THE DEVELOPMENT OF A METHOD TO ASSIST IN THE TRANSFORMATION FROM PROCEDURAL LANGUAGES TO OBJECT ORIENTED LANGUAGES WITH SPECIFIC REFERENCE TO COBOL AND JAVA

by

Jeanette Wendy Wing

A Research dissertation submitted in complete fulfilment for the requirements of the degree of Master of Technology in Information Technology in the Department of Computer Studies, Durban Institute of Technology, Durban, South Africa.

Durban, December 2002

APPROVED FOR FINAL SUBMISSION

_____ _____

Mr. D. A. Hunter, M.Sc.(UND) DATE
SUPERVISOR

ABSTRACT

Computer programming has been a science for approximately 50 years. It this time there have been two major paradigm shifts that have taken place. The first was from “spaghetti code” to structured programs. The second paradigm shift is from procedural programs to object oriented programs. The change in paradigm involves a change in the way in which a problem is approached, can be solved, as well as a difference in the language that is used.

The languages that were chosen to be studied, are COBOL and Java. These programming languages were identified as key languages, and the languages that software development are the most reliant on. COBOL, the procedural language for existing business systems, and Java the object oriented language, the most likely to be used for future development.

To complete this study, both languages were studied in detail. The similarities and differences between the programming languages are discussed. Some key issues that a COBOL programmer has to keep in mind when moving to Java were identified.

A transformation method is proposed. The transformation method identifies the creation of two distinct types of Java classes from one COBOL program. One is called the Java Data Class (JDC) which develops from the DATA DIVISION, and includes all data input and output methods. The second type is called a Java Processing Class (JPC), and includes the PROCEDURE DIVISION processes, which become methods, to complete the processing functions on the data.

The transformation method is applied to COBOL programs, in order to create Java programs.

Working examples of the programs discussed are presented in the APPENDIXES.

PREFACE

This study represents original work by the author under the supervision of Mr. D A Hunter. Two papers have been presented in completing this study.

The first paper was presented at the Technikon Natal Reasearch Day, 13 September 2000, entitled “The transformation of programming languages: Procedural to Object Oriented”. This paper focussed on the choice of COBOL as the procedural language, and Java as the Object Oriented language, and the initial similarities and differences identified. Some of this paper was presented in Chapter 2.

The second paper was presented on 26 September 2001, at “Going Global”, Faculty of Commerce Research Day, Technikon Natal. The title was “Transformation: COBOL to Java”. At this presentation, the paper was chosen to be presented at the Technikon Natal research day held later in the year. This paper proposed a methodology for the transformation, and presented an example of the transformation method applied to a COBOL program. This is expanded in Chapter 3 and 4.

ACKNOWLEDGMENTS

First I would like to thank Mr. D A Hunter for his supervision of this project. His understanding, in the completion of a project, as a part time student and full time staff member has been invaluable.

To the late Mr. F N Heukelman, who encouraged the start of this project, and whose influence on my life, gave me the determination to complete this study.

To my colleagues in the Department of Computer Studies, thank you for your support.

To my friends and family who assisted in many ways, sometimes in just giving me the freedom to ignore them, your support was much appreciated.

To God, the Father, who is part of all I do.

Lastly, I wish to dedicate this thesis to Lloyd, Martin and Stacey. Thank you for your understanding and support.

TABLE OF CONTENTS

Chapter 1 - Introduction.

b) The need for the study.

The following quote from Information week (1998), captures in essence, the purpose of this study, "Even in progressive IT shops, many developers use procedural languages such as COBOL or C, typically for maintenance of existing apps. But with the rise of HTML-based apps and the move to mixed media on networked applications, it's reasonable to expect objects and components to continue their slow rise to dominance". There are many computer systems currently in use that have been written in procedural languages. With the development of the object oriented paradigm, and more reliable programming languages that use the object oriented methodology, it is now the time to convert "older systems" in order to benefit from the object-oriented methodology. There is a need to study methods of re-engineering existing computer systems, to take advantage of object oriented methodologies.

This study is significant in that it will identify a methodology which should be able to assist in this process. As most programmers will be familiar with the procedural methodology, and the computer systems to be changed are also written using procedural methodology, it is important to use this as a starting point, and wherever possible take elements through from the old systems into the new.

This study aims to identify similarities and differences between the two methodologies and then develop a methodology (series of procedures and rules to follow) which

would assist in the transformation from procedural to object oriented applications.

There have been studies done that address the transformation from one programming language to another. In an article in Midrange Systems, Dennis Callaghan (1999) lays out some proposals on transforming from RPG400 to Java. The same problem is addressed by Merritt (1997) who published an academic textbook on the migration from Pascal to C++. Although the languages chosen are different these texts could give some key ideas on how to formulate a methodology for the transformation. Doke and Hardgrave (1999) present the Java language, in terms that the COBOL programmer can understand. There is no transformation methodology presented, however.

Many authors discuss the principles of programming languages and this will assist in the study. Eliens (1995) who discusses the principles of object oriented software development. Sebasta (1993) also has a published text on the concepts of programming languages. These will be valuable in comparing the two languages.

A SABINET search was conducted on 14 September 1999. This showed no current or previous research done in South African institutions comparing these programming languages.

1.2 The identification of COBOL and Java as the languages to be studied.

1.2.1 The paradigm shift.

Computer programming is a relatively new science. The first programs were coded in machine language in the 1950's. Third generation programming languages, which made programming a little easier than 0's and 1's developed in the late 1950's and early 60's. COBOL was one of the first third generation languages. These third generation languages, opened up the science of computer programming to a larger group of people. The first computer programs were described as unstructured. This means there was no method to the way in which the problem was solved. The first major paradigm shift in programming, was the advent of structured programming. Structured programming theory developed over a time period of approximately ten years. The history is presented by Welburn (1983) as follows. The first step towards structured programming theory was the structure theorem presented by Bohm and Jacopini in 1964. This was followed by papers presented by Dijkstra on the harmful effects of the GO TO statement, as well as a pilot project, in 1969, named the “New York Times”project, where Dijkstra demonstrated that structured programming methods make for far more productive programmers. Structured programming theory was presented in a paper entitled “Structured Design” in 1974 by Stevens, Myers and Constantine. This completed the ten years development, and structured programming theory become a part of programming methods. In essence structured programming involves “a program design, documentation, coding, and testing methodology that utilizes techniques in program development to create proper, reliable, and maintainable software products on a cost effective basis.” (Welburn, 1983). Notice the emphasis on maintenance. A key issue in existing business systems is that they often need to change. In a program written in COBOL, it is necessary for the programmer to be able to understand the program (or system of programs) in order to make changes to it. This

can be a very costly exercise, if the programs are poorly designed, and therefore difficult to change. As can be seen from this short description, it took approximately fifteen years from the beginning of the science of programming, for the paradigm shift of structured programming to be defined. Welburn (1983) discusses the development of structured programming theory. In essence this was to create programs using only the three control structures (Bohm and Jacopini), that did not include a GO TO statement (Dijkstra), and that implemented the principles of functional cohesion, (each module executing a single function), and being as loosely coupled as possible.

In the 1980's the second paradigm shift took place. The advent of object oriented programming. The first major conference on Object Oriented programming took place in 1996, OOPSLA, an acronym for the conference on Object-Oriented Programming Systems, Languages and Applications (Budd, 2000). This is a major shift in the way of thinking. Before, in a program language like COBOL, all the data had to be defined, and be globally available to all modules (procedures). In object oriented programming, the data is encapsulated by the methods (procedures).

1.2.2 Other contributing factors.

Many texts also confirm the need to move forward into the object oriented environment. As discussed by Chapin (1997) the need to be able to reuse program code has been evident since the 1950's. This sentiment of being able to reuse code is also supported by Wilson (1993) . The need to change is emphasised by Grady (1997) “businesses involving software are caught up in a crowd moving forward, whether we

like it or not. If we don't move with it, we'll be pushed aside and left behind." Other authors that discuss object oriented programming in general, and also in some cases identify advantages of object orientation as well as some methods for evaluating languages are Henderson-Sellers (1994), Ledgard (1996) and Martin (1995).

There are many commercial systems coded in COBOL. It is estimated that COBOL accounts for 80% of commercial applications developed in the 1970's and 1980's. Nick Langley (1999) states "COBOL is still the most widely used language after 40 years. Billions of lines of COBOL exist, used by more than a million companies.". These legacy systems are business applications, and companies rely on them for their daily functions. COBOL systems are therefore key systems in the business environment. This view, the popularity of COBOL and the business world's dependance on it are discussed by Chapin (1997) in the introduction to his book on object-oriented COBOL.

Due to the fact that COBOL is so dominant in the business world, it is the language chosen to be studied as a procedural language.

The choice of Java as the object oriented language to move to is supported by a number of articles. The Computer Weekly has an article where Black (1998) describes that a study completed by Bloor Research shows that more than two thirds of the large IT industry approached in the UK expect to adopt Java within the next two years. An article in Computerworld (1998) also discussed that Java has registered 750000 developers and worked its way into major corporate houses within two years. In Wall Street and Technology (1999) Cavanaugh states that Java is a significant addition to

the Wall Street IT development scene. In a paper presented by Hunter (1999) at the recent Technikon Natal Commerce Research Conference, he clearly identified that the way forward seems to be with Java and its application development products. Charles Babcock (2000) states “there are currently over 1.3 million developers using Java and that number is expected to reach 4.4 million by 2003".

A factor which plays a part in the choice of COBOL and Java is the consistency of the languages. COBOL was the first language that had its specifications defined by a committee and enforced by the American National Standards Institute, hence the name ANSI COBOL. Each compiler that was produced by any company was required to conform to the ANSI standard. This made COBOL a language that essentially behaved the same on different environments, as the compilers adhered to the ANSI standard. This consistency helped in making COBOL the language of choice for the development of business systems. Java has achieved the same consistency but in a different way. Sun Microsystems developed Java, and decided to publish Java on the Internet, making the code available to all programmers. There is therefore a single standard for Java as there was for COBOL. According to Clifford Swift, “a key factor of Java which will enable its acceptance is the fact that it provides a single technology that enables corporations to develop, maintain and enhance heterogeneous applications with a single skill base.” This broad standard of the programming language is what seems to be a unique factor for both COBOL and Java. This factor influences the languages acceptance in industry, as the programmers, which are the most expensive component in the software development equation, once they have gained the skills of the language, can easily adapt to each development environment.

Goodridge (2000), in an article on Information Technology Labour, discusses COBOL as the most broad skill base available, and Java, HTML and Web development as the most required new skills. This clearly indicates that COBOL to Java is a worthwhile study, and the programming languages are key to the IT industry.

1.2.3 The changing nature of computer systems.

The business systems written in COBOL, were generally for systems that worked within a company structure, within a distributed computer network, or other similar system. The advent of the Internet, and its broad acceptance as a business tool, is changing the nature of computer systems. It is this change that may finally close the chapter on COBOL programming. The emphasis is far more than transferring existing systems to OO technology, it is a re-think of the way in which business processes take place. If the new approach to business, dramatically changes the business process, the need to transform the legacy systems no longer exists. The systems need to be completely re-designed. The new design would have to consider the new business needs, and if this involves event driven applications, available to a broad base of users, distributed around the world, the development should be completed in a language that was designed for this type of application. From this study it has become clear that Java is ideally suited to the new business world.

1.3 The objectives of the study.

The following objectives have been identified. They are presented in the sequence required for the study.

1.3.1 To identify key elements that define a procedural language with direct reference to COBOL.

This is discussed in Chapter 2. Section 2.1 gives a description of the COBOL programming language, from its history to features of the language. The key elements that define COBOL as a procedural language are identified, and discussed in 2.1.4.

1.3.2 To identify key elements that define an object oriented language with direct reference to Java.

This is discussed in Chapter 2. Section 2.2 gives a description of the Java programming language, from its history to features of the language. The key elements that define Java as an object oriented language are identified, and discussed in 2.2.4.

1.3.3 The identification of similarities between the two programming languages.

This discussion takes place in Chapter 3. Section 3.1 notes the similarities

identified between the programming languages.

1.3.4 The identification of the differences between the two programming languages.

This discussion takes place in Chapter 3. Section 3.2 notes the differences identified between the programming languages.

1.3.5 The development of a methodology (series of procedures and rules to follow) to assist in the transformation from procedural languages to object oriented languages with specific reference to COBOL and Java.

This is accomplished as a result of the analysis of the programming languages. Chapter 4 presents a methodology in section 4.1 and examples of the implementation of the methodology are given in section 4.2.

The first four objectives, are set in order to assist with objective five. They will need to be accomplished first in order to achieve the objective of developing a methodology to assist in the transformation from COBOL programs to Java classes (procedural languages to object oriented languages).

1.4 The study design.

This is theoretical creative research. Literature survey will be used to gain an insight into the background of each programming language. Analytical work will be done in

identifying the key features of each language, the similarities and differences. Creative research is achieved in that a transformation methodology will be proposed. Theoretical creative research is defined by Melville (1996), “Theoretical creative research is about the discovery or creation of new models, theorems, algorithms, etc.” . Olivier (1997) states “most of IT research endeavours to realise ‘theories’ to guide construction of automated systems” which supports that this is the type of research required in the field of Information Technology.

An extensive study of both languages will need to be done. COBOL is mainly a procedural language, and Java is in essence an object oriented language. This study will be via literature study and by practically coding programs. COBOL programs as well as Java programs will be coded in order to gain familiarity with the programming languages. This coding will be done using the RM COBOL 85 compiler and the Sun Java version 1.3.1. As the study aims to produce a series of procedures and rules to follow in the conversion of COBOL programs to Java programs, the methodology should be applicable to most versions of the two languages.

Comparisons will be made between the programming languages COBOL and Java. Again a literature search will be needed to determine any similar comparisons that have been done between other languages. Then to propose a methodology to assist in the change from one programming paradigm to the other. This is creative, since a methodology (set of rules and procedures) will be proposed as a result of the study.

2.1 General description of COBOL

This section has been completed with reference to the following authors Welburn (1995; 1983; 1981), Philippakis (1987), Stern (1991), Yourdon (1979), as well as twelve years experience in lecturing COBOL.

2.1.1 A brief History of COBOL.

COBOL is described as a third generation language. The first computers were programmed in machine code, and this was followed by assembler. To program in machine code, it was necessary to give the computer instructions in binary. Assembler uses three, and four character codes to accomplish commands. The first and second generations of programming languages required a high level of expertise in order to use the languages effectively. The languages were also very closely related to the hardware that was being used, resulting in programs that are not portable. The computer scientists then developed third generation languages. A third generation language is easier to use than machine language or assembly language, as the instructions can be given to the computer in a more English type language. A computer programming language has strict rules of syntax (the construction of a statement) and certain reserved words. These reserved words are used in giving instructions to the computer. A program coded in a third generation language is “compiled” into machine code (or object code). The machine code is what is executed by the computer. Programs coded

in a third generation language are easier to transfer to another computer system. The program would be re-compiled, on the new hardware system, thereby generating machine code that works with the new hardware. A number of third generation languages were developed, each with a slightly different emphasis. COBOL is a third generation language that was developed as a commercial language to deal with large volumes of data. A major focus in the development of COBOL was that it would be an English type language, and therefore easy to understand. COBOL was developed in the late 1950's, by a committee, the CODASYL[1] committee. (A committee that actually achieved their goal). COBOL received the support of the United States military, this resulted in a policy which required a COBOL compiler to be available on computer in order to be awarded government contracts. (Wilson, 1993). This resulted in COBOL becoming the most widely used programming language in the world in the 1960's and 1970's. The first release of COBOL was named COBOL 60. Revised versions were released in 1961 and 1962 (by the US Department of Defense) The language was standardised in 1968 by the American National Standards Institute (ANSI). Revisions were released in 1974 and 1985 by ANSI. (Sebasta, 1999). Programming languages were the first truly international languages. COBOL maintained dominance for almost thirty years.

[1] CODASYL Conference on Data Systems Languages. This set standards and rules for the COBOL language. (Wilson, 1993)

2.1.2 The Structure of a COBOL program. (A main program.)

2.1.2.1 The Four Divisions.

A COBOL program is structured in four sections defined as divisions. They are the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION and PROCEDURE DIVISION. Each of these divisions must be clearly defined in the sequence given. There are rules for the types of statements that can appear in each division.

The IDENTIFICATION DIVISION identifies the program to the operating system. The rest of the IDENTIFICATION DIVISION serves mainly as documentation, with statements that identify the programmer, and date written etc.

The ENVIRONMENT DIVISION identifies the environment in which the program is run, as well as establishing the operating system name for any files that will be accessed. This is done in the SELECT statement. All files that are going to be used in the program must be named in a SELECT statement.

The DATA DIVISION is where the each files record format, and any variables that are to be used in the program must be defined. The DATA DIVISION has two parts. The FILE SECTION which must be defined first, followed by a WORKING-STORAGE SECTION. The FILE SECTION must have an FD

(file definition) for each file that is given in a SELECT statement. The record format must be described giving the exact description (position, length, and data type) of each field that is to be used in the program. It is important that the record length is correctly defined. The WORKING-STORAGE SECTION is used for all temporary fields. This includes flags (a field with two states that are used to control processing), line and page counters to control the printing of reports, the definition of all print lines and any other temporary fields for calculation or control purposes (control break programs).

The PROCEDURE DIVISION is where the programming logic is found. Generally the program logic will deal with the functions of input, processing and output. The Input would be via a screen or data file (or both) the processing is done on the data that has been input, and the output is then displayed to screen, or written to a printer or data file.

2.1.2.2 The program logic.

The logic for a COBOL program is in the procedure division. As mentioned in 2.1.1 this usually involves working with data files. Two common program types have been chosen for discussion: a control break report, and a relative file maintenance program.

c) A control break report.

Generally a control break report program was written to support management decision making. In essence this report produces a summary of data that is present on a master file or a transaction file. A master file is defined as containing permanent and semi permanent data. An example would be an employee file. The permanent data being the employee number, and the semi permanent data is data that can change, eg. Address, salary scale and so on. A transaction file contains a record of day to day transactions. For an employee system these may be sick leave or vacation leave changes that need to be made to those employees that have used their leave. A control break report program would be written to summarize the data in some way. For example at a tertiary institution academic staff fall within a department, and each department is part of a Faculty. A report showing the total employees in each department as well as the total employees in each Faculty would be a control break report. This would run off the master file, as this contains the details of each employee.

To write a COBOL program to do this you would need to define the data and the procedure to be applied to the data.

The data is defined in the data division. In the FILE SECTION you would define the master file and the report file (where the report is to be printed). Control fields that identify when a faculty or department has changed would be defined in the WORKING-STORAGE SECTION. The fields required to

accumulate the counts of employees would also be defined in working storage. Page counters and line counters as well as flags identifying the end of file also must be defined in working storage. Definitions of all the heading, detail and total lines to be used in the report would also be defined in working storage.

Once the data has been defined the PROCEDURE DIVISION is coded. This is the logic section. Generally this would always start with a control paragraph. This opens the files, does any initialisation routines, reads the first record from the master file, and calls a more detailed process (another paragraph) which will execute until there is no more data on the master file. The files are then closed and processing is terminated. The detailed process must produce the report. A control field is set up that will indicate when a department has changed. A similar field will also be set up to indicate when a faculty has changed. It is assumed that when a faculty changes, a department will also change. The report headings must be printed as well as total lines whenever a control field changes. The detailed process will generally call any number of paragraphs in order to produce the report.

d) A relative file maintenance program.

A file maintenance program will be written for each master file that is present in a system. This program takes care of adding records to the master file (eg add a new employee), changing records on the master file (e.g. change an

employees address), and deleting records from the master file (eg an employee has left the company).

Again the DATA DIVISION must be defined. In the FILE SECTION you would define the master file, the transaction file (file containing the changes) and a report file (given that a report will be printed showing the changes made to the master). In the WORKING-STORAGE SECTION you would have the control flags identifying if there is more data to process, and all the report lines for the report.

The PROCEDURE DIVISION would start with a control paragraph. This Opens the files, does any initialisation routines, reads the first record from the transaction file. The control paragraph then calls a more detailed process (another paragraph) which will execute until there is no more data on the transaction file. The files are then closed and processing is terminated. The detailed process must maintain the master file records. A transaction record has been read, and this can be an add, change or delete. For a change or a delete, this would involve finding the master record that matches the transaction. If a master record is not found then an error has occurred, and the error - *change or delete where no master exists*, would be written to the report. The next transaction record is read. For an add transaction a matching master record should not be found in order to affect the add. If there is already an existing master then this error - *cannot add, a record already exists* must be written to the report. If the report is an error report then it will contain only error

messages, if it is an audit/error report, then transactions that have been correctly processed as well as errors are written to the report. The detailed process will generally call any number of paragraphs in order to maintain the master file and produce the report.

2.1.2.3 PROCEDURE DIVISION design.

The procedure division, is a breakdown by function, of the task that is to be done. There are a number of methods that can be used, to design the procedure division. The one that is most commonly used is top down design. In top down program design the program problem is broken down by function. Then each function is again broken down. This is continued until each module (or paragraph) in the program only has a single function. Once this design has been completed the PROCEDURE DIVISION modules are coded in COBOL. A structure chart is usually used to assist in the functional breakdown during the program design phase. As presented here the COBOL PROCEDURE DIVISION is concerned only with function. The data to be used has all been defined in the DATA DIVISION. This is where there is such a large paradigm shift, as far as programming is concerned. “Developers need to shift from traditional development thinking that used to split data and function, towards new object oriented thinking which combines aspects of data and function.” (Labuschagne, 1995)

2.1.3 A COBOL subroutine.

Subroutines are used for common program logic that needs to be used by a number of programs. A subroutine would be written for date conversion eg. Transforming a Gregorian date into a Julianised date, or validating a check digit on a number (eg. student number) and so on. The structure of a subroutine is different in that data can be passed to a subroutine and also received back from a subroutine. In most cases a subroutine would also not process any files. A subroutine's function is to receive data, process it and send a result back to a calling program. The four division headers are still required, even although some of the divisions will not contain much or any code. For example the FILE SECTION in the DATA DIVISION will not be used. A new header in the DATA DIVISION is required for passing the data between the main and calling program. This is the LINKAGE SECTION, and it appears after the WORKING-STORAGE section. All data items that are communicated between the main and the calling program must be defined here. The procedure division header must list the data items that are communicated. This is done as PROCEDURE DIVISION USING *data-item-1, data-item*-2. This is matched to the CALL statement that is used to call the subroutine. When a call to a subroutine is executed in a main program, the data values listed in the call statement are passed through to the linkage section. The PROCEDURE DIVISION code is then executed. When a RETURN statement is encountered in the subroutine, control is returned to the main program, to the statement following the call. The data in the linkage section is also passed back to the main program. Any changes that have been made to the LINKAGE SECTION data items would therefore be transferred back to the main program.

2.1.4 Identify the key elements that define a procedural language with reference to COBOL.

COBOL has basically two elements that a programmer must define. The Data to be used in the DATA DIVISION and the logic required to process the data in the PROCEDURE DIVISION. The data is defined and available for all paragraphs in the procedure division. A problem is solved by developing the logic modules (paragraphs) that will process the data as per the specification. Usually top down design is employed to achieve this breakdown of the overall problem into smaller manageable tasks. As most programs will be processing files, the high level control paragraphs will deal with opening and closing the files, as well as the reading of each record on file. As you can see the focus of the programming effort is on procedures.

2.1.5 Key areas - where the language is used best.

COBOL provides a very stable environment for data processing. Many systems have been developed and are still currently running in COBOL. These are often referred to as legacy systems. The data is managed and large volume processing of data can be achieved. Any system which is dependant on a lot of file handling with large volumes of data can still effectively use a COBOL system. Basically anywhere where large volume data processing supports the daily business transactions COBOL can be very effectively used.

As COBOL processed large volumes of data, indexed files were developed in order to

facilitate fast and easy access to records. When an indexed file is created in COBOL, a field that is part of the data record is defined as the record key, this must be unique. Alternate record keys that are also data fields within the record can also be defined. There may be a number of alternate keys, but there is an overhead associated with each. Each COBOL compiler would specify a limit to the number of alternate indexes that could be defined (usually five). COBOL then creates indexes that allow alternate access to each data record. For a customer file, the primary index, which must be a unique field for each record, would be the customer number. An alternate index on customer surname is also defined. COBOL would create a primary index with each customer number, and the records address, as well as an alternate index with each customer surname (duplicates allowed) and the records address. When accessing data records from the indexed file, you would specify the index to be used in the READ, or START statement. There are a number of rules that apply, and where no index was specified, the default of the primary index would be assumed. To access a record from the customer file, there are now a number of options:-

- To access all records on file, sequential access can be used. The COBOL system would use the primary index, which results in all records being accessed in customer number sequence. The sequential access therefore refers to the records being accessed in a true sequence.
- To access a single record, where the customer number is known, the record can be directly accessed, by supplying the customer number, via the primary index. This results in fast data access, especially for online systems.
- To access a single record, where the customer number is not known, only the customer surname, access is via the alternate index. The alternate index would

however have a number of customers with the same surname, as duplicates are allowed, and would normally exist. This does however provide a very powerful searching tool for quickly accessing records, where a primary key is not known.

These three examples, illustrate the processing options available when using indexed files.

