List of Slides

Title

Java Just in Time

John Latham

December 6, 2018

Chapter 15

Exceptions

Chapter aims

- So far made unreasonable assumptions about end user
  - no mistakes
  - programs had little/no code to guard against erroneous input.
- Here look at **exception**s
  - how we may avoid
    - but why we do not!
- Then Java's **exception catching** mechanism
  - let them happen
  - recover from them.
- Many kinds of exception
  - may treat different kinds differently.
- Also can **throw** exceptions in own code.

Section 2

Example:

Age next year revisited

Aim

AIM: To take a closer look at **run time error**s, or as Java calls them, **exception**s.

Age next year revisited

- Revisit AgeNextYear: see what can go wrong.

```
001: // Gets current age from first argument, and reports age next year.
002: public class AgeNextYear
003: {
004:   public static void main(String[] args)
005:   {
006:     int ageNow = Integer.parseInt(args[0]);
007:     int ageNextYear = ageNow + 1;
008:
009:     System.out.println("Your age now is " + ageNow);
010:     System.out.println("Your age next year will be " + ageNextYear);
011:   } // main
012: } // class AgeNextYear
```

Age next year revisited

- Two ways user can make it fail
  - run it without **command line argument**
    - can't access `args[0]`
  - supply argument which is not string representation of `int`
    - `Integer.parseInt()` will fail.
- When **exception**al circumstance occurs
  - **instance** of **class** `Exception` created.

Exception

- Java calls **run time error**s **exception**s.
- Standard **class** `java.lang.Exception`
  - used to record and handle exceptions.
- When exceptional situation happens
  - **instance** of `Exception` created
  - contains information about problem
    - **stack trace**: source line number, **method** name, class name, etc..

Trying it

- No **command line arguments**:

Console Input / Output

```
$ java AgeNextYear
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at AgeNextYear.main(AgeNextYear.java:6)
$ _
```

Run

- See kind of exception: `ArrayIndexOutOfBoundsException`.
  - `args[0]` fails in **main method**
  - **stack trace** contains only one entry.
- Default action of **virtual machine** for exceptions in **main thread**:
  - print details of associated `Exception` **object**
  - end the **thread**
    - program terminates unless another thread running.

Trying it

- A string not representing **`int`** value:

Console Input / Output

```
$ java AgeNextYear ""
Exception in thread "main" java.lang.NumberFormatException: For input string: ""
        at java.lang.NumberFormatException.forInputString(NumberFormatException.
java:48)
        at java.lang.Integer.parseInt(Integer.java:470)
        at java.lang.Integer.parseInt(Integer.java:499)
        at AgeNextYear.main(AgeNextYear.java:6)
$ java AgeNextYear 4.25
Exception in thread "main" java.lang.NumberFormatException: For input string: "4.25"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:458)
        at java.lang.Integer.parseInt(Integer.java:499)
        at AgeNextYear.main(AgeNextYear.java:6)
$ _
```

Run

- Kind of exception: `NumberFormatException`.
- Detected within `Integer.parseInt()`, called from **main method**.

Trying it

- Also two cases of 'bad' input that do not cause exception:
  - representation of a negative number
  - more than one argument.

Console Input / Output

```
$ java AgeNextYear -4
Your age now is -4
Your age next year will be -3
$ java AgeNextYear 60 4
Your age now is 60
Your age next year will be 61
$ _
```

Run

- Let us make this program robust. . . .

Coursework: `FishTankVolume` robustness analysis

(Summary only)

Take a program you have seen before and analyse where it can go wrong.

Section 3

Example:

Age next year with exception avoidance

Aim

AIM: To show how we can avoid **exception**s using **conditional execution**. We also meet the `Character` **class**.

Age next year with exception avoidance

- Could add code to avoid **exception**s.
- First add **method** to check
  - `String` contains only digits,
  - and is not empty.

Standard API: `Character`

- `java.lang.Character` has **class methods**, including:

Public method interfaces for class `Character` (some of them).

Method	Return	Arguments	Description
`isWhitespace`	**`boolean`**	**`char`**	Returns **`true`** if the given **`char`** is a **white space** character, (e.g. **space character**, **tab character**, **new line character**), or **`false`** otherwise.

Standard API: Character

Public method interfaces for class Character (some of them).

Method	Return	Arguments	Description
isDigit	**boolean**	**char**	Returns **true** if the given **char** is a digit (e.g. '0', '8'), or **false** otherwise.
isLetter	**boolean**	**char**	Returns **true** if the given **char** is a letter (e.g. 'A', 'a'), or **false** otherwise.
isLetterOrDigit	**boolean**	**char**	Returns **true** if the given **char** is a letter or a digit, or **false** otherwise.

Standard API: Character

Public method interfaces for class Character (some of them).

Method	Return	Arguments	Description
isLowerCase	**boolean**	**char**	Returns **true** if the given **char** is a lower case letter, or **false** otherwise.
isUpperCase	**boolean**	**char**	Returns **true** if the given **char** is an upper case letter, or **false** otherwise.

Standard API: `Character`

Public method interfaces for class `Character` (some of them).

Method	Return	Arguments	Description
`toLowerCase`	**`char`**	**`char`**	Returns the lower case equivalent of the given **`char`** if it is an upper case letter, or the given **`char`** if it is not.[a]
`toUpperCase`	**`char`**	**`char`**	Returns the upper case equivalent of the given **`char`** if it is a lower case letter, or the given **`char`** if it is not.[a]

[a] For maximum portability of code to different regions of the world, it is better to use the `String` versions of these methods.

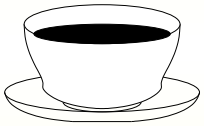
Age next year with exception avoidance

```
001: // Gets current age from first argument, and reports age next year.
002: // Gives an error message if age is not a valid number.
003: public class AgeNextYear
004: {
005:   // Returns true if and only if given string is all digits and not empty.
006:   private static boolean isNonEmptyDigits(String shouldBeDigits)
007:   {
008:     boolean okaySoFar = shouldBeDigits.length() != 0;
009:     int index = 0;
010:     while (okaySoFar && index < shouldBeDigits.length())
011:     {
012:       okaySoFar = Character.isDigit(shouldBeDigits.charAt(index));
013:       index++;
014:     } // while
015:     return okaySoFar;
016:   } // isNonEmptyDigits
```

Age next year with exception avoidance

```
019:   // Check argument and compute result or report error.
020:   public static void main(String[] args)
021:   {
022:     if (args.length > 0 && isNonEmptyDigits(args[0]))
023:     {
024:       int ageNow = Integer.parseInt(args[0]);
025:       int ageNextYear = ageNow + 1;
026:
027:       System.out.println("Your age now is " + ageNow);
028:       System.out.println("Your age next year will be " + ageNextYear);
029:     } // if
030:     else
031:       System.out.println("Please supply your age, as a whole number.");
032:   } // main
033:
034: } // class AgeNextYear
```

Age next year with exception avoidance



Coffee time: What would happen if we swapped the order of the **con-junct**s in the if else statement condition above?

