

20.2 File processing and exception handling

WHAT YOU SHOULD ALREADY KNOW

In [Chapter 10, Section 10.3](#), you learnt about text files, and in [Chapter 13, Section 13.2](#), you learnt about file organisation and access. Review these sections, then try these three questions before you read the second part of this chapter.

- 1 a) Write a program to set up a text file to store records like this, with one record on every line.

```
TYPE
TstudentRecord
    DECLARE name : STRING
    DECLARE address : STRING
    DECLARE className : STRING
ENDTYPE
```

- b) Write a procedure to append a record.
 - c) Write a procedure to find and delete a record.
 - d) Write a procedure to output all the records.
- 2 Describe **three** types of file organisation
 - 3 Describe **two** types of file access and explain which type of files each one is used for.

Key terms

Read – file access mode in which data can be read from a file.

Write – file access mode in which data can be written to a file; any existing data stored in the file will be overwritten.

Append – file access mode in which data can be added to the end of a file.

Open – file-processing operation; opens a file ready to be used in a program.

Close – file-processing operation; closes a file so it can no longer be used by a program.

Exception – an unexpected event that disrupts the execution of a program.

Exception handling – the process of responding to an exception within the program so that the program does not halt unexpectedly.

20.2.1 File processing operations

Files are frequently used to store records that include data types other than string. Also, many programs need to handle random access files so that a record can be found quickly without reading through all the preceding records.

A typical record to be stored in a file could be declared like this in pseudocode:

```
TYPE
TstudentRecord
    DECLARE name : STRING
    DECLARE registerNumber : INTEGER
    DECLARE dateOfBirth : DATE
    DECLARE fullTime : BOOLEAN
ENDTYPE
```

Storing records in a serial or sequential file

The algorithm to store records sequentially in a serial (unordered) or sequential (ordered on a key field) file is very similar to the algorithm for storing lines of text in a text file. The algorithm written in pseudocode below stores the student records sequentially in a serial file as they are input.

Note that PUTRECORD is the pseudocode to **write** a record to a data file and GETRECORD is the pseudocode to **read** a record from a data file.

```

DECLARE studentRecord : ARRAY[1:50] OF TstudentRecord
DECLARE studentFile : STRING
DECLARE counter : INTEGER
counter ← 1
studentFile ← "studentFile.dat"
OPEN studentFile FOR WRITE
REPEAT
    OUTPUT "Please enter student details"
    OUTPUT "Please enter student name"
    INPUT studentRecord.name[counter]
    IF studentRecord.name <> ""
        THEN
            OUTPUT "Please enter student's register number"
            INPUT studentRecord.registerNumber[counter]
            OUTPUT "Please enter student's date of birth"
            INPUT studentRecord.dateOfBirth[counter]
            OUTPUT "Please enter True for fulltime or
            False for part-time"
            INPUT studentRecord.fullTime[counter]
            PUTRECORD, studentRecord[counter]
            counter ← counter + 1

        ELSE
            CLOSEFILE(studentFile)
    ENDIF
UNTIL studentRecord.name = ""
OUTPUT "The file contains these records: "
OPEN studentFile FOR READ
counter ← 1
REPEAT
    GETRECORD, studentRecord[counter]
    OUTPUT studentRecord[counter]
    counter ← counter + 1
UNTIL EOF(studentFile)
CLOSEFILE(studentFile)

```

Identifier name	Description
studentRecord	Array of records to be written to the file
studentFile	File name
counter	Counter for records

Table 20.10

If a sequential file was required, then the student records would need to be input into an array of records first, then sorted on the key field `registerNumber`, before the array of records was written to the file.

Here are programs in Python, VB and Java to write a single record to a file.

