Prosjektet

Hvordan bør et program deles opp?

Ingen faste regler men

- Hvilken oppdeling virker naturlig?
- Hvilken oppdeling gir få aksesser mellom modulene?
- Hvordan flyter data?

Noen programmeringsspråk har mekanismer for store moduler – men langt fra alle. Java har `package`.

Begrunnelse

Anta at vi skal utvikle et CAD-system. Et firma i India har laget en god GUI-modul.
Men, begge har en klasse `Bar`.

Moduler kan løse dette problemet.

Alle filene som skal inngå i en Java-pakke starter med «package *navn*».

Pakkenavn bør bestå av opphavsstedets internettadresse (baklengs) og et lokalt navn. Vårt kompilatorprosjekt heter

no.uio.ifi.pascal2100

Eksempel

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

(Under kompileringen må klassen ligge i et fil-tre som tilsvarer leddene i pakkenavnet; våre filer ligger i

no/uio/ifi/pascal2100/scanner/Scanner.java

og tilsvarende.)

Vi kan hente klasser fra alle pakker så lenge de finnes i CLASSPATH:

B.java

```
class B {
    public static void main (String arg[]) {
        System.out.println("P1.A.x = " + P1.A.x);
    }
}
```

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

Ifis standard CLASSPATH er:

```
$ printenv CLASSPATH
/usr/share/java:/usr/share/java/postgresql-jdbc.jar:.
```

Beskyttelse

Klasser kan beskyttes:

— er usynlig utenfor pakken.

`public` kan brukes fra andre pakker.

For klasseelementer (dvs metoder og variabler) gjelder:

`private` er bare tilgjengelige i klassen.

`protected` er for klassen og subklasser.

— er bare for bruk innen pakken.

`public` kan benyttes overalt.

For å unngå å skrive mange lange navn som `no.uio.ifi.pascal2100.main.Main.version`, kan vi importere klasser fra pakker:

B.java

```
import P1.A;

class B {
    public static void main (String arg[]) {
        System.out.println("P1.A.x = " + A.x);
    }
}
```

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

Vi kan også importere alle klassene fra en pakke:

B.java

```
import P1.*;

class B {
    public static void main (String arg[]) {
        System.out.println("P1.A.x = " + A.x);
    }
}
```

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

En siste mulighet er å importere *statiske* deklarasjoner i en klasse:

B.java

```
import static P1.A.*;

class B {
    public static void main (String arg[]) {
        System.out.println("P1.A.x = " + x);
    }
}
```

P1/A.java

```
package P1;

public class A {
    public static int x = 1;
}
```

Hva kan en pakke inneholde?

En Java-pakke kan bare inneholde klasser.

Men det er også nyttig noen ganger å ha data og metoder som er «globale» for prosjektet vårt. Disse legger vi i `Main`-objektet som `public static`.

Enum-klasser

Noen ganger har man diskrete data som kun kan ha et begrenset antall fast definerte verdier:

Kortfarge	Kløver, ruter, hjerter, spar
Tippetegn	Hjemmeseier, uavgjort, borteseier
Ukedag	Mandag, tirsdag, onsdag, torsdag, fredag, lørdag, søndag

Å representere disse med heltall er en halvgod løsning.

Java tilbyr **enum-klasser**:

Tippetegn.java

```
enum Tippetegn {
    Hjemmeseier, Uavgjort, Borteseier;
    // ...
}
```

Tipping.java

```
class Tipping {
    public static void main (String arg[]) {
        Tippetegn rekke[] = new Tippetegn[12+1];

        rekke[1] = Tippetegn.Hjemmeseier;
        rekke[2] = Tippetegn.Borteseier;
        rekke[3] = Tippetegn.Borteseier;
    }
}
```

En nyttig metode

Til sist et lite tips:

