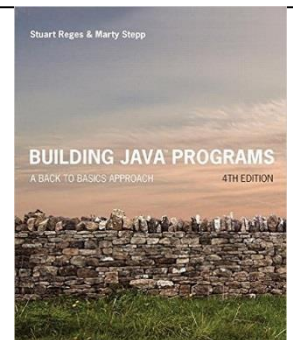


Coding Design, Style, Documentation and Optimization

Reading Assignment: Read Chapters 1.2, 8.1-8.3
Building Java Programs (Stuart Reges and Marty Stepp)



Overview

- ☐ Coding Design.
- ☐ Stepwise Refinement
- ☐ Top-Down Design
 - Advantages
 - Example
- ☐ Example: Print Calendar
- ☐ Why Style and Documentation are important?
- ☐ Coding Optimization
- ☐ Lab 3: Important notes & demo

Coding Design

- ❑ Coding design refers to the low-level program design, i.e. coding.
- ❑ Java supports so called structured programming
 - At the low level, a structured program is composed of simple, hierarchical program flow structures. These are sequence, selection, and repetition.
 - No spaghetti programs, i.e. no gotos.
- ❑ A simple and commonly used method: Step-wise refinement.

Stepwise Refinement

- ❑ The oldest software designing method. It was published by Niklaus Wirth in Communications of the ACM, Vol. 14, No. 4, April 1971, pp. 221-227.
- ❑ A low-level designing method, i.e. a method for designing small programs.
- ❑ The basic idea is to repeatedly decompose pseudocode statements until each pseudocode statement can be coded in a couple of programming language statements.

Top-Down Design

- If we look at a problem as a whole, it may seem impossible to solve because it is so complex.

Examples:

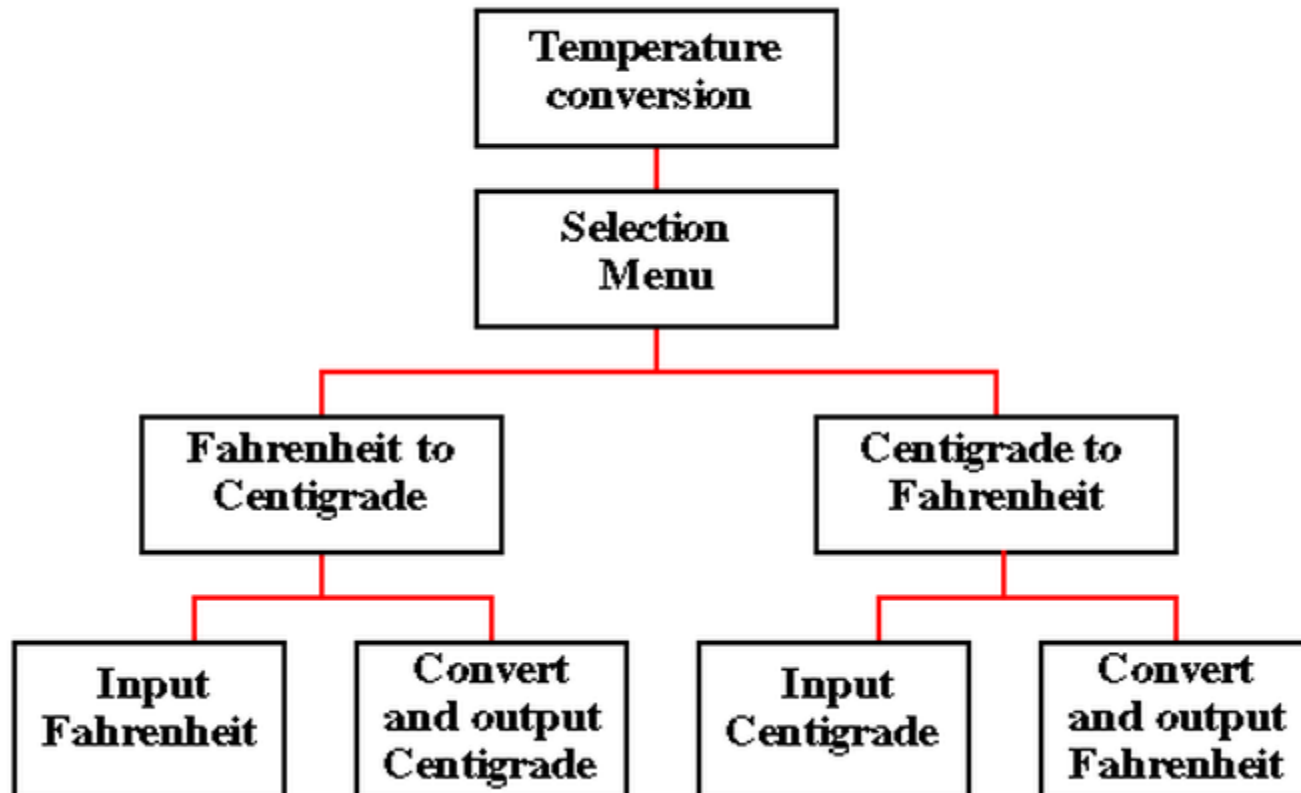
- writing a tax computation program
- writing a word processor
- Complex problems can be solved using **top-down design**, also known as **stepwise refinement**, where
 - We break the problem into parts
 - Then break the parts into parts
 - Soon, each of the parts will be easy to do