Python

```
import pickle
class student:
    def __init__(self):
        self.name = ""
        self.registerNumber = 0
        self.dateOfBirth = datetime.datetime.now()
        self.fullTime = True
studentRecord = student()
studentFile = open('students.DAT','w+b')
print("Please enter student details")
studentRecord.name = input("Please enter student name ")
studentRecord.registerNumber = int(input("Please enter student's register number "))
year = int(input("Please enter student's year of birth YYYY "))
month = int(input("Please enter student's month of birth MM "))
day = int(input("Please enter student's day of birth DD "))
```

Library to use binary files

Create a binary file to store the data

```
studentRecord.dateOfBirth = datetime.datetime(year, month, day)
studentRecord.fullTime = bool(input("Please enter True for full-time or False for
part-time "))
pickle.dump (studentRecord, studentFile)
print(studentRecord.name, studentRecord.registerNumber, studentRecord.dateOfBirth,
studentRecord.fullTime)
studentFile.close()
studentFile = open('students.DAT','rb')
studentRecord = pickle.load(studentFile)
print(studentRecord.name, studentRecord.registerNumber, studentRecord.dateOfBirth,
studentRecord.fullTime)
studentFile.close()
```

The diagram consists of three callout boxes with purple borders and rounded corners, connected to specific lines of code by thin purple lines. The first callout, labeled "Write record to file", points to the line `pickle.dump (studentRecord, studentFile)`. The second callout, labeled "Open binary file to read from", points to the line `studentFile = open('students.DAT','rb')`. The third callout, labeled "Read record from file", points to the line `studentRecord = pickle.load(studentFile)`.

VB

Option Explicit On

Imports System.IO

Library to use Input and Output

Module Module1

Public Sub Main()

Dim studentFileWriter As BinaryWriter

Dim studentFileReader As BinaryReader

Dim studentFile As FileStream

Dim year, month, day As Integer

Dim studentRecord As New student()

studentFile = New FileStream("studentFile.DAT", FileMode.Create)

studentFileWriter = New BinaryWriter(studentFile)

Console.WriteLine("Please enter student name ")

studentRecord.name = Console.ReadLine()

Console.WriteLine("Please enter student's register number ")

studentRecord.registerNumber = Integer.Parse(Console.ReadLine())

Console.WriteLine("Please enter student's year of birth YYYY ")

year =Integer.Parse(Console.ReadLine())

Console.WriteLine("Please enter student's month of birth MM ")

month =Integer.Parse(Console.ReadLine())

Console.WriteLine("Please enter student's day of birth DD ")

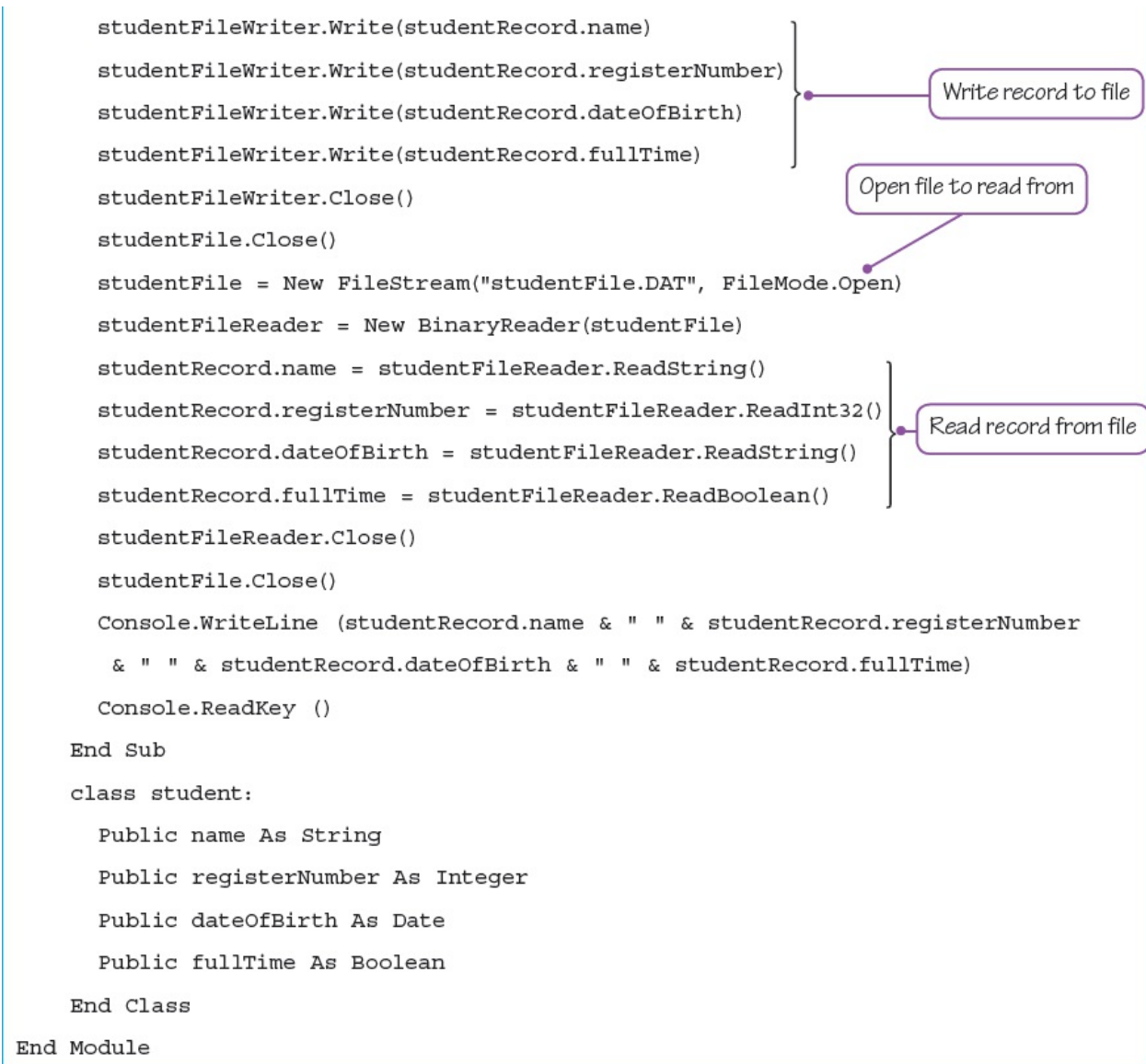
day =Integer.Parse(Console.ReadLine())