Trying it

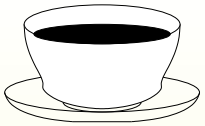
Console Input / Output

```
$ java AgeNextYear
Please supply your age, as a whole number.
$ java AgeNextYear ""
Please supply your age, as a whole number.
$ java AgeNextYear 4.25
Please supply your age, as a whole number.
$ _
```

Run

- Program robust against **exception**s, (or is it?) but
  - code has doubled in size
  - our checks also being done
    by parts that caused exceptions in first place!

Trying it



Coffee time: Worse still, we haven't even avoided all possible exceptions – what command line argument could we present that passes our test and yet still causes `Integer.parseInt()` to **throw** a `NumberFormatException`?

Coursework: `FishTankVolume` exception avoidance

(Summary only)

Take a program you have seen before and make it avoid **exception**s.

Section 4

Example:
Age next year with exception catching

Aim

AIM: To introduce **exception catching** using the **try statement**. We also take a look at **standard error**.

Age next year with exception catching

- Better approach than trying to avoid **exception**s:
  - allow them to happen
  - but **catch** them
    - simpler code
    - less duplication of checks.
- We use this idea here.
- Also have some error messages go to **standard error**.

Operating environment: standard error

- Programs have **standard output** and **standard input**.
- Also **standard error**.
  - intended for output about errors.
- E.g. might redirect standard output to **file**
  - and standard error to different one, etc..

Standard API: `System`: `err.println()`

- `java.lang.System` has **class variable**s called `out` and `in`.
- Also one called `err`
  - contains **reference** to **object** representing **standard error**.
- So we have
  - `System.err.println()`
  - `System.err.print()`
  - `System.err.printf()`

Statement: try statement

- The **try statement** implements **exception catching**.
- E.g.

```
try
{
  ... Code here that might cause an exception to happen.
} // try
catch (Exception exception)
{
  ... Code here to deal with the exception.
} // catch
```

- Two parts – **try block** and **catch clause**.
  - (N.B. – bodies must be **compound statement**s. . . .)

Statement: try statement

- Try block obeyed as usual.
- If **exception** occurs
  - **instance** of `java.lang.Exception` created
  - control passed to catch clause.
  - `Exception` **object** is **exception parameter**
    - like **method parameter**
    - thus declare name (and **type**) for exception after reserved word **`catch`**.
- E.g. **method** to compute mean average of **`int`** **array**...

Statement: try statement

```
private double average(int[] anArray)
{
  try
  {
    int total = anArray[0];
    for (int i = 1; i < anArray.length; i++)
      total += anArray[i];
    return total / (double) anArray.length;
  } // try
  catch (Exception exception)
  {
    // Report the exception and carry on.
    System.err.println(exception);
    return 0;
  } // catch
} // average
```

Exception: `getMessage()`

- An **instance** of `java.lang.Exception`, when created may be given text message describing reason for the error.
- Can be retrieved via `getMessage()` **instance method**.

Age next year with exception catching

- Decide to report error messages to standard output, but also report exception itself to standard error.

```
001: // Gets current age from first argument, and reports age next year.
002: // Gives an error message if age is not a valid number.
003: public class AgeNextYear
004: {
005:   public static void main(String[] args)
006:   {
007:     try
008:     {
009:       int ageNow = Integer.parseInt(args[0]);
010:       int ageNextYear = ageNow + 1;
011:
012:       System.out.println("Your age now is " + ageNow);
013:       System.out.println("Your age next year will be " + ageNextYear);
014:     } // try
```

Age next year with exception catching

```
015:    catch (Exception exception)
016:    {
017:      System.out.println("Please supply your age, as a whole number.");
018:      System.out.println("Exception message was: '"
019:                         + exception.getMessage() + "'");
020:      System.err.println(exception);
021:    } // catch
022:  } // main
023:
024: } // class AgeNextYear
```

Trying it

Console Input / Output

```
$ java AgeNextYear
Please supply your age, as a whole number.
Exception message was: ‘0’
java.lang.ArrayIndexOutOfBoundsException: 0
$ java AgeNextYear ""
Please supply your age, as a whole number.
Exception message was: ‘For input string: ""’
java.lang.NumberFormatException: For input string: ""
$ java AgeNextYear 4.25
Please supply your age, as a whole number.
Exception message was: ‘For input string: "4.25"’
java.lang.NumberFormatException: For input string: "4.25"
$ _
```

Run

Trying it

- Now redirect **standard output** to `/dev/null`.

Console Input / Output

```
$ java AgeNextYear > /dev/null
java.lang.ArrayIndexOutOfBoundsException: 0
$ java AgeNextYear "" > /dev/null
java.lang.NumberFormatException: For input string: ""
$ java AgeNextYear 4.25 > /dev/null
java.lang.NumberFormatException: For input string: "4.25"
$ _
```

Run

Trying it

- Now redirect **standard error** to `/dev/null`.

Console Input / Output

```
$ java AgeNextYear 2> /dev/null
Please supply your age, as a whole number.
Exception message was: '0'
$ java AgeNextYear "" 2> /dev/null
Please supply your age, as a whole number.
Exception message was: 'For input string: ""'
$ java AgeNextYear 4.25 2> /dev/null
Please supply your age, as a whole number.
Exception message was: 'For input string: "4.25"'
$ _
```

Run

- Ideally would like to give different error messages for different kinds of error. . . .

Coursework: `FishTankVolume` exception catching

(Summary only)

Take a program you have seen before and make it **catch exceptions**.

Section 5

Example:

Age next year with multiple exception catching

Aim

AIM: To observe that there are many kinds of **exception** and introduce the idea of multiple **exception catching** by having a **try statement** with many **catch clause**s.

Age next year with multiple exception catching

- Improve `AgeNextYear`
  - give user different error messages
    for the two different causes of **exception**.
  - Java has many kinds of exception. . . .

Exception: there are many types of exception

- `java.lang.Exception` is general model of **exception**s
  - also many classes for more specific kinds of error.
- E.g.

Exception class	Example use
`ArrayIndexOutOfBoundsException`	When some code tries to access an **array element** using an **array index** which is not in the range of the **array** being indexed.
`IllegalArgumentException`	When a **method** is passed a **method argument** which is inappropriate in some way.

Exception: there are many types of exception

Exception class	Example use
`NumberFormatException`	In the `parseInt()` method of the `java.lang.Integer` class when it is asked to interpret an invalid `String` method argument as an **`int`**. (Actually, `NumberFormatException` is a particular kind of the more general `IllegalArgumentException`.)
`ArithmeticException`	When an **integer division** has a denominator which is zero.
`NullPointerException`	When we have code that tries to access the **object referenced** by a **variable**, but the variable actually contains the **null reference**.

Statement: try statement: with multiple catch clauses

- A **try statement** may have more than one **catch clause**
  - each for **catch**ing different kind of **exception**.
- When exception occurs in **try block**
  - execution transfers to first matching catch clause
  - or out of try statement if no matching one.
- E.g. . . .

