Lecture 16
Selections(If-else)

COMP101: Introduction to Programming in JAVA

Clare Dixon
with materials from Michele Zito and Frans Coenen
Department of Computer Science
cldixon@liv.ac.uk
www2.csc.liv.ac.uk/∼clare/teaching/COMP101

Our Programs So Far

Until now our Java programs have included one or two classes (plus references to pre-defined classes like `String` or `Math`). However the methods have been simply sequences of statements eg doing assignment or I/O.

Example. How to make *bruschetta bread*.

```
Slice the bread;
Grill the slices until brown;
Grate a garlic clove on each slice;
Garnish with a little extra-virgin olive oil;
Add half a juicy tomato on top.
```

No scope for variations!!

Example

Prepare a “flexible” *bruschetta bread*.

```
Slice the bread;
Grill the slices until brown;
Grate a garlic clove on each slice;
IF you like it hot
    sprinkle chopped chillies on top of the bread
ELSE IF you like it ‘cheesy’
    grate a little parmesan OR
    some mature cheese on top of the bread;
Garnish with a little extra-virgin olive oil;
IF you have a few rocket leaves
    add them on top of the bread;
Add half a juicy tomato on top.
```

You make a different choice and you get many different bruschetta variants.

Selection

In this lecture we show how such statements are encoded in Java, known in general as selection statements.

A selection statement provides for selection between alternatives.

We can identify two main types of selection construct:

- if-else statements; and
- switch (or case) statements.

Syntax of if-else Statements

The general format of an **`if-else`** statement in Java is

```
if  ( condition )
     { statements }
else                         <=== This part may not
     { statements }          <===  be there
```

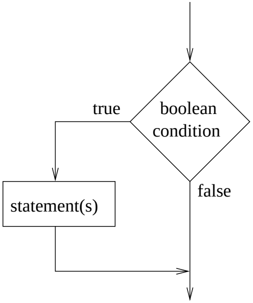
Meaning: if the stated condition is true then the first set of statements is executed, otherwise the second set is executed.

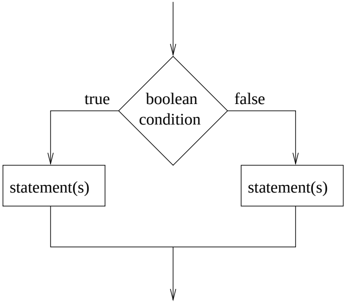
Each set of statements can consist of a single instruction terminated by the semicolon, or a sequence of instructions ENCLOSED IN CURLY BRACKETS.

Semantics of if-else Statements

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

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





Conditions

What goes in the condition part? Any expression that evaluates to either true or false!

Examples.

1. A comparison between two variables ("if A is greater than B then something happens otherwise something else happens")
2. The check that some expression is not larger than a certain threshold ("if the employees earns less than 100,000,000 pounds tax him at 22%").
3. Check that input is within the desired range. (eg disallowing a negative value for a circle radius).

In Java such conditions usually involve variables and, so called, *relational operators*.

Relational Operators

A ***relational operator*** is a function that takes two inputs belonging to a primitive type and returns a “YES/NO” (“TRUE/FALSE”) answer.

Relational operators in Java are normally placed between the two expressions they compare.

A list of relational operators is given below

operator	**meaning**
$>$	greater than
$<$	less than
$==$	equal to
$>=$	greater than or equal to
$<=$	less than or equal to
$!=$	not equal to

Example Problem – Power 4

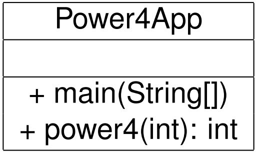
Requirements

Design and create a Java program that takes a single integer as input. If the input is less than 50 add 10 and return the result raised to the power of 4, otherwise simply return the input raised to the power of 4.

$$f(x) = \begin{cases} (x+10)^4 & \text{if } x < 50 \\ x^4 & \text{otherwise} \end{cases}$$

Analysis

As this is such a simple program we will just use one class `Power4App`



Processing

The `main` method has the task of obtaining the integer input from the keyboard, calling the power4 method to perform the calculation and printing the result.

```
METHOD main
INPUT  args
OUTPUT  void
   LOCAL DATA inputInteger
   READ inputInteger from the keyboard
   PRINT the result of applying  the power4 method to inputInt
```

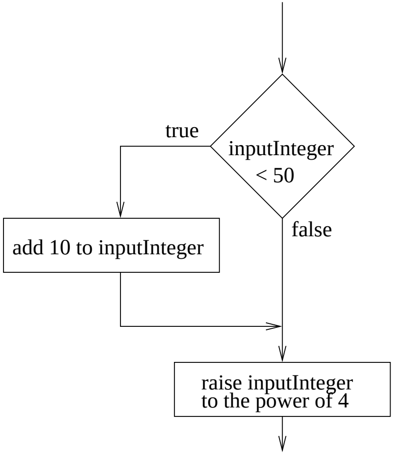
Processing: power4 Method

The `power4` method checks the value of its parameter and adding 10 if necessary, performing the required calculation and returning the result.

```
METHOD power4
INPUT  value
OUTPUT  integer
  IF value < 50
       add 10 to value
   CALCULATE the value of value raised
       to the power of 4
   RETURN the result
```

Diagram of Flow of Control

We provide a flow of control diagram for the if statement.



Implementation

```
// POWER 4 APPLICATION
// Frans Coenen: March 1999, revised 2005
// Revised by Clare Dixon 2011

import java.util.*;
class Power4App {
    // ------------------ FIELDS ------------------------
    // ----------------- METHODS ------------------------
    /* Main method */
    public static void main(String[] args) {
        // Create Scanner class instance
        Scanner input = new Scanner(System.in);
        int inputInteger;
        // Input
        System.out.print("Input␣an␣integer:␣");
        inputInteger = input.nextInt();
        // Output
        System.out.println("Result␣is:␣␣␣␣" + power4(inputInteger));
        }
```

Implementation

```
    // Method which increments its parameter by 10 if it is
    // less than 50  and then raises it to the power of 4.
    public static int power4(int value) {
       if (value < 50)
            value = value + 10;
        value = (int)Math.pow(value,4.0);
        return value;
    }
}
```

Arithmetic Testing

We should derive test cases incorporating positive, negative and zero sample values. An appropriate set of cases is given below.

Test Case	Expected Result
Input Integer	Output
10	160000
0	10000
-10	0

Data validation: In addition we should undertake some data validation testing.

Arithmetic Testing:Output

The output we obtain is as follows.

```
$ java Power4App
Input an integer: 10
Result is:    160000

$ java Power4App
Input an integer: 0
Result is:    10000

$ java Power4App
Input an integer: -10
Result is:    0
```

Path Testing

An ”if” statement should always be tested using at least two test cases, one that causes the if statement to succeed and one that causes it to fail.

Both ”paths” through the procedure should be tested (known as *path testing*).

It is also a good idea, when using discrete data items in the condition expression, to test at the ”cut-off” point where appropriate, i.e. 49 and 50 in this case (in the spirit of range and limit testing). A suitable set of test cases are given below.

Test Case	Expected Result
Input Integer	Output
10	160000
49	12117361
50	6250000
59	12117361

Path Testing: Output

The output we obtain is as follows.

```
$ java Power4App
Input an integer: 10
Result is:    160000

$ java Power4App
Input an integer: 49
Result is:    12117361

$ java Power4App
Input an integer: 50
Result is:    6250000

$ java Power4App
Input an integer: 59
Result is:    12117361
```

Black and White Box Testing

- Techniques such as arithmetic testing are generally regarded as Black Box Testing techniques.
- That is the software to be tested is to be regarded as a box, the internal workings of which cannot be seen, into which we feed input data which outputs data as a result.
- Thus black box test cases are only concerned with input and output and not the nature of the transformation(s) that causes the first to become the second.
- The opposite of black box testing is white box (or glass box) testing.
- White box testing is concerned with the inner workings of software systems.
- Path testing is an example of a white box testing technique.
- Data validation is not generally considered to fall into either category.

Example Problem – Integer Classification

Requirements

Design and create a Java program that takes a single integer as input and classifies it as positive, negative or zero.

Analysis and Design

Lets think about this together.

Implementation

```
/** Author: Clare Dixon July 2010
 * Example of using if statements and relational operators
 **/
import java.util.Scanner;
public class IntegerClassification {
   /*  Main Method */
    public static void main (String[] args) {
       Scanner input = new Scanner(System.in);
        int value;
        System.out.print("Please␣provide␣an␣integer␣value:␣");
        value = input.nextInt(); //read the value from the keyboard
        if (value < 0)
            System.out.println("Your␣number␣is␣negative");
        else {
              if (value == 0)
                  System.out.println("Your␣number␣is␣zero");
              else {
                   if (value > 0)
                       System.out.println("Your␣number␣is␣positive");
              }
         }
    }
```

Testing

For both path testing and arithmetic testing we should derive test cases incorporating positive, negative and zero sample values. An appropriate set of cases is given below.

Test Case	Expected Result
Input Integer	Output
10	positive
1	positive
0	zero
-1	negative
-10	negative

Data validation: In addition we should undertake some data validation testing.

Output

```
$ java IntegerClassification
Please provide an integer value: 10
Your number is positive

$ java IntegerClassification
Please provide an integer value: 1
Your number is positive

$ java IntegerClassification
Please provide an integer value: 0
Your number is zero

$ java IntegerClassification
Please provide an integer value: -1
Your number is negative

$ java IntegerClassification
Please provide an integer value: -10
Your number is negative
```

Common Errors with If-Statements

The following will result in syntax errors. What is wrong?

```
if x < 7
  System.out.println(``Less than seven'');
```

```
if (x < 7)
  x = x + 10
```

```
if (x = 7)
  System.out.println(``Equal to seven'');
```

```
if (x > 7)
  ``Greater than seven'';
```

Common Errors with If-Statements

The following won't cause syntax errors but may not provide the expected results.

```
.....
int x = input.nextInt(); //read the value from the keyboard
if (x > 7)
    System.out.println(``The value of x'');
    System.out.println(``is greater  than seven'');
....
```

Summary

- We introduced the Java syntax for if-statements
- We looked at the conditions used in if-statements and provided relational operators.
- We showed the usage in some example programs.
- We discussed the need for path testing.

Lecture 17
Selections(If-else and Booleans)

COMP101: Introduction to Programming in JAVA

Clare Dixon
with materials from Michele Zito and Frans Coenen
Department of Computer Science
cldixon@liv.ac.uk
www2.csc.liv.ac.uk/∼clare/teaching/COMP101

Overview

- We look further at if-else statements.
- First we consider Boolean expressions and take another look at the Boolean primitive data type and its operations.
- We analyse, design and implement a program to test for leap years.
- We highlight some of the pitfalls when using if-else statements.

Primitive Data Type 4

Type	**Booleans**
Range	values **`false`** and **`true`**.
Keyword	**`boolean`**
Operators	Relational operators (giving a boolean result): `<`, `>`, `==`, `!=`, `<=`, `>=`. Boolean operators (composing boolean values): `!`, `&&`, `||`.

Boolean Expressions

We looked at relational operators last lecture.
One can do calculations with booleans exactly in the same way as one can do calculations with numbers.

```
- 12                34214 + 51              2367*(-85131)
```

```
!false              false || true           true && (!false)
```

`!` is the *logical negation*, `&&` is the *logical conjunction* and `||` the *logical disjunction*.

```
!(4 < a)    (a <= 12) || (a % 2 == 0)    (0 <= a) && (a <= 9)
```

Boolean Operator Tables

A	B	!A	A && B	A || B
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Boolean expressions can be used in the conditions of a selection statement (and in loops).

What is the value of the following for `a=3` or `a=20`?

```
!(4 < a)        (a <= 12) || (a % 2 == 0)        (0 <= a) && (a <= 9)
```

Identification of Leap Years

Problem

Write a program which allows the input of a 4-digit year, checks that the input is in the correct range, checks whether it is a leap year, and outputs this information.

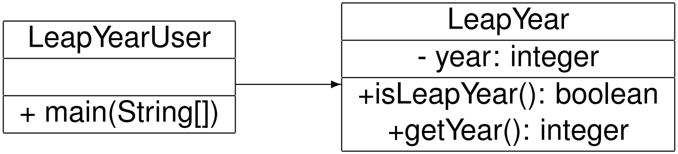
You should check that the input date is a positive integer or zero not greater than 9999.

Analysis

Write a $\boxed{\textit{program}}$ which allows the input an $\boxed{\textit{4-digit year}}$, checks that the $\boxed{\textit{input}}$ is in the $\boxed{\textit{correct range}}$, checks whether it is a $\boxed{\textit{leap year}}$, and outputs this.

- It isn't clear whether we just need one class or two.
- Lets follow what we have done recently and have two classes `LeapYear` and `LeapYearUser`.
- We will store the year as an attribute of `LeapYear`.
- `LeapYear` also needs a method to calculate whether the year is leap or not.
- We may also need a method to get the value of the attribute year.

Class Diagram



Algorithm Design

The `main` method has the usual task of setting up all relevant variables, then calling the isLeapYear method and finally displaying the result of the computation. Note we can check the input data is not negative using an if-else statement.

```
METHOD main
INPUT  args
OUTPUT  void
    LOCAL DATA someYear, MIN_YEAR = 0, MAX_YEAR=9999
    READ someYear from the keyboard
   IF someYear >= MIN_YEAR AND someYear <= MAX_YEAR
      SET up an instance myLeapYear of LeapYear using
          someYear as the year of the structure
       PRINT a message with the result of
          calculating the whether someYear is a Leap Year.
   ELSE
       PRINT a message saying invalid year
```

Operations

How can we check if a year is a leap year?

There exists a simple formula that does the job.

```
if the year is exactly divisible by 4 and
        the year is not a century
    or the year is a century that is divisible by 400
    then the year is a leap year.
```

$\div 4$	$\div 100$	$\div 400$	Eg	Leap
T	T	T	1600	YES
T	T	F	1700	NO
T	F	F	1704	YES
F	F	F	1705	NO

The method `isLeapYear`

The pseudo-code for `leap` can be easily derived from this description:

```
METHOD isLeapYear
INPUT  none (it uses LeapYear attribute year)
OUTPUT boolean
    IF year is divisible by 4 AND
           it is not a century OR
       the year is a century AND divisible by 400
        RETURN true
    ELSE
        RETURN false
```

Implementation:LeapYear

```
/**
* LeapYear.java, Based on code by Michele Zito
* Edited by Clare Dixon June 2010
**/
public class LeapYear {
    // attributes
    private int year;
    // constructors
    public LeapYear(int y) {
        year = y;
    }
        // methods
    public boolean isLeapYear() {
        if (((year % 4 == 0) && (year % 100 != 0)) ||
            ((year % 100 == 0)  && (year % 400 == 0)))
                    return true;
                  else
                      return false;
    }
    public int getYear() {
         return year;
    }
```

Implementation:LeapYearUser

```
/**  LeapYearUser.java
*     July 2010  Clare Dixon  **/
import java.util.Scanner;

public class LeapYearUser {
    public static void main(String [] args) {
        Scanner input = new Scanner(System.in);
        int myYear;
        final int MIN_YEAR = 0;
        final int MAX_YEAR = 9999;
       System.out.print("What's␣the␣year?␣");
        myYear = input.nextInt();
        if ((myYear >= MIN_YEAR) && (myYear <= MAX_YEAR)) {
            LeapYear someLeapYear = new LeapYear(myYear);
            System.out.print("The␣year␣" +  someLeapYear.getYear());
            if (someLeapYear.isLeapYear())
                System.out.println("␣is␣a␣leap␣year.");
            else
                System.out.println("␣is␣not␣a␣leap␣year.");
        }
        else
            System.out.println("␣Not␣a␣valid␣year.");
```

Testing: Path Testing

Path Testing: `LeapYearUser` We should take a positive, negative and zero integer as well as values 9999 and a value greater than 9999 to test paths in the outer if statement of `LeapYear User`.

Test Case	Expected Result
Input Year	Output
-10	Wrong type of input
0	Is Leap Year
10	Is not a Leap Year
9999	Is not a Leap Year
10000	Wrong type of input

Path Testing: `isLeapYear()`

Test Case	Expected Result	Reason
Input Year	Output	
1600	Is Leap Year	Divisible by 400
1700	Is not Leap Year	Divisible by 100 but not by 400
1704	Is Leap Year	Divisible by 4 but not by 100 or 400
1705	Is not Leap Year	Not divisible by 4, 100 or 400

Data validation: In addition we should undertake some data validation testing

Output

```
$ java LeapYearUser
What’s the year? -10
 Not a valid year.

$ java LeapYearUser
What’s the year? 0
The year 0 is a leap year.

$ java LeapYearUser
What’s the year? 10
The year 10 is not a leap year.

$ java LeapYearUser
What’s the year? 9999
The year 9999 is not a leap year.

$ java LeapYearUser
What’s the year? 10000
 Not a valid year.
```

Output

```
$ java LeapYearUser
What’s the year? 1600
The year 1600 is a leap year.

$ java LeapYearUser
What’s the year? 1700
The year 1700 is not a leap year.

$ java LeapYearUser
What’s the year? 1704
The year 1704 is a leap year.

$ java LeapYearUser
What’s the year? 1705
The year 1705 is not a leap year.
```

Meaning of if-else Statements

What will the following code snippet do?

```
if (myValue > 0)
    if (myValue % 2 == 0)
        System.out.println("The␣number␣is␣positive␣and␣even");
else
    System.out.println("␣Not␣positive.");
```

Swapping Two Values

A common task in programming is to swap the values of two variable if the second is smaller that the first.

```
IF b < a
   swap the values of a and b
```

The following is incorrect (for example take a = 13 and b = 10). Why?

```
if (b < a) {
    b = a;
    a = b;
}
```

Swapping Two Values

We must introduce a temporary variable to store the value of b. Again take a = 13 and b = 10.

```
if (b < a) {
    int tmp = b;
    b = a;
    a = tmp;
}
```

Summary

- We have shown how to detect leap years using if-else statements.
- We have also shown how to do simple checking of input using if-else statements.
- We have looked at how to swap the values of two variables given some condition.

Lecture 18
Selections: Switch

COMP101: Introduction to Programming in JAVA

Clare Dixon
with materials from Michele Zito and Frans Coenen
Department of Computer Science
cldixon@liv.ac.uk
www2.csc.liv.ac.uk/∼clare/teaching/COMP101

Overview

- In this lecture we consider the Java `switch` statement.
- This is another type of selection statement.
- After giving the details of this we add to the class considering leap years we looked at last lecture to a date validation class.

Switch

If a selection is based on the values of some expression, belonging to an ordinal type (a value that belongs to a set of ordered items eg integers or characters) then a switch statement may be used instead of a nested `if-else`.

General format of a `switch` statement.

```
switch ( expression ) {
    case constant-expression :
        statements
    break;
    case constant-expression :
        statements
    break;
        ...
    default :
    statements
}
```

Semantics of Switch Statements

The **`default`** clause is optional as are the **`break`** statements but removing the **`break`** statement may have undesired consequences.

1. The `expression` is evaluated.
2. Control passes to the statements following the **`case`** label whose value equals the `expression`, or, if no cases apply, to the **`default`** case.
3. Beginning at the selected label or at the default, all of the statements up to the first of either a **`break`** statement or the end of the switch are executed.

If the **`break`** statements are missing the **`switch`** will execute *all* the statements following the label that matches the `expression`.

Switch Example

```
int number = input.nextInt();

// Process number using a case statement
switch (number) {
    case 0:
           System.out.println("Number␣is␣0");
           break;
    case 1:
           System.out.println("Number␣is␣1");
           break;
    case 2:   case 3:  case 4:
           System.out.println("Number␣is␣2,␣3␣or␣4");
           break;
    default:
          System.out.println("Number␣is␣less␣than␣0␣or␣greater␣than␣
    }
```

What happens if `number` is 1 or 7? What happens if we remove the **`break`** statements and allow the same input?

Case Study: Validation of Dates including Leap Years

Problem

We want to write a program that validates a date. The format of the date DDMMYYYY is a single integer representing day, month, and year. The single integer is split into individual integers representing DD, MM, and YYYY. The program checks that the number of months in a year should not exceed 12, and that the number of days in each month has not been exceeded. The program also reports on leap years.

Class Analysis

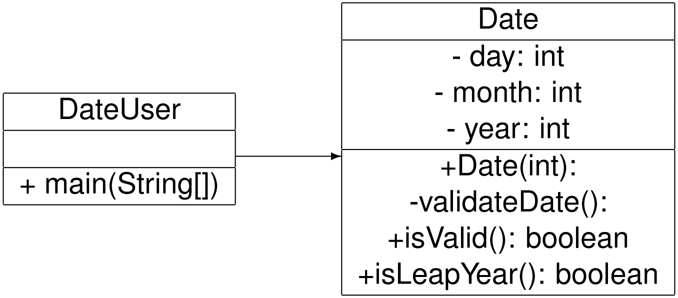
Problem. We want to write a $\boxed{\text{program}}$ that validates a $\boxed{\text{date}}$. The format of the date DDMMYYYY is a single integer representing $\boxed{\text{day}}$, $\boxed{\text{month}}$, and $\boxed{\text{year}}$. The single integer is split into individual integers representing DD, MM, and YYYY. The program checks that the number of months in a year should not exceed 12, and that the number of days in each month has not been exceeded. The program also reports on leap years.

Note the largest integer is 2,147,483,647 which is bigger than the 8 digits we require for input.

Class Analysis

- We start assuming that `DateUser` (corresponding to the word program) and `Date` are classes.
- `month` (an integer between 1 and 12), `day` (an integer in an appropriate range depending on the value of `month` and on whether `year` is a leap year), and `year` (from 0 to 9999) would then be attributes of `Date`.
- We recognize three operations in the system: the setting up of the `Date`'s attributes, the validation of the date (the class constructor should probably have responsibility for these), and the checking on leap years (which we have already done) again `Date` should have responsibility for that.

Class Diagram (2)



Splitting the DDMMYYYY into parts

We can use use a combination of / (integer division) and % (remainder) to obtain the days, months and years separately. Consider an input date `rawDate`

1. Divide `rawDate` by 1,000,000 (using ”integer division”) to obtain the day.
2. Divide the remainder of `rawDate`/1,000,000 by 10,000 to obtain the month.
3. Calculate the remainder of `rawDate`/10,000 to obtain the year.

Example, given `rawDate`=24,092,010 (24-09-2010):

$$\begin{aligned}
\mathit{day} &= 24,092,010/1,000,000 &&= 24\\
\mathit{month} &= (24,092,010 \,\%\, 1,000,000)/10,000 &&=\\
&\ 92,010/10,000 &&= 9\\
\mathit{year} &= 24,092,010 \,\%\, 10,000 &&= 2010
\end{aligned}$$

Setting up `Date`'s Attributes

```
METHOD Date (Constructor)
INPUT rawDate (an integer)
OUTPUT none
    SET day to rawDate divided by 1,000,000
    SET month to the result of dividing by 10,000
       the remainder of rawDate divided by 1,000,000
    SET year to the remainder of rawDate divided by 10,000
    CALL validateDate
```

Implementation of `Date`

```
/**
* Date - validates dates of the form DDMMYYYY
* Michele Zito
* Edited by Clare Dixon September 2010
**/
public class Date {
   // attributes
    private int day;
    private int month;
    private int year;

    // constructor
    public Date(int rawDate) {
        //     split rowDate into month, day, and year
        day = rawDate / 1000000;
        month = (rawDate % 1000000) / 10000;
        year = rawDate % 10000;
        validateDate();
        }
 .........
```

Date validation

The validation of the date has a three-part solution.

The first part is to validate a month as an integer in the range $\{1, 2, \ldots, 12\}$. If the month is treated as an ordinal value of a **`switch`** expression, with case labels occurring for each of the twelve months, then should a month not be in the range 1 to 12, the error can be trapped as the **`default`** value.

The second part involves the checking on leap years, which can be done calling the `isLeapYear` method.

The third part of the solution involves the calculation of the number of days in a month (this depends on the value of `months`, and whether `year` is a leap year or not).

validateDate

```
METHOD validateDate
INPUT
OUTPUT  void
   IF month  is 1, 3, 5, 7, 8, 10, or 12
        numberOfDays is 31
   ELSE IF month is 4, 6, 9, or 11
             numberOfDays is 30;
           ELSE IF month is 2
              IF it is a leap year
                 numberOfDays is  29
              ELSE
                 numberOfDays is 28
                  ELSE
                     print error message
 check day against numberOfDays
```

Implementation of `Date`

```
private void validateDate() {
    int numberOfDays = 0;
    //     sets numberOfDays depending on month and
    //      whether it's a leap year
    switch (month) {
    case 1 : case 3 : case 5 : case 7 : case 8 : case 10 : case 12
        numberOfDays = 31;
        break;
    case 4 : case 6 : case 9 : case 11 :
        numberOfDays = 30;
        break;
    case 2 :
        if (isLeapYear())
            numberOfDays = 29;
        else
            numberOfDays = 28;
        break;
    default :
        day = month = 0; year = -1;
        break;
    }
```

Implementation of `Date` (cont)

```
    //     check day against numberOfDays
    if (day > numberOfDays) {
       day = month = 0;
       year = -1;
     }
}
```

Constructors

We can now build clever constructors.

Constructors don't have to just take some parameters and assign them to the attributes.

We can *check* the validity of our data (not just *date*!) For instance with a simple if statement we can discard values outside a certain range.

Our programs become more robust.

Implementation of `isValid` and `isLeapYear`

The design and implementation for `isLeapYear` is exactly the same as previously.

In the constructor if we have an invalid date we have assigned year to be -1 so to implement `isValid` all we need to do is check for this.

```
public boolean isValid() {
    return (year != -1);
}
```

DateUser

```
/*
 * DateUser by Michele Zito
 * Edited by Clare Dixon Sep 2010
 */
import java.util.Scanner;

public class DateUser {

    public static void main(String [] args) {
        Date date;
        Scanner input = new Scanner(System.in);
        System.out.print("Input␣a␣date␣in␣the␣format␣DDMMYYYY␣");
        date = new Date(input.nextInt());
        if (date.isValid()) {
            System.out.print("It␣is␣a␣valid␣date␣and␣the␣year␣is");
            if (!date.isLeapYear())
                        System.out.print("n't");
            System.out.println("␣a␣leap␣year");
        }
        else
            System.out.println("Invalid␣date!");
    }
```

Testing

We have only one input.

- Arithmetic Testing eg -10121970, 0, 10121970
- Path Testing: for the data validation there are 13 cases (including the default) to test each which may have either the correct or incorrect number of days.
- The `isLeapYear` method should also be tested (but we have already done that).

Testing

Test Case	Expected Result	
Input Integer	Output	
-10121970	invalid date	-
0	invalid date	-
10121970	valid date	not leap
32012011	invalid date	-
31012011	valid date	not leap
29022011	invalid date	-
28022011	valid date	not leap
29022012	valid date	leap
30022012	invalid date	not leap
...	...	...

Summary

- In this lecture we looked at the syntax and semantics of the Java `switch` statement.
- Experiment with switch statements with and without using `break`.
- Next we designed and implemented a date validation class.
- We used a more complex constructor than we have previously to break the input down into the three components (day, month, year) and also to validate the date.