studentRecord.dateOfBirth = DateSerial(year, month, day)

Console.WriteLine("Please enter True for full-time or False for part-time ")

studentRecord.fullTime = Boolean.Parse(Console.ReadLine())

Create a file to store the data



Java

(Java programs using files need to include exception handling – see [Section 20.2.2](#) later in this chapter.)

```
import java.io.File;
import java.io.FileWriter;
import java.util.Scanner;
import java.util.Date;
import java.text.SimpleDateFormat;
class Student {
    private String name;
    private int registerNumber;
    private Date dateOfBirth;
```



```

private boolean fullTime;
Student(String name, int registerNumber, Date dateOfBirth, boolean fullTime) {
    this.name = name;
    this.registerNumber = registerNumber;
    this.dateOfBirth = dateOfBirth;
    this.fullTime = fullTime;
}
public String toString() {
    return name + " " + registerNumber + " " + dateOfBirth + " " + fullTime;
}
}
public class StudentRecordFile {
    public static void main(String[] args) throws Exception {
        Scanner input = new Scanner(System.in);
        System.out.println("Please Student details");
        System.out.println("Please enter Student name ");
        String nameIn = input.next();
        System.out.println("Please enter Student's register number ");
        int registerNumberIn = input.nextInt();
        System.out.println("Please enter Student's date of birth as YYYY-MM-DD ");
        String DOBIn = input.next();
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
        Date dateOfBirthIn = format.parse(DOBIn);
        System.out.println("Please enter true for full-time or false for part-time ");
        boolean fullTimeIn = input.nextBoolean();
        Student studentRecord = new Student(nameIn, registerNumberIn, dateOfBirthIn,
fullTimeIn);
        System.out.println(studentRecord.toString());
        // This is the file that we are going to write to and then read from
        File studentFile = new File("Student.txt");
        // Write the record to the student file
        // Note - this try-with-resources syntax only works with Java 7 and later
        try (FileWriter studentFileWriter = new FileWriter(studentFile)) {
            studentFileWriter.write(studentRecord.toString());
        }
        // Print all the lines of text in the student file
        try (Scanner studentReader = new Scanner(studentFile)) {

```

```
        while (studentReader.hasNextLine()) {
            String data = studentReader.nextLine();
            System.out.println(data);
        }
    }
}
```

ACTIVITY 20L

In the programming language of your choice, write a program to

- input a student record and save it to a new serial file
- read a student record from that file
- extend your program to work for more than one record.