Statement: try statement: with multiple catch clauses

- If array empty: get `ArrayIndexOutOfBoundsException`
  or an array element is not **`int`** representation: get `NumberFormatException`.

```
private int maximum(String[] anArray)
{
  try
  {
    int maximumSoFar = Integer.parseInt(anArray[0]);
    for (int i = 1; i < anArray.length; i++)
    {
      int thisNumber = Integer.parseInt(anArray[i]);
      if (thisNumber > maximumSoFar)
        maximumSoFar = thisNumber;
    } // for
    return maximumSoFar;
  } // try
```

Statement: try statement: with multiple catch clauses

```
    catch(NumberFormatException exception)
    {
      System.err.println("Cannot parse item as an int: "
                         + exception.getMessage());
      return 0;
    } // catch
    catch(ArrayIndexOutOfBoundsException exception)
    {
      System.err.println("There is no maximum, as there are no numbers!");
      return 0;
    } // catch
  } // maximum
```

Statement: try statement: with multiple catch clauses

- But what if **method argument** is **null reference**?
  - Get `NullPointerException`

```
int maximumSoFar = Integer.parseInt(anArray[0]);
```

- `anArray[0]` means
  - "follow reference in `anArray` to array referenced by it
  - then get value stored at **array index** `0`."
- We have no catch clause matching `NullPointerException`
  - execution transfers out of try statement altogether
  - and out of the method.

Statement: try statement: with multiple catch clauses

- If **method call** was inside following try statement `NullPointerException` would get caught there.

```
try
{
  int max = maximum(null);
  ...
} // try
catch (NullPointerException exception)
{
  System.err.println("Silly me!");
} // catch
```

Age next year with multiple exception catching

- New `AgeNextYear` has catch clause for each exception we expect to get
  - also general one to **catch** any other exceptions
    - makes program robust against overlooking other sources of errors.

```
001: // Gets current age from first argument, and reports age next year.
002: // Gives an error message if age is not a valid number.
003: public class AgeNextYear
004: {
005:   public static void main(String[] args)
006:   {
007:     try
008:     {
009:       int ageNow = Integer.parseInt(args[0]);
010:       int ageNextYear = ageNow + 1;
011:
012:       System.out.println("Your age now is " + ageNow);
013:       System.out.println("Your age next year will be " + ageNextYear);
014:     } // try
```

Age next year with multiple exception catching

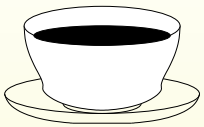
```
015:    catch (ArrayIndexOutOfBoundsException exception)
016:    {
017:      System.out.println("Please supply your age.");
018:      System.err.println(exception);
019:    } // catch
020:    catch (NumberFormatException exception)
021:    {
022:      System.out.println("Your age must be a whole number!");
023:      System.out.println("Exception message was: ‘"
024:                         + exception.getMessage() + "’");
025:      System.err.println(exception);
026:    } // catch
```

Age next year with multiple exception catching

```
027:    // Other exceptions should not happen,
028:    // but we catch anything else, lest we have overlooked something.
029:    catch (Exception exception)
030:    {
031:      System.out.println("Something unforeseen has happened. :-(");
032:      System.out.println("Exception message was: ‘"
033:                         + exception.getMessage() + "’");
034:      System.err.println(exception);
035:    } // catch
036:  } // main
037:
038: } // class AgeNextYear
```

Age next year with multiple exception catching

Coffee time: How can we test the third catch clause in the code above? For example, could we create a `NullPointerException` somehow? Would that need us to alter the code of the program, just for that test, or is there a way we could test the code without altering it? (Hint: think how you could get the **main method** to be given the **null reference** as its **method argument**, using a different **class**.)



Trying it

Console Input / Output

```
$ java AgeNextYear
Please supply your age.
java.lang.ArrayIndexOutOfBoundsException: 0
$ java AgeNextYear ""
Your age must be a whole number!
Exception message was: 'For input string: ""'
java.lang.NumberFormatException: For input string: ""
$ java AgeNextYear 4.25
Your age must be a whole number!
Exception message was: 'For input string: "4.25"'
java.lang.NumberFormatException: For input string: "4.25"
$ _
```

Run

Coursework: `FishTankVolume` multiple exception catching

(Summary only)

Take a program you have seen before and make it **catch** multiple **exception**s.

Section 6

Example:
Age next year throwing an exception

Aim

AIM: To introduce the idea of creating an **exception** and **throw**ing an exception when we have detected a problem, using the **throw statement**.

Age next year throwing an exception

- Still haven't dealt with the other erroneous conditions
  - negative age
  - more than one **command line argument**.
- Cause inappropriate behaviour rather than **exception**s
  - deal with in same way as others:
    * create **instance**s of `Exception`!

Exception: creating exceptions

- `java.lang.Exception` has number of **constructor method**s
  - one takes no **method argument**s
    - `Exception` with no associated message.
  - one takes `String` message.
- Other kinds of **exception**
  - (`ArrayIndexOutOfBoundsException`, `IllegalArgumentException`, `NumberFormatException`, `ArithmeticException` and `NullPointerException`, etc.)

  also have these two constructor methods.

Statement: throw statement

- The **throw statement** used when wish to trigger **exception** mechanism
  - **reserved word** `throw`
  - followed by **reference** to `Exception` **object**.
- Java **virtual machine** finds closest **try statement** currently being executed
  - with matching **catch clause**
  - transfers execution to that catch clause.
- If no matching clause found
  - exception reported
  - **thread** terminated.

Statement: throw statement

- E.g.:

```
throw new Exception();
```

- E.g. with message:

```
throw new Exception("This is the message associated with the exception");
```

- E.g.

```
NumberFormatException exception
  = new NumberFormatException("Only digits please");
throw exception;
```

Age next year throwing an exception

- New `AgeNextYear` throws
  - `ArrayIndexOutOfBoundsException` if too many arguments
  - `NumberFormatException` if age negative.
- Catches them with corresponding **catch clause**.

Age next year throwing an exception

```
001: // Gets current age from first argument, and reports age next year.
002: // Gives an error message if age is not a valid number.
003: public class AgeNextYear
004: {
005:   public static void main(String[] args)
006:   {
007:     try
008:     {
009:       int ageNow = Integer.parseInt(args[0]);
010:       if (args.length > 1)
011:         throw new ArrayIndexOutOfBoundsException
012:                     ("You have supplied " + args.length + " arguments!");
013:       if (ageNow < 0)
014:         throw new NumberFormatException
015:                     ("Your age of " + ageNow + " is negative!");
016:
017:       int ageNextYear = ageNow + 1;
018:       System.out.println("Your age now is " + ageNow);
019:       System.out.println("Your age next year will be " + ageNextYear);
020:     } // try
```

Age next year throwing an exception

```
021:     catch (ArrayIndexOutOfBoundsException exception)
022:     {
023:       System.out.println("Please supply your age, and nothing else.");
024:       System.out.println("Exception message was: ‘"
025:                          + exception.getMessage() + "’");
026:       System.err.println(exception);
027:     } // catch
028:     catch (NumberFormatException exception)
029:     {
030:       System.out.println("Your age must be a non-negative whole number!");
031:       System.out.println("Exception message was: ‘"
032:                          + exception.getMessage() + "’");
033:       System.err.println(exception);
034:     } // catch
```

Age next year throwing an exception

```
035:     // Other exceptions should not happen,
036:     // but we catch anything else, lest we have overlooked something.
037:     catch (Exception exception)
038:     {
039:       System.out.println("Something unforeseen has happened. :-(");
040:       System.out.println("Exception message was: ‘"
041:                          + exception.getMessage() + "’");
042:       System.err.println(exception);
043:     } // catch
044:   } // main
045:
046: } // class AgeNextYear
```

Trying it

Console Input / Output

```
$ java AgeNextYear 60 4
Please supply your age, and nothing else.
Exception message was: 'You have supplied 2 arguments!'
java.lang.ArrayIndexOutOfBoundsException: You have supplied 2 arguments!
$ java AgeNextYear -4
Your age must be a non-negative whole number!
Exception message was: 'Your age of -4 is negative!'
java.lang.NumberFormatException: Your age of -4 is negative!
$ _
```

Run

Coursework: `FishTankVolume` throwing exceptions

(Summary only)

Take a program you have seen before and make it **throw** its own **exception**s and **catch** them.

Section 7

Example: Single times table with exception catching

Aim

AIM: To illustrate the use of **exception catching** in **graphical user interface** (**GUI**) programs.

Single times table with exception catching

- `TimesTable` **GUI**
  - deal with user entering multiplier which is not **`int`** representation.
- Previous version
  - **throw** an **exception** in `parseInt()` during `actionPerformed()`
  - caught by **GUI event thread**
    - report on **standard error**
    - go back to sleep: wait for more GUI **event**s.
- We shall **catch** exception within `actionPerformed()`
  - report error message in the results `JTextArea`.

Single times table with exception catching

```
001: import java.awt.BorderLayout;
002: import java.awt.Container;
003: import java.awt.event.ActionEvent;
004: import java.awt.event.ActionListener;
005: import javax.swing.JButton;
006: import javax.swing.JFrame;
007: import javax.swing.JTextArea;
008: import javax.swing.JTextField;
009:
010: // Program to show a times table for a multiplier chosen by the user.
011: public class TimesTable extends JFrame implements ActionListener
012: {
013:   // A text field for the user to enter the multiplier.
014:   private final JTextField multiplierJTextField = new JTextField(5);
015:
016:   // A text area for the resulting times table, 15 lines of 20 characters.
017:   private final JTextArea displayJTextArea = new JTextArea(15, 20);
018:
019:
```

Single times table with exception catching

```
020:   // Constructor.
021:   public TimesTable()
022:   {
023:     setTitle("Times Table");
024:
025:     Container contents = getContentPane();
026:     contents.setLayout(new BorderLayout());
027:
028:     contents.add(multiplierJTextField, BorderLayout.NORTH);
029:     contents.add(displayJTextArea, BorderLayout.CENTER);
030:
031:     JButton displayJButton = new JButton("Display");
032:     contents.add(displayJButton, BorderLayout.SOUTH);
033:     displayJButton.addActionListener(this);
034:
035:     setDefaultCloseOperation(EXIT_ON_CLOSE);
036:     pack();
037:   } // TimesTable
```

Single times table with exception catching

```
040:   // Act upon the button being pressed.
041:   public void actionPerformed(ActionEvent event)
042:   {
043:     try
044:     {
045:       // Empty the text area to remove any previous result.
046:       displayJTextArea.setText("");
047:
048:       int multiplier = Integer.parseInt(multiplierJTextField.getText());
049:
050:       displayJTextArea.append("--------------------------------\n");
051:       displayJTextArea.append("| Times table for " + multiplier + "\n");
052:       displayJTextArea.append("--------------------------------\n");
053:       for (int thisNumber = 1; thisNumber <= 10; thisNumber++)
054:         displayJTextArea.append("| " + thisNumber + " x " + multiplier
055:                                 + " = " + thisNumber * multiplier + "\n");
056:       displayJTextArea.append("--------------------------------\n");
057:     } // try
```

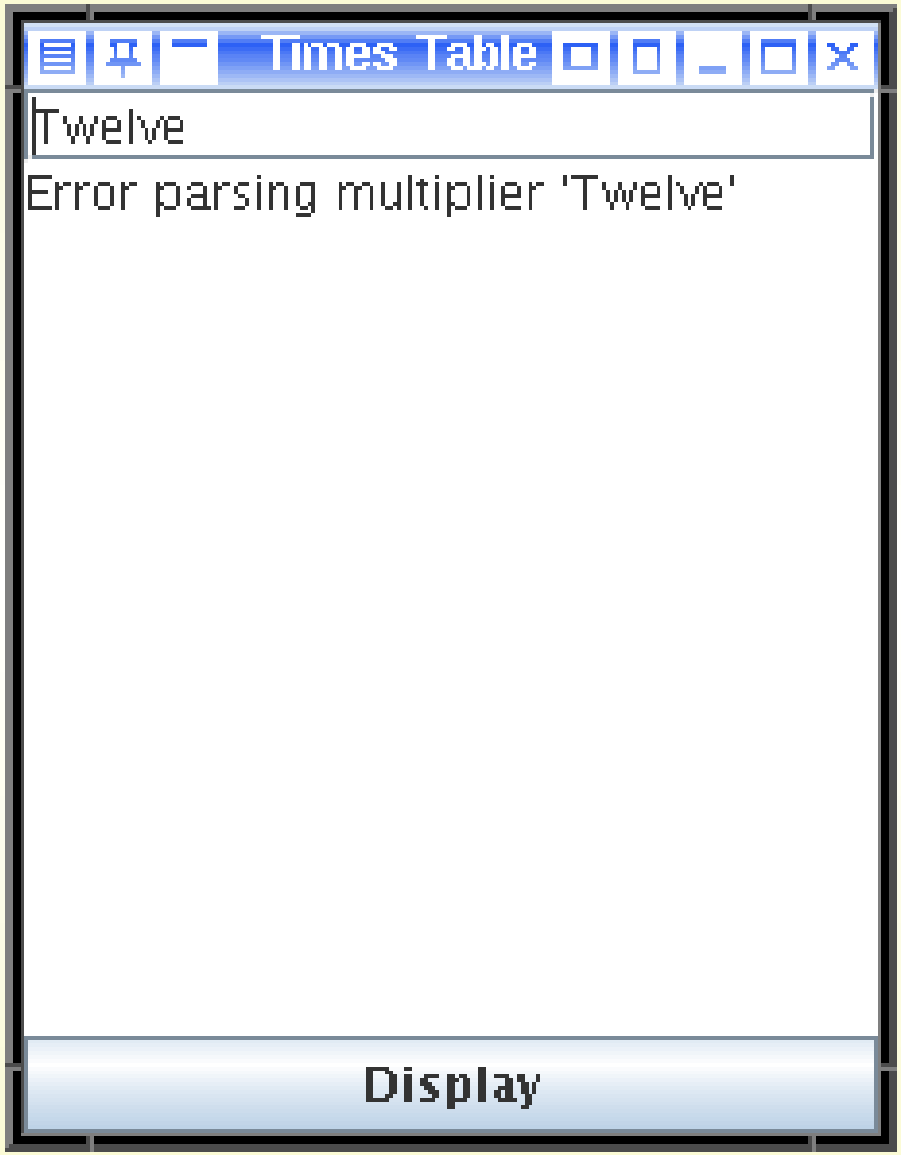
Single times table with exception catching

```
058:    catch (NumberFormatException exception)
059:    {
060:      displayJTextArea.setText("Error parsing multiplier '"
061:                               + multiplierJTextField.getText() + "'");
062:    } // catch
063:  } // actionPerformed
```

- The **main method** is the same as before.

```
066:  // Create a TimesTable and make it appear on the screen.
067:  public static void main(String[] args)
068:  {
069:    TimesTable theTimesTable = new TimesTable();
070:    theTimesTable.setVisible(true);
071:  } // main
072:
073: } // class TimesTable
```

Trying it



Coffee time: What would we do if there was no handy place in our **GUI** to display our error message? How easy would it be for us to make a separate window appear in which we display the error?