The use of alternate indexes provides for flexibility in processing, and a powerful access tool especially for online systems. Data base systems that developed later (historically), use the principles of indexes and alternate indexes extensively. There are two factors that must be taken into consideration when using COBOL indexed files:-

- All the indexes required for the data file must be defined when the file is first created. If an alternate index is to be added to a file, the data file must be re-created, and all programs that use that data file must be modified and re-compiled.
- The indexed data file must be re-organised on a regular basis. Whenever records are added and deleted to/from the data file, the indexes are updated, usually by inserting pointers where changes have taken place. Eventually data access will slow down. Re-organising the file is a simple process of re-creating the file, but it is important that this is done regularly.

There may still be a place for COBOL where systems are confidential, say a personnel system, or customer processing. However as technology advances customers and employees want access to their own data in the form of information. In many cases this access to information would be requested over the Internet. The actual processing in

these systems can still be done by COBOL, but the ability to query etc should be allowed to the customer.

2.1.6 Shortfalls of COBOL.

In COBOL programming there is a lot of code that is repeated in each program. The DATA DIVISION must be repeated. ie. there will be any number of programs that work with a master file. Each time a program is coded that works with the master file the DATA DIVISION code must be repeated. To some extent the use of copy libraries took care of this repetition. Programmers were then however forced to use predefined data names. Where a different data structure was required the data then needed to be manipulated within the program in order to modify the format given in the data structure.

The PROCEDURE DIVISION also has its set of repetitive code. The high level control paragraph always opens and closes the files, sets up the initial read and control loop. Generally the rest of the procedure would be performed until there is no more data on file. If a report program is being written, then the controls for headings and page breaks are very similar in each program. This code would have to be coded in each program's PROCEDURE DIVISION. COBOL programming therefore becomes rather tedious as large portions of code which have been coded before, are repeated in each program.

When designing and coding the procedure division, as mentioned previously a top down design methodology is often used. This means that the end product is envisaged, and

then broken down until we have manageable components, and the components are then coded. Unfortunately due to the nature of COBOL the components are coded in the light of achieving the final product. The components cannot be used in constructing another similar product. This is what object oriented programming intends to solve, where components can be independently developed and then assimilated in order to solve a problem. The structure of the COBOL programming language, where data is separate from procedure does not allow for this to happen very easily.

2.1.7 Online versus batch processing.

Batch processing, is the type of processing that COBOL was initially designed for. By definition, a batch process involves collecting data over a period of time, and then processing the data against a master file in one computer run, a batch run. The period of time over which data is collected could range from a day, a week or a month. Basically any system where data is not immediately processed is termed a batch system. With a batch system for accounts receivable, when a customer makes a purchase, the transaction is recorded, and then processed against the master file at a later stage. The new customer balance will only reflect the new amount after the batch run has been completed.

In contrast online systems have the data available to a user immediately. In an online system, when a customer makes a purchase their account is updated immediately, and the new customer balance is reflected at the end of the transaction. The move from batch processing to online processing has come about as the hardware supporting

computer systems has developed. Online processing has a far larger overhead on memory and disk access as opposed to batch systems. Many systems are now combinations of both processing types. The daily transactions tend to be processed online, but there are processes, for example printing monthly statements that are processed as a batch at the end of each month. When it comes to large volume data processing, and the printing of reports, COBOL was designed for this type of processing, and works well in producing high volume reports.

2.2 General description of Java.

The following has been completed with reference to the authors Arnold and Gosling (1998), Lemay (1997), Dietel and Dietel (1998; 2002), Naughton (1996), Reed Doke and Hardgrave (1999), Savitch (1999; 2001) and Wigglesworth and Lumby (2000). Included is eighteen months lecturing experience in Java.

2.2.1 Brief History of Java.

Java was developed by Sun Microsystems. They had a design team working on software for small appliances. This eventually changed to become Java. James Gosling has been given credit as the father of this programming language (Arnold et. al., 1998), although originally a design team was working on the project. Some of the other key players in the development of Java were Patrick Naughton, James Sheridan, Wayne Rosing, Bill Joy and Jonathan Payne (Naughton, 1996). Initially the language was called Oak. When it was discovered that a language called Oak already existed there

was an urgent need for a name change. A discussion in this regard took place in a coffee shop, and hence the name Java. Java was officially released by Sun in May 1995. Java is an object oriented language. When Java was first released it was designed mainly for use with Web applications, where applets are activated within a Web page (Naughton, 1996). As the use of the Java Virtual Machine (JVM) in executing Java applications introduced a simplicity or facility to execute applications on different operating systems without a need to modify the program code in any way. It is the Java bytecode that is executed. As Sun published Java on the Web, allowing users to download the software free of charge, more users started using the language and it has developed to be far more robust than when first released. Java applets (small applications) that run within a web page are only one facet of the language. Java can also be used for the development of enterprise software, interfacing to database applications and so on. Generally a broad band of all computer processing can be done using Java, even as far as setting up and controlling networks. To quote from Dietel and Dietel (2002, pg12), "Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of World Wide Web servers (the computers that provide content we see on our Web browsers), to provide applications for consumer devices (such as cell phones, pagers and personal digital assistants) and for many other purposes.".

The Java language is still evolving, at a rapid rate. The Java language itself has 48 reserved words, and in this way is relatively simple (Reed Doke and Hardgrave, 1999), the complexity lies in the Java Applications Programming Interface (API), also referred to as the Java class libraries. It is these class libraries that continue to develop. In Java

1 the original API (or set of core classes) included about 200 classes, release 1.1 included about 500 API's, and Java 2 about 1600 (Wigglesworth and Lumby, 2000) . The Java 2 release includes Java Foundation Classes (JFC) of which the swing class is one. Some of the API's are described in 2.2.2.2.b) below. The full API for Java 2 can be viewed at http://java.sun.com/j2se/1.3/docs/api/index.html. (12 April 2002).

2.2.2 The structure of a Java program.

Java is an object oriented programming language, as it views everything as an object. Each Java program is an object. Each object is defined as a class in Java. A class has fields (instance variables) and methods, the class represents a set of objects that have a common structure, and behavior. An instance of a class is created (instantiated), and represents a specific object. Each method may also have its own internal variables which are then only available to that method. Each method provides processing actions that can take place on the fields (instance variables). The relationship between the instance variables and methods is referred to as encapsulation. The fields (instance variables) are encapsulated by the class, the only way in which you can gain access to the fields is through the methods that are within that class. It is important that you declare instance variables of a class that you want protected in this way as private, as this will enforce the protection of data. If an instance variable is declared public it will be able to be accessed directly from outside the class when an object is instantiated.

Classes may also inherit from another class. This promotes the reuse of code. Java allows for single inheritance, where a class may only inherit from one base class (or

super class). However the current class being defined also inherits from all the classes above that base class. ie. all that the base class has already inherited is then also inherited. To inherit from a base class the keyword extends is used in the Java programs class header. Given below is the basic structure for a Java class, that does not consider any inheritance. Each Java class does however inherit from the class object. This is a standard default that is part of the Java language. Below is a basic structure for a Java class.

```
Class header
{ beginning of the class
      variable definitions for this class
      ┌──────────────────────────────────┐
      │ Method header                    │
      │ {    beginning of method         │
      │                                  │
      │      variable definitions        │
      │      java procedural statements  │
      │                                  │
      │ }    end of method               │
      └──────────────────────────────────┘
      ... any number of methods may be coded
} end of the class
```

This is the basic structure for each Java class. There may be any number of methods in a class. Generally in all Java applications except applets and abstract classes, Java will look for a **`main`** method, as this is the point at which the execution of the procedural statements in a class begins. The instance variables described in the constructor for the class will be instantiated before the main method is execured. The main method header is as follows:

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

The java interpreter will look for this method header in order to execute the application. This method is described as void as it does not return any data . An abstract class is used as a base class for inheritance (see 2.2.2.2 b). An abstract class cannot have an object instantiated. An abstract class therefore does not have a main method. There are many possible applications of Java classes that will not have main methods.

An applet is coded to be executed within a web page, and will generally have an `init()` method. The `init()` method initialises the applet within the web page, and is called only once by the appletviewer or browser. To continue, the applet must either have an event listener assigned, through an interface, which will initiate some action when an event is fired, or a `start( )` method that will be called whenever the user returns to the Web page containing the applet.

2.2.2.1 The program logic (Defining a Java class).

The entire approach to solving a problem using Java is an object oriented approach. Firstly the objects need to be designed, for the best implementation, the inheritance between objects also needs to be designed. The methods surrounding each object are then defined. You can now define each class in the solution to the problem. This overall description of object oriented design can be used in the approach to solving both complex and simple problems. Where larger, more complex systems are designed, a larger base of classes, where

inheritance is carefully considered would be designed. More simple systems will have fewer classes.

To define a Java class, you need to define the variables (attributes) of the class, and the methods that work with that class's attributes. The first methods that you define in the class would be constructors. There are one or more constructors in a Java class, that have the same name as the class. A constructor provides values for the instance variables. When an object of a class is created (instantiated), one of the constructors in the class would be used to provide the values for the instance variables. When there is more than one constructor, the constructor methods are described as overloaded. ie. The method names (constructors) will be the same, but the method signature (parameters supplied in the method call) will be different. When an object is instantiated, according to the parameters supplied, the appropriate constructor will be used to initialise the variables for the given object. Accessor and mutator methods are also then defined. Again this would be dependent on the nature of the system. Accessor methods provide access to each variable, and mutator methods allow for a variable to be changed. Methods that perform processes related to the instance variables, would then also be defined in the class. In larger systems, a separate class defining the methods could then be defined, that would then use an object of this class.

For example lets look at an employee. An employee object is to be used in a system. Each employee object will have an employee number, name, address,

salary scale, sick leave, vacation leave and so on. The constructors would be set up to initialise the instance variables with values supplied or default values. Accessor methods[1] would be set up that provide access to certain of the variables. The accessor methods would be used by other classes that use the employee object in order to gain access to the data values. Mutator methods[2] would be written to change the values in the instance variables, for example a change address method to facilitate the change of an address for an employee. Other processing methods could be defined such as allocate sick leave, or increase salary that would implement changes on the employee object. When an employee is instantiated, this can be from another Java class, or in a main method. To utilise object oriented technology as effectively as possible, it would be better to work with the employee object from another class that is focused on the processing of the employee, and not have all these methods defined in the employee class. These methods, that perform a process, are procedural in nature. However as can be seen from this description it is the overall change in design which is vastly different from COBOL where the entire focus is on procedure. In Java the focus is on the objects. A huge design effort needs to take place as far as the overall object design is concerned, only once that is in place can there be a focus on the procedure that must take place within a method. When it comes to the procedure design, all the design principals that are used in COBOL can be used in Java, it is the access to the

---

[1] An accessor method is also referred to as a **get** method. As a programmer standard the method is called getVariableName (eg getAddress).

[2] A mutator method is also referred to as a **set** method. As a programmer standard the method is called setVariableName (eg. SetAddress).

data, and overall design that is so different.

2.2.2.2 Using existing Java classes.

This can be divided into two sections. The first is using the Java classes that are part of the Java package, and the second is using classes that have been designed within an organization.

a) Classes within the Java language.

To bring in the Java environment, you are required to code an import statement (or statements) at the beginning of the program. Although the language of Java itself is simple, you are required to use classes that have already been defined so that there is no need to keep on developing program code from scratch. These are known as Java API's. In order to do this you import the classes that you require. The import statements will be different depending on the type of application you are coding. A default import that takes place for each Java application is the Java language package. To mention some other API's, the graphical user interface (awt), and the extended graphical user interface (swing), the input-output package for file handling (io), the event handler for event driven programs (event) and the applet class (applet) for coding applets. Each of these has been defined as a package. A package is a group of classes that are saved as a folder within the Java application. These are also referred to as the Java class libraries. At the beginning of each Java class the import

statements must be coded. It is not a good idea to import classes that are not used. The import statement makes all the classes defined in each of the packages available to your program. According to your application you will make use of these predefined classes. For example if you want to work with files you would code `import java.io.*` , and you can use the classes already available that work with data streams, you would not need to work with files starting from scratch. When a program is written that works with a graphical user interface there are classes available that describe a window frame, buttons, menus and whatever else you need for an effective user interface. To use all these classes you would `import javax.swing.*` (Java 2). If however you are going to change colors etc. , you need to `import java.awt.*` to facilitate these changes (this is a Java 1 package which also allows you to build an effective graphical user interface (GUI), as implied by the x, the swing classes in Java 2 extends this feature.) With a graphical user interface you would also import `java.awt.event.*` in order to use all the existing event handlers that have been coded. During compilation whatever is needed from these classes will be referenced by your program.

b) Classes designed within an organization.

Each program you code in Java is a class. Referring back to the example in 2.2.2.1 Designing a Java class, an example was given of an employee class. Any other application that needs to refer to an employee instance would then use the

Employee class. This means that programmers do not have to repeat code, defining the variables for an employee in a new application, merely create an instance of the existing employee class in their application. The programmers work should therefore become easier. As program code is not repeated there is less opportunity for errors to occur in the code. In addition should any changes be made to the employee object (class), the changes would automatically be implemented wherever the employee code is used. This would happen when the employee object is instantiated, there is no need for recompilation of source code as in procedural languages.

2.2.3 The Java alternative to subroutines.

There are no subroutines in Java, only classes. A class may be used as a subroutine. The java classes are objects, and these objects can be used as they are needed. There is much more flexibility in using a Java class as opposed to a COBOL subroutine.

Where there are a number of Java programs, or classes that have utility functions, and may be needed by a number of applications, these classes could be organised into a package. This involves defining each class as being a member of the package, and all the classes are then placed in one directory. To use these classes you then code an import statement followed by the package name at the beginning of the application where you would want to use part of the package. The import statement is the same as the statement that is used for

the already defined Java classes.eg. Import java.io.*. This is useful, as it becomes easy to use the classes, by importing the package rather than ensuring you have the classes present in the current directory at runtime.

2.2.4 Identify the key elements that define an object Oriented language with reference to Java.

An object oriented programming language is defined by its ability to satisfy three criteria. These are encapsulation, inheritance, and polymorphism. Sebasta states “A language that is object oriented must provide support for three key language features: abstract data types, inheritance, and a particular kind of dynamic binding” (Sebasta, 1999)

Encapsulation is what Sebasta refers to as abstract data types. This is where data is encapsulated or surrounded by the methods associated with it. This is satisfied by the Java programming language. Each class can define an object. The methods defined in the class are encapsulated with the data to form an object. The only way in which the data can be accessed, or changed is through the methods defined in that class.

Inheritance is the ability of a class to inherit from another. Java supports single inheritance. A class may inherit from another. This is implemented with the use of the extends clause in the class header. Although single inheritance may appear restrictive, any inheritance that the parent class has will also be inherited.

Multiple inheritance is accommodated through interfaces. Multiple interfaces may be implemented within a class. In programs where in is required that a common group of methods be applied, an interface can be used. This is common with event driven programming. Eg.

```
public class MyWindow extends JFrame
                implements ActionListener,
                           MouseListener
```

This will inherit data and methods from the class JFrame, as well as implement all the event handling methods in the interface ActionListener, and MouseListener.

Polymorphism, is accounted for in Java in a number of ways. Polymorphism refers to one object, having many shapes. (Naughton, 1996). This is a straightforward concept that is implemented effectively in Java. A method may have many implementations, and the implementation required is selected, based on which type of object is passed during method invocation. Overriding methods is an example of polymorphism. Where the same method name appears in a base class, and a sub-class (that inherits from the base class). When the method is called, the type of the calling object at run time will determine which method is executed.

Dynamic binding is one example of how Java implements polymorphism.

Inheritance is necessary in order to support dynamic binding. Given a class shape, that has a method printShape. . Further classes that extend shape are box, triangle, rectangle. These also each have a method called printShape. When the method printShape is invoked, the method associated with the actual object that invokes the method will be executed. Shape may hold an object of any type that extends shape.

```
Shape N = new Shape( );
Box B = new Box( );
N = B;   //assign the box object to the shape object
N.printShape(  );       // this will invoke printShape
                        // of the Box class
Triangle T = new Triangle( );
N = T;
N.printShape( );      // this would now invoke the
                    // printShape method for the
                    //  Triangle class, this is
                    // dynamic binding
```

Although the above code is procedural, to demonstrate the principal of dynamic binding, you can see the power in this feature. Given an array of objects that are type Shape. Any shape could be held in each element of the array. Yet to print each shape you need only code the call to one method. Java would determine which method to use depending on the type of the object that is held in the shape object in the array. Later when more shapes are added that further extend the shape class, there is no need to modify the program code. As long as a printShape method is coded in each class that extends shape, the processing of

the array will still work. Without dynamic binding, it would be necessary to have large nested if statements that would have to be updated each time a new shape is added.

Java therefore meets the three requirements of encapsulation, inheritance and polymorphism, for object oriented programming languages very effectively.

2.2.5 Key areas - where the language is used best.

Java originally emerged as a language for the Internet. The ability to embed Java code within a HTML page , so that the program becomes active when the HTML page is displayed is what caused a lot of initial interest in the language. These Java programs are known as applets, originally derived to represent a small application. However from this Java has developed into an all encompassing language.

A definition of Java as taken form the white paper entitled, "The Java™ Language: An overview" is: "Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language." (Sun, 2002). The paper continues to clearly define each of the key attributes given in this short definition. As can be seen from this definition, the objectives of Java are far more extensive than those set out for the design of the COBOL programming language. Java has therefore been developed to solve problems that involve far more than processing large volumes of data. Java has also developed far beyond being a programming language for the WEB. Java can be used in any application requiring a

good GUI user interface. However for Java to be really effective within an organization, it must be implemented with overall OO design as a basis within the organization. This overall OO perspective is what will supply the reusability of code and hence the benefit of being able to reuse code. The entire set of data maintained by the organization needs to be analysed as objects. Developing a set of stand alone systems, which do not interface with each other, and share objects is going to lead to repetition of code and data as occurs in a system based on procedural design principles.

2.2.6 Shortfalls of Java.

The extensive API classes could be viewed as a shortfall in that a programmer needs to constantly remain up to date with what is being published in Java. These are classes that are then available for use in developing systems.

COBOL has over 600 reserved words Java has 48 (Reed, 1999). As far as knowledge of the syntax of the language is concerned, learning the Java syntax is far simpler than learning COBOL syntax. Learning the Java syntax is however only a very small part of Java knowledge. When you have learned all the COBOL reserved words and statement formats you know all there is about the COBOL language. In Java you need to learn about the API classes. These have grown at a phenomenal rate from 200 in the first Java release to about 1600 in the Java 2 release (Wigglesworth, 2000). As a programmer it is only as you implement and use each API that you will effectively gain insight into the features and facilities already coded. Some of the API's you may never use. If you never do any network programming for example you would not work with

java.net, and so on. This need to keep on finding out about new classes in the Java API is part of the nature of computing today.

Although listed as a shortfall, this could equally be listed as an advantage, and argued in such a way as to support this viewpoint.

2.2.7 Event driven programming.

Whereas COBOL's objectives were narrow and well defined, Java's are very broad. This was discussed in 2.2.5. The emphasis in data processing has moved from batch processing to event driven processing. There are still requirements for batch processing programs, for example printing monthly statements is regarded as batch processing. This can be effectively programmed in Java. Event driven processing however is a form of processing that is quite different to batch processing. Instead of the programmer being able to procedurally lay down the sequence in which the program steps are executed, the user decides on the sequence of events by the selection made by the user. The programmer programs what is to be done in the case of each event, but the sequence of events are not in the programmers control. This is a key issue that programmers that are from a procedural background must deal with, and that is to merely code for each event and not on to the next action. The user is in control of the sequence. Java is designed to cater for this type of programming with ease. There are a number of event interfaces in Java that can be used for event driven applications.

Chapter 3 - Analysis, a comparison of the program code.

This chapter compares the programming languages COBOL and Java. This includes the presentation of program code that facilitates the comparison between the programming languages. This is necessary in order to establish how transformations could be accomplished, and the corresponding program statements that could be used.

3.1 Identify the similarities between the programming languages.

COBOL is a procedural language. Java also has procedural statements. The procedural part of both languages are very similar. This will be described further in this section. The way in which the procedural statements are coded, and executed are quite different. In COBOL all procedural code is in the PROCEDURE DIVISION, and the statements can act on any data defined in the DATA DIVISION. In Java, the way in which the programs are created is very different. The focus is on objects, a concept which is very new to a COBOL programmer. The data is defined as an object (or class) and the methods (procedures) that are to act on that data is encapsulated with it. The methods form part of the class. The only way in which the data (object) can be worked with is through those methods.

In COBOL understanding, or being able to use the procedural statements constitutes most of the language knowledge. Defining the data in the DATA DIVISION, is prescriptive. Usually the data files already exist, and it is clear how the records are to be described. The focus of the program effort is in the PROCEDURE DIVISION.

When a programmer has mastered the procedural statements, their knowledge of COBOL could be described as complete. There is a finite set of program statements that can be used in the PROCEDURE DIVISION. In Java all the corresponding procedural statements are defined in the Java programming language. Knowledge of these statements however, only constitutes a small part in being able to complete a Java class. There are other elements that must be understood, for example; inheritance and encapsulation as well being able to effectively use the many Java API's (more of which are being published all the time). The approach to an object oriented program is entirely different to the approach for a procedural program. It is this, the difference in the approach to problem solving that has been identified by both Labuschagne (1995) and Jansen van Rensburg (1998) as a stumbling block for established programmers to move into OO. Their studies focused mainly on systems analysis and design, which confirms that it is the approach to the problem solving which is very different. Deitel et al. (1998, 2002) in the fourth edition of "Java How to Program" have placed a large emphasis on object-oriented design and design patterns. This clearly addresses the approach to programming and problem solving. The procedural code within methods (Java) and the PROCEDURE DIVISION (COBOL) is very similar. Deitel et al. (1998, 2002) describe methods for algorithm development that have been used in procedural program development for many years, it is the difference in the approach to problem solving that is significant. It is therefore feasible that good procedural programmers will become good object oriented programmers. It is the approach to problem solving that must change. Mark Cathcart (1999) supports this viewpoint. A comparison of the procedural elements in the two languages follows.

Where there is a significant difference in the languages, these have been identified in *italics*.

3.1.1 Structure theorem - cohesion and coupling.

The **structure theorem** is the basis for structured programming theory. The structure theorem was developed from a paper by two mathematicians, Bohm and Jacopini who presented a paper in 1964 (Welburn, 1983). Basically they presented mathematical proof that any logic problem can be solved using only three control structures sequence, selection and iteration. These three control structures are present in COBOL and Java in a very similar fashion. This therefore indicates the similarity in the procedural statements in the languages. It is what surrounds these statements that is so different ie. the data.

COBOL	**Java**
Sequence MOVE “Y” TO NEW-PAGE. ADD 1 TO PAGE-COUNT. Statements are executed one after another.	**Sequence** String newPage = “Y”; pageCount += 1; Statements are executed one after another.

Selection	**Selection**

IF MINIMUM-BALANCE < 500
MOVE 5 TO SERVICE-CHARGE
ELSE
MOVE ZERO TO SERVICE-CHARGE
ENDIF.
 	
An EVALUATE statement could also be used for selection.
	if (minimumBalance < 800)
	serviceCharge = 8.00F;
	else
	ServiceCharge = 0.00F;
	
	A CASE statement could also be used for selection.

Iteration	**Iteration**

MOVE 1 TO COUNT.
PERFORM UNTIL COUNT > 5
DISPLAY COUNT
ADD 1 TO COUNT
END-PERFORM.
 	
There are other iteration statements available.
	int count = 1;
	while (count <= 5)
	{
	System.out.println (count);
	count = count + 1;
	}
	There are other iteration statements available.

Structured programming theory also describes the principles of cohesion and coupling. Cohesion refers to how closely related the statements within a module are. Here the more strongly related the statements the better. The highest form of cohesion is functional cohesion where the statements within a module perform a single function. Coupling refers to the strength of relationship between modules. Here the more loosely related the better. The lowest level of coupling is data coupling. See appendix A for a table of cohesion and coupling levels.

With reference to the COBOL programming language, a module would be referred to as a **paragraph**. Functional cohesion can be achieved by coding statements that only achieve a single function in a paragraph. In a Java program a module would be referred to as a **method**. A method generally only executes a single function. Therefore functional cohesion within a module can be coded in both languages.

Coupling, however is very different. A COBOL program cannot reach the lowest level of coupling, data coupling. "Data coupling is exhibited when two or more modules refer to the same nonglobal fields." Welburn (1983). A COBOL program cannot do this as the data is always defined in the DATA DIVISION and all modules (paragraphs coded in the PROCEDURE DIVISION) can access the data defined there. The next best level of coupling is stamp coupling. This is where two or more modules access the same nonglobal data structure. In COBOL this can be achieved using subroutines. The level of coupling that is evidenced in most COBOL programs is common coupling., the weakest level of coupling. Common coupling is when two modules reference the same global data. In COBOL all the modules reference the global data in the DATA DIVISION and therefore have common coupling.

Due to the object oriented nature of the Java language a programmer will tend to naturally develop program code that has data coupling. The only way in which data can be changed is through a method, and a method that has one function will generally work with one element of data.

3.1.2 Assignment of data.

In COBOL data can be placed in a field in three ways: as the result of a Read statement, a move statement, or as a result field in a calculation.

3.1.2.1 READING in data.

READ file-name INTO working-storage area AT END imperative statement.

This would place an entire record's data into the record description given in working storage. With the read statement an entire record from a file is being worked with (a logical record). The working-storage description must therefore have sufficient fields described so as to hold the entire record description.

The corresponding statements in Java are defined in a different manner. When working with data streams, a read statement can be issued that relates to each data type. In COBOL there are basically two data types: numeric and alphanumeric. The size of each field is defined in the record description. In Java there are primitive data types and complex data types. The size of the field is defined by the data type. A different read statement is given according to which data type is being read. Usually you would code a method, that reads the group of fields that define each record in a file. This read method would be defined within the class that has the definition of the object. Each read method then reads in a single field according to the data type defined in the read method (Roberts, 1999) Examples of these methods are given below :

Read Boolean()
readByte()
readChar()
readDouble()
readFloat()
readInt()

readLong()

readShort()

readUTF()

An example of a “read”method defined within a class is given in the example programs in chapter 4. It is important to note that the DATA DIVISION and READ statement code would be repeated in each COBOL program that refers to the data file. In contrast, in Java when an instance is created of the class that defines the data record, the “read” method can be used to read in the data, as the read method is defined within the class. The class can be used by a number of programs and the program code defining the object and the read does not have to be repeated.

3.1.2.2 MOVEing data.

MOVE field-name-1 TO field-name-2.

The move statement does exactly what it appears to do in moving data from field-name-1 to field-name-2. The data types that can be held in these fields can be either alphanumeric, or numeric. Both fields must be of the same data type in order for the statement to compile correctly. In the case of an alphanumeric move, the characters are moved such that the leftmost character from field-name-1 is moved to the leftmost character of field-name-2. This would continue with the next character working from the left and so on. When there are no more characters to transfer, ie. At the end of field-name-1, the transfer stops. If there are additional character positions available in field-name-2 then they are left as they were before the MOVE. If there are not enough positions in field-name-2 to hold all the data, then the transfer of characters stops as soon as the

receiving field (field-name-2) is full. If a numeric MOVE is executed, then the move will be aligned according to the position of the decimal point in both fields. If there is insufficient space for all the digits in the receiving field (field-name-2), then the excess digits will be truncated. This truncation will occur even if there are significant digits, to the left of the decimal point.

The maximum length for an alphanumeric field is 256 characters, and 18 digits for a numeric field. COBOL does therefore allow for a high degree of accuracy in an 18 digit numeric field. The size of each field is however governed by the data definition given by the programmer in the DATA DIVISION. When a numeric field is used that is too small, significant digits will be lost. A warning will only be given if the programmer has places an ON SIZE ERROR clause on the calculation, with appropriate messages should this error occur.

There are no MOVE statements in Java, you would use the assignment operator. The assignment operator, the equals symbol (=) is what is used to place a value in a variable, (or field). A simple assignment uses the = symbol as follows:

```
int a = 0;
```

The assignment works from right to left, so a multiple assignment can be given in one statement and the assignments would take place from right to left.

```
a = b = c = 0;
```

Firstly **c** gets the value zero, then **b** gets the value zero, and then **a** gets the value zero.

All variables must be of the same data type else errors can occur depending on the data type defined. In Java there are primitive data types and complex data types. The primitive data types are defined as follows:

Type Name	Value type	Memory used	Range of values
byte	integer	1 byte	-128 to 127
short	integer	2 bytes	-32768 to 32767
int	integer	4 bytes	-2147483648 to 2147483647
long	integer	8 bytes	-9223372036854775808 to 9223372036854775807
float	floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
double	floating-point	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
char	single character	2 bytes	all Unicode characters
boolean	true or false	1 bit	not applicable

(Information in this table from Savitch (2001; 57).)

With the primitive data types, the assignment of data always occurs with the exact same number of bits, so there are no truncations. Java wraps around in the numeric fields should the limits be exceeded during program execution. For example where a positive integer is exceeded, the answer would be the maximum negative integer less the number that the positive value was exceeded

by. No error will be given. It is therefore very important that the programmer choose the data type carefully.

COBOL works with the number in binary coded decimal format, where the number is defined as usage DISPLAY (the default). This uses 8 bits to hold the code for each digit in a number, up to a maximum of 18 digits. This allows for accurate calculations up to a maximum of 18 digits. A packed decimal number may be held, that uses approximately half the internal storage required when the usage is COMP-3, and the number is stored in its true binary value when the usage is COMP. Java works with the number in its true binary value, using the primitive data types. This results in Java doing faster calculations, as no data conversion is required. The number of bits used in storing the number is given in the table above. There is however a compromise in accuracy. COBOL calculations have a higher degree of accuracy, and very little rounding error. It is therefore recommended that double is used for all currency amounts in order to lessen the effect of rounding errors as far as possible, as float would only produce accuracy to six digits.

If an assignment is required where the data types are different, a type cast must be done to ensure that the data is correctly transferred. There are two types of type cast that can occur with primitive data types, implicit and explicit. An implicit type cast is done automatically by Java. This happens when a primitive data type that uses less bytes is converted to a data type of larger size. An explicit type cast is required when the new data type is smaller than the original.

In this case bits are truncated from the left (low order bits are retained). This may result in data values being changed.

An example of an implicit type cast.

```
int i = 7;
double d =  i ;      // the type cast is implied
                // there would be no error in coding
                // double d = (double) i;
```

An example of an explicit type cast.

```
int i = 7;
byte b = (byte) i ;
// without the type cast there would be a compiler //
error.
// ie. byte b = i;
// also if the value in i was greater than 127, or //
less  than -128, the significant right most bits //
would be lost, and the data value changed.
```

A type cast can be done with any assignment statement, including those that involve calculations.

Type casts can also be done on objects, arrays and interfaces. As there is no comparative code in COBOL, this will not be presented here.

3.1.3 Procedural code.

As has been discussed in 3.1.1 under the heading of the structure theorem there is a high correlation between COBOL and Java, as regards procedural code. In the interests of completeness, a comparison is presented. The following elements of procedural code will be discussed: branching, looping, calculations, arrays and tables.

3.1.3.1 Branching.

Both languages use an if statement to facilitate branching. COBOL refers to a condition, whereas Java refers to an expression. In COBOL the condition can be relational, a class test, sign test or condition-name test (boolean value). In Java the expression is any valid Java expression, that results in a true or false condition. The easiest to compare is the relational conditions. Remember the relational conditions apply to primitive data types. For a string value (complex data type) , you must use a string method eg. equals to compare values, you may not use a relational operator. In both COBOL and Java complex conditions can be evaluated using AND (&&) and OR (||) operators.(COBOL and (Java) respectively).

The COBOL if statement.	The Java if statement.

IF CONDITION
Statement-1
ELSE
Statement-2
END-IF.
	if (expression)
	Statement-1;
	else
	Statement-2;

In both COBOL and Java, statement-1 will be executed when the condition/expression is true, and statement-2 will be executed when the condition/expression is false. Both programming languages support nested if statements.

In COBOL the full stop at the end of the END-IF statement is what terminates the IF. Any number of statements may be listed in the place of statement-1 and statement-2. Only one full stop may appear at the end. If a full stop is coded anywhere in the IF statement, that is where the IF statement will end. This could result in a logic error, or where a full stop appears before the ELSE statement, will result in a compilation error.

In Java, the expression that results in a true or false condition must be enclosed in brackets ( ). When it is required that multiple statements are executed for the true or false result of the expression, then the statements must be placed within braces {}, forming a block of code. *Remember that the terminator ; is present at the end of each statement. This is a significant difference, in that a COBOL*

programmer is used to the terminator (.) terminating the if. This is not so in Java.

Given below is a table that shows the comparative relational conditions.

COBOL	**Java**
AND	&&
EQUAL TO, =	= =
GREATER THAN, >	>
GREATER THAN OR EQUAL TO, >=	>=
LESS THAN, <	<
LESS THAN OR EQUAL TO, <=	<=
NOT	!
NOT EQUAL TO, NOT = , <>	!=
OR	||

As can be seen there is a high correlation as regards branching in COBOL and Java. The statements within the true or false branch may call other paragraphs (COBOL) or methods (Java). Remember that the relational operators can only be applied to primitive data types in Java. It is common practice to compare alphanumeric values in COBOL. In Java, you would place the alphanumeric field in a String. You would then use the predefined String method equals to

compare String values. An example is given below:-

The COBOL if statement.	The Java if statement.
IF CO-BRANCH = “PAV” MOVE “PAVILLION” TO PR-BRANCH END-IF.	if (companyBranch.equals(“PAV”)) printBranch = “PAVILLION”;

COBOL has three other categories of condition testing, apart from relational. These are the class test, sign test and condition-name test. The class test determines if the data in a field is NUMERIC or ALPHABETIC. In Java there is no corresponding class test. The sign test in COBOL tests a numeric field for being POSITIVE, NEGATIVE or ZERO. In Java this is replaced by the corresponding relational condition, >0, <0, = 0. A condition-name test in COBOL, is where the value for a true condition has been indicated in working storage with an 88 level where the data field is defined. The condition-name becomes a boolean value that has a result condition of true when the 88 level value is present, and false if the value is not that given in the 88 level. The condition name test is easily replaced by a corresponding relational test in Java.

Branching can also be achieved with the use of the EVALUATE statement (COBOL) and switch statement (Java). These statements are different in that the values that can be tested for may be any relational condition in COBOL , but may only be an equal condition in Java of an integer (int) or character (char)

data type. In each case however multiple branches can be listed. In COBOL once an evaluated condition is true, all other conditions are ignored, and control passes to the end of the evaluate statement. In Java, a break statement must be coded to pass control to the end of the switch statement, if no other conditions are required to be tested.

3.1.3.2 Looping.

Looping in COBOL is done through different formats of the PERFORM verb.

```
PERFORM paragraph-name n TIMES.
PERFORM paragraph-name UNTIL terminate-condition.
PERFORM paragraph-name VARYING variable-name FROM initial-value BY
        increment-value UNTIL terminate-condition.
```

In all versions of COBOL up to and including COBOL 74, the terminate-condition is tested before the paragraph is executed. In COBOL 85 a programmer may specify WITH TEST AFTER, in the PERFORM statement, to force the paragraph to be executed at least once (as in the Java do .. while). Another very important feature is that in all formats of the PERFORM, the loop is exited when the terminate-condition becomes true.

Java has three looping structures, these are; while, do while, and the for loop.

```
while (boolean-expression)
        {
        statements
        }
```

The while loop test the boolean-expression first. The loop will execute only if the boolean-expression is true. When the boolean-expression is false the loop is exited. It is therefore possible that the statements in the loop will not be executed at all.

```
do      {
        statements
        } while (boolean-expression);
```

The do statement ensures that the statement block will always be executed at least once. The loop terminates when the boolean expression is false.

```
for (initialise-statement; boolean-expression; increment-statement)
        {
        statements
        }
```

The Java for loop has all the features of the PERFORM VARYING. The for loop terminates when the boolean expression is false.

Given this comparison of program code, it appears as if the loop structures available are very similar in execution. This is true, but there is a very significant difference. *In COBOL the loop terminates when the condition specified becomes true. In Java the loop terminates when the condition specified becomes false. It is important that the COBOL programmer takes note of this, as the way of thinking must be reversed in determining the termination condition for loops. This is a key difference in the languages.*

3.1.3.3 Calculations.

As the result of a calculation, data is moved into a result field. In COBOL there are a number of ways that a calculation can be done. There are a number of statements that perform calculations. Each statement has a defined result field. In the following examples, the result field is underlined. (Note: all possible arithmetic statements are not listed.)

a) COMPUTE identifier ROUNDED = arithmetic expression.

b) ADD identifier-1 [identifier-2] ... TO identifier-m.

c) ADD identifier-1 [identifier-2] ... GIVING identifier-m.

d) DIVIDE identifier-1 BY identifier-2 GIVING identifier-m REMAINDER identifier-n.

e) DIVIDE identifier-1 INTO identifier-2 GIVING identifier-m.

f) DIVIDE identifier-1 INTO identifier-2 .

g) MULTIPLY identifier-1 BY identifier-2 GIVING identifier-m.

h) SUBTRACT identifier-1 [identifier-2] ... FROM identifier-m.

i) SUBTRACT identifier-1 [identifier-2] ... FROM identifier-m GIVING identifier-n.

In Java, the assignment operator, the equals symbol (=) is used to place a value in a variable, (or field). A simple assignment has been discussed as a MOVE in 3.1.2.2, however in most cases multiple functions will be carried out on data before it is assigned to the variable stated on the left of the = symbol. For calculations an arithmetic expression is used, as in the COBOL COMPUTE statement. There are no individual arithmetic statements in Java. The calculations will store a result according to the java data type. If you require rounding or truncation of answers you can use the Math.ceil( ), Math.floor( ) methods respectively. This returns the closest integer value from a double value. The data types of the variables included in a calculation are significant. All variables should be of the same type else errors can occur. A type cast must be included where necessary so that the correct number of bytes is used for each variable in the calculation. A corresponding Java statement for each of the example COBOL statements is (the identifier names have been kept the same for simplicity, these are not names that would generally be used in Java):-

a) identifier = Math.ceil(arithmetic expression);

b) identifier-m = identifier-1+ [identifier-2] + ... + identifier-m;

c) identifier-m = identifier-1+ [identifier-2] + ... ;

d) identifier-m = identifier-1 / identifier-2;

identifier-n = identifier-1 % identifier-2;

e) identifier-m = identifier-2 / identifier-1 ;

f) identifier-2 = identifier-2 / identifier-1;

g) identifier-m = identifier-1 * identifier-2;

h) identifier-m = identifier-m - identifier-1 - [identifier-2] - ... ;

i) identifier-n = identifier-m - identifier-1 - [identifier-2] - ... ;

Therefore although there is not a corresponding separate statement for each COBOL calculation statement, any arithmetic can be taken care of in an arithmetic expression. A corresponding Java statement can be coded for all COBOL calculation statements. *A key difference that a COBOL programmer must get used to is that what would usually be coded as a number of separate calculation statements in COBOL , is completed in one line of code, in Java.*

3.1.3.4 Arrays and tables.

COBOL refers to tables, and Java refers to arrays. In COBOL the table is defined in the DATA DIVISION. Data can be loaded into the table in the DATA DIVISION, this is known as a compile time table, or from the PROCEDURE DIVISION, this is known as an execution time table, as the data is only present during program execution. The OCCURS clause is used to define the multiple occurrences of data. To refer to the elements in the table you would use a subscript. The subscript value runs from 1 to n (where n is the number of elements in an array). Generally the PERFORM...VARYING can be

used to search for, or access elements of the table. An example of a compile time TABLE that holds the number of days in each month is given below.

```
01      WS-DAYS-IN-MONTH-TABLE.
        05      WS-DATA                         PIC X(24) VALUE
                "312831303130313130313031".
        05      WS-DAYS-IN-MONTH REDEFINES WS-DATA PIC 99
                                                OCCURS 12.
```

Given that WS-MM holds the month number, WS-DAYS-IN-MONTH (WS-MM) would access the days in the month for the given month number. To access the days in the month for January you could code WS-DAYS-IN-MONTH (1).

A very similar example can be given in Java code. In Java multiple occurrences, or repeating groups, of data are referred to as arrays, not tables. To reference an element of the array, an index is referred to. (Please note, that this does not refer to an INDEX, which is a special data type in COBOL, but merely the terminology in referring to entries in an array). In COBOL the term subscript is used for the field that points to the appropriate entry in the table.. An index value runs from 0 to n-1, where n is the number of elements in the array. To define the array of number of days in each month as an array of integer values, and place the values in the array, you could code:-

```
int daysInMonth [] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

OR

```
int daysInMonth [] = new int [12];
daysInMonth [0] = 31;
daysInMonth [1] = 28;          // and so on for each of the 12 months
```

Given that monthNumber is the month number, daysInMonth [monthNumber-1] would access the days in the month for the given month number. To access the days in the month for January you could code daysInMonth [0].

Tables of multiple dimensions, defined in COBOL can also be coded in Java. A key difference is that each subscript is listed separately in Java. For example a table that is used to accumulate the total sales for each week, within each month of the year.

```
01      WS-SALES-TABLE.
        05      WS-MONTH            OCCURS 12 TIMES.
                10      WS-WEEK     OCCURS 5 TIMES.
                        15      WS-SALE      PIC 9(8)V99.
```

To access the sales for the first week in February you would refer to WS-SALE (2, 1). The first subscript refers to the month and the second to the week. The same code in Java, would appear as follows:-

```
int monthWeekSales [] [] = new int [12] [5];
```

To access the sales for the first week in February you would refer to

monthWeekSales [1] [0]. Notice that the index values are listed separately. The way in which the multiple dimensions are worked with is however very similar. In both COBOL and Java, more than two dimensions can be defined. Each new dimension becomes another subscript/index when accessing elements of the table/array.

In COBOL there is a feature to define tables so that a SEARCH or SEARCH ALL can be executed in order to find elements in the table. There is no corresponding feature in Java. A for loop, however, can be used just as effectively in searching for elements in an array. Java also has additional features that are very useful. One of these features is that Java maintains the length of an array. You can use the length to control a for loop etc. it is not necessary to look back at the array definition to determine its length. For example daysInMonth.length would return a value of 12. (Note to access the last element in the array you could use daysInMonth [daysInMonth.length-1]). This makes working with arrays more dynamic. If the array length changes, all procedural code will automatically change to refer to the new length.

COBOL tables can be converted to Java arrays very effectively. The key issues that need to be faced (memorised) by the COBOL programmer are the following:-

- *Refer to arrays not tables.*
- *Refer to an index, not a subscript.*
- *With multiple dimensions, list the indexés separately, in square*

brackets.

- *The first element of the array is at position zero, and the last element is the number of elements less one.*

3.2 Identify the differences between the two programming languages.

The following topics are areas where a difference between the programming languages has been identified. In some cases this is also an area where computing is developing and Java provides a solution where COBOL does not.

3.2.1 Data definition.

In COBOL all data is defined in the DATA DIVISION. This may be in the FILE SECTION, where the record format for each data file is defined, or in the WORKING-STORAGE SECTION, where calculation fields, and other work fields (eg page and line counters, end-of-file condition fields etc) are defined. The record definitions in the FILE SECTION will hold the current data for the most recent record read or written. The data held in the record definition will change with each READ operation that is executed. The WORKING-STORAGE fields are of a more permanent nature during program execution. Constant values can be defined in WORKING STORAGE by using a VALUE clause. All tables are also defined in WORKING STORAGE. All fields are available to be accessed by the program code in the PROCEDURE DIVISION. Each field is described with its own PICTURE clause. The picture clause defines the field as alphanumeric or numeric, giving the number of characters in each case, the position of

the decimal point, and presence or absence of a sign for numeric fields. As can be seen from this description the COBOL programmer has to be very specific when defining data fields. Giving the wrong definition for a single field in a record description will result in the data not being correctly read from the file. Remember that each program that used a data file had to repeat the record descriptions. The use of copy libraries did alleviate the problem of record definition to some extent, in that it did save the programmer the time of repeating the record description, but in each case there was a large repetition of procedural code. Due to this requirement for detail, and the necessity of defining all variables in the DATA DIVISION for use in the PROCEDURE DIVISION, a COBOL programmer has become accustomed to define all possible variables that they may need, before beginning with the PROCEDURE DIVISION, ie. The function of the program. This is a habit which has to change when a Java program is written. In order to explain this, the data definition methods used in Java need to be summarised.

Java variables can be defined in different ways. The definition of a variable determines which parts of the program have access to it. It is possible for a variable to be defined so that it is only available within a statement. An example of this would be a for loop counter which is only available to the block of statements within the for loop. The integer variable i is only available within the for loop given below.

```
for (int i = 1, i < = 10, i++)
        System.out.println ( “i = ” + i );
```

This is to such a degree, that you could define another i variable in a second for loop, and there would be no conflict identified by the compiler. This variable is local to the program block where it is defined, and is therefore commonly referred to as a local variable. A local variable is only available in the program block in which it is defined. Java programmers tend to define local variables as they are needed within the program code. Instance variables (also called class variables) are defined in the outermost block of a Java program, so that they are associated with the class header. In some cases this traditionally was at the end of the class, however most texts now present the instance variables at the beginning of a class under a class header. The instance variables define each attribute that is to exist for each object.(each instance of the class). If ten objects are instantiated (created), ten separate instances of the same variable will be created. Local variables must be initialised when they are used, instance variables are automatically initialised The instance variables are available to all the methods in the class. Instance variables, can also be declared with an access modifier of private or public. It is strongly recommended that instance variables are declared as private. This makes them only accessible to the methods of the class in which they are defined. This ensures encapsulation, where the data can only be accessed by the methods defined in that class. A class variable may also be defined as static. In the case of a static variable there is only one instance of this variable. Each object that is instantiated for the class will refer to this single variable. You can draw a parallel between the instance variables of a Java class as relating to a data record (the DATA DIVISION), the static and local variables are what would have been in the WORKING-STORAGE storage of a COBOL program. Constants are defined in Java as being final. This means they cannot be modified. eg. Static final double PRIME_INTEREST_RATE = 0.13 ; would declare

a constant interest rate. Traditionally constant values are named in capitals in Java.

Java variables can be a primitive data type, see 3.1.2.2 for the table of primitive data types. This works for COBOL numeric fields. The alphanumeric fields in COBOL would be defined as strings in Java. Java has a String class which defines a string object as well as a number of methods that work with strings. These methods enable a lot of flexibility in string manipulation. All COBOL processing/manipulation done on alphanumeric fields can be coded in Java using the string methods. It is also important to note that when working with GUI's in Java, all data displayed on the screen is in string format. To convert a string to a primitive data type the Java wrapper classes are used. Exception handling is used extensively to control errors in data type conversion. Exception handling is discussed in topic 3.2.3. The Java string offers far more flexibility to the programmer, than a COBOL alphanumeric field.

Apart from the instance variables that relate to an instance of an object for a class, a COBOL programmer must learn to define variables only when they are needed. Java will then remove these variables from memory when they are no longer required. This means that the COBOL programmer must rely on Java to clean up memory, and not be in total control of all that is taking place.

3.2.2 Event driven programming.

In COBOL, when a program is written to interact with a user via a screen, the programmer has to write the program code to control the interaction. If a menu is

presented to the user, the COBOL programmer has to set up a loop structure that controls the exit to the loop before writing the code that deals with the menu choices. All options must be controlled by the programmer. In Java, the event handling program logic has already been set up. The programmer must work with what is already there. Define components that can fire events, and then write program code that can handle each event. The Java programmer need only concern themselves with defining these relationships, and then writing the procedural code that needs to take place in the case of each event. The overall structure is taken care of in the Java Language, as you define Events, and Event Listeners, and implement the appropriate interface to handle each event. When a Java programmer defines a GUI (graphical user interface) component, be it a button, items on a menu etc., the programmer will assign an action listener. When a user interacts with the screen, for example clicking on a button with a mouse, an event is fired. ie. Java creates an object containing information about the event. Java also calls the appropriate event-handling method. An example is given below, showing the definition of a button, assigning an action listener, and then a portion of the event handling method - actionPerformed. For the full program listing please see APPENDIX B - Event driven program example - changing colors on the screen.

```
        :
        Button magentaButton = new Button("Magenta");
        magentaButton.setBackground(Color.magenta);
        magentaButton.addActionListener(this);
        buttonPanel.add(magentaButton);
        :
public void actionPerformed(ActionEvent e)
{
     Container newPane = getContentPane();
```

```
if (e.getActionCommand( ).equals("Magenta"))
          {
          newPane.setBackground(Color.magenta);
          }
     else if (e.getActionCommand().equals .....
```

Another example of an event driven program is listed in APPENDIX C - a program that plays a game where the player guesses a number between 1 and 1000.

The key issue is that the overall logic that controls the listening for an event is no longer the concern of the programmer. The Java programmer need only assign the listener, and the processing required when each event is fired. The listening and passing of control to the method to handle the event is controlled by Java. The event could be clicking on a button, selecting from a menu, moving the mouse etc.

3.2.3 Exception handling.

Exception handling is Java's method of dealing with errors. Exceptions are thrown by the Java language, and the programmer can catch or ignore the exception. If the exceptions are ignored, a program will often terminate abnormally with the exception listed by Java at the time of program termination. If the exception is caught and dealt with by the programmer, processing can continue. This is especially true in event driven programs, where after an appropriate error message, the user would typically respond to an error message, and then continue processing.

There are three elements involved in Java exception handling. An exception is thrown, a try block and a catch block. When an exception is thrown, this may be the result of a coded throws statement (in programmer defined exceptions), or thrown from a predefined Java class. The exception has an identity, and Java will identify that an

exception has been thrown. The section of program code that can throw the exception must be placed in a try block, or the throws clause must be defined on the method header. Immediately after the try block, or calling the method with the throws caluse, a catch block must be coded for each exception that can occur in the try block. (Note: One try block can throw multiple exceptions.) The value of this is that the Java programmer can code the procedure to be executed, as if there would be no error condition. Should an error occur, and an exception be thrown, control is immediately passed to the catch block for that particular exception. This means that Java will not try to continue the normal processing . Once the exception has been caught, program execution will continue from the point before the try block. If the program is event driven, that program will wait for the user to fire the next event. An example of some of the exceptions that can be thrown in the java.lang package are:-

Exception

ClassNotFoundException

IllegalAccessException

InstantiationException

NoSuchFieldException

NoSuchMethodWxception

RuntimeException

ArithmeticException

ArrayStoreException

ClassCastException

IllegalArgumentException

NumberFormatException

IndexOutOfBoundsException

ArrayIndexOutOfBoundsException

StringIndexOutOfBoundsException

NegativeArraySizeException

Indentation has been used to indicate inheritance, for example all these exceptions inherit from the class exception. The Java programmer must know where in the inheritance hierarchy, an exception fits in, as the more specific (lowest level in the hierarchy) exception must always be caught first, when there are multiple catch blocks for a try block. As all the exceptions listed above have been defined in Java, a programmer would not have to code a throw statement for them. These exceptions, should an error occur, would automatically be thrown. The programmers work is therefore to ensure that a try block is present, and that a catch block has been coded for the exception should it occur. The listing of the Java program MiniCalc, which catches the NumberFormatException is listed in APPENDIX D.

When an exception can be thrown in a method, and that method is to be used by a number of programmers, it is appropriate to include a throws clause in the method header followed by the list of exceptions that can be thrown in the method. It then becomes appropriate for the programmer that calls the method, to place the method call in a try block, and then follow the try block with a catch block for each exception.

A Java programmer may define their own exception classes, by extending the Exception class. In this case the throw clause must be coded when the exception occurs. The programs listed in APPENDIX E, show two programmer defined exceptions, as well as using the throws clause on the method header.

When working with data streams there is an anomaly, in that exceptions can be a normal part of program code. For example when processing binary files, an EOF exception is thrown when the end of the data file is reached. This is a normal data processing situation, which identifies that there is no more data on file. In this case when the programmer “catches”the exception, they can go ahead and close the data

stream as there is no more data to process. This has been used however, as it means when there is no more data, control is immediately passed to the catch, as the exception is thrown. The program code is simpler for the programmer, as they need to do no condition testing to determine if there is data to process. The program code is placed in a try block, and then coded as if there is always new data to process

Error processing in a COBOL program is part of the program code throughout the procedure division. In working with files, a programmer may code a DECLARATIVES SECTION. This can be used to process error conditions relating to file handling only. The DECLARATIVES SECTION in COBOL, although limited in use, does have a similar function in that should an error occur, control is passed immediately to this section of program code. A difference is that, once the error is identified and reported on as specified by the programmer, the COBOL program will terminate. The COBOL programmer has to test for, identify and process all error conditions at the point when they could occur during program execution.

Java's exception handling is a very powerful programming tool, that should be used to ensure that programs are as robust as possible, and will not terminate abnormally when unexpected errors occur (Deitel et al., 2002).

A key issue for the COBOL programmer to assimilate is that once an exception has been thrown and caught, normal program execution can continue. Also, as far as program control flow is concerned, when an exception is thrown, control passes immediately to the catch block.

3.2.4 Working with the web.

Java applets provide an easy connection to the Internet. This is what attracted a huge

interest in Java, when it was first released. Java applets (named to represent small applications), can run in any web page, and be viewed through a browser. The requirement is that the Java virtual machine, that executes the Java code is available where the applet is being viewed. Most texts, teach Java by starting with applets. The newer texts do not, but rather approach Java application programs first. This can be seen in the latest texts presented by Deitel and Deitel (2002) and Savitch (2001). As a COBOL programmer, there is no way to activate a COBOL program for use on the Internet. There is therefore no comparative program code.

A Java application can be changed to an applet, and then be run through a web page. An example of this is given in APPENDIX F, where the program to guess a number has been changed to an applet. The changes that were made to the program are as follows:-

- change the extends to extend Applet and not Frame.
- remove the **main** method. Most of the initialisation done in main is automatically applied in and applet. (eg setting window size, making a window visible)
- replace the constructor with an **init** method.
- delete any use of the WindowListener, when the html document is closed, the window is also automatically closed.
- remove **setTitle** and **setSize**, as applets have no titles, although you may give a title to the html page, and the sizing is controlled by the html page.
- code an html file, that will refer to the compiled applet (.class), and call it to be executed within the page. (An example is listed in APPENDIX F).

Once the applet has been successfully compiled, it can be executed from within an html page. To test an applet, you must activate the html page using appletviewer, or click

on the html file though a web browser. Most java applications can be changed to applets using the above steps. There are many security features that are set up with Java applets. One such feature is that an applet may not read or write to a disk, thus ensuring that when a web page is viewed by someone on the Internet, and an applet is activated, their computer resources would not be accessed, or written to. This means that any Java application that works with files would not be able to be created as an applet.

Java has many classes defined that can make applets or other GUI applications very interactive, and user friendly. The benefit is that the Java programmer can use these classes, and does not have to keep working from scratch. Graphics files in .gif and .jpeg format can be imported and manipulated from within a Java program. Animation and sound can be added through classes that already exist. This is a large area of study, due to COBOL having no comparative code, programming for the Internet has not been extensively studied.

3.2.5 Online documentation.

Java has a feature available, in which online documentation can be generated for a Java program. The documentation is generated by using javadoc, and referring to the java source file. The command would be:-

```
>javadoc GuessNumber.java
```

An html document is generated that can be viewed through a web browser. This document explains many features of the Java program. Any comments in the program source code that are enclosed between /*...comments here....*/, are also included in the html document. The java documentation file generated by javadoc for the GuessNumber program is listed in APPENDIX G.

This facility saves time and effort. Program documentation is a necessary part of programming, which is usually left to after the program is completed, or not at all. In Java, as a program is coded, the comments for the html file can be inserted. The documentation generated by javadoc includes many other aspects of the java program. As the documentation is an html file it is easy to publish so that another programmer can easily access it. All these features are not available for a COBOL program.

Documentation is what should be done as a program is developed. Usually this gets left to last, if it is done at all. Effective documentation is part of the definition of structured programming. In a COBOL program the documentation was placed as comments within the program. This is fine, but if the program was a subroutine, and another programmer wanted to use it, they would need to access the COBOL programs source code in order to read the documentation. Due to the time pressure, publication of documentation on a COBOL program, so that other programmers can gain access to the documentation was rarely done. This results in a lot of work being done again, as how can a programmer use a subroutine, if they do not know that it exists. Weak documentation (Chapin, 1997) has been the downfall of many computer systems.

Java goes a long way towards solving this problem. The Java API, which has many classes that can be used by a programmer, is available. As Java is published, and freely available at the Sun Microsystems web site, so is the documentation available on all the Java API's at http://java.sun.com/j2se/1.3/docs/api/index.htmlSun Microsystems. The javadoc feature however is a tool that can be used to effectively publish documentation on any Java class. This is a very powerful feature that will improve programmer productivity if used correctly. Instant online documentation is available for each Java class, over the Internet.

3.2.6 Developing models/simulations.

Java is ideally suited to developing models, or simulations of events. The graphics and imaging capabilities are extensive. There is also so much that has already been developed in Java, and is available for use. In many cases Java source code is released so that programmers can make their own changes, or use the objects as they are. Inheritance allows programmers to make their own additions or extensions to objects that have already been defined. The possibilities are endless for the types of models that could be developed. Also in Java the models can be developed as interactive, with sound and animation, the models therefore become very life like, and are therefore a useful tool. COBOL has not got the extensive graphics, animation, sound and reusability features, and is therefore not recommended as a modeling language.

Chapter 4 - Proposed Methodology for the transformation, COBOL to Java.

4.1 The proposed methodology.

From the study of COBOL and Java, the similarities and main differences in the programming languages have been identified. The areas of significance, as regarding program code, were discussed in chapter 3.

The nature of the programming languages is very different. This has been clearly identified in this study. Whereas COBOL was designed for commercial programming, with extensive file handling, and a large number of reports, Java was designed with very different objectives in mind. These differences were discussed in chapter 2, see 2.1.1, 2.1.5 for the COBOL objectives, and 2.2.1, 2.2.5, for the Java objectives. As these are clearly very different, the proposed methodology is for applications that are common in the COBOL programming Language. The proposed methodology will be applied to the following applications:- a control break report, a relative file maintenance application and a subroutine.

As this transformation is COBOL to Java, the COBOL programs will be transformed to Java programs. The two main components that need to be studied in the COBOL program, are the DATA DIVISION and then the PROCEDURE DIVISION. As Java works with objects that incorporate both data and methods, it is not possible to transform each section independently. The proposed transformation steps, have been placed in two sections. The first deals with the transformation of the DATA

DIVISION, and will include some code from the PROCEDURE DIVISION, and the second step deals with the remainder of the PROCEDURE DIVISION..

4.1.1 Transforming the DATA DIVISION.

The DATA DIVISION should be transformed into a separate class for each file that is defined. As each data file would contain information that describes an object of a different type. To identify the Java classes that define the data, they will be referred to as a Java Data Class (JDC), but these classes will include methods, not only data.

4.1.1.1 **Step 1** Define an object (JDC) for each record description.

Identify the record description/ or descriptions that the program processing is dependent on. In some cases there may be more than one record description, as more than one data file can be used. Each record description, must be defined in a Java class as an object. Each field in the data record will be an instance variable in the new Java class. To adhere to structured programming principles, each instance variable should be declared as private.

The alphanumeric fields will be defined as strings, and the numeric fields as the appropriate primitive data type. This would be int (9 digit accuracy), or long (18 digit accuracy) for numeric fields without a decimal point, and double(18 digit accuracy) for those numeric fields

with a decimal point. All Java numeric fields are automatically signed.. Boolean can be used for fields with only two states.

4.1.1.2 **Step 2** Include accessor and mutator methods for the appropriate instance variables in each class (JDC).

Some analysis needs to be done as regards which fields can be modified and which should not. A mutator method changes a value, replacing it with a new value. For a customer accounts object (record) it would be acceptable to have a mutator method for an address. The customers current balance however, should not be changed by a mutator method, the customer balance would be changed by a process, such as a purchase, receipt or return. A mutator method therefore is not necessarily defined for each instance variable.

In most cases an accessor method would be coded for each instance variable. Given the recommendation, that the instance variables are defined as private, the only way to access their value is through methods available in the defining class. To access the value in an instance variable, the accessor method would be used.

4.1.1.3 **Step 3** Define input and output methods that would work through a data stream.

As the record descriptions in the COBOL file section, are for data files (or report files), it is necessary to code the input and/or output methods that would place the instance variables for each object in a data stream, or retrieve the instance variables from a data stream. The READ and WRITE paragraphs in COBOL would indicate the type of processing that is going to be required for each object. These input/output methods are defined in the JDC. Java works with data files, as data streams. The JDC therefore contains the object, with each instance variable (the COBOL record is the object, and the data fields are the instance variables), as well as the input/output method for that object. The input/output method is directly associated with the object, as each instance variable must be referred to in the exact same sequence in the input/output method. The COBOL READ and WRITE statements work with the entire data record, the Java input/output method must therefore do the same by working with all instance variables of an object.

4.1.1.4 **Step 4** Include static variables and constants.

In order to do this, you need to study the WORKING-STORAGE SECTION in the COBOL program, as all the constants and working

fields needed in the COBOL program have been defined here. The working fields should not be defined as part of the new JDC, as working fields in Java are defined as they are needed in a process by the Java programmer. This saves space in memory, as the working fields are only allocated memory when they are needed, and then removed by the Java garbage collector when they are no longer required. In COBOL, the memory is permanently allocated for the fields in WORKING-STORAGE, and the memory remains allocated to the COBOL program until it terminates. There may be fields that are required to be processed with the data class, for example a vat percentage figure. This constant can be defined as a static final variable in the Java class (JDC), and will then be available when each Java object is instantiated. Only one instance of a static variable is created, and all instances of the class will refer to the single variable.

4.1.2 Transforming the remainder of the PROCEDURE DIVISION.

The Java Data Classes that have been defined are now available to be used by many applications. The processes that are executed in the PROCEDURE DIVISION that do not directly deal will data input or output, now become a separate Java class, this will be referred to as a Java Processing Class (JPC).

4.1.2.1 **Step 5** Define an a Java Processing Class (JPC) to complete the processing for each object, defined as a Java Data Class (JDC).

All the procedures that are defined to operate on each data file in the COBOL program that have not been transferred to a JDC, must be identified. These now become part of a second Java class. This class will instantiate objects of the JDC and use them to complete the processing required. Where the COBOL program has been well designed, and the paragraphs have functional cohesion, each COBOL paragraph will almost directly transform to a Java method. The high correlation between the procedural statements in each language will facilitate this transformation. The key points raised in chapter 3 will have to be kept in mind to make sure the transformation from COBOL to Java is correctly coded. These points are:-

- Branching.

  Remember that the terminator ; is present at the end of each statement. This is a significant difference, in that a COBOL programmer is used to the terminator (.) terminating the if. This is not so in Java.

- Looping.

  In COBOL the loop terminates when the condition specified becomes true. In Java the loop terminates when the condition specified becomes false. It is important that the COBOL programmer takes note of this, as the way of thinking must be reversed in determining the termination condition for loops.

This is a key difference in the languages.

- Calculations.

  A key difference that a COBOL programmer must get used to is that what would usually be coded as a number of separate calculation statements in COBOL , is completed in one line of code, in Java.

- Tables transformed to arrays.

  Refer to arrays not tables. Refer to an index, not a subscript. With multiple dimensions, list the indexes separately, in square brackets. The first element of the array is at position zero, and the last element is the number of elements less one.

4.1.2.2 **Step 6** More than one Java Processing class (JPC) may be defined from one PROCEDURE DIVISION.

Where a COBOL program uses multiple data files, a separate Java class (JDC) would be defined for each file. This would imply that the PROCEDURE DIVISION will have many processes that work with each data file. It is therefore feasible that multiple Java classes to process the data would be created.

An analysis of which process belongs to which data file would need to

be done. In some cases the processes will apply to both, or all data, and then a single processing class can be defined. However, where the processes can be identified as belonging to one data file only, it would be better as far as program design is concerned , to create separate processing classes.

4.1.3 **Step 7** Transforming a system (collection) of COBOL programs.

Where systems have been written in COBOL, there are a number of programs that work with the same data files. In COBOL programming, for each application, the data definition had to be repeated in each COBOL program. With Java this is not required. If the JDC has been correctly coded, all applications that work with the data, can refer to the same JDC, by instantiating the object. The work done transforming the first COBOL program in a system would not therefore be repeated for subsequent COBOL programs that work with the same data. Where the same data is used there is no need to recode the JDC, it will only be necessary to repeat step 4.1.2, to transform the PROCEDURE DIVISION of each program in the COBOL system.

4.2 Implementing the methodology.

4.2.1 Transformation: A Control Break Report.

Step 1 Identify the record description for the data file. The file being worked with in the COBOL program is a Sales transaction file. This is a LINE

SEQUENTIAL file, and therefore will be able to be read as a TEXT file in the Java program. Refer to APPENDIX H for the COBOL program code, and the output produced when the program is run. In APPENDIX I, the Java Data Class (JDC) is defined. This identifies that each record in the Sales Transaction file contains three elements of data. These are then defined as the instance variables in the SalesRecord object in the Java class (JDC) SalesRecord.java. Each data item is defined as private. This means that the data can only be changed by the methods in the SalesRecord class.

Step 2 The accessor and mutator methods are included for each variable. These can be clearly identified in the Java class in APPENDIX I. The accessor methods are identified by using “get”in the method header, and the mutator methods are identified by using “set”in the method header. An additional variable indicating each line of text had to be defined. This is in order to control the end of file condition which is indicated by a null value being returned from the read method. The Java utility class StringTokenizer was used to break down the individual fields from the input line. The utility class therefore had to be imported at the beginning of this class, SalesRecord.java.

Step 3 Identify the routines that deal with the input, and output when working with the data files record description. The Sales transaction file is read in order to print the control break report. Therefore only a read method is coded in SalesRecord.java. , this takes care of 400-READ-A-REC in APPENDIX H, the COBOL program.

Step 4 There are no static variables and constants that need be included in SalesRecord.java.

Step 5 Create further Java classes, that work with the first, and complete the processing of the COBOL program. The SalesRecord class does nothing active, it is an object that defines the instance variables in a record, and the methods to work with that data. The SalesRecord class may be used by other programs that need to work with the Sales transaction file, this is a good representation of what has been defined as a Java Data Class. To complete the processing, the Java Processing Class (JPC) WriteSalesReport.java has been coded. The program listing and output is presented in APPENDIX J. As this is a single processing requirement there is no need for more that one processing class. **Step 6** therefore is not required.

Step 7 would be completed for all other COBOL applications working with the employee data file.

WriteSalesReport.java (APPENDIX J) includes the COBOL procedure division code, as comments, to demonstrate the correlation in the procedural statements. It can be seen that the logical breakdown of the problem, can be

coded in a very similar fashion. There was a requirement to use some additional Java classes. The import statements at the beginning of the program are:-

```
import java.io.*;
import java.util.*;
import java.text.NumberFormat;
```

The **io** package is required to facilitate working with data streams. In this application, a text file is read (96ASS12.DAT), this is the same data file, that is read by the COBOL program. A text file is also created, that contains the control break report. (cbreport.txt).

The **util** package is required, in order to access the date from the operating system for the headings. The Date class is used as follows:

```
Date toDay = new Date();
outputStream.println( "\f"  +   toDay.getDay( ) + "/"
        + ( toDay.getMonth() + 1 ) + "/"+ ( toDay.getYear() +1900 ) );
```

(Note: \f is the escape sequence for a new page).

The java Date class, has the methods getDay( ), getMonth( ), getYear( ). The method getMonth( ) returns the month number , starting at zero. ie. January is month number 0. This means that if you were using the month number as an array index, it would not need to be manipulated. For output, one must be added to the month number. The method getYear( ), returns the year number

with the year 1900 as zero. The year 2002, is therefore year number 102, hence the necessity to add 1900 for output. Using the date class in the Java program does create warning messages, that the Date class has been updated in later versions of Java. This is noted as a deprecated API, and the compilation requires -deprecated as a parameter to the compilation. The program will compile successfully, but with warnings as regards the date class.

The text.NumberFormat class is required for the $ symbol, and two decimal places on the double value for saleAmount, when it is printed in the report.

```
NumberFormat moneyFormat =
                                NumberFormat.getCurrencyInstance ( Locale.US );
outputStream.println(  moneyFormat.format(salesData.getSaleAmount() );
```

All features that are required for the report, are available in Java, and the same output can be generated. A control break report is a very common COBOL application. With Java, if the application was being designed in Java, and not converted, it would typically work with a screen through a GUI, rather than a printed report.

4.2.2 Transformation: Relative File Maintenance.

Step 1 Identify the record description for the data file. The file being worked with in the COBOL programs is an Employee data file. Refer to APPENDIX K and L for the program code. In APPENDIX K, the record description is

given. This identifies that each record in the Employee master file contains four elements of data. These are then defined as the instance variables in the Record object in the java program (JDC) Record.java in APPENDIX M. Each data item is defined as private. This means that the data can only be changed by the methods in the Record class.

Step 2 The accessor and mutator methods are included for each variable. These can be clearly identified in the Java class in APPENDIX M. The accessor methods are identified by using “get”in the method header, and the mutator methods are identified by using “set”in the method header.

Step 3 Identify the routines that deal with the input, and output when working with the data files record description. In the CREMP program in APPENDIX K, the setting up of empty data fields, and writing out the record is given in the paragraph 200-PROC. In the ADDEMP program in APPENDIX L, the paragraphs 230-WRITE-MASTER and 420-READ-MAST work with the Employee master file. In APPENDIX M, you can see how this is implemented in Java. There are two read methods, and two write methods. These methods have been defined in the Record class, and are overloaded. In each case the sequential read and write method has no parameter passed to it. The direct read and write methods have the employee number passed as a parameter. The direct read and write methods use the seek method (defined in the Java RandomAcessFile class) to position the record pointer for the direct read, or write The record class can now be used by any program wishing to work with

the Employee master file. The read and write methods do not need to be repeated in any other classes. The direct Read is used in the Java class WriteRandomFile.java (to add an employee, APPENDIX O), and the sequential Read is used in ReadRandomFile.java (APPENDIX O) to display all the employees on file.

The linking of the data stream to the actual data file is also a method (openFile) in Record.java. This replaces the OPEN satement, and SELECT statement in the COBOL program. This means that the actual data file does not need to be defined in the classes that use the employee object. The open mode is however supplied as a parameter to the openFile method.

A closeFile method is also defined in Record.java. The JDC therefore has all the methods defined that would be required by any number of applications that use the employee random access file.

Step 4 There are no static variables and constants that need be included in Record.java.

Step 5 Create further Java classes, that work with the first, and complete the processing of the COBOL program. The Record class does nothing active. It is an object that defines the instance variables in a record, and the methods to work with that data. The Record class will be used by other programs that need to work with the Employee record, this is a good representation of what has

been defined as a Java Data Class. Given the two COBOL programs, the functions are to create a file of 100 blank records, that have an employee number of zero (APPENDIX K), and add active records to the employee file (APPENDIX L). These functions are then accomplished in the programs CreateRandomFile (APPENDIX N) which creates a file of 100 empty records, and WriteRandomFile (APPENDIX O), which places active records on file. An additional JPC that displays all active records has been given in APPENDIX O, ReadRandomFile.java. As more than one processing class has been created, **Step 6** has also been completed.

In a COBOL processing system there would be many other application programs that work with the employee file. As these are sample programs, a small representation of the data and processing has been given. **Step 7** would be completed for all other COBOL applications working with the employee data file. These further applications would typically include deleting employees that leave the company, changing details of employees, processing salaries, leave, tax returns, and many other reports.

4.2.3 Transformation: A COBOL subroutine.

The COBOL subroutine, is one which checks a number to ensure if the check digit is correct according to the modulus-11 method. The COBOL subroutine is listed in APPENDIX P. An example calling program is listed in APPENDIX Q. The transformation is of the COBOL subroutine. As there is no real data

defined in the subroutine, only parameters, the following applies for each of the transformation steps:-

Step 1 Identify the record description for the data file.

Step 2 The accessor and mutator methods are included for each variable.

Step 3 Identify the routines that deal with the input, and output when working with the data files record description.

Step 4 There are no static variables and constants that need be included in the Java Data Class.

In the case of the subroutine not referring to a data file, these first four steps fall away, as there is no Java Data Class to develop.

Step 5 Create further Java classes, that work with the first, and complete the processing of the COBOL program. The Java class CheckDigit is created. This class is listed in APPENDIX R. The CheckDigit class can be instantiated, whenever it is necessary to validate a number using the modulus-11 method. To use this class, you must provide the number as a string value when instantiating the class. The method isValid will return a boolean value of true, if the number is valid, and false if the number is not. A demonstration calling class has been coded, and is listed as APPENDIX S DemoCeckDigit.java.As this is a single processing requirement there is no need for more that one processing class. **Step 6** therefore is not required. **Step 7** is also not applicable.

4.3 Methodology evaluation.

This methodology highlights the advantages of object oriented programming. The object to be worked on, the data record is identified. The methods that work with the instance variables in the object are identified and become encapsulated with the data. These methods are defined here so that any changes to the instance variables can only be done from within the Java Data Class. Where there is more than one data file in the DATA DIVISION, it is probable that a Java Data Class would be developed for each file. This relates to object oriented programming, where each object is clearly defined. From then on all applications that work with that file (object) can use the appropriate Java Data Class. This effectively achieves reuse of program code. Where a COBOL system is transformed, the benefits of this will be clearly seen, in that once the Java Data Class is established, there is no need to repeat the data definition code. The data definition code was always repeated in each COBOL program's DATA DIVISION.

In the Java classes that need to use the data files, an instance of the JDC is created. There is no need to repeat the definition of the fields, or the read and write methods, only use them. This methodology therefore ensures that the benefits of object oriented programming are incorporated in the Java programs. A Java Processing Class is developed from the PROCEDURE DIVISION of each COBOL program. Again some analysis is necessary. Where the processing can be divided, as it relates to separate functions, separate Java Processing classes should be created.

The methodology proposed worked for the control break processing application, and

the relative file application. The subroutine application was a little different in that there was no need for a Java Data Class. The Java subroutine, is far simpler than the COBOL subroutine, and easy to use in any application. The Java code even allows for a field of any length, whereas in COBOL you could only use a seven digit number. This is a key area where Java code can be so much more useful, as it is easy to code flexible code. COBOL program code is very restrictive.

The disadvantage of this methodology is that it relies on a good knowledge of both COBOL and Java in order to be effectively implemented.

Chapter 5 - Object Oriented COBOL.

5.1 The origins of OO COBOL.

COBOL was developed in the late 1950's and became an ANSI standard in 1968. The second main release of COBOL was the COBOL-74 standard. Then came COBOL-85, which implemented quite a few changes in the language. (For example the test before/after on the PERFORM verb, and the inline PERFORM). Then the move to OBJECT ORIENTED COBOL. Remember that COBOL is controlled by a committee. The standards are published by the American National Standards Institute. Then the software developer, like MicroFocus, needs to ensure that they implement the language as specified by the committee. This is a very slow process. The OO COBOL standard was first released in 1997 (Chapin, 1997). At the OOPSLA conference 1993, there was a panel discussion on the status of OO COBOL (Van Stee et al., 1993). As can be seen the discussion, and intent was there, but it was still four years to publication. A long process, when you compare this to the rapid development that has taken place in Java since its first publication in 1995, the slow progress of COBOL into the field of object oriented programming can be seen. Releasing COBOL to run on the Internet has not even reached the discussion stage (Chapin, 1997). So Object Oriented COBOL has had a slow emergence into the market place, as opposed to other object oriented development languages.

5.2 Features of OO COBOL.

This has been completed with reference to the following authors, Chapin (1997), DeWard Brown (1999), and Grauer et al. (1998). The features of an object oriented language are described as encapsulation, inheritance and polymorphism. Object Oriented COBOL is discussed as regards these three features.

OO COBOL provides encapsulation. You can define data in a WORKING-STORAGE SECTION that is only accessible through the methods in that class. The reference is to a COBOL class, as the PROGRAM-ID statement is replaced by a CLASS-ID statement. The methods are listed in the PROCEDURE DIVISION, and each is identified by a **METHOD-ID. Method-name.** header, and **END METHOD Method-name.** at the end of the method. Each method has its own LINKAGE SECTION. , for the data fields that are local to the method. The entire class ends with an END CLASS class-name. The basic structure for an OO COBOL program is shown in APPENDIX T - OO COBOL program structure.

As can be seen in the class header, the COBOL class can inherit from a base class. Objects are created by using the reserved word NEW to instantiate a class. The inheritance in OO COBOL works in the same way as it does in Java. Descendant classes are referred to as subclasses, and ascendant classes as base classes or superclasses. All subclasses inherit the data and methods from the classes above them. A superclass may not reference methods in a subclass.

Polymorphism is provided for in that if two methods have been defined with the same name, COBOL will refer to the OBJECT-REFERENCE defined with the pointer to determine which method to call.

So OO COBOL does provide for the requirements of object oriented programming.

5.3 Features that OO COBOL does not provide.

Object oriented COBOL does not provide event driven programming, Internet applications, graphics, and many other features that have been included in Java from the beginning of its design. This means that although the benefits of object oriented programming can be gained by using OO COBOL, there are many other features in the programming languages that have been developed as pure OO languages that are just not available in COBOL.

A disadvantage of OO COBOL presented by Chapin (1997), is weak documentation. In order to benefit from OO design, the program code, or objects must be re-used. In order to use objects, programmers must know of the objects existence and features. There are two main ways in which a class may be used. 1) For processing purposes, where the class is used as is, or 2) as a base class that this then further extended by a programmer. Lamping (1993) discusses these as two interface methods, 1) a client interface for users of the objects of a class and 2) the specialisation interface, where the class is extended and overriding is implemented, in the paper entitled “Typing the Specialization Interface”. In each case the documentation required by the programmer

is different. In order to use a class, there must be documentation available on a class. There is no automation of documentation in OO COBOL. Java takes care of this problem with the javadoc feature.

5.4 Implications of implementing OO COBOL?

OO COBOL means that existing legacy systems in COBOL can be re-engineered to enjoy the benefits that object oriented systems provide. The transfer of data should be relatively easy to accomplish. The key is the difference between procedural COBOL and OO COBOL. The re-engineering of the system, to be properly implemented, must involve OO analysis of the entire system. The OO analysis is required in order to achieve the reuse of objects, and thereby gain the benefits of an OO system. To do this the programmers will have to be trained in OO principles. As has been identified in this study, it is the difference in approach which is so vast. Instead of breaking problems down into smaller and smaller processes, the system must be looked at as objects, composed of data and methods, and so on. Object oriented programming is substantially different from structured programming. A programmer must undergo a learning process to understand OO, and be able to implement it successfully. With this in mind, and given the need to re-engineer the system anyway, it may be better to re-engineer with an object oriented programming language, that provides a far greater scope than OO COBOL.

5.5 The impact of OO COBOL on this study.

The programming world, had moved into the new paradigm of object oriented programming in the 1980's. Java and the Internet exploded into the programming world in the 1990's. OO COBOL was published in 1997. In view of the other OO languages that have become available within the same time period, OO COBOL is a little late. It is trying to play catch up to a programming paradigm that has been around for a while. As this study has looked on the transformation from structured programming to OO programming, with reference to COBOL and Java, it was necessary to look at OO COBOL as an alternative for OO development. Unfortunately once the scope, and flexibility that is provided in the Java programming language has been experienced, OO COBOL appears very dull. There are so many benefits in using Java as opposed to OO COBOL. The learning still has to take place, whatever OO language is chosen, as a replacement for COBOL. If the effort is going to be made to transform a legacy system so as to take advantage of OO, it may be better to choose an OO language that has more features than OO COBOL.

Chapter 6 - Conclusion.

6.1 Were the objectives met?

To repeat some of the discussion that took place in Chapter 1:-

Theoretical creative research is defined by Melville (1996), "Theoretical creative research is about the discovery or creation of new models, theorems, algorithms, etc.". Olivier (1997) states "most of IT research endeavours to realise 'theories' to guide construction of automated systems" which supports that this is the type of research required in the field of Information Technology.

This has been accomplished in that a transformation methodology has been proposed, and tested as a result of this study. In order to achieve this, the following objectives were met.

.

6.1.1 To identify key elements that define a procedural language with direct reference to COBOL.

The key elements that define a procedural language were identified, and presented in Chapter 2, Section 2.1. The key elements that define COBOL as a procedural language are identified, in this study.

6.1.2 To identify key elements that define an object oriented language with direct reference to Java.

The key elements that define an object oriented language, and how Java meets these requirements is accomplished. This has been discussed in Chapter 2, Section 2.2.

6.1.3 The identification of similarities between the two programming languages.

The similarities between COBOL and Java have been highlighted and presented in Chapter 3, Section 3.1. The clear identification of the similarities is a key requirement towards developing a transformation methodology. This was therefore a key objective that was achieved and thereby enabled a methodology to be proposed.

6.1.4 The identification of the differences between the two programming languages.

This discussion takes place in Chapter 3. Section 3.2 the differences between the programming languages were clearly identified. These differences influenced the development of the proposed methodology.

6.1.5 The development of a methodology (series of procedures and rules to follow) to assist in the transformation from procedural languages to object oriented languages with specific reference to COBOL and Java.

This is accomplished as a result of the successful completion of the first four objectives. The examples of the implementation of the methodology are given in section 4.2. These examples illustrate a successful implementation of the methodology proposed.

6.2 The lessons learned.

6.2.1 Structured programming principals can be applied in the development of Java systems.

This dissertation has highlighted the fact that structured programming principals can be applied in the development of Java systems. The change to OO, changes the way in which the overall problem is approached. The separate objects are identified. These objects describe data, a JDC or Java Data Class, with the methods to access and manipulate the data, then the main processing functions become another object the JPC, or Java Processing class. The principals of structured programming theory can be applied just as effectively within the Java class, as they can be applied in a COBOL program. In terms of coupling, the principals can be adhered to, to a far higher standard than in COBOL. A good structured programming background, is a background in problem solving methods. This can be used just as effectively to solve Java problems, as COBOL problems. The correlation in procedural statements, makes the changeover from COBOL to Java straightforward. The few key

differences, as regards the actual statements were highlighted in chapter 3, topic 3.1.

6.2.2 OO design should be applied to systems, rather than programs.

As can be seen from the discussion in 6.1, to effectively transfer a system to OO, the changeover needs to be done for the system, not the individual programs. This study was approached from the point of view of transforming a COBOL program into a Java program. A methodology was proposed and implemented, and seems to be effective. This approach however, does not take into account a computer system. To effectively use OO technology, the overall system must be re-engineered to use OO. After implementing the transformation methodology, you will have a system written in Java, but to gain the full benefit of OO , it would be better to redesign, rather than work from the COBOL programs. The key is that the business processes presented in the COBOL programs have been well tested. A possible approach could be to redesign the data classes, and then complete steps 5, 6 and 7, as relates to the PROCEDURE DIVISION to get the Java Processing Classes. The COBOL paragraphs can become methods in Java, and the business processes would effectively be transferred to the new system.

6.3 Areas of further study.

The following are areas that could be further investigated as a result of this study.

6.3.1 Java and database.

Java provides easy database access through the JDBC class in the Java API. An interface has been written, and it is necessary to import java.sql.* in order to work with a database. Most texts that were studied did not give comprehensive examples of working with the JDBC. This is however an area where there are many business applications. There is a need for more work to be done on the practical implementation on working with data base applications from Java. The classes are available, but not discussed in the general text books. Other areas, such as animation, sound, multithreading, and networking are. An interesting project would be to set up a Java system that works with an MS Access database. There are also interfaces available that work with an Oracle database.

6.3.2 Working with COBOL'S indexed files.

Indexed files are used extensively in COBOL programs. The advantages of indexed files were discussed 2.1.5, as an area where COBOL is used best. In searches conducted, there was no reference found to processing indexed files from a Java program. In this study, COBOL LINE SEQUENTIAL files were effectively processed as text files in Java. The development of Java classes to

process indexed sequential files, is a study that could be conducted.

A key objective would be to ensure that the data files that are read by the COBOL programs could then be read by the Java programs. Classes that navigate the indexes, as well as maintain them would have to be coded. The transfer of the data files from the COBOL system to the Java system without having to modify them would be a very significant advantage.

It must be considered however, that in many cases with the change over to object oriented, it may be advantageous to also change the data structures to a database format.

6.3.3 A study of design patterns (relating to Java Programs).

In the development of the transformation methodology, COBOL to Java, two types of Java classes were described. These were referred to as a Java Data Class (JDC) and Java Processing Class (JPC). The design of these classes are inherently different, as the JDC is developed primarily from the COBOL DATA DIVISION, and only contains methods relating directly to the input and or output of data. The JPC is developed from the COBOL PROCEDURE DIVISION, and focuses on the business processes applied to the data. The JPC would work with the JDC to process the data. This is a type of design pattern. A number of design patterns have been recognised in object oriented software development. Jia (2000) mentions design patterns as being in three main

classifications, creational patterns, structural patterns and behavioral patterns. Deitel et al.(2002) discusses some behavioral design patterns, these are Chain-of-Responsibility, Command, Observer, Strategy and Template Method design patterns. In describing a design pattern, each pattern describes a problem which occurs over and over again, and then describes the core of the solution to that problem in such a way that you can use the solution to the problem over and over again. Design patterns were first applied to architectural designs by Christopher Alexander in 1979, and these principals were first applied to software design by Gamma et al. in 1995 (Jia, 2000). Budd (2000) also discusses design patterns, and presents a number of examples. The definition of design patterns, and their implementation as Java reusable objects, is a possible field of further study.

6.3.4 Test the proposed methodology on an operational legacy system.

To work in co-operation with industry, and convert a working COBOL system to a Java system, the study would need to note the effectiveness of the transformation methodology, and also measure its effectiveness. Designing a tool in order to measure the effectiveness of the methodology would be a part of the research process.

6.3.5 Work with a group of COBOL programmers, training them to become Java programmers.

This study focused on the transformation COBOL to Java in terms of program code. There are a number of COBOL programmers that need to change to object oriented programming. In this study some key issues were identified, that a COBOL programmer must understand in order to move to OO. A possible study would be to measure if the key issues identified here, being applied in a training program, make the transition easier for a programmer. There would need to be a control group of programmers that do not have training in order to complete the comparison. This would also determine how important these key issues are, in order to assist a programmer in making the change.

In the initial stages of the study, some articles were encountered that were negative as regards a programmers movement from COBOL to Java. Goodridge (2000), in an article discussing the IT labor market, stated that companies are reluctant to hire COBOL programmers for new development in Java, even when the COBOL programmers had undergone some training on the new Technology.

The theory that a COBOL programmer can be an effective Java programmer, would have value in being measured.

6.3.6 The user interface.

The user interface has not been extensively studied here. This is due to the fact that COBOL has limited corresponding features. The COBOL user interface is very much text based. Java provides a vast range of classes that can be used in the user interface. Design studies could range from the corresponding user interface for commercial, business applications, where data is entered via the screen to interactive visual screens that simulate events. An example of the interactive type of user interface, is to view the simulation result, for a modeling system that predicts elephant movement patterns (Duffy, 2001). To use graphics depicting actual elephants moving on the screen rather than color shading, or dots representing each elephant would be of more interest to a person viewing the simulation.

6.4 Final word.

This study has involved the analysis of structured design, of OO methodologies as well as the COBOL and Java programming languages. Upon setting out on the journey, there was no way of predicting the result. It is hoped that this dissertation has provided a way for the transformation to take place. The principals of structured programming, and the ability to solve problems is a requirement for programming skills whether they be applied to a procedural language, or an object oriented programming language. The study may also have resulted in no transformation methodology. I am pleased that there is a proposed methodology that can be used to transform COBOL programs to Java.

REFERENCE LIST

Arnold, K. and Gosling, J. 1998. **The Java™ Programming Language** , Second Edition, Sun Microsystems Inc., California, ISBN 0201310066.

Babcock, C. 2000. Java"Can Sun Control the Flood. **Inter@ctive Week** . June 5, 2000, v7 i22, p116.

Black, G. 1998. Java shadow looms over C++ despite lack of uses. **Computer Weekly** , Sept. 24, 1998 p14(1) .

Budd, T. 2000. **Understanding Object-Oriented Programming with Java**, updated edition, Addison Wesley Longman Inc., ISBN 0-201-61273-9.

Callaghan, D. 1999. Road to Java can take different paths (migrating from RPG to Java), **MIDRANGE Systems** , March 1, 1999 v12 i3 p28(1)).

Cathcart, M. 1999. ÖO alternatives for S/390. **Enterprise Systems Journal**, April 1999v14i4p34(1).

Cavanaugh, K. 1999. Java's Uncharted Waters . **Wall Street & Technology**, March 1999 v17 i3p30(1).

Chapin, N. 1997. **Standard Object-oriented COBOL** . Wiley Computer Publishing, ISBN 0-471-12974-7. p69.

Deitel, H. M., Deitel, P. J. 1998. **Java How to Program**. Prentice Hall Inc. New Jersey. 2nd edition. ISBN 0-13-899394-7.

Deitel, H. M., Deitel, P. J. 2002. **Java How to Program**. Prentice Hall Inc. New Jersey. 4th edition. ISBN 0-13-0345151-7.

DeWard Brown, G. 1999. **Advanced COBOL, for structured and Object-Oriented Programming.** Third Edition. John Wiley & Sons, Inc., ISBN 0-471-31481-1.

Doke, E. R. and Hardgrave B. C. 1999. **Java for the COBOL Programmer**. Cambridge University Press, UK, ISBN 0 521 65892 6.

Duffy, K. 2001.Elephant tracking research. **Technikon Natal Research Day 2001.** 13 September 2001.

Eliens, A. 1995. **Principles of Object-oriented Software Development** . Addison-Wesley, ISBN 0-201-62444-3.

Gaudin, S. 1998. Java holds its own, starts to make inroads. **Computerworld** , Jan. 19, 1998

v32 n3 p32(1) .

Goodridge, E. 2000. Older IT Professionals Struggle with Age Bias - DESPITE THE IT LABOUR SHORTAGE, MANY EXPERIENCED WORKERS SAY THEY CAN'T FIND JOBS, **InformationWeek** , Oct. 9, 2000, p228

Grady, R. B. 1997. **Successful Software Process Improvement** . Hewlett-Packard Professional Books, Prentice-Hall, ISBN 0-13-626623-1. p2.

Grauer, R.T, Vazquez Villar, C., Buss, A. R. 1998. **COBOL, From Micro to Mainframe, Preparing for the New Millennium.** Third Edition. Prentice-Hall. ISBN 0-13-790817-2.

Henderson-Sellers, B. and Edwards, J. M. 1994. **Booktwo of Object-Oriented Knowledge. The Working Object** . Prentice-Hall Australia Pty. Ltd., ISBN 0 13 148 404 4.

Hunter, D. 1999. What should we teach our Commercial programmers in the new millennium, **Technikon Natal, Commerce Research Conference 1999** , p4

InformationWeek, User Survey. 1998. Objects on the rise . **InformationWeek** Sept 28, 1998 n702 p192(1) .

Jansen van Rensburg, M. 1998. **Pitfalls and guidelines in the transition to object oriented software design methodologies.** University of the Witwatersrand, M.Sc Engineering.

Jia, X. 2000. **Object-Oriented Software Development using Java, principles, patterns, and frameworks.** Addison Wesley Longman Inc., ISBN 0-201-35084-X.

Labuschagne, I. L. 1995. **Framework for object oriented analysis: Adopting object oriented analysis in software development** . Rand Afrikaans University, M Sc. Computer Science, p 5.

Lamping, J. 1993. Typing the Specialization Interface. **Conference on Object-Oriented programming systems, languages, and applications. (OOPSLA 1993), Eighth Annual Conference.** , published proceedings.

Langley, N. 1999. The Language of the millennium bug. **Computer Weekly**. Dec. 9, 1999 p58.

Ledgard, H.F. 1996. **The Little Book of Object-Oriented Programming** . Prentice-Hall Inc. New Jersey, ISBN 0-13-396342-X.

Lemay, L. 1997. **Java 1.1™ Interactive Course** , Waite Group Press, California, ISBN 1-57169-083-2.

Martin, J. and Odell, J. J. 1995. **Object-Oriented Methods a Foundation** . Prentice-Hall Inc. USA, ISBN 0-13-630856-2.

Melville, S. 1996. **Research Methodology, An introduction for science and engineering students.** Juta & Co. Ltd., ISBN 0-7021-3562-3 .

Merritt, S. M.. and Stix, A. 1997. **Migrating from Pascal to C++** . Springer-Verlag New York Inc. , ISBN 0-387-94730-2.

Naughton, P. 1996. **The Java HandBook** . Osborne McGraw-Hill, California, ISBN 0-07-882199-1.

Olivier, M. S. 1997. **Information Technology Research, A Practical Guide.** Printed by MS Olivier.

Philippakis, A. S. and Kazmier, L. J. 1987. **Advanced COBOL** . McGraw-Hill, United States of America, 2nd Edition, ISBN 0-07-049813-X.

Reed Doke, E and Hardgrave B. C. 1999. **Java for the COBOL Programmer** . Cambridge University Press, United Kingdom, ISBN 0521 65892 6.

Roberts, S. and Heller, P. and Ernest, M. 1999. **Java™ 2 Certification Study Guide**, Sybex Inc., United States of America, ISBN: 0-7821-2700-2.

SABINET. 1999. www.sabinet.co.za accessed on 14 September 1999.

Savitch, W. 2001. **Java, An Introduction to Computer Science.** Prentice-Hall Inc. New Jersey, 2nd edition, ISBN 0-13-031697-0.

Savitch, W. 1999. **Java, An Introduction to Computer Science.** Prentice-Hall Inc. New Jersey, 1st edition, ISBN 0-13-287426-1.

Sebasta, R. W. 1993. **Concepts of programming Languages.** The Benjamin/Cummings Publishing Company Inc. , ISBN 0-8053-7136-3.

Sebasta, Robert W. 1999. **Concepts of programming Languages.** Addison Wesley Longman Inc., 4th edition, ISBN 0-201-38596-1. p 437

Stern, N. and Stern, R. A. 1991. **Structured COBOL programming** . John Wiley & Sons, United States, 6th edition, ISBN 0-471-53400-5.

Sun Microsystems. 2002. **The Java™ Language: An Overview** . Available at http://java.sun.com/docs/overviews/java/java-overview-1.html. (12 April 2002).

Swift, C. 1999. “The Rise of Java, Enlightening the Enterprise”, **Network Times** , Issue 104, July 1999, Pg 25.

Van Stee, J.G., Clarke, D., Filani, D. Lenkov, D.Obin, R. 1993. Status of Object-Oriented COBOL, a Panel discussion, **Conference on Object-Oriented programming systems, languages, and applications. (OOPSLA 1993), Eighth Annual Conference.** , published proceedings.

Welburn, T. 1981. **Structured COBOL, Fundamentals and Style** . Mayfield Publishing Company, First Edition, ISBN 0-87484-543-2.

Welburn, T. 1983. **Advanced Structured COBOL** . Mayfield Publishing Company, First Edition, ISBN 0-97484-558-0.

Welburn, T. 1983. **Advanced Structured COBOL** . Mayfield Publishing Company, First Edition, ISBN 0-97484-558-0. Pg 6.

Welburn, T. and Price, W. 1995. **Structured COBOL, Fundamentals and Style** . Mc Graw-Hill, California, 4th Edition, ISBN 0-07-113544-8.

Wilson, L. B. and Clark, R.G. 1993. **Comparative Programming Languages** . Addison-Wesley, 2nd ed., ISBN 0-201-56885-3. p317.

Wigglesworth, J. and Lumby, P. 2000. **Java ™ Programming Advanced Topics** . Course Technology:Thompson Learning, Canada, ISBN 0-7600-1098-6.

Yourdon, E., Gane, C., Sarson, T. and Lister, T.R. 1979. **Learning to Program in Structured COBOL Parts 1 and 2.**, Prentice Hall, New York, ISBN 0-13-527713-2.

APPENDIX A

Levels of Cohesion	**Independence from other modules**	**Desirability**
Coincidental	Low	Least desired
Logical	Low	⎜
Temporal	Low	⎜
Procedural	Medium	⎜
Communicational	Medium	⎜
Sequential	High	↓
Functional	Very high	Most Desired

Levels of Coupling	**Coupling Attribute**	**Desirability**
Content	High (or tight)	Least Desired
Common	⎜	⎜
External	⎜	⎜
Control	⎜	⎜
Stamp	↓	↓
Data	Low (or loose)	Most desired

APPENDIX B Event driven program example, changing colors on the screen.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*************************************************************
 *Simple demonstration of putting buttons in a panel. If you do
 *not see the colored part of the window with text in it, use your
 *mouse to increase the size of the window and it will appear.
 *************************************************************/
public class ColorPanelDemoTut6 extends JFrame implements ActionListener
{
   public static final int WIDTH = 300;
   public static final int HEIGHT = 200;

   public static void main(String[] args)
   {
      ColorPanelDemoTut6 guiWithPanel = new ColorPanelDemoTut6();
      guiWithPanel.setVisible(true);
   }

   public ColorPanelDemoTut6()
   {
      setTitle("Color and Panel Demonstration");
      setSize(WIDTH, HEIGHT);
      setBackground(Color.blue);
      addWindowListener(new WindowDestroyer());

         Container newPane = getContentPane();

// Create the panel to hold the six color buttons
// Set the layout of the panel to gridLayout

      JPanel buttonPanel = new JPanel();
      buttonPanel.setLayout(new GridLayout(3, 2));

      Button redButton = new Button("Red");
      redButton.setBackground(Color.red);
      redButton.addActionListener(this);
      buttonPanel.add(redButton);

      Button greenButton = new Button("Green");
      greenButton.setBackground(Color.green);
      greenButton.addActionListener(this);
      buttonPanel.add(greenButton);
```

```
        Button pinkButton = new Button("Pink");
        pinkButton.setBackground(Color.pink);
        pinkButton.addActionListener(this);
        buttonPanel.add(pinkButton);

        Button yellowButton = new Button("Yellow");
        yellowButton.setBackground(Color.yellow);
        yellowButton.addActionListener(this);
        buttonPanel.add(yellowButton);

        Button magentaButton = new Button("Magenta");
        magentaButton.setBackground(Color.magenta);
        magentaButton.addActionListener(this);
        buttonPanel.add(magentaButton);

        Button cyanButton = new Button("Cyan");
        cyanButton.setBackground(Color.cyan);
        cyanButton.addActionListener(this);
        buttonPanel.add(cyanButton);

// Add the panel to the top of the screen

        newPane.setLayout(new BorderLayout());
        newPane.add(buttonPanel, "North");


        Button memoButton = new Button("Watch me change color");
        memoButton.addActionListener(this);
        newPane.add(memoButton, "South");

    }


    public void actionPerformed(ActionEvent e)
    {
      Container newPane = getContentPane();
      if (e.getActionCommand().equals("Red"))
       {
          newPane.setBackground(Color.red);

       }
       else if (e.getActionCommand().equals("Green"))
       {
         newPane.setBackground(Color.green);

       }
       else if (e.getActionCommand().equals("Pink"))
```

```
        {
          newPane.setBackground(Color.pink);
        }
        else if (e.getActionCommand().equals("Yellow"))
        {
          newPane.setBackground(Color.yellow);
        }
        else if (e.getActionCommand().equals("Magenta"))
        {
          newPane.setBackground(Color.magenta);
        }
       else if (e.getActionCommand().equals("Cyan"))
        {
          newPane.setBackground(Color.cyan);
        }
    }

}
```

APPENDIX C - Guess the number game.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/*************************************************************
 *The Guess a number game
 *************************************************************/
public class GuessNumber extends Frame implements ActionListener
                                 , ItemListener
{
   private Label message1, message2, message3;
   private TextField text2;
   private int rNo;
   private Button button1, button2;
   private JCheckBox bold, italic;
   private String s1, theText = "Watch me!";

   public static final int WIDTH = 400;
   public static final int HEIGHT = 300;

   public static void main(String[] args)
   {
      GuessNumber guiGuessNumber = new GuessNumber();
      guiGuessNumber.setVisible(true);
   }

   public GuessNumber()
   {
      setTitle("Guess the number game");
      setSize(WIDTH, HEIGHT);
      setBackground(Color.cyan);
      addWindowListener(new WindowDestroyer());

      setLayout(new GridLayout(3,1));
// The third entry in the grid is for the background color

      Panel descripPanel = new Panel();
      descripPanel.setLayout(new BorderLayout());

      message1 = new Label("I have a number between 1 and 1000");
      message2 = new Label("Can you guess my number?");
      message3 = new Label("Please enter your first guess:");
      descripPanel.add(message1, "North");
      descripPanel.add(message2, "Center");
      descripPanel.add(message3, "South");
```

```
        add(descripPanel);

        Panel buttonPanel = new Panel();
        buttonPanel.setLayout(new FlowLayout());

        s1 = "Start Game";
        button1 = new Button(s1);
        button1.addActionListener(this);
        buttonPanel.add(button1);

        button2 = new Button ("Am I right?");
        button2.addActionListener(this);
        buttonPanel.add(button2);

        text2 = new TextField(10);
        buttonPanel.add(text2);

         bold = new JCheckBox("Bold");          // this code is to test how a checkbox
         bold.addItemListener(this);       // works
         italic = new JCheckBox("Italic");
         italic.addItemListener(this);
         buttonPanel.add(bold);
         buttonPanel.add(italic);
         add(buttonPanel);
    }

   /** Test for online help

Key in a number of your choice

If you are close to the number
 within 20, the screen will be red

If you are much lower than the number
          the screen will be green

If you are much higher than the number
          the screen will be orange

          end of test  */
```

```
 public void actionPerformed(ActionEvent e)
 {
   if (e.getActionCommand().equals("Start Game"))
    {
     text2.setText("");
        rNo = (int) (Math.random()*1000 + 1);
     button1.setVisible(false);
     theText = "You can start guessing now  ";
    }
    else
    if (e.getActionCommand().equals("Am I right?"))
       {  String s = text2.getText();
          int val = Integer.parseInt(s);
               int diff = Math.abs(val-rNo);
               text2.setText("");
               if(val < rNo)
                  {
                  setBackground(Color.green);
                  theText = "Too low";
             message3 = new Label("Too low");
                  }
                else  if (val > rNo)
                         {
                         setBackground(Color.orange);
                         theText = "Too high";
                         }
                        else
                        {
                         setBackground(Color.cyan);
                          theText = "YOU'VE GOT IT  " + s ;
                          text2.setText(s);
                          button1.setVisible(true);
                          }
           if(diff < 20)
                 setBackground(Color.red);

       }


   repaint(); //force color and text change
 }

       public void itemStateChanged ( ItemEvent e)   // this is for a check box
{                                                                                  // event
   if ( e.getSource() == bold )
         if ( e.getStateChange ( ) == ItemEvent.SELECTED )
                  button2.setFont( new Font ("Serif", Font.BOLD , 12 ) );
```

```
            else
                        button2.setFont( new Font ("Serif", Font.PLAIN , 12 ) );
     if ( e.getSource() == italic )
            if ( e.getStateChange ( ) == ItemEvent.SELECTED )
                       button2.setFont( new Font ("Serif", Font.ITALIC , 12 ) );
            else
                        button2.setFont( new Font ("Serif", Font.PLAIN , 12 ) );

   }
    public void paint (Graphics g)
    {
          g.drawString(theText, 140, 280);
    }


}
```

APPENDIX D - MiniCalc, a demonstration of the NumberFormatException.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MiniCalc extends JFrame implements ActionListener
{
   public static final int WIDTH = 400;
   public static final int HEIGHT = 300;

   private JTextField theText, numberOne, numberTwo, numberAns;
   private String helpMessage =  "Enter a numeric value in Number1 and Number 2"
                                       + " \n " + " and then select a numeric operation "
                                       + "\n the answer will then be displayed. \n";
   private String theMessage = "Please enter Number 1, Number2 and select function.";
   private double n1, n2;

   public MiniCalc()
   {
      setSize(WIDTH, HEIGHT);
      addWindowListener(new WindowDestroyer());
      setTitle("Mini Calculator JW");
      Container contentPane = getContentPane();
      contentPane.setLayout(new BorderLayout());


      JPanel textPanel = new JPanel();
      textPanel.setBackground(Color.magenta);

      theText = new JTextField(theMessage, 30);
      theText.setBackground(Color.white);
      theText.setEditable(false);
      theText.setToolTipText("You may not enter data here");
      textPanel.add(theText);
      contentPane.add(textPanel, BorderLayout.NORTH);

      JPanel mainPanel = new JPanel();
      mainPanel.setBackground(Color.cyan);
      mainPanel.setLayout(new GridLayout(4,1,5,10));

     JPanel panelOne = new JPanel();
     JLabel labelOne = new JLabel( "Enter number 1" );
     numberOne = new JTextField(10);
     numberOne.setEditable(true);
     panelOne.add(labelOne);
     panelOne.add(numberOne);
```

```
mainPanel.add(panelOne);

JPanel panelTwo = new JPanel();
JLabel labelTwo = new JLabel( "Enter number 2"  );
numberTwo = new JTextField(10);
numberTwo.setEditable(true);
panelTwo.add(labelTwo);
panelTwo.add(numberTwo);
mainPanel.add(panelTwo);

JPanel functionPanel = new JPanel();
functionPanel.setLayout(new GridLayout(1,5,5,10));

 JButton bAdd = new JButton("+");
 bAdd.addActionListener(this);
 functionPanel.add(bAdd);

 JButton bSub = new JButton("-");
 bSub.addActionListener(this);
 functionPanel.add(bSub);

 JButton bMult = new JButton("X");
 bMult.addActionListener(this);
 functionPanel.add(bMult);

 JButton bDiv = new JButton("/");
 bDiv.addActionListener(this);
 functionPanel.add(bDiv);

 JButton bClear = new JButton("Clear");
 bClear.addActionListener(this);
 functionPanel.add(bClear);

mainPanel.add(functionPanel);

JPanel panelAns = new JPanel();
JLabel labelAns = new JLabel("Answer");
numberAns = new JTextField(10);
numberAns.setEditable(false);
panelAns.add(labelAns);
panelAns.add(numberAns);
mainPanel.add(panelAns);

contentPane.add(mainPanel, BorderLayout.CENTER);

 JPanel buttonPanel = new JPanel();
 buttonPanel.setBackground(Color.blue);
```

```
    buttonPanel.setLayout(new FlowLayout());
    JButton exitButton = new JButton("Exit");
    exitButton.addActionListener(this);
    buttonPanel.add(exitButton);
    JButton helpButton = new JButton("Help");
    helpButton.addActionListener(this);
    buttonPanel.add(helpButton);
    contentPane.add(buttonPanel, BorderLayout.SOUTH);

}

public void actionPerformed(ActionEvent e)
{
    String actionCommand = e.getActionCommand();
    if (actionCommand.equals("+"))
        { convert( );
          numberAns.setText(Double.toString(n1+ n2));
         }
    else if (actionCommand.equals("-"))
        { convert();
          numberAns.setText(Double.toString(n1- n2));
         }
    else if (actionCommand.equals("X"))
        { convert();
          numberAns.setText(Double.toString(n1* n2));
        }
    else if (actionCommand.equals("/"))
        {
         convert();
         numberAns.setText(Double.toString(n1/ n2));
         }
      else if (actionCommand.equals("Clear"))
        { numberOne.setText("");
          numberTwo.setText("");
          numberAns.setText("");
          theText.setText(theMessage); }
    else  if (actionCommand.equals("Exit"))
          System.exit(0);
    else if (actionCommand.equals("Help"))
        JOptionPane.showMessageDialog(
                null, helpMessage,"Help information",
                JOptionPane.INFORMATION_MESSAGE );
   else
        theText.setText("Error in memo interface");
 }

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

```
    {
        MiniCalc guiMemo = new MiniCalc();
        guiMemo.setVisible(true);
    }
    public void convert()
    {
        try
         {
         n1 = Double.parseDouble(numberOne.getText());
         }
        catch (NumberFormatException e)
         {
            theText.setText("Number 1 must be numeric");
         }
        try
         {
           n2 = Double.parseDouble(numberTwo.getText());
         }
        catch (NumberFormatException e2)
         {
            theText.setText("Number 2 must be numeric");
         }

  }


}
```

APPENDIX E - JNDateConverter uses two programmer defined exception classes, MonthExceptin and DayException. The use of the throws clause on the method header is also demonstrated.

```
/*****************************************************************
 *
 * File name: JNDateConverter.java
 *
 * Converts numerical day/month/year format to alphabetic day month year.
 *
 * The date is also validated.
 *
 *****************************************************************/

public class JNDateConverter
{
  private static String input, s1, s2;//To be thrown away
  private static int month, day, year;
  private static int i, j;//month and day position of / separator
  private static boolean validDay, validMonth;//Flag to convert or not
  private static boolean isLeapYr;
  private static char repeat;// Program do/while loop control
  private static String[] monthName = {"January",
                       "February",
                       "March",
                       "April",
                       "May",
                       "June",
                       "July",
                       "August",
                       "September",
                       "October",
                       "November",
                       "December"};

  public static void main(String[] args)
  {
     do // Repeat while user says 'yes'.
      {
  try
    {

       validMonth = false;   // initialise boolean values for
       validDay = false;     // each execution of the loop
       isLeapYr = false;

       System.out.println();
```

```
        System.out.println("Enter the date (format dd/mm/yyyy)");
      s1 = SavitchIn.readLine();
      System.out.println();

      s1 = s1.trim();              // get rid of additional spaces

      i = s1.indexOf( "/" ) + 1;  // find the / position
      s2 = s1.substring( i );     // and extract the day month year
       j = s2.indexOf( "/" ) + 1;  // convert the strings to integers
      day = Integer.valueOf(s1.substring(0, i-1)).intValue();
      month = Integer.parseInt(s1.substring(i, s1.length()-5));
      year = Integer.parseInt(s2.substring(j, j+4));

      // check for a leap year
      isLeapYr = leap(year);

      // validate day
      dayCheck();

      // validate month
      monthCheck();

      if(validDay && validMonth)
        {
        System.out.println();
          System.out.println( "The date is:" + day + " " + monthName[month-1] + " " + year
);
        }
      else
        {
        System.out.println();
        System.out.println( "The date is invalid");
        }
  }//end try block

  catch(DayException e2)
         {
         System.out.println();
         System.out.println(e2.getMessage());
         System.out.println();
         }

  catch(MonthException e1)
          {
          System.out.println();
          System.out.println(e1.getMessage());
          System.out.println();
```

```
            }

        System.out.println();
        System.out.println("Again?(y/n)");
        repeat = SavitchIn.readLineNonwhiteChar();

      }while ((repeat == 'y') || (repeat == 'Y'));

  }//end main


  private static void monthCheck() throws MonthException
  {
        if(month < 0)
            {
         validMonth = false;
            throw new MonthException
            ("Month format error: must be m/d with m from 1 to 12.");
            }
           else if(month < 1 || month > 12)
            {
         validMonth = false;
            throw new MonthException(month);
            }
        else      //Valid month
            {
         validMonth = true;
            }
} // end of monthCheck

   private static boolean leap (int yr)
   {     boolean leap = true;
         if ( yr % 4 != 0 )
              leap = false;
         else
              if ( yr % 100 == 0 )
                   if ( yr % 400 == 0)
                        leap = true;
                   else
                        leap = false;
              else
                   leap = true;
          return leap;
    }

   private static void dayCheck() throws DayException
   {
```

```
//Check for valid day
switch(month)
{
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
          if(day < 1 || day > 31)
          {
     validDay = false;
            System.out.println("1 through 31 only, please.");
            throw new DayException(month);
          }
          else
          {
     validDay = true;
   }
          break;
  case 4:
  case 6:
  case 9:
  case 11:
          if(day < 1 || day > 30)
          {
   validDay = false;
          System.out.println("1 through 30 only, please.");
          throw new DayException(month);
          }
          else
          {
   validDay = true;
          }
          break;
  case 2:
   int noofdays = 28;
   if ( isLeapYr )
      noofdays++;
      if(day < 1 || day > noofdays)
          {
   validDay = false;
          System.out.println("1 through 29 only, please.");
          throw new DayException(month);
          }
          else
```

```
                {
            validDay = true;
                }
                break;
            default:
            validDay = false;
                System.out.println("Invalid month in day-check.");
                throw new
                DayException("Impossible: Invalid month in day-check.");
                }
        }//end dayCheck

}//end class
```

```
/**************************************************************
 *
 * File name: MonthException.java
 *
 * Exception class that prints either a message passed by the caller
 * or the default message that only 1 through 12 are legitimate
 * numbers for a month.
 *************************************************************/
public class MonthException extends Exception
{
   public MonthException()
   {
         super("Invalid input for month.");
   }

   public MonthException(int monthNumber)
   {
      super
        (monthNumber + " is invalid: month number must be from 1 to 12.");
   }

   public MonthException(String message)
   {
      super(message);
   }
}
```

```
/**************************************************************
 *
 * File name: DayException.java
 *
 * Exception class that prints either a message passed by the caller
 * or the default message that says the number of days is out of range.
 *
 **************************************************************/
public class DayException extends Exception
{
   public DayException()
   {
                  super("Invalid entry for day.");
    }

   public DayException(int monthNumber)
   {
      super("You have entered an invalid day for month number "
           + monthNumber);
   }

   public DayException(String message)
   {
      super(message);
   }
}
```

APPENDIX F - Guess the number game as an applet, followed by a small html file that runs the applet.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/*****************************************************************
 *The Guess a number game
 *****************************************************************/
public class GuessNumberApplet extends Applet implements ActionListener
{
   private Label message1, message2, message3;
   private TextField text2;
   private int rNo;
   private Button button1, button2;
   private String s1, theText = "Watch me!";

//   ***   The program code that is not required in the applet has been commented in order to
//   ***   highlight the difference between a Java application and a Java applet.


//   public static final int WIDTH = 400;
//   public static final int HEIGHT = 300;

//   public static void main(String[] args)
//   {
//      GuessNumber guiGuessNumber = new GuessNumber();
//      guiGuessNumber.setVisible(true);
//   }

   public void init()
   {
//      setTitle("Guess the number game");
//      setSize(WIDTH, HEIGHT);
        setBackground(Color.cyan);
//      addWindowListener(new WindowDestroyer());

        setLayout(new GridLayout(3,1));
// The third entry in the grid is for the background color

       Panel descripPanel = new Panel();
      descripPanel.setLayout(new BorderLayout());

      message1 = new Label("I have a number between 1 and 1000");
      message2 = new Label("Can you guess my number?");
      message3 = new Label("Please enter your first guess:");
```

```
    descripPanel.add(message1, "North");
    descripPanel.add(message2, "Center");
    descripPanel.add(message3, "South");

    add(descripPanel);

    Panel buttonPanel = new Panel();
    buttonPanel.setLayout(new FlowLayout());

    s1 = "Start Game";
    button1 = new Button(s1);
    button1.addActionListener(this);
    buttonPanel.add(button1);

    button2 = new Button ("Am I right?");
    button2.addActionListener(this);
    buttonPanel.add(button2);

    text2 = new TextField(10);
    buttonPanel.add(text2);

    add(buttonPanel);
}


public void actionPerformed(ActionEvent e)
{
   if (e.getActionCommand().equals("Start Game"))
    {
      rNo = (int) (Math.random()*1000 + 1);
      button1.setVisible(false);
      theText = "You can start guessing now  " ;
    }
    else
    if (e.getActionCommand().equals("Am I right?"))
       {  String s = text2.getText();
          int val = Integer.parseInt(s);
               int diff = Math.abs(val-rNo);
               text2.setText("");
      if(val < rNo)
          {
           setBackground(Color.green);
           theText = "Too low";
           message3 = new Label("Too low");
           repaint();
           }
      else  if (val > rNo)
```

```
                {
                setBackground(Color.orange);
                theText = "Too high";
                repaint();
                }
             else {
                 setBackground(Color.cyan);
                 theText = "YOU'VE GOT IT  " + s ;
                 text2.setText(s);
                 button1.setVisible(true);
                 repaint();
                 }
            if(diff < 5) {
                    setBackground(Color.red);
               repaint();
          }
          }


      repaint(); //force color and text change
   }

   public void paint (Graphics g)
   {
         g.drawString(theText, 40, 10 );
   }


}
```

GNum2.html - to view the applet through a browser, or using appletviewer

```
<HTML>
<HEAD>
<TITLE>
Guess the Number Game
</TITLE>
</HEAD>
<APPLET CODE="GuessNumberApplet.class" WIDTH=400 HEIGHT=200>
</APPLET>
</HTML>
```

APPENDIX G - the GuessNumber.html file generated by javadoc.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Frameset//EN""http://www.w3.org/TR/REC-html40/frameset.dtd">
<!--NewPage-->
<HTML>
<HEAD>
<!-- Generated by javadoc on Mon Feb 04 09:25:55 GMT+02:00 2002 -->
<TITLE>
: Class  GuessNumber
</TITLE>
<LINK REL ="stylesheet" TYPE="text/css" HREF="stylesheet.css" TITLE="Style">
</HEAD>
<BODY BGCOLOR="white">

<!-- ========== START OF NAVBAR ========== -->
<A NAME="navbar_top"><!-- --></A>
<TABLE BORDER="0" WIDTH="100%" CELLPADDING="1" CELLSPACING="0">
<TR>
<TD COLSPAN=2 BGCOLOR="#EEEEFF" CLASS="NavBarCell1">
<A NAME="navbar_top_firstrow"><!-- --></A>
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="3">
  <TR ALIGN="center" VALIGN="top">
  <TD BGCOLOR="#FFFFFF" CLASS="NavBarCell1Rev">  <FONT
CLASS="NavBarFont1Rev"><B>Class</B></FONT> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="overview-tree.html"><FONT
CLASS="NavBarFont1"><B>Tree</B></FONT></A> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="deprecated-list.html"><FONT
CLASS="NavBarFont1"><B>Deprecated</B></FONT></A> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="index-all.html"><FONT
CLASS="NavBarFont1"><B>Index</B></FONT></A> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="help-doc.html"><FONT
CLASS="NavBarFont1"><B>Help</B></FONT></A> </TD>
  </TR>
</TABLE>
</TD>
<TD ALIGN="right" VALIGN="top" ROWSPAN=3><EM>
</EM>
</TD>
</TR>

<TR>
<TD BGCOLOR="white" CLASS="NavBarCell2"><FONT SIZE="-2">
```

```
 PREV CLASS 
 NEXT CLASS</FONT></TD>
<TD BGCOLOR="white" CLASS="NavBarCell2"><FONT SIZE="-2">
  <A HREF="index.html" TARGET="_top"><B>FRAMES</B></A>   
 <A HREF="GuessNumber.html" TARGET="_top"><B>NO
FRAMES</B></A></FONT></TD>
</TR>
<TR>
<TD VALIGN="top" CLASS="NavBarCell3"><FONT SIZE="-2">
  SUMMARY:  <A
HREF="#inner_classes_inherited_from_class_java.awt.Frame">INNER</A> | 
<A HREF="#field_summary">FIELD</A> | <A
HREF="#constructor_summary">CONSTR</A> | <A
HREF="#method_summary">METHOD</A></FONT></TD>
<TD VALIGN="top" CLASS="NavBarCell3"><FONT SIZE="-2">
DETAIL:  <A HREF="#field_detail">FIELD</A> | <A
HREF="#constructor_detail">CONSTR</A> | <A
HREF="#method_detail">METHOD</A></FONT></TD>
</TR>
</TABLE>
<!-- =========== END OF NAVBAR =========== -->

<HR>
<!-- ======== START OF CLASS DATA ======== -->
<H2>
Class  GuessNumber</H2>
<PRE>
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
                          |
                          +--<B>GuessNumber</B>
</PRE>
<DL>
<DT><B>All Implemented Interfaces:</B> <DD>javax.accessibility.Accessible,
java.awt.event.ActionListener, java.util.EventListener, java.awt.image.ImageObserver,
java.awt.MenuContainer, java.io.Serializable</DD>
</DL>
<HR>
<DL>
<DT>public class <B>GuessNumber</B><DT>extends java.awt.Frame<DT>implements
```

```
java.awt.event.ActionListener</DL>

<P>
The Guess a number game
<P>
<DL>
<DT><B>See Also: </B><DD><A
HREF="serialized-form.html#GuessNumber">Serialized Form</A></DL>
<HR>

<P>
<!-- ======== INNER CLASS SUMMARY ======== -->

<A NAME="inner_classes_inherited_from_class_java.awt.Frame"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Inner classes inherited from class java.awt.Frame</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>java.awt.Frame.AccessibleAWTFrame</CODE></TD>
</TR>
</TABLE>
 <A NAME="inner_classes_inherited_from_class_java.awt.Window"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Inner classes inherited from class java.awt.Window</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>java.awt.Window.AccessibleAWTWindow</CODE></TD>
</TR>
</TABLE>
 <A NAME="inner_classes_inherited_from_class_java.awt.Container"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Inner classes inherited from class java.awt.Container</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>java.awt.Container.AccessibleAWTContainer</CODE></TD>
</TR>
</TABLE>
 <A NAME="inner_classes_inherited_from_class_java.awt.Component"><!--
--></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Inner classes inherited from class java.awt.Component</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
```

```
<TD><CODE>java.awt.Component.AccessibleAWTComponent</CODE></TD>
</TR>
</TABLE>
 
<!-- =========== FIELD SUMMARY =========== -->

<A NAME="field_summary"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#CCCCFF" CLASS="TableHeadingColor">
<TD COLSPAN=2><FONT SIZE="+2">
<B>Field Summary</B></FONT></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD ALIGN="right" VALIGN="top" WIDTH="1%"><FONT SIZE="-1">
<CODE>static int</CODE></FONT></TD>
<TD><CODE><B><A
HREF="GuessNumber.html#HEIGHT">HEIGHT</A></B></CODE>

<BR>
           </TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD ALIGN="right" VALIGN="top" WIDTH="1%"><FONT SIZE="-1">
<CODE>static int</CODE></FONT></TD>
<TD><CODE><B><A
HREF="GuessNumber.html#WIDTH">WIDTH</A></B></CODE>

<BR>
           </TD>
</TR>
</TABLE>
 <A NAME="fields_inherited_from_class_java.awt.Frame"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Fields inherited from class java.awt.Frame</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>CROSSHAIR_CURSOR, DEFAULT_CURSOR, E_RESIZE_CURSOR,
HAND_CURSOR, ICONIFIED, MOVE_CURSOR, N_RESIZE_CURSOR,
NE_RESIZE_CURSOR, NORMAL, NW_RESIZE_CURSOR, S_RESIZE_CURSOR,
SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, TEXT_CURSOR,
W_RESIZE_CURSOR, WAIT_CURSOR</CODE></TD>
</TR>
</TABLE>
 <A NAME="fields_inherited_from_class_java.awt.Component"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
```

<TD><B>Fields inherited from class java.awt.Component</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>BOTTOM_ALIGNMENT, CENTER_ALIGNMENT,
LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT</CODE></TD>
</TR>
</TABLE>
 <A NAME="fields_inherited_from_class_java.awt.image.ImageObserver"><!--
--></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Fields inherited from interface java.awt.image.ImageObserver</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>ABORT, ALLBITS, ERROR, FRAMEBITS, PROPERTIES,
SOMEBITS</CODE></TD>
</TR>
</TABLE>

<!-- ======== CONSTRUCTOR SUMMARY ======== -->

<A NAME="constructor_summary"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#CCCCFF" CLASS="TableHeadingColor">
<TD COLSPAN=2><FONT SIZE="+2">
<B>Constructor Summary</B></FONT></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE><B><A
HREF="GuessNumber.html#GuessNumber()">GuessNumber</A></B>()</CODE>

           </TD>
</TR>
</TABLE>

<!-- ========== METHOD SUMMARY ========== -->

<A NAME="method_summary"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#CCCCFF" CLASS="TableHeadingColor">
<TD COLSPAN=2><FONT SIZE="+2">
<B>Method Summary</B></FONT></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD ALIGN="right" VALIGN="top" WIDTH="1%"><FONT SIZE="-1">
<CODE> void</CODE></FONT></TD>

<TD><CODE><B><A
HREF="GuessNumber.html#actionPerformed(java.awt.event.ActionEvent)">actionPerform
ed</A></B>(java.awt.event.ActionEvent e)</CODE>

          Test for online help

Key in a number of your choice

If you are close to the number
within 20, the screen will be red

If you are much lower than the number
the screen will be green

If you are much higher than the number
the screen will be orange

end of test</TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD ALIGN="right" VALIGN="top" WIDTH="1%"><FONT SIZE="-1">
<CODE>static void</CODE></FONT></TD>
<TD><CODE><B><A
HREF="GuessNumber.html#main(java.lang.String[])">main</A></B>(java.lang.String[]&
nbsp;args)</CODE>

           </TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD ALIGN="right" VALIGN="top" WIDTH="1%"><FONT SIZE="-1">
<CODE> void</CODE></FONT></TD>
<TD><CODE><B><A
HREF="GuessNumber.html#paint(java.awt.Graphics)">paint</A></B>(java.awt.Graphics
 g)</CODE>

           </TD>
</TR>
</TABLE>
 <A NAME="methods_inherited_from_class_java.awt.Frame"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Methods inherited from class java.awt.Frame</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">

<TD><CODE>addNotify, finalize, getAccessibleContext, getCursorType, getFrames, getIconImage, getMenuBar, getState, getTitle, isResizable, paramString, remove, removeNotify, setCursor, setIconImage, setMenuBar, setResizable, setState, setTitle</CODE></TD>
</TR>
</TABLE>
 <A NAME="methods_inherited_from_class_java.awt.Window"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Methods inherited from class java.awt.Window</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>addWindowListener, applyResourceBundle, applyResourceBundle, dispose, getFocusOwner, getGraphicsConfiguration, getInputContext, getListeners, getLocale, getOwnedWindows, getOwner, getToolkit, getWarningString, hide, isShowing, pack, postEvent, processEvent, processWindowEvent, removeWindowListener, setCursor, show, toBack, toFront</CODE></TD>
</TR>
</TABLE>
 <A NAME="methods_inherited_from_class_java.awt.Container"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Methods inherited from class java.awt.Container</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>add, add, add, add, add, addContainerListener, addImpl, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getInsets, getLayout, getMaximumSize, getMinimumSize, getPreferredSize, insets, invalidate, isAncestorOf, layout, list, list, locate, minimumSize, paintComponents, preferredSize, print, printComponents, processContainerEvent, remove, remove, removeAll, removeContainerListener, setFont, setLayout, update, validate, validateTree</CODE></TD>
</TR>
</TABLE>
 <A NAME="methods_inherited_from_class_java.awt.Component"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Methods inherited from class java.awt.Component</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addPropertyChangeListener, addPropertyChangeListener, bounds, checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, disable, disableEvents, dispatchEvent, enable, enable, enableEvents, enableInputMethods,

firePropertyChange, getBackground, getBounds, getBounds, getColorModel,
getComponentOrientation, getCursor, getDropTarget, getFont, getFontMetrics,
getForeground, getGraphics, getHeight, getInputMethodRequests, getLocation,
getLocation, getLocationOnScreen, getName, getParent, getPeer, getSize, getSize,
getTreeLock, getWidth, getX, getY, gotFocus, handleEvent, hasFocus, imageUpdate,
inside, isDisplayable, isDoubleBuffered, isEnabled, isFocusTraversable, isLightweight,
isOpaque, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus,
mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move,
nextFocus, paintAll, prepareImage, prepareImage, printAll, processComponentEvent,
processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent,
processInputMethodEvent, processKeyEvent, processMouseEvent,
processMouseMotionEvent, removeComponentListener, removeFocusListener,
removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener,
removeKeyListener, removeMouseListener, removeMouseMotionListener,
removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint,
repaint, requestFocus, reshape, resize, resize, setBackground, setBounds, setBounds,
setComponentOrientation, setDropTarget, setEnabled, setForeground, setLocale,
setLocation, setLocation, setName, setSize, setSize, setVisible, show, size, toString,
transferFocus</CODE></TD>
</TR>
</TABLE>
 <A NAME="methods_inherited_from_class_java.lang.Object"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Methods inherited from class java.lang.Object</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>clone, equals, getClass, hashCode, notify, notifyAll, wait, wait,
wait</CODE></TD>
</TR>
</TABLE>
 <A NAME="methods_inherited_from_class_java.awt.MenuContainer"><!--
--></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#EEEEFF" CLASS="TableSubHeadingColor">
<TD><B>Methods inherited from interface java.awt.MenuContainer</B></TD>
</TR>
<TR BGCOLOR="white" CLASS="TableRowColor">
<TD><CODE>getFont, postEvent</CODE></TD>
</TR>
</TABLE>

<P>

<!-- ============ FIELD DETAIL =========== -->

<A NAME="field_detail"><!-- --></A>

```
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#CCCCFF" CLASS="TableHeadingColor">
<TD COLSPAN=1><FONT SIZE="+2">
<B>Field Detail</B></FONT></TD>
</TR>
</TABLE>

<A NAME="WIDTH"><!-- --></A><H3>
WIDTH</H3>
<PRE>
public static final int <B>WIDTH</B></PRE>
<DL>
</DL>
<HR>

<A NAME="HEIGHT"><!-- --></A><H3>
HEIGHT</H3>
<PRE>
public static final int <B>HEIGHT</B></PRE>
<DL>
</DL>

<!-- ========= CONSTRUCTOR DETAIL ======== -->

<A NAME="constructor_detail"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#CCCCFF" CLASS="TableHeadingColor">
<TD COLSPAN=1><FONT SIZE="+2">
<B>Constructor Detail</B></FONT></TD>
</TR>
</TABLE>

<A NAME="GuessNumber()"><!-- --></A><H3>
GuessNumber</H3>
<PRE>
public <B>GuessNumber</B>()</PRE>
<DL>
</DL>

<!-- ============ METHOD DETAIL ========== -->

<A NAME="method_detail"><!-- --></A>
<TABLE BORDER="1" CELLPADDING="3" CELLSPACING="0" WIDTH="100%">
<TR BGCOLOR="#CCCCFF" CLASS="TableHeadingColor">
<TD COLSPAN=1><FONT SIZE="+2">
<B>Method Detail</B></FONT></TD>
</TR>
```

```
</TABLE>

<A NAME="main(java.lang.String[])"><!-- --></A><H3>
main</H3>
<PRE>
public static void <B>main</B>(java.lang.String[] args)</PRE>
<DL>
<DD><DL>
</DL>
</DD>
</DL>
<HR>

<A NAME="actionPerformed(java.awt.event.ActionEvent)"><!-- --></A><H3>
actionPerformed</H3>
<PRE>
public void <B>actionPerformed</B>(java.awt.event.ActionEvent e)</PRE>
<DL>
<DD>Test for online help

Key in a number of your choice

If you are close to the number
within 20, the screen will be red

If you are much lower than the number
the screen will be green

If you are much higher than the number
the screen will be orange

end of test<DD><DL>
<DT><B>Specified by: </B><DD><CODE>actionPerformed</CODE> in interface
<CODE>java.awt.event.ActionListener</CODE></DL>
</DD>
</DL>
<HR>

<A NAME="paint(java.awt.Graphics)"><!-- --></A><H3>
paint</H3>
<PRE>
public void <B>paint</B>(java.awt.Graphics g)</PRE>
<DL>
<DD><DL>
<DT><B>Overrides:</B><DD><CODE>paint</CODE> in class
<CODE>java.awt.Container</CODE></DL>
</DD>
```

```
</DL>
<!-- ========= END OF CLASS DATA ========= -->
<HR>

<!-- ========== START OF NAVBAR ========== -->
<A NAME="navbar_bottom"><!-- --></A>
<TABLE BORDER="0" WIDTH="100%" CELLPADDING="1" CELLSPACING="0">
<TR>
<TD COLSPAN=2 BGCOLOR="#EEEEFF" CLASS="NavBarCell1">
<A NAME="navbar_bottom_firstrow"><!-- --></A>
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="3">
  <TR ALIGN="center" VALIGN="top">
  <TD BGCOLOR="#FFFFFF" CLASS="NavBarCell1Rev">  <FONT
CLASS="NavBarFont1Rev"><B>Class</B></FONT> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="overview-tree.html"><FONT
CLASS="NavBarFont1"><B>Tree</B></FONT></A> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="deprecated-list.html"><FONT
CLASS="NavBarFont1"><B>Deprecated</B></FONT></A> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="index-all.html"><FONT
CLASS="NavBarFont1"><B>Index</B></FONT></A> </TD>
  <TD BGCOLOR="#EEEEFF" CLASS="NavBarCell1">    <A
HREF="help-doc.html"><FONT
CLASS="NavBarFont1"><B>Help</B></FONT></A> </TD>
  </TR>
</TABLE>
</TD>
<TD ALIGN="right" VALIGN="top" ROWSPAN=3><EM>
</EM>
</TD>
</TR>

<TR>
<TD BGCOLOR="white" CLASS="NavBarCell2"><FONT SIZE="-2">
 PREV CLASS 
 NEXT CLASS</FONT></TD>
<TD BGCOLOR="white" CLASS="NavBarCell2"><FONT SIZE="-2">
  <A HREF="index.html" TARGET="_top"><B>FRAMES</B></A>   
 <A HREF="GuessNumber.html" TARGET="_top"><B>NO
FRAMES</B></A></FONT></TD>
</TR>
<TR>
<TD VALIGN="top" CLASS="NavBarCell3"><FONT SIZE="-2">
  SUMMARY:  <A
HREF="#inner_classes_inherited_from_class_java.awt.Frame">INNER</A> | 
```

<A HREF="#field_summary">FIELD</A> | <A
HREF="#constructor_summary">CONSTR</A> | <A
HREF="#method_summary">METHOD</A></FONT></TD>
<TD VALIGN="top" CLASS="NavBarCell3"><FONT SIZE="-2">
DETAIL: <A HREF="#field_detail">FIELD</A> | <A
HREF="#constructor_detail">CONSTR</A> | <A
HREF="#method_detail">METHOD</A></FONT></TD>
</TR>
</TABLE>
<!-- =========== END OF NAVBAR =========== -->

<HR>

</BODY>
</HTML>

APPENDIX H - Example Control Break Report, written in COBOL, from Mrs. C R White, Department of Computer Studies, Technikon Natal.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DS1-EXAM-98.
AUTHOR. C R WHITE.
DATE-WRITTEN. 26 SEPTEMBER 1996 (modified 10 Sept '98).

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT SALES-FILE ASSIGN TO DISK
          "96ASS12.DAT"
        ORGANIZATION IS LINE SEQUENTIAL.
   SELECT REPORT-FILE ASSIGN TO PRINTER
                "EXAM.RPT".

DATA DIVISION.
FILE SECTION.
FD  SALES-FILE
   RECORD CONTAINS 80 CHARACTERS
   DATA RECORD IS SALES-REC.
01   SALES-REC.
   02  DEPT-NUM-SR      PIC 9(4).
   02                PIC X.
   02  TRANS-DATE-SR    PIC 9(6).
   02                PIC X.
   02  AMOUNT-SR        PIC 9(3)V99.
   02                PIC X(63).

FD  REPORT-FILE
   RECORD CONTAINS 80 CHARACTERS
   DATA RECORD IS REPORT-REC.
01  REPORT-REC           PIC X(80).

WORKING-STORAGE SECTION.
01  HEADING-REC.
   02  DATE-HR.
      05  DAY-HR         PIC 99/.
      05  MONTH-HR       PIC 99/.
      05  YEAR-HR        PIC 9999.
   02                  PIC X(14) VALUE SPACES.
   02                  PIC X(28)
                  VALUE "XYZ WHOLESALE SUPPLY COMPANY".
```

```
   02                PIC X(09) VALUE SPACES.
   02                PIC X(05) VALUE "PAGE ".
   02  PAGE-NUM-HR        PIC Z9.
   02                PIC X(12) VALUE SPACES.

01  COL-HEADING-REC.
   02                PIC X(03)  VALUE SPACES.
   02                PIC X(08)  VALUE "DEPT NUM".
   02                PIC X(03)  VALUE SPACES.
   02                PIC X(10)  VALUE "TRANS DATE".
   02                PIC X(03)  VALUE SPACES.
   02                PIC X(12)  VALUE "ORDER AMOUNT".
   02                PIC X(41)  VALUE SPACES.

01  DETAIL-LINE.
   02                PIC X(05)   VALUE SPACES.
   02  DEPT-NUM-DL        PIC 9(4)    BLANK WHEN ZERO.
   02                PIC X(6)    VALUE SPACES.
   02  TRANS-DATE-DL      PIC 99/99/99.
   02                PIC X(7)   VALUE SPACES.
   02  AMOUNT-DL          PIC $(3)9.99.
   02                PIC X(43)  VALUE SPACES.

01  DEPT-TOTALS.
   02                PIC X(15)   VALUE SPACES.
   02                PIC X(30)
                VALUE "TOTAL AMOUNT FOR DEPARTMENT: ".
   02  DEPT-NUM-DT        PIC 9999.
   02  TOT-AMOUNT-DT      PIC $(5)9.99.
   02                PIC X(22)  VALUE SPACES.

01  TOTAL-LINE.
   02                PIC X(34)    VALUE SPACES.
   02                PIC X(15)
                 VALUE "COMPANY TOTAL: ".
   02  TOT-AMOUNT-TL     PIC $(6)9.99.
   02                PIC X(22)   VALUE SPACES.

01  ACCUMULATED-TOTALS.
   02  DEPT-TOTAL-WS       PIC 9(5)V99  VALUE ZERO.
   02  COMPANY-TOTAL-WS     PIC 9(6)V99  VALUE ZERO.

01  CALCULATION-FIELDS.
   02  PAGE-NUM-WS        PIC 99  VALUE 01.
   02  DEPT-COUNT-WS      PIC 9   VALUE 00.

01  DATE-WS.
```

```
   02  YEAR-WS            PIC 99.
   02  MONTH-WS           PIC 99.
   02  DAY-WS             PIC 99.

01  HOLD-DEPT-NUM         PIC 9(4).

01  FLAGS.
   02  DEPT-FLAG              PIC XXX VALUE "YES".
      88 FIRST-RECORD           VALUE "YES".
   02  ARE-THERE-MORE-RECORDS  PIC XXX  VALUE "YES".
      88  END-OF-FILE           VALUE "NO ".
      88  MORE-RECORDS          VALUE "YES".


PROCEDURE DIVISION.

100-MAIN-MODULE.
   PERFORM 200-INITIALIZE.
   PERFORM 300-PROCESS-A-RECORD
         UNTIL END-OF-FILE.
   PERFORM 700-WRITE-TOTALS.
   CLOSE SALES-FILE,
       REPORT-FILE.
   STOP RUN.

200-INITIALIZE.
   OPEN INPUT  SALES-FILE
      OUTPUT REPORT-FILE.
   PERFORM 400-READ-A-REC.
   MOVE DEPT-NUM-SR TO HOLD-DEPT-NUM.
   PERFORM 250-GET-DATE.
   PERFORM 500-HEADING-RTN.

250-GET-DATE.
   ACCEPT DATE-WS FROM DATE.
   MOVE DAY-WS TO DAY-HR.
   MOVE MONTH-WS TO MONTH-HR.
   IF YEAR-WS < 97
      ADD 2000 TO YEAR-WS GIVING YEAR-HR
   ELSE
      ADD 1900 TO YEAR-WS GIVING YEAR-HR.


300-PROCESS-A-RECORD.
   PERFORM 350-DETAIL-LINE
         UNTIL (END-OF-FILE)
             OR (HOLD-DEPT-NUM NOT EQUAL TO DEPT-NUM-SR).
```

```
    PERFORM 350-TOTAL-A-DEPARTMENT.

350-DETAIL-LINE.
    ADD AMOUNT-SR TO DEPT-TOTAL-WS.
    PERFORM 400-WRITE-DETAIL-LINE.
    PERFORM 400-READ-A-REC.


350-TOTAL-A-DEPARTMENT.
    ADD 1 TO DEPT-COUNT-WS.
    ADD DEPT-TOTAL-WS TO COMPANY-TOTAL-WS.
    MOVE DEPT-TOTAL-WS TO TOT-AMOUNT-DT.
    WRITE REPORT-REC FROM DEPT-TOTALS
              AFTER ADVANCING 2 LINES.
    MOVE SPACES TO REPORT-REC.
    WRITE REPORT-REC.
    IF MORE-RECORDS
      IF DEPT-COUNT-WS = 2
          PERFORM 500-HEADING-RTN
          MOVE ZERO TO DEPT-COUNT-WS
      END-IF
      MOVE "YES" TO DEPT-FLAG
      MOVE ZERO TO DEPT-TOTAL-WS
      MOVE DEPT-NUM-SR TO HOLD-DEPT-NUM.

400-WRITE-DETAIL-LINE.
    PERFORM 450-MOVE-DATA.
    WRITE REPORT-REC FROM DETAIL-LINE.

400-READ-A-REC.
    READ SALES-FILE
        AT END MOVE "NO " TO ARE-THERE-MORE-RECORDS.

450-MOVE-DATA.
    IF FIRST-RECORD
       MOVE DEPT-NUM-SR TO DEPT-NUM-DL
       MOVE "NO " TO DEPT-FLAG
    ELSE MOVE ZEROS TO DEPT-NUM-DL.
    MOVE TRANS-DATE-SR TO TRANS-DATE-DL.
    MOVE AMOUNT-SR TO AMOUNT-DL.

500-HEADING-RTN.
    MOVE PAGE-NUM-WS TO PAGE-NUM-HR.
    IF PAGE-NUM-WS = 01
        WRITE REPORT-REC FROM HEADING-REC
    ELSE WRITE REPORT-REC FROM HEADING-REC
              AFTER ADVANCING PAGE.
```

```
    ADD 1 TO PAGE-NUM-WS.
    PERFORM 600-WRITE-COL-HEADING.

600-WRITE-COL-HEADING.
    WRITE REPORT-REC FROM COL-HEADING-REC
            AFTER ADVANCING 2 LINES.
    WRITE REPORT-REC FROM SPACES.

700-WRITE-TOTALS.
    PERFORM 750-MOVE-TOTALS.
    WRITE REPORT-REC FROM TOTAL-LINE
            AFTER ADVANCING 2 LINES.
    MOVE SPACES TO REPORT-REC.
    WRITE REPORT-REC AFTER 2.

750-MOVE-TOTALS.
    MOVE COMPANY-TOTAL-WS TO TOT-AMOUNT-TL.
```

DEPT NUM	TRANS DATE	ORDER AMOUNT
1000	01/03/91	$100.00
	01/03/91	$120.00
	01/03/91	$30.00
	01/04/91	$44.00
	01/04/91	$5.90
	01/05/91	$234.00

TOTAL AMOUNT FOR DEPARTMENT: $533.90

2000	01/03/91	$200.00
	01/03/91	$234.00
	01/04/91	$340.00

TOTAL AMOUNT FOR DEPARTMENT: $774.00

DEPT NUM	TRANS DATE	ORDER AMOUNT
3000	01/04/91	$200.00
	01/04/91	$39.99
	01/05/91	$44.00
	TOTAL AMOUNT FOR DEPARTMENT:	$283.99
4000	01/03/91	$50.00
	TOTAL AMOUNT FOR DEPARTMENT:	$50.00

COMPANY TOTAL: $1641.89

APPENDIX I - Java Data Class for the Control Break Report Program

```
//  SalesRecord.java
//  SalesRecord class for the ControlBreakReport.
import java.io.*;
import java.util.*;

public class SalesRecord {
   private String line;
   private int departmentNumber;
   private String saleDate;
   private double saleAmount;

   // Read a sales record from the specified file

   public void read( BufferedReader file ) throws IOException
   {
      line = null;
      line = file.readLine();
      if ( line != null )
          {
           System.out.println(line);
           StringTokenizer fieldFinder = new StringTokenizer(line);
           departmentNumber = Integer.parseInt((fieldFinder.nextToken()).trim());
           saleDate = fieldFinder.nextToken();
           saleAmount = Double.parseDouble((fieldFinder.nextToken()).trim()) / 100 ;
          }
   }

   public void setLine( String ln ) { line = ln; }

   public String getLine() { return line; }

   public void setDepartmentNumber( int a ) { departmentNumber = a; }

   public int getDepartmentNumber() { return departmentNumber; }

   public void setSaleDate( String d ) { saleDate = d; }

   public String getSaleDate() { return saleDate; }

   public void setSaleAmount( double  a ) { saleAmount = a; }

   public double getSaleAmount() { return saleAmount; }

 }
```

APPENDIX J - The Java Processing Class to create the control break report.

```
// WriteSalesReport.java
// This program through a Text file of sale records to print a control break report.

import java.io.*;
import java.util.*;
import java.text.NumberFormat;


public class WriteSalesReport
{

   // use the class SalesRecord to define the data, and access the records
   private SalesRecord salesData;

   private BufferedReader inputStream;

   private PrintWriter outputStream;

   int holdDepartmentNumber;

   Date toDay = new Date();

   int pageNo = 0;

   int deptCount = 0;

   double deptTotal = 0;

   double companyTotal = 0;

   boolean firstRecord = true;

   NumberFormat moneyFormat = NumberFormat.getCurrencyInstance ( Locale.US );


   public static void main( String args[] )
   {

   WriteSalesReport  rep = new WriteSalesReport();
    rep.writeReport();
    }

/*      100-MAIN-MODULE.
          PERFORM 200-INITIALIZE.
          PERFORM 300-PROCESS-A-RECORD
```

```
            UNTIL END-OF-FILE.
        PERFORM 700-WRITE-TOTALS.
        CLOSE SALES-FILE,
              REPORT-FILE.
        STOP RUN.
*/
  public void writeReport()
  {


  salesData = new SalesRecord();

    // Open the text file, and report file

    initialise();

    try {

      while ( salesData.getLine() != null )
        {
          processReport( );
        }

      companyTotals( );

      closeFiles();


        }

  catch(IOException e)
        {
        System.out.println("Error reading from text file 96ASS12.DAT");
        }
  }

/*      200-INITIALIZE.
        OPEN INPUT  SALES-FILE
             OUTPUT REPORT-FILE.
        PERFORM 400-READ-A-REC.
        MOVE DEPT-NUM-SR TO HOLD-DEPT-NUM.
        PERFORM 250-GET-DATE.
        PERFORM 500-HEADING-RTN.
*/

  public void initialise()
  {
```

```
  try {
       inputStream = new BufferedReader( new FileReader( "96ASS12.DAT"));
       outputStream = new PrintWriter( new FileOutputStream("cbreport.txt"));
        try {
            salesData.read(inputStream);
            holdDepartmentNumber = salesData.getDepartmentNumber();
            printHeading();
             }
         catch(IOException e)
         {
         System.out.println("Error reading from text file 96ASS12.DAT");
         }
         }
  catch(FileNotFoundException e)
         {
           System.out.println("Text file cannot be found - 96ASS12.DAT");
         }

  }


  public void closeFiles()
 {
  try {
       inputStream.close();
       outputStream.close();
       }
  catch(IOException e)
         {
           System.out.println("Files cannot be closed");
         }
  }
/*
    300-PROCESS-A-RECORD.
         PERFORM 350-DETAIL-LINE
               UNTIL (END-OF-FILE)
                    OR (HOLD-DEPT-NUM NOT EQUAL TO DEPT-NUM-SR).
         PERFORM 350-TOTAL-A-DEPARTMENT.
*/


  public void processReport() throws IOException
  {
   while ( ( holdDepartmentNumber ==  salesData.getDepartmentNumber() )
             && ( salesData.getLine() != null ) )
         {
         detailLine();
```

```
        }
    departmentTotal();
    }

/*      350-DETAIL-LINE.
        ADD AMOUNT-SR TO DEPT-TOTAL-WS.
        PERFORM 400-WRITE-DETAIL-LINE.
        PERFORM 400-READ-A-REC.
*/
  public void detailLine( ) throws IOException
  {
        deptTotal += salesData.getSaleAmount();
        printDetailLine();
        salesData.read(inputStream);
  }

/*     500-HEADING-RTN.
        MOVE PAGE-NUM-WS TO PAGE-NUM-HR.
        IF PAGE-NUM-WS = 01
            WRITE REPORT-REC FROM HEADING-REC
        ELSE WRITE REPORT-REC FROM HEADING-REC
                AFTER ADVANCING PAGE.
        ADD 1 TO PAGE-NUM-WS.
        PERFORM 600-WRITE-COL-HEADING.
*/

  public void printHeading( )
  {
       pageNo++;
       outputStream.println( "\f" + toDay.getDay( ) + "/" +( toDay.getMonth() + 1) + "/"+ (
toDay.getYear() +1900 )
                                    + "                        "+ "XYZ WHOLESALE SUPPLY
COMPANY" +
                                  "              "+ "PAGE" + " "+ pageNo  );
       columnHeading();
   }

/*   600-WRITE-COL-HEADING.
        WRITE REPORT-REC FROM COL-HEADING-REC
                AFTER ADVANCING 2 LINES.
        WRITE REPORT-REC FROM SPACES.
*/

  public void columnHeading()
  {
      outputStream.println( " " );
        outputStream.println( "      "+ "DEPT NUM"+ "      "+ "TRANS DATE"+ "      "+
```

```
"ORDER AMOUNT");
       outputStream.println( " " );
   }



/*
      250-GET-DATE.
         ACCEPT DATE-WS FROM DATE.
         MOVE DAY-WS TO DAY-HR.
         MOVE MONTH-WS TO MONTH-HR.
         IF YEAR-WS < 97
            ADD 2000 TO YEAR-WS GIVING YEAR-HR
         ELSE ADD 1900 TO YEAR-WS GIVING YEAR-HR.
*/

  public void departmentTotal( )
  {
     deptCount++;
     companyTotal += deptTotal;
     outputStream.println( " ");
     outputStream.print( "                    "+ "TOTAL AMOUNT FOR DEPARTMENT" );
     outputStream.println( "    "+ moneyFormat.format(deptTotal) );
     outputStream.println( );
     outputStream.println( );
     if  ( ( salesData.getLine() != null ) && ( deptCount == 2 ) )
         {
          printHeading( );
               deptCount = 0;
               }
    holdDepartmentNumber = salesData.getDepartmentNumber();
    deptTotal = 0;
    firstRecord = true;
    }

/*       350-TOTAL-A-DEPARTMENT.
         ADD 1 TO DEPT-COUNT-WS.
         ADD DEPT-TOTAL-WS TO COMPANY-TOTAL-WS.
         MOVE DEPT-TOTAL-WS TO TOT-AMOUNT-DT.
         WRITE REPORT-REC FROM DEPT-TOTALS
                   AFTER ADVANCING 2 LINES.
         IF MORE-RECORDS
           IF DEPT-COUNT-WS = 2
              PERFORM 500-HEADING-RTN
              MOVE ZERO TO DEPT-COUNT-WS
           END-IF
           MOVE "YES" TO DEPT-FLAG
```

```
           MOVE ZERO TO DEPT-TOTAL-WS
           MOVE DEPT-NUM-SR TO HOLD-DEPT-NUM.

       400-WRITE-DETAIL-LINE.
           PERFORM 450-MOVE-DATA.
           WRITE REPORT-REC FROM DETAIL-LINE.
*/
   public void printDetailLine()
        {
         if  ( firstRecord )
          {
          outputStream.print( "         " + salesData.getDepartmentNumber( ));
          String salesDate = salesData.getSaleDate();
          outputStream.print( "             "+ salesDate.substring(0, 2) + "/" );
                outputStream.print( salesDate.substring(2, 4) + "/"+ salesDate.substring(4, 6) );
          outputStream.println( "            "+  moneyFormat.format(salesData.getSaleAmount()
) );
          firstRecord = false;
          }
         else
          {
          outputStream.print( "         " + "           " );
          String salesDate = salesData.getSaleDate();
          outputStream.print( "           "+ salesDate.substring(0, 2) + "/");
                outputStream.print(salesDate.substring(2, 4) + "/"+ salesDate.substring(4, 6) );
          outputStream.println( "            "+  moneyFormat.format(salesData.getSaleAmount())
);
          }
        }

/*       400-READ-A-REC.
          READ SALES-FILE
              AT END MOVE "NO " TO ARE-THERE-MORE-RECORDS.

       450-MOVE-DATA.
           IF FIRST-RECORD
              MOVE DEPT-NUM-SR TO DEPT-NUM-DL
              MOVE "NO " TO DEPT-FLAG
           ELSE MOVE ZEROS TO DEPT-NUM-DL.
           MOVE TRANS-DATE-SR TO TRANS-DATE-DL.
           MOVE AMOUNT-SR TO AMOUNT-DL.
*/
   public void companyTotals( )
   {
       outputStream.println( " ");
       outputStream.print( "COMPANY TOTAL              " );
       outputStream.println(  moneyFormat.format(companyTotal) );
```

```
    }

/*      700-WRITE-TOTALS.
        PERFORM 750-MOVE-TOTALS.
        WRITE REPORT-REC FROM TOTAL-LINE
                AFTER ADVANCING 2 LINES.
        MOVE SPACES TO REPORT-REC.
        WRITE REPORT-REC AFTER 2.
      750-MOVE-TOTALS.
        MOVE COMPANY-TOTAL-WS TO TOT-AMOUNT-TL.
*/

 }
```

DEPT NUM	TRANS DATE	ORDER AMOUNT
1000	01/03/91	$100.00
	01/03/91	$120.00
	01/03/91	$30.00
	01/04/91	$44.00
	01/04/91	$5.90
	01/05/91	$234.00
	TOTAL AMOUNT FOR DEPARTMENT	$533.90
2000	01/03/91	$200.00
	01/03/91	$234.00
	01/04/91	$340.00
	TOTAL AMOUNT FOR DEPARTMENT	$774.00

DEPT NUM	TRANS DATE	ORDER AMOUNT	
3000	01/04/91	$200.00	
	01/04/91	$39.99	
	01/05/91	$44.00	
	TOTAL AMOUNT FOR DEPARTMENT		$283.99
4000	01/03/91	$50.00	
	TOTAL AMOUNT FOR DEPARTMENT		$50.00

COMPANY TOTAL $1,641.89

APPENDIX K - COBOL program to create a relative file.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.     CREMP.
       AUTHOR.         J WING.

      ****************************************************************
      *    THIS PROGRAM CREATES A RELATIVE EMPLOYEE FILE             *
      ****************************************************************

       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
          SELECT EMPFILE ASSIGN TO DISK "EMPMAST"
                      ORGANIZATION IS RELATIVE
                      ACCESS MODE IS SEQUENTIAL
                      RELATIVE KEY IS WS-KEY.

       DATA DIVISION.
       FILE SECTION.

       FD  EMPFILE
          RECORD CONTAINS 41 CHARACTERS
          LABEL RECORDS ARE STANDARD
          VALUE OF FILE-ID IS "EMPMAST".
       01  EMP-REC.
          03  EMP-NO              PIC 9(3).
          03  EMP-FIRST-NAME      PIC X(15).
          03  EMP-LAST-NAME       PIC X(15).
          03  EMP-SALARY          PIC 9(6)V99.


       WORKING-STORAGE SECTION.
       01  WS-KEY                 PIC 9(3) VALUE ZEROS.


       PROCEDURE DIVISION.

      ****************************************************************
      *    THE PROGRAM'S MAIN OPERATIONS ARE CONTROLLED IN THIS      *
      *    PARAGRAPH.                                                *
      ****************************************************************

       000-MAIN.
          PERFORM 100-INIT.
          PERFORM 200-PROC VARYING WS-KEY FROM 1 BY 1 UNTIL
```

```
                    WS-KEY > 100.
        PERFORM 300-FINAL.
        STOP RUN.

    ***************************************************************
    *   THE FILES ARE OPENED AND ALL VARIABLES INITIALIZED IN     *
    *   THE FOLLOWING PARAGRAPH.                                  *
    ***************************************************************

     100-INIT.
        OPEN OUTPUT EMPFILE.


    ***************************************************************
    *   THIS PARAGRAPH PERFORMS THE WRITING OF NEW MASTER RECORDS.
8
    ***************************************************************

     200-PROC.
        MOVE ZEROS TO EMP-NO, EMP-SALARY.
        MOVE SPACES TO EMP-FIRST-NAME, EMP-LAST-NAME.
        WRITE EMP-REC.


     300-FINAL.
        CLOSE EMPFILE.
```

APPENDIX L - COBOL program to add an Employee to the relative file.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.     ADDEMP.
 AUTHOR.         J WING.

*****************************************************************
*    THIS PROGRAM ADDS RECORDS TO A RELATIVE EMPLOYEE FILE
 *
*****************************************************************

ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
    SELECT EMPFILE ASSIGN TO DISK "EMPMAST"
                ORGANIZATION IS RELATIVE
                ACCESS MODE IS RANDOM
                RELATIVE KEY IS WS-KEY.
    SELECT IN-FILE ASSIGN TO DISK "INFILE.txt"
                ORGANIZATION IS LINE SEQUENTIAL.

 DATA DIVISION.

 FILE SECTION.

 FD  IN-FILE
    RECORD CONTAINS 41 CHARACTERS
    LABEL RECORDS ARE OMITTED.
 01  IN-REC.
    03  IN-EMP              PIC 9(3).
    03  FILLER              PIC X(38).

 FD  EMPFILE
    RECORD CONTAINS 41 CHARACTERS
    LABEL RECORDS ARE STANDARD.
 01  EMP-REC.
    03  EMP-NO              PIC 9(3).
    03  FILLER              PIC X(38).

 WORKING-STORAGE SECTION.
 01  WS-KEY                 PIC 9(3).

 01  WS-EOF                 PIC X VALUE "N".
    88  EOF                 VALUE "Y".
```

```
 PROCEDURE DIVISION.

*****************************************************************
*    THE PROGRAM'S MAIN OPERATIONS ARE CONTROLLED IN THIS     *
*    PARAGRAPH.                                                 *
*****************************************************************

 000-MAIN.
    PERFORM 100-INIT.
    PERFORM 200-PROCESS UNTIL EOF.
    PERFORM 300-FINAL.
    STOP RUN.

*****************************************************************
*   THE FILES ARE OPENED
*****************************************************************

 100-INIT.
    OPEN INPUT IN-FILE
         I-O EMPFILE.
    PERFORM 410-READ-IN-FILE.

*****************************************************************
*   THIS PARAGRAPH PERFORMS THE WRITING OF NEW MASTER RECORDS
*****************************************************************

 200-PROCESS.
    PERFORM 420-READ-MAST.
    IF EMP-NO = 0
       PERFORM 230-WRITE-MASTER
    ELSE
       DISPLAY "CANNOT ADD RECORD ALREADY EXISTS " EMP-NO.
    PERFORM 410-READ-IN-FILE.

 230-WRITE-MASTER.
    REWRITE EMP-REC FROM IN-REC.

 300-FINAL.
    CLOSE IN-FILE
          EMPFILE.
```

```
*****************************************************************
*    THESE  PARAGRAPHS  CONTROL  THE READING OF THE  FILES*
*****************************************************************

 410-READ-IN-FILE.
    READ IN-FILE
       AT END MOVE "Y" TO WS-EOF.

 420-READ-MAST.
    MOVE IN-EMP TO WS-KEY.
    READ EMPFILE INVALID KEY
              DISPLAY "CANNOT ACCESS MASTER RECORD." IN-EMP.
```

APPENDIX M - Java class to define the employee.

```
// Record.java
// Record class for the RandomAccessFile programs.
// Contains the object, and all associated input/output methods
// This is effectively the JDC (Java Data Class)
// The COBOL equivalents that are in this class, are the
// record description, the OPEN statement, the CLOSE satatement,
// a direct READ, a sequential READ, a direct WRITE,
// and a sequential WRITE

import java.io.*;

public class Record {
   private int employee;
   private String lastName;
   private String firstName;
   private double salary;

   private RandomAccessFile  employeeData;

   public void openFile( String openMode) throws IOException
   {
    // Open the file
       employeeData = new RandomAccessFile( "employee.dat", openMode );
    }

   public void closeFile( ) throws IOException
   {
    // Close the file
       employeeData.close();
    }

   // Read a record from the specified RandomAccessFile,
   // where the record number is supplied - a direct read
   public void read( int empNo ) throws IOException
   {

     employeeData.seek(
              (long) ( empNo -1 ) * Record.size() );

      employee =employeeData.readInt();

      char first[] = new char[ 15 ];

      for ( int i = 0; i < first.length; i++ )
         first[ i ] =employeeData.readChar();
```

```
   firstName = new String( first );

   char last[] = new char[ 15 ];

   for ( int i = 0; i < last.length; i++ )
      last[ i ] =employeeData.readChar();

   lastName = new String( last );

   salary = employeeData.readDouble();
}

// Read a record from the specified RandomAccessFile,
// where the record number is not supplied - a sequential  read
public void read( ) throws IOException
{

   employee =employeeData.readInt();

   char first[] = new char[ 15 ];

   for ( int i = 0; i < first.length; i++ )
      first[ i ] =employeeData.readChar();

   firstName = new String( first );

   char last[] = new char[ 15 ];

   for ( int i = 0; i < last.length; i++ )
      last[ i ] =employeeData.readChar();

   lastName = new String( last );

   salary = employeeData.readDouble();
}

// Write a record to the specified RandomAccessFile,.
// where the employee number is supplied - a direct write
public void write(int empNo ) throws IOException
{

  employeeData.seek(
            (long) ( empNo -1 ) * Record.size() );

   StringBuffer buf;

  employeeData.writeInt( employee );
```

```
    if ( firstName != null )
      buf = new StringBuffer( firstName );
    else
      buf = new StringBuffer( 15 );

    buf.setLength( 15 );

    employeeData.writeChars( buf.toString() );

    if ( lastName != null )
      buf = new StringBuffer( lastName );
    else
      buf = new StringBuffer( 15 );

    buf.setLength( 15 );

    employeeData.writeChars( buf.toString() );

   employeeData.writeDouble( salary );
}

// Write a record to the specified RandomAccessFile,
// where the employee number is not supplied - a sequential write
public void write( ) throws IOException
{

    StringBuffer buf;

   employeeData.writeInt( employee );

    if ( firstName != null )
      buf = new StringBuffer( firstName );
    else
      buf = new StringBuffer( 15 );

    buf.setLength( 15 );

    employeeData.writeChars( buf.toString() );

    if ( lastName != null )
      buf = new StringBuffer( lastName );
    else
      buf = new StringBuffer( 15 );

    buf.setLength( 15 );

    employeeData.writeChars( buf.toString() );
```

```
    employeeData.writeDouble( salary );
  }

  public void setEmployee( int a ) { employee = a; }

  public int getEmployee() { return employee; }

  public void setFirstName( String f ) { firstName = f; }

  public String getFirstName() { return firstName; }

  public void setLastName( String l ) { lastName = l; }

  public String getLastName() { return lastName; }

  public void setSalary( double b ) { salary = b; }

  public double getSalary() { return salary; }

  // NOTE: This method contains a hard coded value for the
  // size of a record of information.
  public static int size() { return 72; }
}
```

APPENDIX N - Java class to create an empty random file of employees.

```
// CreateRandomFile.java
// This program creates a random access file sequentially
// by writing 100 empty records to disk.

import java.io.*;

public class CreateRandomFile {
   private Record blank;

   public CreateRandomFile()
   {

     blank = new Record();



// Open the file
      try
        {
         blank.openFile( "rw");
        }
      catch( IOException e )
         {
         System.err.println( "File not opened properly\n" +
                             e.toString() );
         System.exit( 1 );
          }



// Write the records
       try
         {
         for ( int i = 0; i < 100; i++ )
            blank.write( );
         }
      catch( IOException e )
         {
         System.err.println( "Cannot create empty records\n" +
                             e.toString() );
         System.exit( 1 );
          }
```

```
// Close the file
      try
        {
         blank.closeFile( );
        }
      catch( IOException e )
         {
         System.err.println( "File not closed properly\n" +
                           e.toString() );
         System.exit( 1 );
          }



   }

   public static void main( String args[] )
   {
     CreateRandomFile accounts = new CreateRandomFile();
   }
}
```

APPENDIX O - Java class to write employee records to the random file, and the Java class to display all employees.

```
// WriteRandomFile.java
// This program uses TextFields to get information from the
// user at the keyboard and writes the information to a
// random access file.

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.JOptionPane;

public class WriteRandomFile extends Frame
             implements ActionListener {

   // TextFields where user enters employee number, first name,
   // last name and salary.
   private TextField employeeField, firstNameField,
                     lastNameField, salaryField;

   private Button enter,  // send record to file
                  done;   // quit program

   // Application other pieces

   private Record data;

   // Constructor -- intialize the Frame

   public WriteRandomFile()
   {
      super( "Write to random access file" );

      data = new Record();

      try {
          data.openFile( "rw" );
          }
      catch ( IOException e )
          {
         System.err.println( e.toString() );
         System.exit( 1 );
          }

      setSize( 300, 150 );
      setLayout( new GridLayout( 5, 2 ) );
```

```
   // create the components of the Frame
   add( new Label( "Employee Number" ) );
   employeeField = new TextField();
   add( employeeField );

   add( new Label( "First Name" ) );
   firstNameField = new TextField( 20 );
   add( firstNameField );

   add( new Label( "Last Name" ) );
   lastNameField = new TextField( 20 );
   add( lastNameField );

   add( new Label( "Salary" ) );
   salaryField = new TextField( 20 );
   add( salaryField );

   enter = new Button( "Enter" );
   enter.addActionListener( this );
   add( enter );

   done = new Button( "Done" );
   done.addActionListener( this );
   add( done );

   setVisible( true );
}

public void addRecord()
{
   int employeeNumber = 0;
   Double d;

   if ( ! employeeField.getText().equals( "" ) ) {

      // output the values to the file
      try {
         employeeNumber =
            Integer.parseInt( employeeField.getText() );

         if ( employeeNumber > 0 && employeeNumber <= 100 )

         {
            data.setEmployee( employeeNumber );
            data.setFirstName( firstNameField.getText() );
            data.setLastName( lastNameField.getText() );
            d = new Double ( salaryField.getText() );
```

```
            data.setSalary( d.doubleValue() );
            data.write( employeeNumber );

             // clear the TextFields
             employeeField.setText( "" );
             firstNameField.setText( "" );
             lastNameField.setText( "" );
             salaryField.setText( "" );
          }

         else

          {
            JOptionPane.showMessageDialog(
                 null, "Employee number must be less than 100\n"+
                       "Please re-enter the data \n"+
                       "No record was created" ) ;
          }
        }
        catch ( NumberFormatException nfe )
        {
          System.err.println(
            "You must enter an integer employee number" );
        }
        catch ( IOException io )
        {
          System.err.println(
            "Error during write to file\n" +
            io.toString()  + " ; mployee Number" +  employeeNumber );
          System.exit( 1 );
        }
      }
}

public void actionPerformed( ActionEvent e )
{
   addRecord();

   if ( e.getSource() == done )
     {
      try
      {
        data.closeFile();
       }
      catch ( IOException io )
        {
         System.err.println( "File not closed properly\n" +
```

```
                        io.toString() );
        }
        System.exit( 0 );
      }

    }

    // Instantiate a WriteRandomFile object and start the program
    public static void main( String args[] )
    {
      new WriteRandomFile();
    }
}
```

```
// This program reads a random access file sequentially and
// displays the contents one record at a time in text fields.

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;

public class ReadRandomFile extends Frame
            implements ActionListener {

   // TextFields to display employee number, first name,
   // last name and salary.
   private TextField employeeField, firstNameField,
                     lastNameField, salaryField;

   private Button next,   // get next record in file
                  done;   // quit program

   // Application other pieces

   private Record data;

   // Constructor -- initialize the Frame
   public ReadRandomFile()
   {
      super( "Read Employee File" );

      data = new Record();

      // Open the file
      try
      {
         data.openFile( "r" );
      }
      catch ( IOException e ) {
         System.err.println( e.toString() );
         System.exit( 1 );
      }

      setSize( 300, 150 );
      setLayout( new GridLayout( 5, 2 ) );

      // create the components of the Frame
      add( new Label( "Employee Number" ) );
      employeeField = new TextField();
      employeeField.setEditable( false );
```

```
   add( employeeField );

   add( new Label( "First Name" ) );
   firstNameField = new TextField( 20 );
   firstNameField.setEditable( false );
   add( firstNameField );

   add( new Label( "Last Name" ) );
   lastNameField = new TextField( 20 );
   lastNameField.setEditable( false );
   add( lastNameField );

   add( new Label( "Salary" ) );
   salaryField = new TextField( 20 );
   salaryField.setEditable( false );
   add( salaryField );

   next = new Button( "Next" );
   next.addActionListener( this );
   add( next );

   done = new Button( "Done" );
   done.addActionListener( this );
   add( done );

   setVisible( true );
}

public void actionPerformed( ActionEvent e )
{
   if ( e.getSource() == next )
      readRecord();
   else
      {
       endProcess();
       }

}

public void readRecord()
{
   DecimalFormat twoDigits = new DecimalFormat( "0.00" );

   // read a record and display
   try {
      do {
         data.read( );
```

```
            } while ( data.getEmployee() == 0 );
              // loop through the read until an active record is found

          employeeField.setText(
             String.valueOf( data.getEmployee() ) );
          firstNameField.setText( data.getFirstName() );
          lastNameField.setText( data.getLastName() );
          salaryField.setText( String.valueOf(
                twoDigits.format( data.getSalary() ) ) );
          }
        // An eof exception is an expected exception that will occur
        // at the end of the data file.
        catch ( EOFException eof )
          {
          endProcess();
          }
        catch ( IOException e ) {
          System.err.println( "Error during read from file\n" +
                           e.toString() );
          System.exit( 1 );
        }

    }

    private void endProcess()
    {
      try
        {
        data.closeFile();
        System.exit( 0 );
         }
      catch ( IOException e )
         {
         System.err.println( "Error closing file\n" +
                                  e.toString() );
         System.exit( 1 );
          }
     }

    // Instantiate a ReadRandomFile object and start the program
    public static void main( String args[] )
    {
      new ReadRandomFile();
    }
}
```

APPENDIX P - A COBOL subroutine, validates a seven digit number, for a valid modulus 11 check digit .

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. CHKDIG.
      *
      * THIS PROGRAM IS A SUBROUTINE THAT ACCEPTS A SEVEN DIGIT
      *  NUMBER
      * AND A FLAG FIELD.   THIS SUBROUTINE CHECKS USING MODULUS-11
      * TO SEE IF THE CHECK DIGIT IS CORRECT, AND THEREFORE THAT THE
      * NUMBER HAS BEEN ENTERED CORRECTLY.  IF THE NUMBER IS CORRECT,
      * THE FLAG IS SET TO "Y". IF THE NUMBER IS INCORRECT, THE FLAG
      * IS SET TO "N".
      *
       AUTHOR.  J. WING.
       INSTALLATION.  TECHNIKON NATAL.
       DATE-WRITTEN.  27/4/87.
       DATE-COMPILED.

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.  IBM-PC.
       OBJECT-COMPUTER.  IBM-PC.

       DATA DIVISION.
       FILE SECTION.

       WORKING-STORAGE SECTION.
       01  WS-CALC-FIELDS.
           05  PROD-D1        PIC 999.
           05  PROD-D2        PIC 999.
           05  PROD-D3        PIC 999.
           05  PROD-D4        PIC 999.
           05  PROD-D5        PIC 999.
           05  PROD-D6        PIC 999.
           05  PROD-D7        PIC 999.
       01  WS-ANSWERS.
           05  WS-TOTAL       PIC 9(5).
           05  WS-ANS         PIC 9(5).
           05  WS-REM         PIC 9.

       LINKAGE SECTION.
       01  LK-STU-NO.
           05  LK-D1      PIC 9.
           05  LK-D2      PIC 9.
           05  LK-D3      PIC 9.
           05  LK-D4      PIC 9.
```

```
    05  LK-D5       PIC 9.
    05  LK-D6       PIC 9.
    05  LK-D7       PIC 9.
01  LK-FLAG         PIC X.

PROCEDURE DIVISION
                USING LK-STU-NO LK-FLAG.

A-100-MAINLINE.
    MOVE "Y" TO LK-FLAG.
    PERFORM B-100-CALC-CHKDIG.
    PERFORM B-200-SET-FLAG.

A-200-EXIT.
    EXIT PROGRAM.

B-100-CALC-CHKDIG.
    MULTIPLY LK-D7 BY 1 GIVING PROD-D7.
    MULTIPLY LK-D6 BY 2 GIVING PROD-D6.
    MULTIPLY LK-D5 BY 3 GIVING PROD-D5.
    MULTIPLY LK-D4 BY 4 GIVING PROD-D4.
    MULTIPLY LK-D3 BY 5 GIVING PROD-D3.
    MULTIPLY LK-D2 BY 6 GIVING PROD-D2.
    MULTIPLY LK-D1 BY 7 GIVING PROD-D1.
    ADD PROD-D1 PROD-D2 PROD-D3 PROD-D4 PROD-D5 PROD-D6
                PROD-D7 GIVING WS-TOTAL.
    DIVIDE WS-TOTAL BY 11 GIVING WS-ANS
                        REMAINDER WS-REM.
*
* IF THE REMAINDER IS ZERO THEN THE CHECK DIGIT IS VALID.
*

B-200-SET-FLAG.
    IF WS-REM = ZEROS
       MOVE "Y" TO LK-FLAG
    ELSE
       MOVE "N" TO LK-FLAG.
```

APPENDIX Q - A COBOL main program, to demonstrate calling a subroutine.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. MAINPR.
*
* THIS IS A MAIN PROGRAM THAT CALLS A SUBROUTINE TO CHECK
* WETHER A CHECK DIGIT IS CORRECT. THE SUBROUTINE USES
* MODULUS-11 TO SEE IF THE CHECK DIGIT IS CORRECT.
* IF THE NUMBER IS CORRECT, THE FLAG IS SET TO "Y".
* IF THE NUMBER IS INCORRECT, THE FLAG IS SET TO "N".
*
 AUTHOR.  J. WING.
 INSTALLATION.  TECHNIKON NATAL.
 DATE-WRITTEN.  08/06/1997.
 DATE-COMPILED.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  IBM-PC.
 OBJECT-COMPUTER.  IBM-PC.
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.
 01  WS-STU-NO       PIC 9(7).
 01  WS-FLAG         PIC X.

 PROCEDURE DIVISION.
 A-100-MAINLINE.
    MOVE 6213457 TO WS-STU-NO.
    PERFORM B-100-PROCESS.
    MOVE 8725187 TO WS-STU-NO.
    PERFORM B-100-PROCESS.
    MOVE 8710074 TO WS-STU-NO.
    PERFORM B-100-PROCESS.
    STOP RUN.

 B-100-PROCESS.
    PERFORM C-100-CALL.
    PERFORM C-200-DISPLAY.

 C-100-CALL.
    CALL "CHKDIG" USING WS-STU-NO, WS-FLAG.

C-200-DISPLAY.
    IF WS-FLAG = "Y"
       DISPLAY "STUDENT NUMBER " WS-STU-NO " IS VALID"
    ELSE
       DISPLAY "STUDENT NUMBER " WS-STU-NO " IS NOT VALID".
```

APPENDIX R - A Java Class that can be used to validate a number according to the modulus-11 check digit method.

```
public class CheckDigit
{

private String studentNumber;

// Constructor

CheckDigit ( String sNo)
    {
      studentNumber = sNo;
    }

public boolean isValid( )
    {
            int weight = studentNumber.length( );
        int sum = 0;

        for ( int i = 0; i<=( (studentNumber.length() -1) ) ; i++ )
                {
            sum += ( Integer.parseInt(studentNumber.substring(i, i+1) ) * weight );
                weight = weight - 1;
                }

        if ( (sum % 11) == 0 )
             return true;
        else
             return false;
    }
}
```

APPENDIX S - A Java class to demonstrate the use of the class CheckDigit.

```
public class DemoCheckDigit
{

      CheckDigit valNo;


        public static void main (String[] args)
        {
                DemoCheckDigit testSubroutine = new DemoCheckDigit( );
                String stNum;

                stNum = "6213457";
                testSubroutine.callAndPrint( stNum);

                stNum = "8725187";
                testSubroutine.callAndPrint( stNum);

                stNum = "8710674";
                testSubroutine.callAndPrint( stNum);

                stNum = "8403201";
                testSubroutine.callAndPrint( stNum);

        }

        public void callAndPrint( String stNum )
        {
                valNo = new CheckDigit( stNum );
                if (valNo.isValid() )
                    System.out.println("Student number "+ stNum + " is valid ");
                        else
                    System.out.println("Student number "+ stNum + " is NOT valid");
        }
}
```

APPENDIX T - OO COBOL program structure.

```
CLASS-ID.     Class-name INHERITS Class-name.
ENVIRONMENT DIVISION.....
.....
DATA DIVISION.
WORKING-STORAGE SECTION.
        (The data for class, global to all methods in the class )
PROCEDURE DIVISION.
METHOD-ID.          Method-name.
DATA DIVISION.
LINKAGE SECTION.
        (The data that is local to the method )
PROCEDURE DIVISION USING parameters passed to the method,
                        RETURNING data returned from the method.
END METHOD    Method-name.
(There may be any number of methods)
```