Advantages of Top-Down Design

- Breaking the problem into parts helps us to clarify what needs to be done.
- At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
- Parts of the solution may turn out to be reusable.
- Breaking the problem into parts allows more than one person to work on the solution.

A graphical example

Design and create an Java program that, given a temperature in Centigrade converts it to Fahrenheit and vice-versa.



Example: Write a program to print a calendar

❑ Top level design

- To print a calendar.

❑ First refinement

- Read a year
- Print the calendar of the year

❑ Second refinement

- Read a year
- Print the calendar of the year
 - Label the year
 - Print months

2019

January

S	M	Tu	W	Th	F	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

February

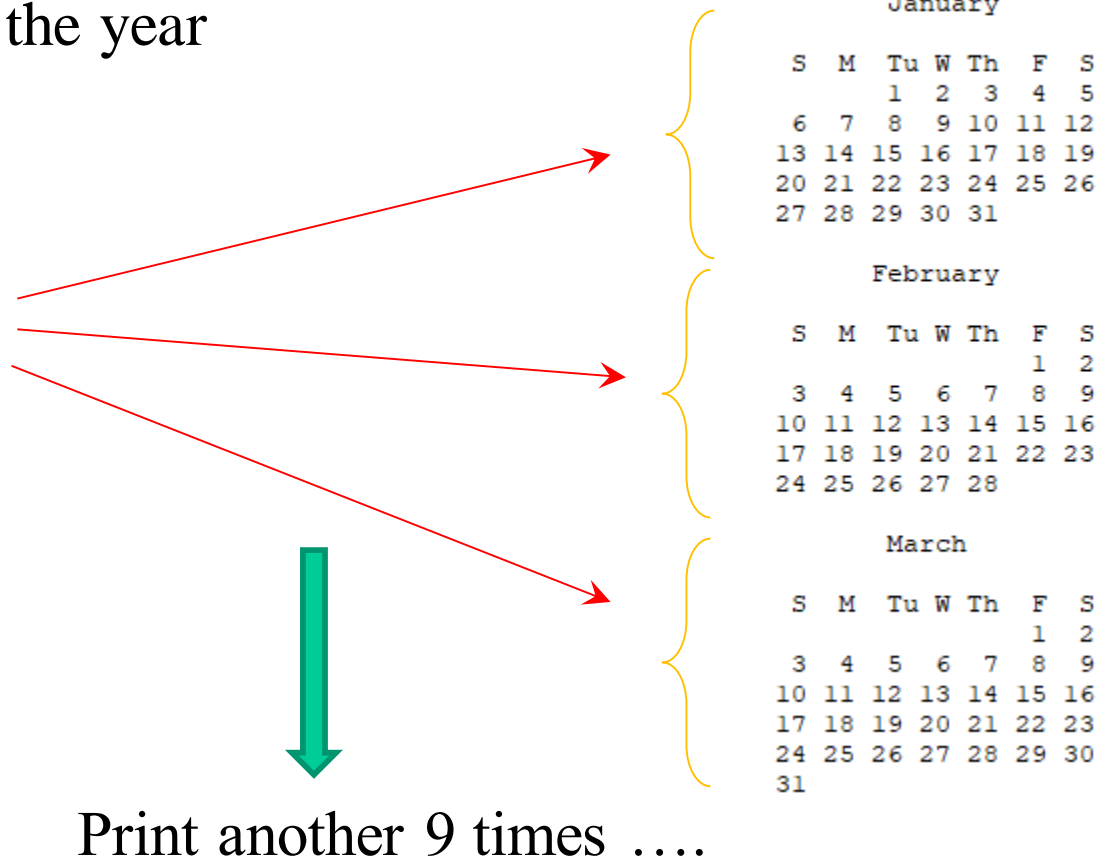
S	M	Tu	W	Th	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28		