Coursework: `TimesTable` with a `ScrollPane` catching exceptions

(Summary only)

Take a program with a **GUI**, that you have seen before, and make it **catch exception**s.

Section 8

Example:
A reusable `Date` class with exceptions

Aim

AIM: To introduce the **throws clause** together with its associated **doc comment tag**. We also look at supplying an **exception cause** when we create an **exception**, and discuss the use of `RuntimeExceptions`.

A reusable Date class with exceptions

- Improve our reusable Date by adding **exception**s.

```
001: /**
002:  * This class represents calendar dates and provides certain
003:  * manipulations of them.
004:  *
005:  * @author John Latham
006:  */
007: public class Date
008: {
009:   // Class variable to hold the present date.
010:   private static Date presentDate = null;
```

- setPresentDate() will now **throw** an exception if called more than once.

Method: that throws an exception

- If body of **method** can cause an **exception**
  - either directly or indirectly

  which is not caught by it
  - then method must have a **throws clause** in heading.
- Write **reserved word `throws`** followed by kind(s) of exception
- E.g. `charAt()` of `java.lang.String`
  - **throws** an exception if illegal **string index**.

```
public char charAt(int index) throws IndexOutOfBoundsException
{
  ...
} // charAt
```

Method: that throws an exception

- Suppose we have a **class** which provides **mutable objects**
  - representing customer details.
- An **instance** is allowed to have customer name changed
  - but new name not allowed to be empty.

```
public class Customer
{
  private String familyName, firstNames;
  ...
```

Method: that throws an exception

```
public void setName(String requiredFamilyName, String requiredFirstNames)
            throws IllegalArgumentException
{
  if (requiredFamilyName == null || requiredFirstNames == null
      || requiredFamilyName.equals("") || requiredFirstNames.equals(""))
    throw new IllegalArgumentException("Name cannot be null or empty");

  familyName = requiredFamilyName;
  firstNames = requiredFirstNames;
} // setName
...
} // class Customer
```

Java tools: javadoc: throws tag

- Another **doc comment tag**
  - for describing **exception**s that a **method throws**.

Tag	Meaning	Where used
`@throws` exception name and description	Describes the circumstances leading to an exception.	Before a method.

A reusable Date class with exceptions

```
013:  /**
014:   * Sets the present date.
015:   * The date must not have already been set.
016:   *
017:   * @param requiredPresentDate The required date for the present day.
018:   *
019:   * @throws Exception if present date has already been set
020:   *                   or if given date is null.
021:   */
022:  public static void setPresentDate(Date requiredPresentDate) throws Exception
023:  {
024:    if (requiredPresentDate == null)
025:      throw new Exception("Present date cannot be set to null");
026:    if (presentDate != null)
027:      throw new Exception("Present date has already been set");
028:    presentDate = requiredPresentDate;
029:  } // setPresentDate
```

A reusable `Date` class with exceptions

- Resulting **API** documentation:

Web Browser Window

setPresentDate

```
public static void setPresentDate(Date requiredPresentDate)
                           throws java.lang.Exception
```

Sets the present date. The date must not have already been set.

Parameters:

requiredPresentDate - The required date for the present day.

Throws:

java.lang.Exception - if present date has already been set or if given date is null.

Run

A reusable Date class with exceptions

```
032:   /**
033:    * Gets the present date.
034:    *
035:    * @return The present date.
036:    *
037:    * @throws Exception if present date has not been set.
038:    */
039:   public static Date getPresentDate() throws Exception
040:   {
041:     if (presentDate == null)
042:       throw new Exception("Present date has not been set");
043:     return presentDate;
044:   } // getPresentDate
```

A reusable Date class with exceptions

```
047:   // Instance variables: the day, month and year of a date.
048:   private final int day, month, year;
```

- Previous version **constructor method** 'corrected' illegal date values
  - e.g. if day was zero or negative, was set to one.
- Here we throw exception instead.
- Also, our leap year calculation only works for dates after 1753....

```
051:   /**
052:    * Constructs a date, given the three int components.
053:    *
054:    * @param requiredDay The required day.
055:    * @param requiredMonth The required month.
056:    * @param requiredYear The required year.
057:    *
```

A reusable Date class with exceptions

```
058:    * @throws Exception if the date components do not form a legal date since
059:    *                   the start of 1753 (post Gregorian Reformation).
060:    */
061:   public Date(int requiredDay, int requiredMonth, int requiredYear)
062:         throws Exception
063:   {
064:     year = requiredYear;
065:     month = requiredMonth;
066:     day = requiredDay;
067:     // Now check these components are legal, throw exception if not.
068:     checkDateIsLegal();
069:   } // Date
```

- checkDateIsLegal() just checks, and throws Exception if date is not legal. . . .

A reusable Date class with exceptions

```
072:  // Check legality of date components and throw exception if illegal.
073:  private void checkDateIsLegal() throws Exception
074:  {
075:    if (year < 1753)
076:      throw new Exception("Year " + year + " must be >= 1753");
077:
078:    if (month < 1 || month > 12)
079:      throw new Exception("Month " + month + " must be from 1 to 12");
080:
081:    if (day < 1 || day > daysInMonth())
082:      throw new Exception("Day " + day + " must be from 1 to " + daysInMonth()
083:                          + " for " + month + "/" + year);
084:  } // checkDateIsLegal
```

A reusable `Date` class with exceptions

- If `checkDateIsLegal()` throws exception
  - it will continue to be thrown by constructor
    - constructor does not **catch** it.

A reusable `Date` class with exceptions

- New version has second constructor
  - takes `String` representation of date
    - e.g. `"01/07/2019"`.
- Use `split()` to split string in to three **`int`** values.
- Splitting may fail – e.g. less than 3 values, or not an **`int`** representation.
  - would result in `ArrayIndexOutOfBoundsException` or `NumberFormatException`.
- Catch such 'low level' exceptions
  - throw **new** `Exception` which is more meaningful.
  - new `Exception` caused by the one we caught....

Exception: creating exceptions: with a cause

- Two more **constructor method** in `java.lang.Exception`
  - create **instance** which has another **exception** that caused it
    - with or without a message.
- Many other kinds of exception also have these.

A reusable Date class with exceptions

```
087:  /**
088:   * Constructs a date, given a String holding the
089:   * day/month/year representation of the date.
090:   *
091:   * @param dateString The required date as day/month/year.
092:   *
093:   * @throws Exception if dateString is not legal.
094:   */
095:  public Date(String dateString) throws Exception
096:  {
097:    try
098:    {
099:      String[] dateElements = dateString.split("/");
100:      if (dateElements.length > 3)
101:        // This exception will be caught below.
102:        throw new Exception("Too many date elements");
```

A reusable Date class with exceptions

```
103:      day = Integer.parseInt(dateElements[0]);
104:      month = Integer.parseInt(dateElements[1]);
105:      year = Integer.parseInt(dateElements[2]);
106:    } // try
107:    catch (Exception exception)
108:      { throw new Exception("Date '" + dateString
109:                              + "' is not in day/month/year format",
110:                            exception); }
111:    // If we get to here, we just check the date components are legal.
112:    checkDateIsLegal();
113:  } // Date
```

Coffee time: What if the **method argument** passed to this new constructor method is the **null reference**? Have we overlooked that scenario?

A reusable Date class with exceptions

```
116:   /**
117:    * Yields the day component of this date.
118:    *
119:    * @return The day of this date.
120:    */
121:   public int getDay()
122:   {
123:     return day;
124:   } // getDay
125:
126:
127:   /**
128:    * Yields the month component of this date.
129:    *
130:    * @return The month of this date.
131:    */
132:   public int getMonth()
133:   {
134:     return month;
135:   } // getMonth
136:
137:
```

A reusable Date class with exceptions

```
138:   /**
139:    * Yields the year component of this date.
140:    *
141:    * @return The year of this date.
142:    */
143:   public int getYear()
144:   {
145:     return year;
146:   } // getYear
147:
148:
149:   /**
150:    * Provides the day/month/year representation of this date.
151:    *
152:    * @return A String day/month/year representation of this date.
153:    */
154:   public String toString()
155:   {
156:     return day + "/" + month + "/" + year;
157:   } // toString
```

A reusable `Date` class with exceptions

- Methods that compare with another date
  - might be given **null reference** for other date
  - produce `NullPointerException`
    - particular kind of more general `RuntimeException`.

Method: that throws an exception: `RuntimeException`

- All **exception**s that *possibly* can be **throw**n when running body of a **method**
  - must either be caught by it
  - or declared in its **throws clause**.
- Java relaxes this rule for `RuntimeExceptions`
  - common erroneous situations which are usually avoidable
  - typically write code to ensure they do not happen.
- `java.lang.RuntimeException` is kind of `Exception`
  - more specific kinds of `RuntimeException` include
    - `java.lang.ArrayIndexOutOfBoundsException`
    - `java.lang.IllegalArgumentException`
    - `java.lang.NumberFormatException`
    - `java.lang.ArithmeticException`
    - `java.lang.NullPointerException`.

Method: that throws an exception: `RuntimeException`

- Would be very inconvenient to *have* to always declare these might happen, or explicitly **catch** them
  - when *we* know they will not happen due to way we have written the code.
- So Java lets us choose to declare whether they might be thrown by a method.

Method: that throws an exception: `RuntimeException`

- E.g. in the following: **array reference** and (implicit) **array element** access
  - could give `NullPointerException` and `ArrayIndexOutOfBoundsException`
  - **compiler** not clever enough to reason whether *actually* can occur
    - but we can be sure they won't, so no **throws clause**.

```
public int sum(int[] array)
{
  if (array == null)
    return 0;

  int sum = 0;
  for (int element : array)
    sum += element;
  return sum;
} // sum
```

Method: that throws an exception: `RuntimeException`

- E.g. following method *can* cause some kinds of `RuntimeException`
  - we don't check `array` is not **`null`**
  - nor array is not empty.

```
public double mean(int[] array)
                throws NullPointerException, ArrayIndexOutOfBoundsException
{
  int sum = array[0];
  for (int index = 1; index < array.length; index++)
    sum += array[index];
  return (double)sum / array.length;
} // sum
```

Method: that throws an exception: `RuntimeException`

- For code intended for **software reuse**
  - good idea to be disciplined.
- If method can throw some kind of `RuntimeException`, because
  - does not avoid possibility
  - or even, explicitly throws such exception

  should declare in **throws clause**
  - even though not forced to.

Method: that throws an exception: `RuntimeException`

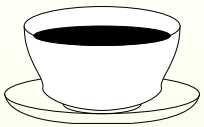
- Kinds of exception for which we *must* either
  - have **catch clause** for
  - or list in **throws clause**

  known as **checked exception**s.
- Those for which rule is relaxed
  - e.g. `RuntimeException` and its specific kinds

  known as **unchecked exception**s.

A reusable `Date` class with exceptions

Coffee time: Why have we been able to get so far through this book without needing to write the **reserved word `throws`** in our programs (except when using a `Scanner` on a **file**)? Now that you know about it, can you think of places where we might include it if we were writing all those programs again?



A reusable Date class with exceptions

```
160:   /**
161:    * Compares this date with a given other one.
162:    *
163:    * @param other The other date to compare with.
164:    *
165:    * @return The value 0 if the other date represents the same date
166:    * as this one; a value less than 0 if this date is less than the
167:    * other; and a value greater than 0 if this date is greater than
168:    * the other.
169:    *
170:    * @throws NullPointerException if other is null.
171:    */
172:   public int compareTo(Date other) throws NullPointerException
173:   {
174:     if (year != other.year)        return year - other.year;
175:     else if (month != other.month) return month - other.month;
176:     else                           return day - other.day;
177:   } // compareTo
```

A reusable `Date` class with exceptions

Web Browser Window

compareTo

```
public int compareTo(Date other)
            throws java.lang.NullPointerException
```

Compares this date with a given other one.

Parameters:

other - The other date to compare with.

Returns:

The value 0 if the other date represents the same date as this one; a value less than 0 if this date is less than the other; and a value greater than 0 if this date is greater than the other.

Throws:

java.lang.NullPointerException - if other is null.

Run

A reusable Date class with exceptions

```
180:   /**
181:    * Compares this date with a given other one, for equality.
182:    *
183:    * @param other The other date to compare with.
184:    *
185:    * @return true if and only if they represent the same date.
186:    *
187:    * @throws NullPointerException if other is null.
188:    */
189:   public boolean equals(Date other) throws NullPointerException
190:   {
191:     return compareTo(other) == 0;
192:   } // equals
193:
194:
```

A reusable Date class with exceptions

```
195:   /**
196:    * Compares this date with a given other one, for less than.
197:    *
198:    * @param other The other date to compare with.
199:    *
200:    * @return true if and only if this date is less than the other.
201:    *
202:    * @throws NullPointerException if other is null.
203:    */
204:   public boolean lessThan(Date other) throws NullPointerException
205:   {
206:     return compareTo(other) < 0;
207:   } // lessThan
208:
209:
```

A reusable Date class with exceptions

```
210:   /**
211:    * Compares this date with a given other one, for greater than.
212:    *
213:    * @param other The other date to compare with.
214:    *
215:    * @return true if and only if this date is greater than the other.
216:    *
217:    * @throws NullPointerException if other is null.
218:    */
219:   public boolean greaterThan(Date other) throws NullPointerException
220:   {
221:     return compareTo(other) > 0;
222:   } // greaterThan
```

A reusable `Date` class with exceptions

- Interesting twist for `addDay()`
  - creates a new `Date`: constructor can throw an exception
    - so `addDay()` must catch or throw it.
- We know newly created `Date` cannot be erroneous
  - but still have to explicitly catch exception!

