

Pascal-Specific Token Classes



- Each class **`PascalWordToken`**, **`PascalNumberToken`**, **`PascalStringToken`**, **`PascalSpecial-SymbolToken`**, 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.