March

S	M	Tu	W	Th	F	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Third Refinement

- ❑ Read a year
- ❑ Print the calendar of the year
 - Label the year
 - Repeat 12 times
 - Print a month



Fourth Refinement

- ❑ Read a year
- ❑ Print the calendar of the year
 - Label the year
 - Repeat 12 times

- Print a month

- ✓ Label the month
- ✓ Label Days of the week.
- ✓ Print days

January

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Fifth Refinement

- ❑ Read a year
- ❑ Print the calendar of the year
 - Label the year
 - Repeat 12 times
 - Print a month
 - ✓ Label the month
 - ✓ Label Days of the week.
 - ✓ Print days
 - Determine number of days
 - Position the 1st day
 - Print days

January

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Sixth Refinement

- ❑ Read a year
- ❑ Print the calendar of the year
 - Label the year
 - Repeat 12 times
 - Print a month
 - ✓ Label the month
 - ✓ Label Days of the week.
 - ✓ Print days
 - Determine number of days
 - Position the 1st day
 - Repeat (number of days) times
 - Print a day

				1	2	3	4
5	6	7	8	9	10	11	
12	13	14	15	16	17	18	
19	20	21	22	23	24	25	
26	27	28	29	30	31		

Why Style and Documentation are important?

- ☐ 80% of the lifetime cost of a piece of software goes to maintenance.
- ☐ Hardly any software is maintained for its whole life by the original author.
- ☐ Good style and documentation improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- ☐ Reference

“Code Conventions for the Java Programming Language”

<http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html>

What are included in style and documentation?

Style and documentation are divide into three categories:

- ☐ Naming
- ☐ Comments
- ☐ Format

Naming

- ❑ Use meaningful names.
- ❑ Make names long enough to be meaningful but short enough to avoid being wordy.
- ❑ Java naming conventions:
 - Packages: Names should be in lowercase.
 - Classes & interfaces: Names should be in CamelCase, i.e. each new word begins with a capital letter (e.g. CamelCase, CustomerAccount)
 - Methods & variables: Names should be in mixed case, i.e. first letter of the name is in lowercase (e.g. hasChildren, customerFirstName).
 - Constants: Names should be in uppercase, e.g. MAX_HEIGHT

Comments

There are four styles of comments:

1. **Block comments** are used to provide descriptions of files, methods, data structures and programs. Example:

```
/*  
 * Lab number: 1  
 * Your name:  
 * Section number: 4  
 */
```

2. Short comments can appear on a single line indented to the level of the code that follows.

```
if (condition) {  
    /* Handle the condition. */  
    ...  
}
```


Comments conti.

- 3. Trailing Comments:** Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements.

```
if (a == 2) {  
    return true;           /* special case */  
} else {  
    return isPrime(a);     /* works only for odd number a */  
}
```

- 4. End-Of-Line Comments:** The // comment delimiter can comment out a complete line or only a partial line.

```
if (foo > 1) {  
    // Do a double-flip.  
    ...  
} else {  
    return false; // Explain why here.  
}
```

Format

- ☐ To make the logical organization of the code stand out.
- ☐ To ensure that the source code is formatted in a consistent, logical manner.

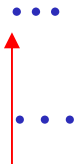
Indentation

- ☐ Establish a standard size for an indent, such as four spaces, and use it consistently.
- ☐ Align sections of code using the prescribed indentation.