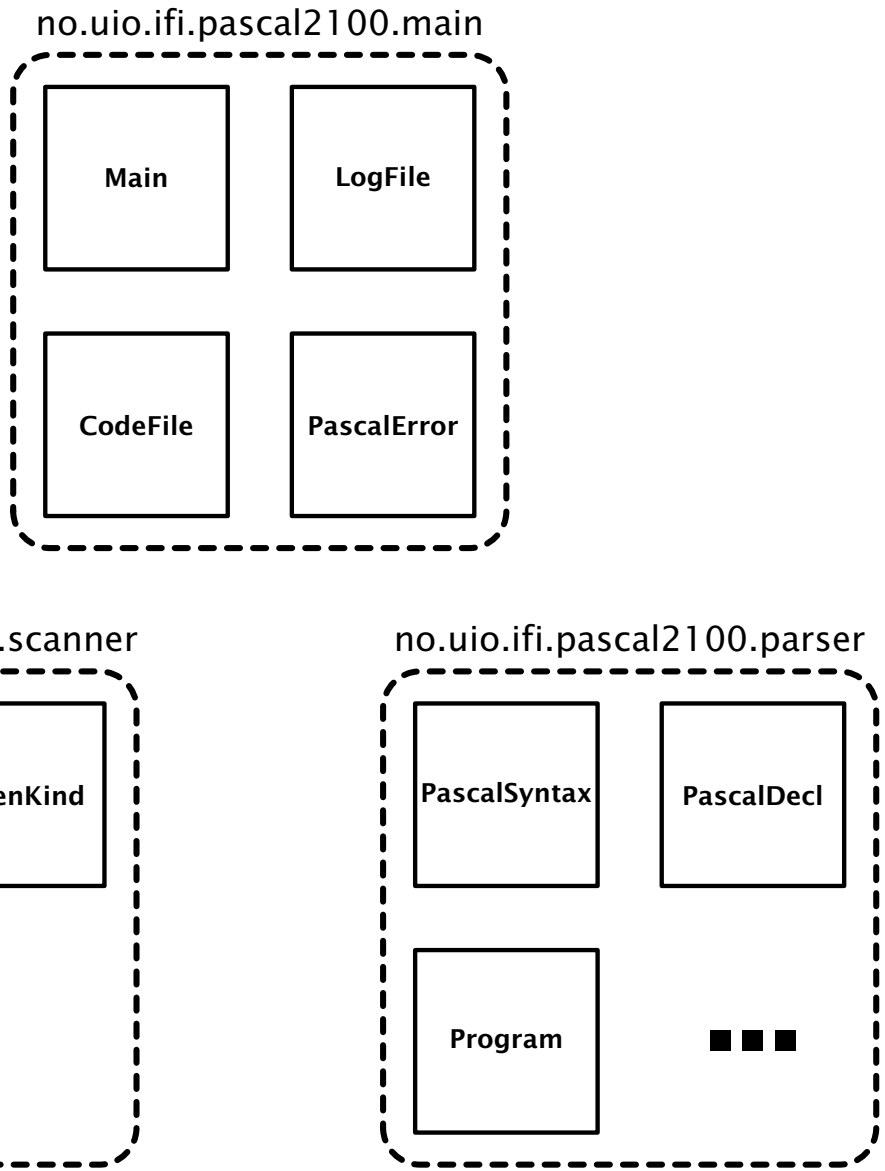
Når man skriver oo-programmer, er det nyttig om alle objektene kan identifisere seg selv. Nøyaktig hvilken informasjon de skal gi, avhenger av programmet.

I dette prosjektet skal vi la alle våre klasser inneholde en metode `identify` som gir nok informasjon til å identifisere objektene.