EXTENSION ACTIVITY 20E

In the programming language of your choice, extend your program to sort the records on `registerNumber` before storing in the file.

Adding a record to a sequential file

Records can be appended to the end of a serial file by opening the file in append mode. If records need to be added to a sequential file, then the whole file needs to be recreated and the record stored in the correct place.

The algorithm written in pseudocode below inserts a student record into the correct place in a sequential file.

```
DECLARE studentRecord : TstudentRecord
DECLARE newStudentRecord : TstudentRecord
DECLARE studentFile : STRING
DECLARE newStudentFile : STRING
DECLARE recordAddedFlag : BOOLEAN
recordAddedFlag ← FALSE
studentFile ← "studentFile.dat"
newStudentFile ← "newStudentFile.dat"
CREATE newStudentFile // creates a new file to write to
OPEN newStudentFile FOR WRITE
OPEN studentFile FOR READ
OUTPUT "Please enter student details"
OUTPUT "Please enter student name"
INPUT newStudentRecord.name
OUTPUT "Please enter student's register number"
```

```

INPUT newStudentRecord.registerNumber
OUTPUT "Please enter student's date of birth"
INPUT newStudentRecord.dateOfBirth
OUTPUT "Please enter True for full-time or False for part-time"
INPUT newStudentRecord.fullTime
REPEAT
    WHILE NOT recordAddedFlag OR EOF(studentFile)
        GETRECORD, studentRecord // gets record from existing file
        IF newStudentRecord.registerNumber > studentRecord.registerNumber
            THEN
                PUTRECORD studentRecord
                // writes record from existing file to new file
            ELSE
                PUTRECORD newStudentRecord
                // or writes new record to new file in the correct place
                recordAddedFlag ← TRUE
            ENDIF
        ENDWHILE
    IF EOF (studentFile)
    THEN
        PUTRECORD newStudentRecord
        // add new record at end of the new file
    ELSE
        REPEAT
            GETRECORD, studentRecord
            PUTRECORD studentRecord
            //transfers all remaining records to the new file
        ENDIF UNTIL EOF(studentRecord)
    CLOSEFILE(studentFile)
    CLOSEFILE(newStudentFile)
    DELETE(studentFile)
    // deletes old file of student records
    RENAME newStudentfile, studentfile
    // renames new file to be the student record file

```

Identifier name	Description

studentRecord	record from student file
newStudentRecord	new record to be written to the file
studentFile	student file name
newStudentFile	temporary file name

Table 20.11

Note that you can directly **append** a record to the end of a file in a programming language by opening the file in append mode, as shown in the table below.

Opening a file in append mode	Language
<code>myFile = open("fileName", "a")</code>	Opens the file with the name <code>fileName</code> in append mode in Python
<code>myFile = New FileStream("fileName", FileMode.Append)</code>	Opens the file with the name <code>fileName</code> in append mode in VB.NET
<code>FileWriter myFile = new FileWriter("fileName", true);</code>	Opens the file with the name <code>fileName</code> in append mode in Java

Table 20.12

ACTIVITY 20M

In the programming language of your choice, write a program to

- put a student record and append it to the end of a sequential file
- find and output a student record from a sequential file using the key field to identify the record
- extend your program to check for record not found (if required).

EXTENSION ACTIVITY 20F

Extend your program to input a student record and save it to in the correct place in the sequential file created in Extension [Activity 20E](#).

Adding a record to a random file

Records can be added to a random file by using a hashing function on the key field of the record to be added. The hashing function returns a pointer to the address where the record is to be added.

In pseudocode, the address in the file can be found using the command:

```
SEEK <filename>,<address>
```

The record can be stored in the file using the command:

```
PUTRECORD <filename>,<recordname>
```