Example: loops

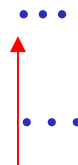
```
for (i = 0; i < 100; i++) {
```

```
    ...  
    ...  
}
```




```
for (i = 0; i < 100; i++)  
{
```

```
    ...  
    ...  
}
```




Example: if statements

```
if ( condition ) {  
    statements;  
} else {  
    statements;  
}
```

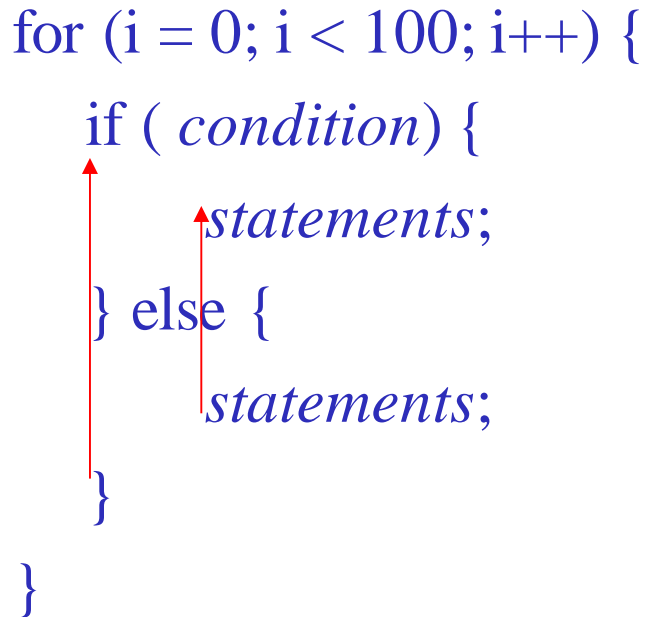


```
if ( condition )  
{  
    statements;  
}  
else  
{  
    statements;  
}
```

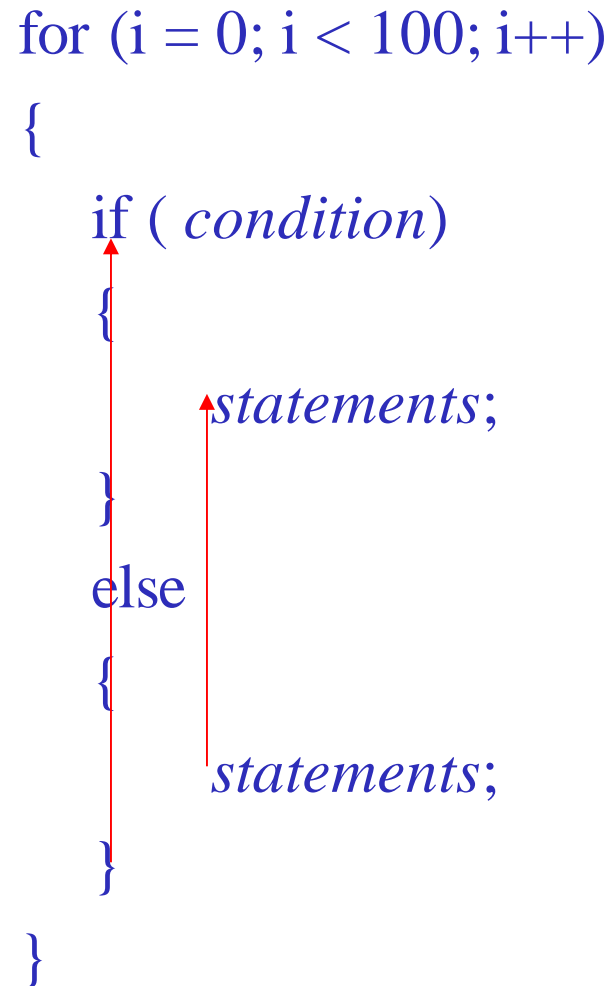


Example: nested statements

```
for (i = 0; i < 100; i++) {  
    if ( condition) {  
        ↑  
        statements;  
    } else {  
        ↑  
        statements;  
    }  
}
```



```
for (i = 0; i < 100; i++)  
{  
    if ( condition)  
    {  
        ↑  
        statements;  
    }  
    else  
    {  
        ↑  
        statements;  
    }  
}
```



Coding optimization

- Code optimization is any method of code modification to improve code quality and efficiency.
- A program may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations.
- Sometimes, these are tradeoffs (i.e. performance vs. readability)
- Types:
 - Intermediate code level
 - We are looking at this part now
 - Machine code level
 - Instruction selection, register allocation, etc.

Coding optimization (cont)

- ❑ Donald Knuth, *premature optimization is the root of all evil*
 - Optimization can introduce new, subtle bugs
 - Optimization usually makes code harder to understand and maintain
- ❑ Get your code right first, then, if really needed, optimize it
 - Document optimizations carefully
 - Keep the non-optimized version handy, or even as a comment in your code

The 80/20 rules

- ❑ In general, *80% percent of a program's execution time is spent executing 20% of the code.*
 - ❑ This means that a small part of the code is running most of the time, and the bigger part of the code is running seldom.
- ❑ 90%/10% for performance-hungry programs.
 - ❑ 90 percent of a program's execution time is spent running 10 percent of the code.
- ❑ Spend your time optimizing the important 10/20% of your program.
- ❑ Optimize the common case even at the cost of making the uncommon case slower.

General optimization techniques

❑ Strength reduction

- Use the faster and cheaper version of an operation

- *E.g.*


`x >> 2` *instead of* `x / 4` // Note: readability issue here!
`x << 1` *instead of* `x * 2`

❑ Common sub expression elimination

- Reuse results that are already computed and store them for use later, instead of re-computing them.

- *E.g.*

```
double x = d * (limit / max) * sx;  
double y = d * (limit / max) * sy;
```

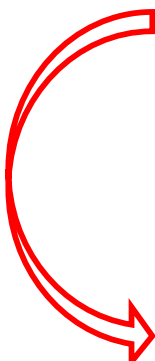


```
double depth = d * (limit / max);  
double x = depth * sx;  
double y = depth * sy;
```

General optimization techniques conti.

□ Code motion

- *Invariant* expressions should be executed only once
- *E.g.*



```
for (int i = 0; i < x.length; i++)  
    x[i] *= Math.PI * Math.cos(y);
```

```
double picosy = Math.PI * Math.cos(y);  
for (int i = 0; i < x.length; i++)  
    x[i] *= picosy;
```

General optimization techniques conti.

❑ Eliminate unnecessary loads and stores

`x = y + 5;`

`z = y;`

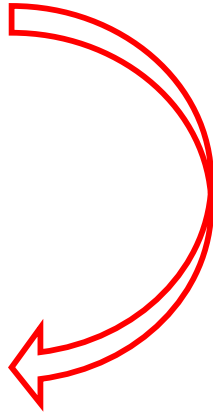
`w = z;`

`u = w * 3;`

`x = y + 5;`

`z = y;`

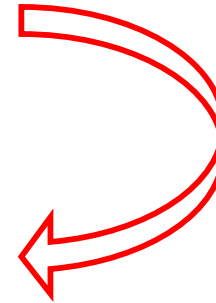
`u = z * 3;`



`x = 5;`

`z = y+x;`

`z = y+5;`



Eliminate unneeded assignment statement! (middleman)

General optimization techniques_{conti.}

❑ Factoring out invariants

- If an expression is carried out **both** when a condition is met and is not met, it can be written just once outside of the conditional statement.

- *E.g.*

```
if (condition) {  
    do A;  
    do B;  
    do C;  
} else {  
    do A;  
    do D;  
    do C;  
}
```



```
do A;  
if (condition) {  
    do B;  
} else {  
    do D;  
}  
do C;
```

Lab 3 Important Notes

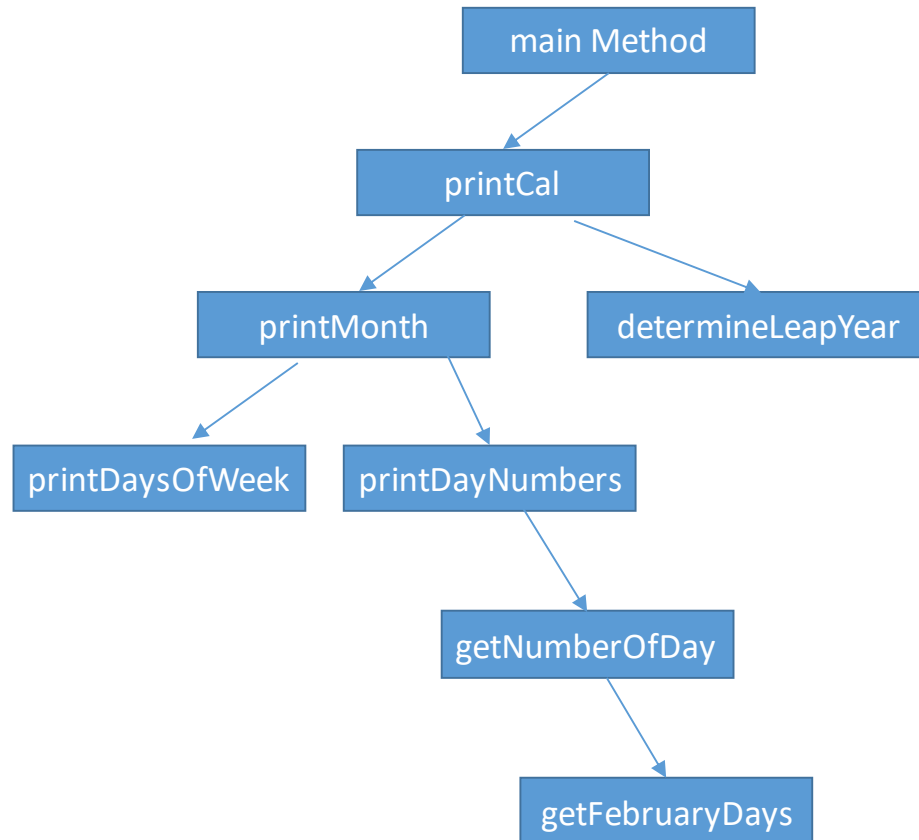
(Your Lab Instructors will provide more context in the lab sections)

- ☐ Program Structure (1-6 steps)
- ☐ Getting input year into main methods
- ☐ Compute leap year
- ☐ Position to first day of the month
- ☐ Some recommendations

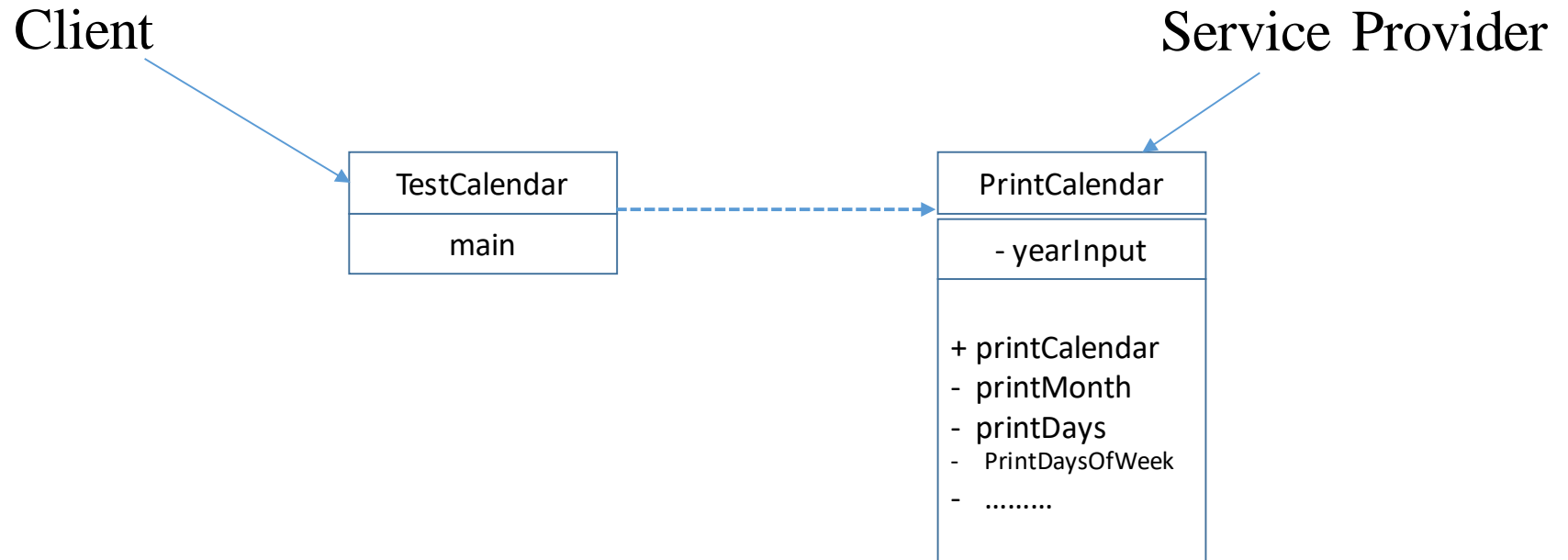
Print Calendar - Sixth Refinement

- ☐ Read a year
- ☐ Print the calendar of the year
 - Label the year
 - Repeat 12 times
 - Print a month
 - ✓ Label the month
 - ✓ Label Days of the week.
 - ✓ Print days
 - Determine number of days
 - Position the 1st day
 - Repeat (number of days) times
 - Print a day

Print Calendar – Call Sequencing (Example Only)



Class Organization (Example Only)



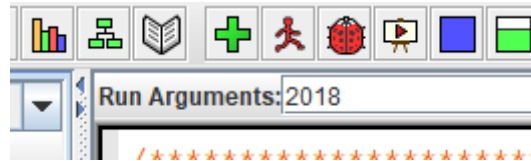
Can you define the responsibilities of each class ?

Getting input year into main method

- ❑ Recall main method: `void main(String[] args)`
- ❑ Check `args.length == 0` ?
 - If true (no user input), create a Calendar object with a year with System's current year
 - Else, create a Calendar with user's input year
- ❑ Get System's current year:
 - `Calendar.getInstance().get(Calendar.YEAR)`
(note: `import java.util.Calendar;`)
- ❑ Get User Input year:
 - `Integer.parseInt(args[0])`

Getting input year into main method (Cont)

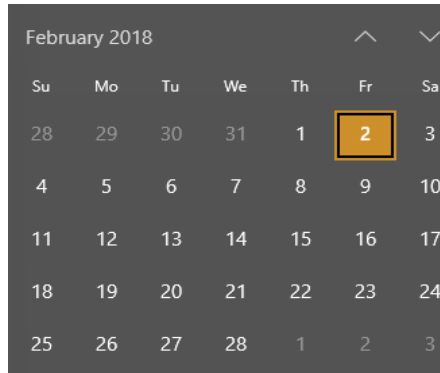
- ❑ In Jgrasp, go to menu -> Build -> Run Argument .
- ❑ Then you should see an input box:



Compute Leap year

- ❑ Every year that is exactly divisible by four is a leap year, **except** for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400. For example, the years 1700, 1800, and 1900 were not leap years, but the years 1600 and 2000 were.
- ❑ Source: https://en.wikipedia.org/wiki/Leap_year

❑ This year:



Su	Mo	Tu	We	Th	Fr	Sa
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	1	2	3

❑ Next leap year:

Saturday, February 29

Leap Day 2020

Compute Leap year (Cont)

❑ Algorithms:

- ❑ if (year is not divisible by 4)
 - then (it is a common year)
- ❑ else if (year is **not** divisible by 100)
 - then (it is a leap year)
- ❑ else if (year is **not** divisible by 400)
 - then (it is a common year)
- ❑ else
 - (it is a leap year)

Compute first date of a month

- ❑ Computing first date of a month.

```
// Determine number of days  
int date = jd.toJulian(year, month, 1);  
int dayOfWeek = (date+1)%7; // 0 means Sunday, 1 means Monday, etc.
```

The screenshot shows a Java IDE window with tabs for Messages, jGRASP Messages, Run I/O, and Interactions. The main area displays a calendar for January 2015. The days of the week are listed as S, M, Tu, W, Th, F, S. The first day of the month, January 1st, is highlighted with a blue box, and a black arrow points to it. To the right of the calendar is a clock showing the time as 1:20:13 PM.

Ex: `System.out.printf("%" + (dayOfWeek*3) + "c", ' ');`

- ❑ Please be sure to place the provided Julian class file in the same directory with your program.

Lab 3 Recommendations

- ❑ Learn to use **correct indentation.**
 - ❑ Part of your lab + grade + future work
- ❑ Learn to decompose a problem into sub-problems
- ❑ Learn to isolate syntax error(s) and correct them.
 - ❑ Use your debugger to step into program/method.
 - ❑ Error resolved by falling back to what we know worked before!
- ❑ Piecemeal approach
 - ❑ Not all at once, work on individual simple parts and construct more complex parts on top.

Lab 3 Demo Scenarios

- ☐ Using system year
- ☐ Using user input year
- ☐ Leap year (2020)
- ☐ First date of the month