Som eksempel: Scanner:

```
public String identify() {
    return "Scanner reading " + sourceFileName;
}
```

Våre pakker


no.uio.ifi.pascal2100.main
Main
LogFile
CodeFile
PascalError
no.uio.ifi.pascal2100.scanner
Token
TokenKind
Scanner
no.uio.ifi.pascal2100.parser
PascalSyntax
PascalDecl
Program

Hovedprogrammet Main.java

```
package no.uio.ifi.pascal2100.main;

import no.uio.ifi.pascal2100.parser.*;
import no.uio.ifi.pascal2100.scanner.*;
import static no.uio.ifi.pascal2100.scanner.TokenKind.*;

import java.io.*;

public class Main {
    public static final String version = "2015-08-23";

    public static Library library;
    public static LogFile log = new LogFile();

    private static String sourceFileName, baseFileName;
    private static boolean testParser = false, testScanner = false;
```

Modulen «main»

```
public static void main(String arg[]) {
    System.out.println("This is the Ifi Pascal2100 compiler (" +
                       version + ")");

    int exitStatus = 0;
    try {
        readArgs(arg);
        log.init(baseFileName + ".log");

        Scanner s = new Scanner(sourceFileName);
        if (testScanner)
            doTestScanner(s);
        else if (testParser)
            doTestParser(s);
        else
            doRunRealCompiler(s);
    } catch (PascalError e) {
        System.out.println();
        System.err.println(e.getMessage());
        exitStatus = 1;
    } finally {
        log.finish();
    }

    System.exit(exitStatus);
}
```

Modulen «main»

```
private static void doTestScanner(Scanner s) {
    while (s.nextToken.kind != eofToken)
        s.readNextToken();
}


private static void doTestParser(Scanner s) {
    Program prog = Program.parse(s);
    if (s.curToken.kind != eofToken)
        error("Scanner error: Garbage after the program!");

    prog.prettyPrint();
}


private static void doRunRealCompiler(Scanner s) {
    System.out.print("Parsing...");
    Program prog = Program.parse(s);

  ⋮
```

Skanner

En kompilator *kan* lese og tolke en program tegn for tegn, men det er mye lettere om det kan gjøres *symbol for symbol*. Dette ordner en **skanner**.

En skanner gjør følgende:

- Leser programkoden fra en fil
- Fjerner alle kommentarer
- Deler resten av teksten opp i symboler («tokens»)

```
/* Et minimalt Pascal-program */
program Mini;
begin
   write('x');
end.
```

har disse symbolene:

program	name: Mini	;	begin	name: write	(	'x'
)	;	end	.			

I vår skanner

Vår skanner kan levere **token** som definert i klassen Token:

```
package no.uio.ifi.pascal2100.scanner;

import static no.uio.ifi.pascal2100.scanner.TokenKind.*;

public class Token {
    public TokenKind kind;
    public String id, strVal;
    public int intVal, lineNum;
```

TokenKind er definert i en enum-klasse:

```
public enum TokenKind {
    nameToken("name"),
    intValToken("number"),
    stringValToken("text string"),

    addToken("+"),
    assignToken(":="),
    colonToken(":"),
    commaToken(","),


     ⋮


    eofToken("e-o-f");

    private String image;

    TokenKind(String im) {
        image = im;
    }


    public String identify() {
        return image + " token";
    }
```

Klassen Scanner

```
public class Scanner {
    public Token curToken = null, nextToken = null;

    private LineNumberReader sourceFile = null;
    private String sourceFileName, sourceLine = "";
    private int sourcePos = 0;

    public Scanner(String fileName) {
        sourceFileName = fileName;
        try {
            sourceFile = new LineNumberReader(new FileReader(fileName));
        } catch (FileNotFoundException e) {
            Main.error("Cannot read " + fileName + "!");
        }

        readNextToken();  readNextToken();
    }


    public String identify() {
        return "Scanner reading " + sourceFileName;
    }
```

Symbolene leses inn i curToken og nextToken i metoden readNext:

```
public void readNextToken() {
    curToken = nextToken;  nextToken = null;

    ⋮

    Main.log.noteToken(nextToken);
}


private void readNextLine() {
    if (sourceFile != null) {
        try {
            sourceLine = sourceFile.readLine();
            if (sourceLine == null) {
                sourceFile.close();  sourceFile = null;
                sourceLine = "";
            } else {
                sourceLine += " ";
            }
            sourcePos = 0;
        } catch (IOException e) {
            Main.error("Scanner error: unspecified I/O error!");
        }
    }
    if (sourceFile != null)
        Main.log.noteSourceLine(getFileLineNum(), sourceLine);
}
```

Hva er en god feilmelding?