A reusable Date class with exceptions

```
225:   /**
226:    * Constructs a new date which is one day later than this one.
227:    *
228:    * @return A new date which is one day later than this one.
229:    */
230:   public Date addDay()
231:   {
232:     int newDay = day + 1;
233:     int newMonth = month;
234:     int newYear = year;
235:     if (newDay > daysInMonth())
236:     {
237:       newDay = 1;
238:       newMonth++;
239:       if (newMonth > 12)
240:       {
241:         newMonth = 1;
242:         newYear++;
243:       } // if
244:     } // if
```

A reusable `Date` class with exceptions

```
245:    // This cannot cause an exception, but Java does not know that.
246:    try { return new Date(newDay, newMonth, newYear); }
247:    catch (Exception exception) { return null; }
248:  } // addDay
```

Coffee time: What if we had decided that the constructor should throw a `RuntimeException` rather than an `Exception`. Would that have made a difference to us here?

A reusable Date class with exceptions

```
251:   /**
252:    * Constructs a new date which is one month later than this one.
253:    * If the day is too large for that month, it is truncated to
254:    * the number of days in that month.
255:    *
256:    * @return A new date which is one month later than this one.
257:    */
258:   public Date addMonth()
259:   {
260:     int newDay = day;
261:     int newMonth = month + 1;
262:     int newYear = year;
263:     if (newMonth > 12)
264:     {
265:       newMonth = 1;
266:       newYear++;
267:     } // if
```

A reusable Date class with exceptions

```
268:    if (newDay > daysInMonth(newMonth, newYear))
269:      newDay = daysInMonth(newMonth, newYear);
270:    // This cannot cause an exception, but Java does not know that.
271:    try { return new Date(newDay, newMonth, newYear); }
272:    catch (Exception exception) { return null; }
273:  } // addMonth
```

A reusable Date class with exceptions

```
276:  /**
277:   * Constructs a new date which is one year later than this one.
278:   * If this date is a leap day, it returns 28th February of the next year.
279:   *
280:   * @return A new date which is one year later than this one.
281:   */
282:  public Date addYear()
283:  {
284:    // This cannot cause an exception, but Java does not know that.
285:    try
286:    {
287:      if (day == 29 && month == 2)
288:        return new Date(28, month, year + 1);
289:      else
290:        return new Date(day, month, year + 1);
291:    } // try
292:    catch (Exception exception) { return null; }
293:  } // addYear
```

A reusable Date class with exceptions

```
296:   /**
297:    * Constructs a new date which is one day earlier than this one.
298:    * This can throw an exception
299:    * if the new date is earlier than the start of 1753.
300:    *
301:    * @return A new date which is one day earlier than this one.
302:    *
303:    * @throws Exception if the new date is earlier than the start of 1753.
304:    */
305:   public Date subtractDay() throws Exception
306:   {
307:     int newDay = day - 1;
308:     int newMonth = month;
309:     int newYear = year;
```

A reusable Date class with exceptions

```
310:     if (newDay < 1)
311:     {
312:       newMonth--;
313:       if (newMonth < 1)
314:       {
315:         newMonth = 12;
316:         newYear--;
317:       } // if
318:       newDay = daysInMonth(newMonth, newYear);
319:     } // if
320:     return new Date(newDay, newMonth, newYear);
321:   } // subtractDay
```

A reusable Date class with exceptions

```
324:   /**
325:    * Constructs a new date which is one month earlier than this one.
326:    * This can throw an exception
327:    * if the new date is earlier than the start of 1753.
328:    * If the day is too large for that month, it is truncated to
329:    * the number of days in that month.
330:    *
331:    * @return A new date which is one month earlier than this one.
332:    *
333:    * @throws Exception if the new date is earlier than the start of 1753.
334:    */
335:   public Date subtractMonth() throws Exception
336:   {
337:     int newDay = day;
338:     int newMonth = month - 1;
339:     int newYear = year;
```

A reusable Date class with exceptions

```
340:     if (newMonth < 1)
341:     {
342:       newMonth = 12;
343:       newYear--;
344:     } // if
345:     if (newDay > daysInMonth(newMonth, newYear))
346:       newDay = daysInMonth(newMonth, newYear);
347:     return new Date(newDay, newMonth, newYear);
348:   } // subtractMonth
```

A reusable Date class with exceptions

```
351:   /**
352:    * Constructs a new date which is one year earlier than this one.
353:    * This can throw an exception
354:    * if the new date is earlier than the start of 1753.
355:    * If this date is a leap day, it returns 28th February of the previous year.
356:    *
357:    * @return A new date which is one year earlier than this one.
358:    *
359:    * @throws Exception if the new date is earlier than the start of 1753.
360:    */
361:   public Date subtractYear() throws Exception
362:   {
363:     if (day == 29 && month == 2)
364:       return new Date(28, month, year - 1);
365:     else
366:       return new Date(day, month, year - 1);
367:   } // subtractYear
```

A reusable Date class with exceptions

```
370:   /**
371:    * Calculates how many days this date is from a given other.
372:    * If the other date is less than this one, then the distance
373:    * is negative. It is non-negative otherwise (including zero
374:    * if they represent the same date).
375:    *
376:    * @param other The other date.
377:    *
378:    * @return The distance in days.
379:    *
380:    * @throws NullPointerException if other is null.
381:    */
382:   public int daysFrom(Date other) throws NullPointerException
383:   {
384:     // The code here is a prototype
385:     // -- the result should be computed more efficiently than this!
```

A reusable Date class with exceptions

```
386:     if (equals(other))
387:       return 0;
388:     else if (lessThan(other))
389:     {
390:       Date someDate = addDay();
391:       int noOfDaysDistance = 1;
392:       while (someDate.lessThan(other))
393:       {
394:         someDate = someDate.addDay();
395:         noOfDaysDistance++;
396:       } // while
397:       return noOfDaysDistance;
398:     } // else if
399:     else
```

A reusable Date class with exceptions

```
400:      try // We should not get an exception from subtractDay,
401:          //  because target date is legal. But Java does not know this.
402:      {
403:        Date someDate = subtractDay();
404:        int noOfDaysDistance = -1;
405:        while (someDate.greaterThan(other))
406:        {
407:          someDate = someDate.subtractDay();
408:          noOfDaysDistance--;
409:        } // while
410:        return noOfDaysDistance;
411:      } // try
412:      // Java does not know we cannot get an exception.
413:      catch (Exception e){ return 0; }
414:    } // daysFrom
```

A reusable Date class with exceptions

- daysInMonth() now both instance method and class method.

```
417:   // Calculate the number of days in the month.
418:   private int daysInMonth()
419:   {
420:     return daysInMonth(month, year);
421:   } // daysInMonth
```

A reusable Date class with exceptions

```
424:  // Number of days in each month for normal and leap years.
425:  // The first index (0) is not used.
426:  private static final int[]
427:    DAYS_PER_MONTH_NON_LEAP_YEAR
428:        //  Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
429:      = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
430:    DAYS_PER_MONTH_LEAP_YEAR
431:      = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
432:
433:
434:  // Calculate the number of days in a given month for a given year.
435:  // This will never be called with a month out of range 1 to 12.
436:  private static int daysInMonth(int month, int year)
437:  {
438:    if (isLeapYear(year)) return DAYS_PER_MONTH_LEAP_YEAR[month];
439:    else                  return DAYS_PER_MONTH_NON_LEAP_YEAR[month];
440:  } // daysInMonth
```

A reusable Date class with exceptions

```
443:   // Return true if and only if year is a leap year.
444:   // (We can ignore pre Gregorian Reformation years.)
445:   // Year is a leap year if it is divisible by 4
446:   //                     and is not divisible by 100
447:   //                         or is divisible by 400.
448:   private static boolean isLeapYear(int year)
449:   {
450:     return year % 4 == 0
451:            && (year % 100 != 0 || year % 400 == 0);
452:   } // isLeapYear
453:
454: } // class Date
```

A reusable `Date` class with exceptions

Coffee time: In this exploration of the `Date` example, we added code to **throw** exceptions which are `Exception` objects. We might instead have chosen to use `RuntimeException` objects. What difference would that make? Which would really be the most appropriate?

Coursework: `Date` class with nested try statements

(Summary only)

Modify a **class** so that it uses **nested try statement**s.

Section 9

Example: Date difference with command line arguments

Aim

AIM: To further illustrate the use of **exception**s and introduce the `getCause()` **instance method** in the `Exception` **class**.

Date difference with command line arguments

- Given two dates as **command line argument**s
  - output number of days between them.
- Next section – same program
  - except **data** from **standard input**.
- Interesting how approach to **exception** handling differs.
- Also show causes of **exception**s. . . .

Exception: `getCause()`

- The **exception cause** inside an `Exception`
  - retrieved via `getCause()` **instance method**
  - **returns null reference** if no cause.

Date difference with command line arguments

```
001: // Obtain two dates in day/month/year format from first and second arguments.
002: // Report how many days there are from first to second,
003: // which is negative if first date is the earliest one.
004: public class DateDifference
005: {
006:   public static void main(String[] args)
007:   {
008:     try
009:     {
010:       // The two dates come from args 0 and 1.
011:       Date date1 = new Date(args[0]);
012:       Date date2 = new Date(args[1]);
013:       if (args.length > 2)
014:         throw new ArrayIndexOutOfBoundsException(args.length + " is > 2");
015:       System.out.println("From " + date1 + " to " + date2 + " is "
016:                          + date1.daysFrom(date2) + " days");
017:     } // try
```

Date difference with command line arguments

```
018:     catch (ArrayIndexOutOfBoundsException exception)
019:     {
020:       System.out.println("Please supply exactly two dates");
021:       System.err.println(exception);
022:       if (exception.getCause() != null)
023:         System.err.println("Caused by: " + exception.getCause());
024:     } // catch
025:     catch (Exception exception)
026:     {
027:       System.out.println(exception.getMessage());
028:       System.err.println(exception);
029:       if (exception.getCause() != null)
030:         System.err.println("Caused by: " + exception.getCause());
031:     } // catch
032:   } // main
033:
034: } // class DateDifference
```

Trying it

- Two legal dates.

Console Input / Output

```
$ java DateDifference 01/07/2018 01/07/2019
From 1/7/2018 to 1/7/2019 is 365 days
$ java DateDifference 01/07/2019 01/07/2018
From 1/7/2019 to 1/7/2018 is -365 days
$ _
```

Run

Trying it

- Test `ArrayIndexOutOfBoundsException` **exception**s.

Console Input / Output

```
$ java DateDifference
Please supply exactly two dates
java.lang.ArrayIndexOutOfBoundsException: 0
$ java DateDifference 01/07/2018
Please supply exactly two dates
java.lang.ArrayIndexOutOfBoundsException: 1
$ java DateDifference 01/07/2018 01/07/2019 ExtraArgument
Please supply exactly two dates
java.lang.ArrayIndexOutOfBoundsException: 3 is > 2
$ _
```

Run

Trying it

- Test invalid date format exceptions.

Console Input / Output

```
$ java DateDifference 01/07/2019 "Hello mum"
Date ‘Hello mum’ is not in day/month/year format
java.lang.Exception: Date ‘Hello mum’ is not in day/month/year format
Caused by: java.lang.NumberFormatException: For input string: "Hello mum"
$ java DateDifference 01/07 "Hello mum"
Date ‘01/07’ is not in day/month/year format
java.lang.Exception: Date ‘01/07’ is not in day/month/year format
Caused by: java.lang.ArrayIndexOutOfBoundsException: 2
$ _
```

Run

Trying it

- Test illegal date exceptions.

Console Input / Output

```
$ java DateDifference 13/07/2019 07/13/2019
Month 13 must be from 1 to 12
java.lang.Exception: Month 13 must be from 1 to 12
$ java DateDifference 01/07/2019 2019/07/01
Year 1 must be >= 1753
java.lang.Exception: Year 1 must be >= 1753
$ java DateDifference 01/07/2019 30/2/2019
Day 30 must be from 1 to 28 for 2/2019
java.lang.Exception: Day 30 must be from 1 to 28 for 2/2019
$ _
```

Run

Section 10

Example: Date difference with standard input

Aim

AIM: To introduce the idea of obtaining possibly erroneous information from the end user on **standard input**, detecting problems with it, and requesting it again until it is acceptable.

Date difference with standard input

```
001: import java.util.Scanner;
002:
003: // Obtain two dates in day/month/year format from the user.
004: // Report how many days there are from first to second,
005: // which is negative if first date is earliest one.
006: public class DateDifference
007: {
008:   public static void main(String[] args)
009:   {
010:     // A scanner for reading from standard input.
011:     Scanner input = new Scanner(System.in);
012:     // The two dates are obtained from the user.
013:     Date date1 = inputDate(input, "first");
014:     Date date2 = inputDate(input, "second");
015:
016:     System.out.println();
017:     System.out.println("From " + date1 + " to " + date2 + " is "
018:                        + date1.daysFrom(date2) + " days");
019:   } // main
```

Date difference with standard input

```
022:  // Obtain a date from the user via the given Scanner.
023:  // The second argument is part of the prompt.
024:  // Keep repeating until user has entered a valid date.
025:  private static Date inputDate(Scanner input, String whichDate)
026:  {
027:    // Result will eventually refer to a legal date.
028:    Date result = null;
029:    System.out.print("Please type the " + whichDate + " date: ");
030:    // Keep trying until we get a legal date.
031:    boolean inputValidYet = false;
```

Date difference with standard input

```
032:     do
033:     {
034:       try
035:       {
036:         result = new Date(input.nextLine());
037:         // If we get here then date was valid.
038:         inputValidYet = true;
039:       } // try
040:       catch (Exception exception)
041:       {
042:         System.out.println(exception.getMessage());
043:         System.out.print("Please re-type the " + whichDate + " date: ");
044:       } // catch
045:     } while (!inputValidYet);
046:     // When we get here the result must be a valid date.
047:     return result;
048:   } // inputDate
049:
050: } // class DateDifference
```

Trying it

Console Input / Output

```
$ java DateDifference
Please type the first date: 01/07/2019
Please type the second date: 01/07/2020

From 1/7/2019 to 1/7/2020 is 366 days
$ java DateDifference
Please type the first date: Umm, err...
Date 'Umm, err...' is not in day/month/year format
Please re-type the first date: Oh, a date!
Date 'Oh, a date!' is not in day/month/year format
Please re-type the first date: 01/07/2019
Please type the second date: Another one?
Date 'Another one?' is not in day/month/year format
Please re-type the second date: 01/07/2020

From 1/7/2019 to 1/7/2020 is 366 days
$ _
```

Run

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.