Or it can be retrieved using:

```
GETRECORD <filename>,<recordname>
```

The file needs to be opened as random:

```
OPEN studentFile FOR RANDOM
```

The algorithm written in pseudocode below inserts a student record into a random file.

```

DECLARE studentRecord : TstudentRecord
DECLARE studentFile : STRING
DECLARE Address : INTEGER
studentFile ← "studentFile.dat"
OPEN studentFile FOR RANDOM
// opens file for random access both read and write
OUTPUT "Please enter student details"
OUTPUT "Please enter student name"
INPUT StudentRecord.name
OUTPUT "Please enter student's register number"
INPUT studentRecord.registerNumber
OUTPUT "Please enter student's date of birth"
INPUT studentRecord.dateOfBirth
OUTPUT "Please enter True for full-time or False for
part-time"
INPUT studentRecord.fullTime
address ← hash(studentRecord,registerNumber)
// uses function hash to find pointer to address
SEEK studentFile,address
// finds address in file
PUTRECORD studentFile,studentRecord
//writes record to the file
CLOSEFILE(studentFile)

```

EXTENSION ACTIVITY 20G

In the programming language of your choice, write a program to input a student record and save it to a random file.

Finding a record in a random file

Records can be found in a random file by using a hashing function on the key field of the record to be found. The hashing function returns a pointer to the address where the record is to be found, as shown in the example pseudocode below.

```
DECLARE studentRecord : TstudentRecord
DECLARE studentFile : STRING
DECLARE Address : INTEGER
studentFile ← "studentFile.dat"
OPEN studentFile FOR RANDOM
// opens file for random access both read and write
OUTPUT "Please enter student's register number"
INPUT studentRecord.registerNumber
address ← hash(studentRecord.registerNumber)
// uses function hash to find pointer to address
SEEK studentFile,address
// finds address in file
GETRECORD studentFile,studentRecord
//reads record from the file
OUTPUT studentRecord
CLOSEFILE(studentFile)
```

EXTENSION ACTIVITY 20H

In the programming language of your choice, write a program to find and output a student record from a random file using the key field to identify the record.

20.2.2 Exception handling

An **exception** is an unexpected event that disrupts the execution of a program. **Exception handling** is the process of responding to an exception within the program so that the program does not halt unexpectedly. Exception handling makes a program more robust as the exception routine traps the error then outputs an error message, which is followed by either an orderly shutdown of the program or recovery if possible.

An exception may occur in many different ways, for example

- dividing by zero during a calculation
- reaching the end of a file unexpectedly when trying to read a record from a file
- trying to open a file that has not been created
- losing a connection to another device, such as a printer.

Exceptions can be caused by

- programming errors
- user errors
- hardware failure.

Error handling is one of the most important aspects of writing robust programs that are to be used every day, as users frequently make errors without realising, and hardware can fail at any time. Frequently, error handling routines can take a programmer as long, or even longer, to write and test as the program to perform the task itself.


The structure for error handling can be shown in pseudocode as:

```
TRY
    <statements>
EXCEPT
    <statements>
ENDTRY
```

Here are programs in Python, VB and Java to catch an integer division by zero exception.

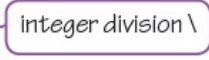
Python

```
def division(firstNumber, secondNumber):
    try:
        myAnswer = firstNumber // secondNumber
        print('Answer ', myAnswer)
    except:
        print('Divide by zero')
division(12, 3)
division(10, 0)
```



VB

```
Module Module1
    Public Sub Main()
        division(12, 3)
        division(10, 0)
        Console.ReadKey()
    End Sub
    Sub division(ByVal firstNumber As Integer, ByVal secondNumber As Integer)
        Dim myAnswer As Integer
        Try
            myAnswer = firstNumber \ secondNumber
            Console.WriteLine("Answer " & myAnswer)
        Catch e As DivideByZeroException
            Console.WriteLine("Divide by zero")
        End Try
    End Sub
End Module
```



Java

```
public class Division {
    public static void main(String[] args) {
        division(12, 3);
        division(10, 0);
    }
    public static void division(int firstNumber, int secondNumber){
        int myAnswer;
        try {
            myAnswer = firstNumber / secondNumber;
            System.out.println("Answer " + myAnswer);
        }
        catch (ArithmeticException e){
            System.out.println("Divide by zero");
        }
    }
}
```

Automatic Integer division because there are integers on both sides of the division operator

ACTIVITY 20N

In the programming language of your choice, write a program to check that a value input is an integer.

ACTIVITY 20O

In the programming language of your choice, extend the file handling programs you wrote in [Section 20.2.1](#) to use exception handling to ensure that the files used exist and allow for the condition unexpected end of file.

End of chapter questions

1 A declarative programming language is used to represent the following facts and rules: