List of Slides

1 Title
2 **Chapter 8:** Separate methods and logical operators
3 Chapter aims
4 **Section 2:** Example:Age history with two people
5 Aim
6 Age history with two people
11 Trying it
12 Coursework: `WorkFuture2`
13 **Section 3:** Example:Age history with a separate method
14 Aim
15 Method
16 Method: private
17 Age history with a separate method
18 Method: accepting parameters
22 Age history with a separate method
23 Method: calling a method

26 Age history with a separate method
27 Method: void methods
28 Age history with a separate method
29 Age history with a separate method
31 Age history with a separate method
33 Trying it
34 Warning: do not forget **`static`**
35 Warning: do not forget **`static`**
36 Coursework: `WorkFuture4`
37 **Section 4:** Example:Dividing a cake with a separate method for GCD
38 Aim
39 Method: returning a value
42 Dividing a cake with a separate method for GCD
46 Dividing a cake with a separate method for GCD
47 Method: changing parameters does not affect arguments
48 Changing values of method parameters
49 Coursework: `DivideCake4`
50 **Section 5:** Example:Multiple times table with separate methods

51 Aim
52 Multiple times table with separate methods
53 Variable: local variables
54 Variable: class variables
56 Multiple times table with separate methods
57 Multiple times table with separate methods
59 Multiple times table with separate methods
60 Multiple times table with separate methods
61 Multiple times table with separate methods
62 Standard API: `System: out.printf()`
64 Multiple times table with separate methods
65 Trying it
66 Coursework: `CommonFactorsTable` with methods
67 **Section 6:** Example:Age history with day and month
68 Aim
69 Age history with day and month
70 Expression: boolean: logical operators
76 Age history with day and month

77 Variable: a group of variables can be declared together
78 Age history with day and month
81 Age history with day and month
82 Age history with day and month
84 Trying it
85 Trying it
86 Trying it
87 Coursework: Reasoning about conditions
88 **Section 7:** Example:Truth tables
89 Aim
90 Truth tables
91 Truth tables
92 Type: `boolean`
93 Truth tables
94 Variable: `boolean` variable
99 Truth tables
100 Truth tables
104 Type: `String`

105 Truth tables
106 Truth tables
107 Statement: for loop: multiple statements in for update
109 Truth tables
111 Truth tables
112 Coursework: `TruthTable34`
113 **Section 8:** Example:Producing a calendar
114 Aim
115 Producing a calendar
116 Producing a calendar
117 Producing a calendar
121 Producing a calendar
122 Producing a calendar
123 Producing a calendar
124 Producing a calendar
125 Standard API: `System`: `out.printf()`: zero padding
126 Producing a calendar
127 Trying it

128 Coursework: `CalendarHighlight`
129 Concepts covered in this chapter

Java Just in Time

John Latham

October 26, 2018

Chapter 8

Separate methods and logical operators

Chapter aims

- Time to stop putting all code in **main method**
  - might want to reuse certain parts instead of copying
    - e.g. in multiple times table...
  - wish to split up big programs into separate, manageable, parts.
- Also meet **logical operator**s
- and some more Java concepts.

Section 2

Example:
Age history with two people

Aim

AIM: To further illustrate the inconvenience of having to copy a chunk of code which is used in different parts of a program, and thus motivate the need for separate **method**s.

Age history with two people

```
001: // Print out an age history of two people.
002: // Arguments: present year, first birth year, second birth year.
003: public class AgeHistory2
004: {
005:   public static void main(String[] args)
006:   {
007:     // The year of the present day.
008:     int presentYear = Integer.parseInt(args[0]);
009:
010:     // The two birth years, which must be less than the present year.
011:     int birthYear1 = Integer.parseInt(args[1]);
012:     int birthYear2 = Integer.parseInt(args[2]);
013:
```

Age history with two people

```
014:    // PERSON 1
015:    // Start by printing the event of birth.
016:    System.out.println("Pn 1 was born in " + birthYear1);
017:
018:    // Now we will go through the years between birth and last year.
019:    int someYear1 = birthYear1 + 1;
020:    int ageInSomeYear1 = 1;
021:    while (someYear1 != presentYear)
022:    {
023:      System.out.println("Pn 1 was " + ageInSomeYear1 + " in " + someYear1);
024:      someYear1++;
025:      ageInSomeYear1++;
026:    } // while
027:
```

Age history with two people

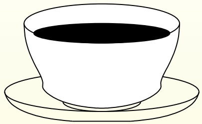
```
028:      // Finally, the age of the person this year.
029:      System.out.println("Pn 1 is " + ageInSomeYear1 + " this year");
030:
```

Age history with two people

```
031:    // PERSON 2
032:    // Start by printing the event of birth.
033:    System.out.println("Pn 2 was born in " + birthYear2);
034:
035:    // Now we will go through the years between birth and last year.
036:    int someYear2 = birthYear2 + 1;
037:    int ageInSomeYear2 = 1;
038:    while (someYear2 != presentYear)
039:    {
040:      System.out.println("Pn 2 was " + ageInSomeYear2 + " in " + someYear2);
041:      someYear2++;
042:      ageInSomeYear2++;
043:    } // while
044:
```

Age history with two people

```
045:     // Finally, the age of the person this year.
046:     System.out.println("Pn 2 is " + ageInSomeYear2 + " this year");
047:   } // main
048:
049: } // class AgeHistory2
```



Coffee time: While this approach works, what are the problems with it? E.g., could we be careless with our editing? What if we wanted to make it work for 10 people?

Trying it

Console Input / Output

```
$ java AgeHistory2 2019 2000 1989
Pn 1 was born in 2000
Pn 1 was 1 in 2001
Pn 1 was 2 in 2002
 (... lines removed to save space.)
Pn 1 was 18 in 2018
Pn 1 is 19 this year
Pn 2 was born in 1989
Pn 2 was 1 in 1990
Pn 2 was 2 in 1991
 (... lines removed to save space.)
Pn 2 was 29 in 2018
Pn 2 is 30 this year
$ _
```

Run

Coursework: `WorkFuture2`

(Summary only)

Write a program to print out all the years from the present day until retirement, for two people.

Section 3

Example: Age history with a separate method

Aim

AIM: To introduce the idea of dividing a program into separate **method**s to enable the reuse of some parts of it. We meet the concepts **private**, **method parameter**, **method argument**, **method call** and **void method**.

Method

- A **method** – section of code for performing particular task.
- Programs have **main method**.
- Can have other methods – any name we like
  - which suits the purpose – describes what it does.
- Convention:
  - method names start with lower case letter
  - first letter of subsequent words capitalized.

Method: private

- Can have **method** with **private** visibility **modifier**.
  - Should be private if not intended to be usable outside defining **class**.
- Use **reserved word** **`private`** instead of **`public`**.