Enda litt bedre

```
ERROR: Syntax error found in line 217:
    if (x == y+1)
```

Melding med mening

Meldingen bør fortelle hva som er galt:

```
ERROR in line 217: Illegal expression.
    if (x == y+1)
```

Den beste meldingen

Meldingen bør angi hvorledes kompilatoren «tenker»:

```
ERROR in line 217:
  Expected a value but found '='.
    if (x == y+1)
```

Feil

Hva gjør man med feil?

- Før prøvde man å finne så mange feil som mulig.
- Vi skal stoppe med melding ved første feil.

Metoden `Main.error`

```
    public static void error(String message) {
        log.noteError(message);
        throw new PascalError(message);
    }
```

PascalError.java

```
public class PascalError extends RuntimeException {

    PascalError(String message) {
        super(message);
    }
}
```

Noen ganger tabber *vi* oss ut!

```
public void panic(String where) {
    error("PANIC! Programming error in " + where);
}
```

Testutskrifter

Alle vil gjøre feil under arbeidet med kompilatoren. For enklere å oppdage feilene når de skjer, skal vi bygge inn ulike testutskrifter som brukeren enkelt kan slå på:

Opsjon	Hva dumpes?	Del
-logB	Navnebindingen	3
-logP	Parseringen	2
-logS	Skanneren	1
-logT	Typesjekkingen	3
-logY	«Pretty-printing»	2

Modulen log

Brukeren kan slå av og på logging.

```
public class LogFile {
    boolean doLogBinding = false, doLogParser = false, doLogPrettyPrint = false,
        doLogScanner = false, doLogTypeChecks = false;



    public void noteSourceLine(int lineNum, String line) {
        if (doLogParser || doLogScanner)
            writeLogLine(String.format("%4d: %s",lineNum,line));
    }



    public void noteToken(Token tok) {
        if (doLogScanner)
            writeLogLine("Scanner: " + tok.identify());
    }
```

Klassen LogFile

```
$ ~inf2100/pascal2100 -testscanner mini.pas
$ more mini.log

   1:
   2: /* Et minimalt Pascal-program */
   3: program Mini;
Scanner: program token on line 3
Scanner: name token on line 3: mini
Scanner: ; token on line 3
   4: begin
Scanner: begin token on line 4
   5:    write('x');
Scanner: name token on line 5: write
Scanner: ( token on line 5
Scanner: text string token on line 5: 'x'
Scanner: ) token on line 5
Scanner: ; token on line 5
   6: end.
Scanner: end token on line 6
Scanner: . token on line 6
Scanner: e-o-f token
```

- Siden utskriften på forrige side kommer fra to kilder, vil flettingen av den kunne variere.

NB!

Variasjoner i fletting er helt normalt og må forventes.

- Når hele programmet er lest, vil skanneren bare returnere **eofToken**.

Mål for del 1

1. Hent ned, pakk ut og kompiler prekoden.
2. Gjør nødvendige endringer slik at skanneren fungerer og at den skriver ut loggmeldinger som vist når vi kjører kompilatoren med opsjonen **–testscanner**.

Siste innspill

- Skanneren er dum! Den lager symboler uten tanke på sammenhengen.
- Skanneren er grådig! Den lager så lange symboler som mulig; for eksempel:

`... ifa ...` blir til `name: ifa`

- En del av jobben er å skjønne basiskoden, resten er å programmere `Scanner.readNextToken`.
- Les kompendiet!
- Det er lov å endre basiskoden.
- Du kan bruke alt du vil fra Java-biblioteket.
- Gruppelærerne er der for å hjelpe dere.
- Begynn i tide!