THE UNIVERSITY of MANCHESTER

Age history with a separate method

- Our separate method heading, so far:

```
private static ... printAgeHistory ...
```

Method: accepting parameters

- A **method** may have **method parameter**s
  - enable variation of effect based on given values.
- Similar to same idea with program **command line argument**s
  - indeed: those are passed as parameter to **main method**.
- Parameters declares in method heading, in brackets after name.
- E.g.

```
public static void main(String[] args)
```

- Can have zero or more parameters
  - separated by commas (,)
  - each has **type** and name.

Method: accepting parameters

- E.g.

```
private static void printHeightPerYear(double height, int age)
{
  System.out.println("At age " + age + ", height per year ratio is "
                     + height / age);
} // printHeightPerYear
```

- Parameters are like **variable**s declared inside method
  - but given initial values before method body **execute**d.
- E.g. `String[] args` on main is variable
  - already given list of string command line arguments.

Method: accepting parameters

- Parameter names not important to Java – except must be different.
- But should be meaningful to human reader.
- E.g.

```
private static void printHeightPerYear(double howTall, int howOld)
{
  System.out.println("At age " + howOld + ", height per year ratio is "
                     + howTall / howOld);
} // printHeightPerYear
```

- First or second better? Subjective.

Method: accepting parameters

- So what about this version?

```
private static void printHeightPerYear(double d, int i)
{
  System.out.println("At age " + i + ", height per year ratio is "
                     + d / i);
} // printHeightPerYear
```

- Hardly better than using `x` and `y`.
- Java too dumb to have understanding of problem
  - so it cannot care
  - but we must – or are we as dumb? ;-)

Age history with a separate method

- Our separate method heading, so far:

```
private static ... printAgeHistory(int presentYear,
                                   int personNumber, int birthYear)
```

Method: calling a method

- Body of **method execute**d when some other code has **method call**.
- E.g. `System.out.println("Hello world!")`.
- E.g. assume `printHeightPerYear`

```
printHeightPerYear(1.6, 14);
```

- We supply **method argument** for each **method parameter**
  - separated by commas (,).
- How does it know which value is age and which is height?
  - associated by order:
    - first argument goes into first parameter,
    - second into second, ...

Method: calling a method

- Arguments may be current values of **variable**s.
- E.g.

```
double personHeight = 1.6;
int personAge = 14;

printHeightPerYear(personHeight, personAge);
```

Method: calling a method

- In fact, arguments are **expressions**
  - get **evaluated** when method is called.
- E.g.

```
double growthLastYear = 0.02;

printHeightPerYear(personHeight - growthLastYear, personAge - 1);
```

Age history with a separate method

- We have four method calls in main method:

```
...
printAgeHistory(presentYear, 1, birthYear1);
printAgeHistory(presentYear, 2, birthYear2);
printAgeHistory(presentYear, 3, birthYear3);
printAgeHistory(presentYear, 4, birthYear4);
```

Method: void methods

- A **method** might calculate a result
  - perhaps based on **method parameter**s

  and **return** that answer.
- Might be **`int`**, **`double`** or some other **type**.
- If method returns result then write **return type** in heading.
- If not write **`void`** – meaning ‘without contents’.
- E.g. **main method** does not return a result – it is a **void method**.

```
public static void main(String[] args)
```

Age history with a separate method

- Our separate method heading:

```
private static void printAgeHistory(int presentYear,
                                    int personNumber, int birthYear)
```

Age history with a separate method

```
001: // Print out an age history of four people.
002: // Arguments: present year, first birth year, second, third, fourth.
003: public class AgeHistory4
004: {
005:   // Print the age history of one person, identified as personNumber.
006:   // Birth year must be less than present year.
007:   private static void printAgeHistory(int presentYear,
008:                                       int personNumber, int birthYear)
009:   {
010:     // Start by printing the event of birth.
011:     System.out.println("Pn " + personNumber + " was born in " + birthYear);
012:
013:     // Now we will go through the years between birth and last year.
014:     int someYear = birthYear + 1;
015:     int ageInSomeYear = 1;
```

Age history with a separate method

```
016:      while (someYear != presentYear)
017:      {
018:        System.out.println("Pn " + personNumber + " was "
019:                           + ageInSomeYear + " in " + someYear);
020:        someYear++;
021:        ageInSomeYear++;
022:      } // while
023:
024:      // Finally, the age of the person this year.
025:      System.out.println("Pn " + personNumber + " is "
026:                         + ageInSomeYear + " this year");
027:    } // printAgeHistory
```

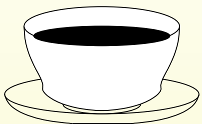
- Next comes main method
  - order does not matter to Java.

Age history with a separate method

```
030:  // The main method: get arguments and call printAgeHistory.
031:  public static void main(String[] args)
032:  {
033:    // The year of the present day.
034:    int presentYear = Integer.parseInt(args[0]);
035:
036:    // The four birth years, which must be less than the present year.
037:    int birthYear1 = Integer.parseInt(args[1]);
038:    int birthYear2 = Integer.parseInt(args[2]);
039:    int birthYear3 = Integer.parseInt(args[3]);
040:    int birthYear4 = Integer.parseInt(args[4]);
041:
```

Age history with a separate method

```
042:     // Now print the four age histories.
043:     printAgeHistory(presentYear, 1, birthYear1);
044:     printAgeHistory(presentYear, 2, birthYear2);
045:     printAgeHistory(presentYear, 3, birthYear3);
046:     printAgeHistory(presentYear, 4, birthYear4);
047:   } // main
048:
049: } // class AgeHistory4
```



Coffee time: Why did we need to write the **reserved word** **`static`** in the heading of `printAgeHistory()`? What do you think would happen if we omitted it?

Trying it

Console Input / Output

```
$ java AgeHistory4 2019 2000 1989 1959 2018
Pn 1 was born in 2000
Pn 1 was 1 in 2001
 (... lines removed to save space.)
Pn 1 is 19 this year
Pn 2 was born in 1989
Pn 2 was 1 in 1990
Pn 2 was 2 in 1991
 (... lines removed to save space.)
Pn 2 is 30 this year
Pn 3 was born in 1959
Pn 3 was 1 in 1960
 (... lines removed to save space.)
Pn 3 is 60 this year
Pn 4 was born in 2018
Pn 4 is 1 this year
$ _
```

Run

Warning: do not forget `static`

- What happens if omit **`static`**?
  - You'll do that at some point. . . .
- Here is compiling `AgeHistoryOops` – same as `AgeHistory4` except no **`static`**.

Warning: do not forget `static`

Console Input / Output

```
$ javac AgeHistoryOops.java
AgeHistoryOops.java:43: non-static method printAgeHistory(int,int,int) cannot be
 referenced from a static context
    printAgeHistory(presentYear, 1, birthYear1);
    ^
AgeHistoryOops.java:44: non-static method printAgeHistory(int,int,int) cannot be
 referenced from a static context
    printAgeHistory(presentYear, 2, birthYear2);
    ^
AgeHistoryOops.java:45: non-static method printAgeHistory(int,int,int) cannot be
 referenced from a static context
    printAgeHistory(presentYear, 3, birthYear3);
    ^
AgeHistoryOops.java:46: non-static method printAgeHistory(int,int,int) cannot be
 referenced from a static context
    printAgeHistory(presentYear, 4, birthYear4);
    ^
4 errors
$ _
```

Run

Coursework: `WorkFuture4`

(Summary only)

Write a program, with a separate **method**, to print out all the years from the present day until retirement, for four people.

Section 4

Example: Dividing a cake with a separate method for GCD

Aim

AIM: To introduce the idea of using **method**s merely to split the program into parts, making it easier to understand and develop. We also meet the **return statement** for use in **non-void method**s, and see that altering a **method parameter** does not change its argument.

Method: returning a value

- A **method** can **return** a result
  - we declare **return type** in heading (intead of **`void`**).
- Often called **non-void methods**.
- E.g. return corresponding Fahrenheit for given Celsius.

```
private static double celsiusToFahrenheit(double celsiusValue)
{
  double fahrenheitValue = celsiusValue * 9 / 5 + 32;
  return fahrenheitValue;
} // celsiusToFahrenheit
```

- Method declared with return type **`double`**.
- The **return statement** specifies value to be returned
  - causes execution control to go back to after method call.

Method: returning a value

- Result of non-void method can be used in **expression**s.
- E.g.

```
double celsiusValue = Double.parseDouble(args[0]);
System.out.println("The Fahrenheit value of "
                   + celsiusValue + " Celsius is "
                   + celsiusToFahrenheit(celsiusValue) + ".");
```

Method: returning a value

- Return statement can have any expression – not just a variable.
- E.g.

```
private static double celsiusToFahrenheit(double celsiusValue)
{
  return celsiusValue * 9 / 5 + 32;
} // celsiusToFahrenheit
```

Dividing a cake with a separate method for GCD

```
001: // Program to decide how to divide a cake in proportion to the age of two
002: // persons, using the minimum number of equal sized portions.
003: // The two arguments are the two positive integer ages.
004: public class DivideCake
005: {
006:   // Find the GCD of two positive integers.
007:   private static int greatestCommonDivisor(int multiple1OfGCD,
008:                                            int multiple2OfGCD)
009:   {
010:     // Both multiple1OfGCD and multiple2OfGCD must be positive.
011:     // While the two multiples are not the same, the difference
012:     // between them must also be a multiple of the GCD.
013:     // So we keep subtracting the smallest from the largest
014:     // until they are equal.
```

Dividing a cake with a separate method for GCD

```
015:     while (multiple1OfGCD != multiple2OfGCD)
016:       if (multiple1OfGCD > multiple2OfGCD)
017:         multiple1OfGCD -= multiple2OfGCD;
018:       else
019:         multiple2OfGCD -= multiple1OfGCD;
020:
021:     // Now multiple1OfGCD == multiple2OfGCD
022:     // which is also the GCD of their original values.
023:     return multiple1OfGCD;
024:   } // greatestCommonDivisor
025:
026:
```

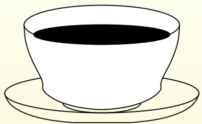
Dividing a cake with a separate method for GCD

```
027:  // Obtain arguments, get GCD, compute portions and report it all.
028:  public static void main(String[] args)
029:  {
030:    // Both ages must be positive.
031:    int age1 = Integer.parseInt(args[0]);
032:    int age2 = Integer.parseInt(args[1]);
033:
034:    int agesGCD = greatestCommonDivisor(age1, age2);
035:    System.out.println("The GCD of " + age1 + " and " + age2
036:                       + " is " + agesGCD);
037:    int noOfPortions1 = age1 / agesGCD;
038:    int noOfPortions2 = age2 / agesGCD;
039:
```

Dividing a cake with a separate method for GCD

```
040:     System.out.println("So the cake should be divided into "
041:                        + (noOfPortions1 + noOfPortions2));
042:     System.out.println
043:       ("The " + age1 + " year old gets " + noOfPortions1
044:        + " and the " + age2 + " year old gets " + noOfPortions2);
045:   } // main
046:
047: } // class DivideCake
```

Dividing a cake with a separate method for GCD



Coffee time: Did you notice that inside the `greatestCommonDivisor()` method, the code changes the values of both `multiple1OfGCD` and `multiple2OfGCD`? These start off as being the ages of the two people, but end up being the GCD of the two ages. Then, after the method has finished executing, the `main()` method prints out the values of `age1` and `age2` in its message.

So, will `age1` and `age2` have had their value changed, causing the program to wrongly report both ages as being the GCD of the original values?

Method: changing parameters does not affect arguments

- A **method parameter** is just like **variable** defined inside **method**
  - except given initial value by method call.
- Method body can change the value – it is a variable
  - changes do not affect where initial value came from.
- Known as **call by value**
  - **method argument** is some **expression**
    - *value* is copied into parameter at method call.

Changing values of method parameters

- So, when:

```
int agesGCD = greatestCommonDivisor(age1, age2);
```

  - values of `age1` and `age2` are *copied* to **method parameters** `multiple1OfGCD` and `multiple2OfGCD`
  - those parameters are changed within the method
    - but no effect on `age1` and `age2`.

Coursework: `DivideCake4`

(Summary only)

Write a program to compute the **greatest common divisor** of *four* numbers, using a separate **method**.

Section 5

Example: Multiple times table with separate methods

Aim

AIM: To introduce the concept of **class variable**s, compared with **local variable**s, and reinforce the ideas of using separate **method**s for reuse and for dividing a program into manageable chunks. We also meet `System.out.printf()`.

Multiple times table with separate methods

- Improve multiple times table program
  - split into separate methods
    - avoid duplicated code
    - make more readable
  - improve flexibility – have size of table in a **variable**
    - easy to change if requirements change.

Variable: local variables

- All **variable**s declared inside **method** – local to that method
  - only exist while method is running
  - cannot be accessed by other methods.
- Known as **local variable**s or **method variable**s.
- Different methods can have variables with same name.

Variable: class variables

- We can declare **variable**s inside a **class**
  - outside of any **method**s.
  - called **class variable**s
  - exist from when class is loaded into **virtual machine**
  - can be accessed by any method in that class.

Variable: class variables

- E.g., perhaps store components of today's date:

```
private static int presentDay;
private static int presentMonth;
private static int presentYear;
```

- Observe **reserved word** `static`
  - they are part of the **static context** memory allocation.
- Also visibility **modifier**
  - if **private** can only be accessed by code inside that class.

Multiple times table with separate methods

```
001: // Program to print out a neat multiplication table.
002: public class TimesTable
003: {
004:   // The size of the table -- the number of rows and columns.
005:   private static int tableSize = 12;
```

- Main method calls several separate methods.
- Some directly access class variable `tableSize`.

THE UNIVERSITY of MANCHESTER

Multiple times table with separate methods

```
008: // The main method implements the top level structure of the table.
009: public static void main(String[] args)
010: {
011:   // Top line.
012:   printLine();
013:
014:   // Column headings.
015:   printColumnHeadings();
016:
017:   // Underline headings.
018:   printLine();
019:
020:   // Now the rows.
021:   for (int row = 1; row <= tableSize; row++)
022:     printRow(row);
023:
```

Multiple times table with separate methods

```
024:     // Bottom line.
025:     printLine();
026:   } // main
```

- Separate method to print a line – accesses `tableSize`.

```
029:   // Print a line across the table.
030:   private static void printLine()
031:   {
032:     // Left side, 5 characters for row labels, separator.
033:     System.out.print("|-----|");
034:     // Across each column.
035:     for (int column = 1; column <= tableSize; column++)
036:       System.out.print("----");
037:     // The right side.
038:     System.out.println("-|");
039:   } // printLine
```

Multiple times table with separate methods

```
042:  // Print the line containing the column headings.
043:  private static void printColumnHeadings()
044:  {
045:    System.out.print("|     |");
046:    for (int column = 1; column <= tableSize; column++)
047:      printNumber(column);
048:    System.out.println(" |");
049:  } // printColumnHeadings
```

Multiple times table with separate methods

```
052:  // Print one row of the table.
053:  private static void printRow(int row)
054:  {
055:    // The left side.
056:    System.out.print("|");
057:    printNumber(row);
058:    // Separator.
059:    System.out.print(" |");
060:
061:    // Now the columns on this row.
062:    for (int column = 1; column <= tableSize; column++)
063:      printNumber(row * column);
064:
065:    // The right side.
066:    System.out.println(" |");
067:  } // printRow
```

Multiple times table with separate methods

- Printing number in style previously used:
  - write once, use in three places.

```
private static void printNumber(int numberToPrint)
{
  if (numberToPrint < 10)
    System.out.print("   " + numberToPrint);
  else if (numberToPrint < 100)
    System.out.print("  " + numberToPrint);
  else
    System.out.print(" " + numberToPrint);
} // printNumber
```

- But still seems a lot of work! Simpler way?

Standard API: `System`: `out.printf()`

- Since Java 5.0, `System` contains **method** `out.printf()`
  - similar to `out.print()`, but produces formatted output.
- E.g. print **integer** with **space padding** to given field width
  - output with leading spaces so at least field width **character**s.

```
System.out.println("1234567890");
System.out.printf("%10d%n", 123);
```

  - produces:

```
1234567890
       123
```

  - `%` – wish to format something
  - `10` – minimum total field width
  - `d` – please format decimal whole number following the **format specifier**.
  - `%n` – output platform dependent **line separator**.

Standard API: `System: out.printf()`

- Can format floating point value, e.g. a **`double`**.

```
System.out.printf("%1.2f%n", 123.456);
```

  - `1` – minimum total field width
  - `.2` – number of decimal places
  - `f` – conversion code for floating point value
  - output needs more than minimum width:

```
123.46
```

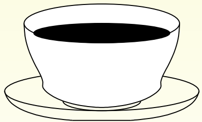
- Whereas:

```
System.out.println("1234567890");
System.out.printf("%10.2f%n", 123.456);
```

  - produces:

```
1234567890
    123.46
```

Multiple times table with separate methods



Coffee time: Are you tempted to pop back to previous example programs and improve their output using `System.out.printf()`?

- Observe no `%n` in **format specifier**.

```
070:  // Print a number using exactly 4 characters, with leading spaces.
071:  private static void printNumber(int numberToPrint)
072:  {
073:    System.out.printf("%4d", numberToPrint);
074:  } // printNumber
075:
076: } // class TimesTable
```

Trying it

Console Input / Output

```
$ java TimesTable
|-----|------------------------------------------------|
|     |   1   2   3   4   5   6   7   8   9  10  11  12 |
|-----|------------------------------------------------|
|   1 |   1   2   3   4   5   6   7   8   9  10  11  12 |
|   2 |   2   4   6   8  10  12  14  16  18  20  22  24 |
|   3 |   3   6   9  12  15  18  21  24  27  30  33  36 |
|   4 |   4   8  12  16  20  24  28  32  36  40  44  48 |
|   5 |   5  10  15  20  25  30  35  40  45  50  55  60 |
|   6 |   6  12  18  24  30  36  42  48  54  60  66  72 |
|   7 |   7  14  21  28  35  42  49  56  63  70  77  84 |
|   8 |   8  16  24  32  40  48  56  64  72  80  88  96 |
|   9 |   9  18  27  36  45  54  63  72  81  90  99 108 |
|  10 |  10  20  30  40  50  60  70  80  90 100 110 120 |
|  11 |  11  22  33  44  55  66  77  88  99 110 121 132 |
|  12 |  12  24  36  48  60  72  84  96 108 120 132 144 |
|-----|------------------------------------------------|
$ _
```

Run

Coursework: `CommonFactorsTable` with methods

(Summary only)

Write a program, with separate **methods**, to produce a table showing pairs of numbers which share common factors.

Section 6

Example: Age history with day and month

Aim

AIM: To introduce the **logical operators**. We also see that a group of **variables** can be declared together.

Age history with day and month

- Adding day and month to age history
  - means comparing dates based on three values
  - **loop condition** complexity explosion!
    - Surprising?

Expression: boolean: logical operators

- Need more complex **conditions** than just **relational operators**.
  - Use **logical operators** to glue simple conditions into bigger ones.
  - Most commonly used: **conditional and**, **conditional or** and **logical not**.

Expression: boolean: logical operators

Operator	Title	Posh title	Description
`&&`	and	**conjunction**	`c1 && c2` is **`true`** if and only if both conditions `c1` and `c2` **evaluate** to **`true`**. Both of the two conditions, known as **conjunct**s, must be **`true`** to satisfy the combined condition.
`||`	or	**disjunction**	`c1 || c2` is **`true`** if and only if at least one of the conditions `c1` and `c2` evaluate to **`true`**. The combined condition is satisfied, unless both of the two conditions, known as **disjunct**s, are **`false`**.
`!`	not	**negation**	`!c` is **`true`** if and only if the condition `c` evaluates to **`false`**. This operator negates the given condition.

Expression: boolean: logical operators

- Can define using **truth tables**
  - ? means the **operand** is not evaluated.

c1	c2	c1 && c2
true	**true**	**true**
true	**false**	**false**
false	?	**false**

c1	c2	c1 || c2
true	?	**true**
false	**true**	**true**
false	**false**	**false**

c	!c
true	**false**
false	**true**

Expression: boolean: logical operators

- E.g.

```
age1 < age2 || age1 == age2 && height1 <= height2
```

- What about **operator precedence** and **operator associativity**?
  - `&&` and `||` lower precedence than relational operators
  - relational operators lower precedence than arithmetic operators
  - `!` has very high precedence
  - `&&` higher precedence than `||`.
- Implicit brackets:

```
(age1 < age2) || ((age1 == age2) && (height1 <= height2))
```

- E.g. **sort**ing people by age then height. . .

Expression: boolean: logical operators

```
if (age1 < age2 || age1 == age2 && height1 <= height2)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");
```

- Less clearly?

```
if (!(age1 < age2 || age1 == age2 && height1 <= height2))
  System.out.println("Please swap over.");
else
  System.out.println("You are in the correct order.");
```

- Another way – same effect (convince yourself!).

```
if (age1 > age2 || age1 == age2 && height1 > height2)
  System.out.println("Please swap over.");
else
  System.out.println("You are in the correct order.");
```

Expression: boolean: logical operators

- In maths: $x \leq y \leq z$
  - in Java: `x <= y && y <= z`
- In English: “my mother’s age is 46 or 47”
  - in Java: `myMumAge == 46 || myMumAge == 47`
- In English: sometimes say “and” when really mean “or”:
  - “the two possible ages for my dad are 49 *and* 53”
    * “my dad’s age is 49 *or* my dad’s age is 53”.

Age history with day and month

- Previously `printAgeHistory()` had three **method parameter**s,
  - present year, the person number, birth year.
- Now, dates need three values, so perhaps seven parameters?
- No! – Present date same for all people, so store in **class variable**s.

Variable: a group of variables can be declared together

- Can declare several **variable**s of same **type** in one declaration.
- E.g.

```
int x, y;
```

- Give values too. E.g.

```
int minimumVotingAge = 18, minimumArmyAge = 16;
```

- Not as useful as might expect
  - typically have a **comment** before each variable...
- But, sometimes can have one comment for group of variables.

Age history with day and month

```
001: // Print out an age history of two people.
002: // Arguments: present date, first birth date, second birth date.
003: // Each date is three numbers: day month year.
004: public class AgeHistory2
005: {
006:   // The present date, stored as three variables.
007:   private static int presentDay, presentMonth, presentYear;
008:
009:
010:   // Print the age history of one person, identified as personNumber.
011:   // The birth date must be less than the present date.
012:   private static void printAgeHistory
013:     (int personNumber, int birthDay, int birthMonth, int birthYear)
014:   {
015:     // Start by printing the event of birth.
016:     System.out.println("Pn " + personNumber + " was born on "
017:                         + birthDay + "/" + birthMonth + "/" + birthYear);
018:
```

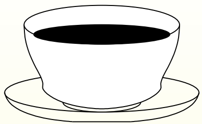
Age history with day and month

```
019:   // Now we will go through the years since birth but before today.
020:   int someYear = birthYear + 1;
021:   int ageInSomeYear = 1;
022:   while (someYear < presentYear
023:          || someYear == presentYear && birthMonth < presentMonth
024:          || someYear == presentYear && birthMonth == presentMonth
025:             && birthDay < presentDay)
026:   {
027:     System.out.println("Pn " + personNumber + " was " + ageInSomeYear
028:                        + " on " + birthDay + "/" + birthMonth
029:                        + "/" + someYear);
030:     someYear++;
031:     ageInSomeYear++;
032:   } // while
033:
```

Age history with day and month

```
034:     // At this point birthDay/birthMonth/someYear
035:     // will be the next birthday, aged ageInSomeYear.
036:     // This will be greater than or equal to the present date.
037:     // If the person has not yet had their birthday this year
038:     // someYear equals presentYear,
039:     // otherwise someYear equals presentYear + 1.
040:
041:     if (birthMonth == presentMonth && birthDay == presentDay)
042:       // then someYear must equal presentYear.
043:       System.out.println("Pn " + personNumber + " is "
044:                          + ageInSomeYear + " today!");
045:     else
046:       System.out.println("Pn " + personNumber + " will be "
047:                          + ageInSomeYear + " on " + birthDay + "/"
048:                          + birthMonth + "/" + someYear);
049:   } // printAgeHistory
```

Age history with day and month



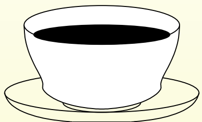
Coffee time: In the code above, did you see how the **condition** of the **while loop** has exploded with complexity, compared with the previous versions of the program that merely had one **relational operator**, i.e. `while (someYear != presentYear)`? Did this surprise you?

Age history with day and month

```
052:   // The main method: get arguments and call printAgeHistory.
053:   public static void main(String[] args)
054:   {
055:     // The present date, stored in three class variables.
056:     presentDay = Integer.parseInt(args[0]);
057:     presentMonth = Integer.parseInt(args[1]);
058:     presentYear = Integer.parseInt(args[2]);
059:
060:     // The dates of birth: these must be less than the present date.
061:     int birthDay1 = Integer.parseInt(args[3]);
062:     int birthMonth1 = Integer.parseInt(args[4]);
063:     int birthYear1 = Integer.parseInt(args[5]);
064:
```

Age history with day and month

```
065:     int birthDay2 = Integer.parseInt(args[6]);
066:     int birthMonth2 = Integer.parseInt(args[7]);
067:     int birthYear2 = Integer.parseInt(args[8]);
068:
069:     // Now print the two age histories.
070:     printAgeHistory(1, birthDay1, birthMonth1, birthYear1);
071:     printAgeHistory(2, birthDay2, birthMonth2, birthYear2);
072:   } // main
073:
074: } // class AgeHistory2
```



Coffee time: Of the nine variable assignments above, why do three of them not start with the word `int`?

Trying it

Born this day and month last year and same day 19 years ago:

Console Input / Output

```
$ java AgeHistory2 01 07 2019 01 07 2018 01 07 2000
 (Output shown using multiple columns to save space.)
Pn 1 was born on 1/7/2018
Pn 1 is 1 today!
Pn 2 was born on 1/7/2000
Pn 2 was 1 on 1/7/2001
Pn 2 was 2 on 1/7/2002
Pn 2 was 3 on 1/7/2003
Pn 2 was 4 on 1/7/2004
Pn 2 was 5 on 1/7/2005
Pn 2 was 6 on 1/7/2006
Pn 2 was 7 on 1/7/2007
Pn 2 was 8 on 1/7/2008
Pn 2 was 9 on 1/7/2009
Pn 2 was 10 on 1/7/2010
Pn 2 was 11 on 1/7/2011
Pn 2 was 12 on 1/7/2012
Pn 2 was 13 on 1/7/2013
Pn 2 was 14 on 1/7/2014
Pn 2 was 15 on 1/7/2015
Pn 2 was 16 on 1/7/2016
Pn 2 was 17 on 1/7/2017
Pn 2 was 18 on 1/7/2018
Pn 2 is 19 today!
$ _
```

Run

Born yesterday and same day 19 years ago:

Console Input / Output

```
$ java AgeHistory2 01 07 2019 30 06 2019 30 06 2000
 (Output shown using multiple columns to save space.)
Pn 1 was born on 30/6/2019
Pn 1 will be 1 on 30/6/2020
Pn 2 was born on 30/6/2000
Pn 2 was 1 on 30/6/2001
Pn 2 was 2 on 30/6/2002
Pn 2 was 3 on 30/6/2003
Pn 2 was 4 on 30/6/2004
Pn 2 was 5 on 30/6/2005
Pn 2 was 6 on 30/6/2006
Pn 2 was 7 on 30/6/2007
Pn 2 was 8 on 30/6/2008
Pn 2 was 9 on 30/6/2009
Pn 2 was 10 on 30/6/2010
Pn 2 was 11 on 30/6/2011
Pn 2 was 12 on 30/6/2012
Pn 2 was 13 on 30/6/2013
Pn 2 was 14 on 30/6/2014
Pn 2 was 15 on 30/6/2015
Pn 2 was 16 on 30/6/2016
Pn 2 was 17 on 30/6/2017
Pn 2 was 18 on 30/6/2018
Pn 2 was 19 on 30/6/2019
Pn 2 will be 20 on 30/6/2020
$ _
```

Run

Trying it

Born a year ago tomorrow and same day 19 years ago:

Console Input / Output

```
$ java AgeHistory2 01 07 2019 2 07 2018 2 07 2000
 (Output shown using multiple columns to save space.)
Pn 1 was born on 2/7/2018
Pn 1 will be 1 on 2/7/2019
Pn 2 was born on 2/7/2000
Pn 2 was 1 on 2/7/2001
Pn 2 was 2 on 2/7/2002
Pn 2 was 3 on 2/7/2003
Pn 2 was 4 on 2/7/2004
Pn 2 was 5 on 2/7/2005
Pn 2 was 6 on 2/7/2006
Pn 2 was 7 on 2/7/2007
Pn 2 was 8 on 2/7/2008
Pn 2 was 9 on 2/7/2009
Pn 2 was 10 on 2/7/2010
Pn 2 was 11 on 2/7/2011
Pn 2 was 12 on 2/7/2012
Pn 2 was 13 on 2/7/2013
Pn 2 was 14 on 2/7/2014
Pn 2 was 15 on 2/7/2015
Pn 2 was 16 on 2/7/2016
Pn 2 was 17 on 2/7/2017
Pn 2 was 18 on 2/7/2018
Pn 2 will be 19 on 2/7/2019
$ _
```

Run

Born this day last month and same day 19 years ago:

Console Input / Output

```
$ java AgeHistory2 01 07 2019 01 6 2019 01 6 2000
 (Output shown using multiple columns to save space.)
Pn 1 was born on 1/6/2019
Pn 1 will be 1 on 1/6/2020
Pn 2 was born on 1/6/2000
Pn 2 was 1 on 1/6/2001
Pn 2 was 2 on 1/6/2002
Pn 2 was 3 on 1/6/2003
Pn 2 was 4 on 1/6/2004
Pn 2 was 5 on 1/6/2005
Pn 2 was 6 on 1/6/2006
Pn 2 was 7 on 1/6/2007
Pn 2 was 8 on 1/6/2008
Pn 2 was 9 on 1/6/2009
Pn 2 was 10 on 1/6/2010
Pn 2 was 11 on 1/6/2011
Pn 2 was 12 on 1/6/2012
Pn 2 was 13 on 1/6/2013
Pn 2 was 14 on 1/6/2014
Pn 2 was 15 on 1/6/2015
Pn 2 was 16 on 1/6/2016
Pn 2 was 17 on 1/6/2017
Pn 2 was 18 on 1/6/2018
Pn 2 was 19 on 1/6/2019
Pn 2 will be 20 on 1/6/2020
$ _
```

Run

Trying it

Born a year ago next month and same day 19 years ago:

Console Input / Output

```
$ java AgeHistory2 01 07 2019 01 8 2018 01 8 2000
 (Output shown using multiple columns to save space.)
Pn 1 was born on 1/8/2018
Pn 1 will be 1 on 1/8/2019
Pn 2 was born on 1/8/2000
Pn 2 was 1 on 1/8/2001
Pn 2 was 2 on 1/8/2002
Pn 2 was 3 on 1/8/2003
Pn 2 was 4 on 1/8/2004
Pn 2 was 5 on 1/8/2005
Pn 2 was 6 on 1/8/2006
Pn 2 was 7 on 1/8/2007
Pn 2 was 8 on 1/8/2008
Pn 2 was 9 on 1/8/2009
Pn 2 was 10 on 1/8/2010
Pn 2 was 11 on 1/8/2011
Pn 2 was 12 on 1/8/2012
Pn 2 was 13 on 1/8/2013
Pn 2 was 14 on 1/8/2014
Pn 2 was 15 on 1/8/2015
Pn 2 was 16 on 1/8/2016
Pn 2 was 17 on 1/8/2017
Pn 2 was 18 on 1/8/2018
Pn 2 will be 19 on 1/8/2019
$ _
```

Run

More tests? What date have we overlooked?

Coursework: Reasoning about conditions

(Summary only)

Do some reasoning to show that two different **condition**s have the same value.

Section 7

Example:
Truth tables

Aim

AIM: To introduce the `boolean` **type**, and reinforce **logical operators**. We also meet the `String` type and see that a **for update** can have multiple **statements**.

Truth tables

- Print out **truth table** for two **hard coded propositional expressions**
  - `p1` : `a && (b || c)`
  - `p2` : `a && b || a && c`

Console Input / Output

```
$ java TruthTable
 _______________________________________
|   a   |   b   |   c   |  p1   |  p2   |
|_______|_______|_______|_______|_______|
| true  | true  | true  | true  | true  |
| true  | true  | false | true  | true  |
| true  | false | true  | true  | true  |
| true  | false | false | false | false |
| false | true  | true  | false | false |
| false | true  | false | false | false |
| false | false | true  | false | false |
| false | false | false | false | false |
|_______|_______|_______|_______|_______|
$ _
```

Run

Truth tables

- Table has 8 lines because 3 **variable**s, `a`, `b` and `c`
  - each can be true or false: $2 \times 2 \times 2$

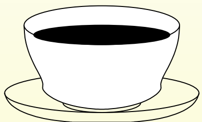
Coffee time: Did you expect the **propositional expression**s to be equivalent? They are – the `p1` and `p2` columns are the same. Make more concrete: replace `a` with `isRaining`, `b` with `haveUmbrella` and `c` with `amWaterproof`:

```
isRaining && (haveUmbrella || amWaterproof)
```

and

```
isRaining && haveUmbrella || isRaining && amWaterproof
```

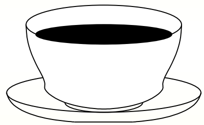
More intuitive?



Type: `boolean`

- Java **type** `boolean`
  - type of all **condition**s
  - named after George Boole.
- two **boolean literal** values: `true` and `false`.
- E.g. `5 <= 5` is a **boolean expression**
  - always `true`.
- E.g. `age1 < age2 || age1 == age2 && height1 <= height2`
  - depends on values of the variables.

Truth tables



Coffee time: What is the value of `5 <= 5`? Is it **`true`** or **`false`**?
What about `5 < 5 || 5 == 5`?

Variable: `boolean` variable

- The `boolean` **type** can be used like `int` and `double`
  - can have **boolean variable**s
  - **method**s can have `boolean` **return type**
  - etc..

Variable: `boolean` variable

- E.g.

```
if (age1 < age2 || age1 == age2 && height1 <= height2)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");
```

Might instead write:

```
boolean correctOrder = age1 < age2 || age1 == age2 && height1 <= height2;
if (correctOrder)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");
```

  - Perhaps more readable code?
  - named **condition** in a helpful way
  - context dependent, ultimately subjective.

Variable: `boolean` variable

- More motive if result used more than once:

```
boolean correctOrder = age1 < age2 || age1 == age2 && height1 <= height2;
if (correctOrder)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");

... Lots of stuff here.

if (!correctOrder)
  System.out.println("Don't forget to swap over!");
```

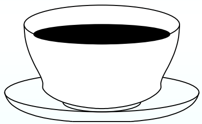
- Novices and some so-called experts may have written. . .

Variable: `boolean` variable

```
boolean correctOrder;
if (age1 < age2 || age1 == age2 && height1 <= height2)
  correctOrder = true;
else
  correctOrder = false;


if (correctOrder == true)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");


... Lots of stuff here.


if (correctOrder == false)
  System.out.println("Don't forget to swap over!");
```

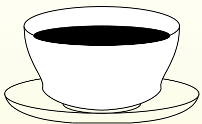
Variable: `boolean` variable

- There are *three terrible* things wrong with the above!



Coffee time: What are they?

Truth tables



Coffee time: Assuming that `be` is some **`boolean`** expression and `bv1` and `bv2` are some **`boolean`** variables, why is it so terrible to write the following?

```
if (bv1 == true)
  if (be) bv2 = true;
  else    bv2 = false;
```

What code should be written instead?

THE UNIVERSITY of MANCHESTER

Truth tables

```
001: // Program to print out the truth table
002: // for two hard coded propositional expressions p1 and p2.
003: // The expressions have three boolean variables, a, b, and c.
004: // Each column of the table occupies 7 characters plus separator.
005: public class TruthTable
006: {
007:   // The first propositional expression, p1.
008:   private static boolean p1(boolean a, boolean b, boolean c)
009:   {
010:     return a && (b || c);
011:   } // p1
012:
013:
```

Truth tables

```
014:   // The second propositional expression, p2.
015:   private static boolean p2(boolean a, boolean b, boolean c)
016:   {
017:     return a && b || a && c;
018:   } // p2
019:
020:
021:   // Print a line of underscores as wide as the truth table.
022:   private static void printStraightLine()
023:   {
024:     System.out.println(" _______________________________________ ");
025:   } // printStraightLine
026:
027:
```

Truth tables

```
028:   // Print the headings for the truth table.
029:   private static void printHeadings()
030:   {
031:     System.out.println("|   a   |   b   |   c   |  p1   |  p2   |");
032:   } // printHeadings
033:
034:
035:   // Print a line of underscores
036:   // with vertical bars for the column separators.
037:   private static void printColumnsLine()
038:   {
039:     System.out.println("|_______|_______|_______|_______|_______|");
040:   } // printColumnsLine
041:
042:
```

Truth tables

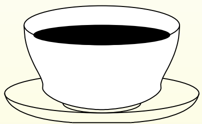
```
043:   // To print a row, we use formatRowItem to make the
044:   // column entries have 7 characters.
045:   private static void printRow(boolean a, boolean b, boolean c)
046:   {
047:     System.out.println("|" + formatRowItem(a) + "|" + formatRowItem(b)
048:                          + "|" + formatRowItem(c)
049:                          + "|" + formatRowItem(p1(a, b, c))
050:                          + "|" + formatRowItem(p2(a, b, c)) + "|");
051:   } // printRow
```

Type: `String`

- Another **type** – `String`
  - type of **text data string**s
  - e.g. **string literal**s
  - **concatenation** results.

Truth tables

```
054:   // Take a boolean row item and return a string of 7 characters
055:   // to represent that item.
056:   private static String formatRowItem(boolean rowItem)
057:   {
058:     return rowItem ? " true  " : " false ";
059:   } // formatRowItem
```



Coffee time: Notice that we did *not* write `rowItem == true` before the `?`. Such code is terrible – every time you are tempted to write it, you should chastise yourself!

Truth tables

- Want three **nested loops**, one for each of `a`, `b` and `c`.
  - each loops twice: once for **`true`** and once for **`false`**.
- Cannot use **boolean variable** to control for loop
  - only two values – need third one to indicate have had both the others.
- So use **int variable** to ensure two executions
  - and make **`boolean`** variable swap from **`true`** to **`false`**.

Statement: for loop: multiple statements in for update

- Can have more than one **statement** in **for update**
  - separated by commas (,)
- E.g. A for loop over the possible values of a **boolean variable** . . . .

Statement: for loop: multiple statements in for update

```
boolean haveUmbrella = true;
boolean isRaining = true;
for (int countU = 1; countU <= 2; countU++, haveUmbrella = !haveUmbrella)
  for (int countR = 1; countR <= 2; countR++, isRaining = !isRaining)
  {
    System.out.println("It is" + (isRaining ? "" : " not") + " raining.");
    System.out.println
       ("You have " + (haveUmbrella ? "an" : "no") + " umbrella.");
    if (isRaining && !haveUmbrella)
      System.out.println("You get wet!");
    else
      System.out.println("You stay dry.");
    System.out.println();
  } // for
```

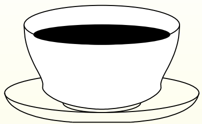
Truth tables

```
062:  // The main method has nested loops to generate table rows.
063:  public static void main(String[] args)
064:  {
065:    printStraightLine();
066:    printHeadings();
067:    printColumnsLine();
068:
069:    // Start off with all three variables being true.
070:    boolean a = true, b = true, c = true;
071:
```

Truth tables

```
072:     // Execute twice for the 'a' variable,
073:     // and ensure 'a' goes from true to false.
074:     for (int aCount = 1; aCount <= 2; aCount++, a = !a)
075:       // Do the same for 'b', for each 'a' value.
076:       for (int bCount = 1; bCount <= 2; bCount++, b = !b)
077:         // Do the same for 'c', for each 'b' value.
078:         for (int cCount = 1; cCount <= 2; cCount++, c = !c)
079:           // Print a row for each a, b and c combination.
080:           printRow(a, b, c);
081:
082:     printColumnsLine();
083:   } // main
084:
085: } // class TruthTable
```

Truth tables



Coffee time: In some programming languages, such as Perl(**?**), it is possible to treat **data** as program code at **run time**. But this is not so in Java (maybe that is a good thing?). How easy would it be to alter this program so that the propositional expressions are supplied as **command line arguments** rather than being hard coded?

Coursework: `TruthTable34`

(Summary only)

Write a program to test the equivalence of three **propositional expressions**, each having four **variables**.

Section 8

Example:
Producing a calendar

THE UNIVERSITY of MANCHESTER

Aim

AIM: To reinforce much of the material presented in this chapter. We also revisit `System.out.printf()`.

Producing a calendar

- Wish to produce monthly calendar, given start day and number of days.

Console Input / Output

```
$ java Calendar 3 28
 --------------------
|Su Mo Tu We Th Fr Sa|
|         01 02 03 04 05|
|06 07 08 09 10 11 12|
|13 14 15 16 17 18 19|
|20 21 22 23 24 25 26|
|27 28               |
|                    |
 --------------------
$ _
```

Run

- Just to be different, declare each method after it is used.

Producing a calendar

```
001: // Program to print a calendar for a single given month.
002: // The first argument is the number of the start day, 1 to 7
003: // (Sunday = 1, Monday = 2, ..., Saturday = 7).
004: // The second argument is the last date in the month, e.g. 31.
005: public class Calendar
006: {
007:   public static void main(String[] args)
008:   {
009:     printMonth(Integer.parseInt(args[0]), Integer.parseInt(args[1]));
010:   } // main
```

Producing a calendar

```
013:   // Print the calendar for the month.
014:   private static void printMonth(int monthStartDay, int lastDateInMonth)
015:   {
016:     // Keep track of which day (1-7) is the next to be printed out.
017:     int nextDayColumnToUse = monthStartDay;
018:
019:     // Keep track of the next date to be printed out.
020:     int nextDateToPrint = 1;
021:
022:     // The top line of hyphens.
023:     printMonthLineOfHyphens();
024:     // The column headings.
025:     printDayNames();
026:
```

Producing a calendar

```
027:    // We always print out as many rows as we need,
028:    // but with a minimum of 6 to encourage consistent format.
029:    int noOfRowsPrintedSoFar = 0;
030:    while (nextDateToPrint <= lastDateInMonth || noOfRowsPrintedSoFar < 6)
031:    {
032:      // Print one row.
033:      System.out.print("|");
034:      for (int dayColumnNo = 1; dayColumnNo <= 7; dayColumnNo++)
035:      {
036:        // Print a space separator between day columns.
037:        if (dayColumnNo > 1)
038:          System.out.print(" ");
039:
```

Producing a calendar

```
040:         // We either print spaces or a date.
041:         if (dayColumnNo != nextDayColumnToUse
042:             || nextDateToPrint > lastDateInMonth)
043:           printDateSpace();
044:         else
045:         {
046:           printDate(nextDateToPrint);
047:           nextDayColumnToUse++;
048:           nextDateToPrint++;
049:         } // else
050:       } // for
051:
052:       // End the row.
053:       System.out.println("|");
054:       noOfRowsPrintedSoFar++;
```

Producing a calendar

```
055:
056:        // Get ready for the next row.
057:        nextDayColumnToUse = 1;
058:      } // while
059:
060:      // The bottom line of hyphens.
061:      printMonthLineOfHyphens();
062:    } // printMonth
```

Producing a calendar

```
065:  // Print a line of hyphens as wide as the table,
066:  // starting and ending with a space so the corners look right.
067:  private static void printMonthLineOfHyphens()
068:  {
069:    System.out.print(" ");
070:    for (int dayColumnNo = 1; dayColumnNo <= 7; dayColumnNo++)
071:    {
072:      if (dayColumnNo > 1)
073:        System.out.print("-");
074:      printDateHyphens();
075:    } // for
076:    System.out.println(" ");
077:  } // printMonthLineOfHyphens
```

THE UNIVERSITY of MANCHESTER

Producing a calendar

```
080:   // Print the day name headings.
081:   private static void printDayNames()
082:   {
083:     System.out.print("|");
084:     for (int dayColumnNo = 1; dayColumnNo <= 7; dayColumnNo++)
085:     {
086:       if (dayColumnNo > 1)
087:         System.out.print(" ");
088:       printDayName(dayColumnNo);
089:     } // for
090:     System.out.println("|");
091:   } // printDayNames
```

THE UNIVERSITY of MANCHESTER

Producing a calendar

```
094:   // Print the day name of the given day number, as two characters.
095:   private static void printDayName(int dayNo)
096:   {
097:     // Our days are numbered 1 - 7, from Sunday.
098:     switch (dayNo)
099:     {
100:       case 1: System.out.print("Su"); break;
101:       case 2: System.out.print("Mo"); break;
102:       case 3: System.out.print("Tu"); break;
103:       case 4: System.out.print("We"); break;
104:       case 5: System.out.print("Th"); break;
105:       case 6: System.out.print("Fr"); break;
106:       case 7: System.out.print("Sa"); break;
107:     } // switch
108:   } // printDayName
```

Producing a calendar

```
111:   // Print spaces as wide as a date, i.e. two spaces.
112:   private static void printDateSpace()
113:   {
114:     System.out.print("  ");
115:   } // printDateSpace
```

and

```
118:   // Print hyphens as wide as a date, i.e. two hyphens.
119:   private static void printDateHyphens()
120:   {
121:     System.out.print("--");
122:   } // printDateHyphens
```

Standard API: `System`: `out.printf()`: zero padding

- `System.out.printf()` can produce **zero padding** instead of **space padding**
  - place leading zero on minimum width in **format specifier**:

```
System.out.println("1234567890");
System.out.printf("%010d%n", 123);
```

produces:

```
1234567890
0000000123
```

- Also:

```
System.out.println("1234567890");
System.out.printf("%010.2f%n", 123.456);
```

produces:

```
1234567890
0000123.46
```

Producing a calendar

```
125:   // Print a date, using two characters, with a leading zero if required.
126:   private static void printDate(int date)
127:   {
128:     System.out.printf("%02d", date);
129:   } // printDate
130:
131: } // class Calendar
```

Trying it

Console Input / Output

```
$ java Calendar 6 29; java Calendar 7 31; java Calendar 3 30
 (Output shown using multiple columns to save space.)
 --------------------   --------------------   --------------------
|Su Mo Tu We Th Fr Sa| |Su Mo Tu We Th Fr Sa| |Su Mo Tu We Th Fr Sa|
|               01 02| |                  01| |      01 02 03 04 05|
|03 04 05 06 07 08 09| |02 03 04 05 06 07 08| |06 07 08 09 10 11 12|
|10 11 12 13 14 15 16| |09 10 11 12 13 14 15| |13 14 15 16 17 18 19|
|17 18 19 20 21 22 23| |16 17 18 19 20 21 22| |20 21 22 23 24 25 26|
|24 25 26 27 28 29   | |23 24 25 26 27 28 29| |27 28 29 30          |
|                    | |30 31               | |                    |
 --------------------   --------------------   --------------------
$ _
```

Run

Coursework: `CalendarHighlight`

(Summary only)

Modify a calendar month printing program to produce a larger calendar format and to highlight a certain date.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.