abstract void resize();
}

Java code for Subclasses:

class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}

Example 3: Employee payroll system.

Consider the following case study:

A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and salaried-commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants to implement a Java application that performs its payroll calculations polymorphically.

![Employee Hierarchy UML class diagram](image)

Please note that “Employee” is in Italics since it is an abstract class.
### STRUCTURED PROGRAMMING TECHNIQUES

#### TOP-DOWN AND BOTTOM UP APPROACHES

Top-down approach in problem solving involves:

1. Breaking down a complex problem into simpler, manageable sub-problems
2. Each sub-problem is either broken down further or simply resolved

<table>
<thead>
<tr>
<th>Class</th>
<th>Method ‘earnings’</th>
<th>Method ‘toString’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>abstract</td>
<td>firstName, lastName, SSN (social security number)</td>
</tr>
<tr>
<td>SalariedEmployee</td>
<td>weeklySalary</td>
<td>firstName, lastName, SSN, weeklySalary</td>
</tr>
</tbody>
</table>
| HourlyEmployee         | If hours <= 40 wage * hours         | firstName, lastName, SSN, weeklySalary wage = hourly wage  
                         | If hours > 40 40 * wage + (hours – 40) * wage * 1.5  
                         | hours = hours worked                  |
| CommissionEmployee     | commissionRate * grossSales        | firstName, lastName, SSN, weeklySalary, grossSales, commissionRate |
| BasePlusCommissionEmployee | (commissionRate * grossSales) + baseSalary | firstName, lastName, SSN, weeklySalary, grossSales, commissionRate, baseSalary |

This type of approach is ideal in solving complex and large systems.
Disadvantage: since problems are broken down into smaller ones until solved (forming a tree structure), there might be the case that a sub-problem could be common to one or more branches. This leads to waste of resources in trying to solve the same problem. It is not easy to identify common sub-problems and share the solution (re-usability – solution to common problems stored in a library for future reference).

Example:

Consider developing a software system for a large company with several departments. It may be tempting to use a top down design strategy and allocate a separate team for developing software to cater for the IT needs of each department. What is the main problem with this approach?

Among other services, each department is likely to require some word processing facilities; this sub-problem is shared among many departments; solving it independently for each department has several substantial drawbacks; in addition to the unnecessary huge cost, this may lead to serious compatibility and reliability problems.

Conclusion: Top-down approach is essential for solving any complex problem BUT, care should be taken to identify common tasks in the sub-problems generated!

Examples of Top Down approach:

1. Data flow decomposition (chart/diagrammatic decomposition - DFD)
2. Sequential decomposition
3. Recursive decomposition
4. Divide and conquer

Bottom up approach involves:

1. Designing solution to a problem in a sequence of layers starting from the bottom layer
2. Each new layer builds on top of the previous one, utilizing also previous layers (e.g. building of a house)

Example: converting number to text, one should consider all the possibilities, i.e. from 1 to 9, 11 to 19, hundreds, thousands, etc.)

MODULARITY

An approach to developing software (even hardware) that breaks projects into smaller units (or modules). These are designed as standalone units that can work with other sections of the program. The same module can perform the same task in another or several other programs or components. Modifying the way that each module works will have no adverse affects on the other components of a program. High modularity costs some design time but pays back well through clarity, elegance, maintainability and flexibility.

Riccardo Flask B.Ed.(Hons.), Dip.C.S.Ed
SEQUENCE, SELECTION AND ITERATION

Sequence, Selection (conditional) and Iteration (loops) operations can be illustrated as part of the Jackson Structured Programming methodology.

i. Sequence involves moving from one step (operations) to another. This can be illustrated as follows:

   ![Sequence Diagram]

   Step 1 → Step 2 → Step 3

ii. Selection operation occurs where the flow of the program depends on a condition (if-then-else)

   ![Selection Diagram]

   The circle represents the possible paths to take based on the condition (yes/no; true/false)

iii. Iteration involves repeating one or more steps for a number of times (loop)

   ![Iteration Diagram]

ADVANTAGES OF STRUCTURED TECHNIQUES

Structured programming is a sub set of a major programming paradigm, i.e. Procedural Programming. Pascal and C are examples of languages which adopt such programming techniques. Structured Programming can be defined as a technique for organizing and coding computer programs in which a hierarchy of modules is used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used: sequential, selection, and iteration.
The following are advantages of this programming technique:

- **Easy to write:** Modular design increases the programmer’s productivity by allowing them to look at the big picture first and focus on details later. Several Programmers can work on a single, large program, each working on a different module. Studies show structured programs take less time to write than standard programs. Procedures written for one program can be reused in other programs requiring the same task. A procedure that can be used in many programs is said to be reusable.

- **Easy to debug:** Since each procedure is specialized to perform just one task, a procedure can be checked individually. Older unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such programs is cluttered with details and therefore difficult to follow.

- **Easy to understand:** The relationship between the procedures shows the modular design of the program. Meaningful procedure names and clear documentation identify the task performed by each module. Meaningful variable names help the programmer identify the purpose of each variable.

- **Easy to change:** Since a correctly written structured program is self-documenting, it can be easily understood by another programmer.

### PROBLEM SOLVING TECHNIQUES

#### FLOWCHARTS AND PSEUDOCODE

Flowcharts are a graphic representation of the logic or steps in a program or system. A flowchart represents how a program or activity moves through various processes or program routines. It uses symbols to represent the activities, and it uses arrows to represent the direction of activity through the processes. Flowcharts can be used to define the behavior of a single program or a system (a combination of programs).

- Flow Chart Notation (Process, Start/Stop, Decision Box, I/O, Connector, Document, Store, Data Flow)

Pseudocode (false code) is a short hand way of describing a computer program. Rather than use the specific syntax of a computer language, more general wording is used. Using pseudocode, it is easier for a non-programmer to understand the general workings of the program.
SEARCHING AND SORTING ALGORITHMS

Searching algorithms take a problem as input and return a solution to it. Sorting algorithms are essential as they sort data in arrays or lists so that this data can be further manipulated by other algorithms.

LINEAR AND BINARY SEARCH

The linear search algorithm searches each element in an array sequentially. If the search key does not match an element in the array, the algorithm tests each element, and when the end of the array is reached, informs the user that the search key is not present. If the search key is in the array, the algorithm tests each element until it finds one that matches the search key and returns the index of that element.

As an example, consider an array containing the following values

34 56 2 10 77 51 93 30 5 52

and a program that is searching for 51. Using the linear search algorithm, the program first checks whether 34 matches the search key. It does not, so the algorithm checks whether 56 matches the search key. The program continues moving through the array sequentially, testing 2, then 10, then 77. When the program tests 51, which matches the search key, the program returns the index 5, which is the location of 51 in the array. If, after checking every array element, the program determines that the search key does not match any element in the array, the program returns a sentinel value (e.g., -1).

The binary search algorithm is more efficient than the linear search algorithm, but it requires that the array be sorted. The first iteration of this algorithm tests the middle element in the array. If this matches the search key, the algorithm ends. Assuming the array is sorted in ascending order, then if the search key is less than the middle element, the search key cannot match any element in the second half of the array and the algorithm continues with only the first half of the array (i.e., the first element up to, but not including the middle element). If the search key is greater than the middle element, the search key cannot match any element in the first half of the array and the algorithm continues with only the second half of the array (i.e., the element after the middle element through the last element). Each iteration tests the middle value of the remaining portion of the array. If the search key does not match the element, the algorithm eliminates half of the remaining elements. The algorithm ends either by finding an element that matches the search key or reducing the sub-array to zero size.

As an example consider the sorted 15-element array:

2 3 5 10 27 30 34 51 56 65 77 81 82 93 99

And a search key of 65. A program implementing the binary search algorithm would first check whether 51 is the search key (because 51 is the middle element of the array). The search key (65) is larger than 51, so 51 is discarded along with the first half of the array (all elements smaller than 51.) Next, the algorithm checks whether 81 (the middle element of the remainder of the array) matches the search key. The search key (65) is smaller than 81, so 81 is discarded along with the elements larger than 81. After just two tests, the algorithm...
has narrowed the number of values to check to three (56, 65 and 77). The algorithm then checks 65 (which
indeed matches the search key), and returns the index of the array element containing 65. This algorithm
required just three comparisons to determine whether the search key matched an element of the array. Using
a linear search algorithm would have required 10 comparisons. [Note: In this example, we have chosen to use
an array with 15 elements so that there will always be an obvious middle element in the array. With an even
number of elements, the middle of the array lies between two elements. We implement the algorithm to
choose the lower of those two elements.]

HASH TABLES AND FUNCTIONS

We already discussed Hash tables later on in this course, as data structures they are compared to dictionaries.
They allow a fast lookup of a data record, given its key (in a dictionary the keys are the terms and the data
record would be the definitions). The link between keys and data is known as the hashing function. Ideally a
hashing function maps each key to a unique index, but this is not always the case (impossible/impractical), e.g.

```
0   #
1   #  16
2   #
3   #  27
4   #
5   #
6   #  11
7   #  22
```

The above diagram illustrates a hash Table as an array with 8 elements. Each element is a pointer to a linked
list of numeric data. The hash function for this example simply divides the data key by 8, and uses the
remainder as an index into the table. To insert a new item in the table, we hash the key to determine which list
the item goes on, and then insert the item at the beginning of the list. For example, to insert 11, we divide 11
by 8 giving a remainder of 3. Thus, 11 goes on the list starting at hashTable[3]. To find a number, we hash the
number and chain down the correct list to see if it is in the table. To delete a number, we find the number and
remove the node from the linked list.

ITERATIVE AND RECURSIVE ALGORITHMS

In particular situations a problem can be solved either iteratively or recursively. However recursion is usually
much more efficient than iteration. Let us take the factorial function as an example. If we had to write code to
solve this problem we can do it either iteratively or recursively:

**Factorial Function, e.g. the factorial of 5, i.e. 5! = 5 x 4 x 3 x 2 x 1**

**Iterative approach:**

```
Result = 1
for i = 1 to n
   Result = Result * i
```

**Recursive approach:**

```
If x = 1 then Result = 1
Else Result = x * factorial(x-1)
```

Note that the function is calling itself, i.e. a function which uses the same function until the base case is reached.

Note that 1! = 1 (0! = 1 as well), this is known as the base case.
ITERATIVE SORT – BUBBLE AND INSERTION SORT

Bubble Sort: a simple method of sorting. This algorithm starts at the beginning/end of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. Although simple, this algorithm is highly inefficient and is rarely used.

```java
for i ← 1 to length[A]
  do for j ← length[A] downto i + 1
```

Insertion Sort: one of the simplest methods to sort an array is an insertion sort. An example of an insertion sort is while sorting out playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence.

Suppose we have an array ‘a’ with size ‘n’:

For i = 1 to n-1
  1. Set j = i
  2. Set t = a[j]
  3. Repeat
     a. while j>0 and a[j-1]>t
     b. move a[j-1] to the right
  4. Set a[j] = t

Java Code:

```java
insert(array a, int length, value) {
  int i = length - 1;
  while (i >= 0 && a[i] > value) {
    a[i + 1] = a[i];
    i = i - 1;
  }
  a[i + 1] := value;
}
insertionSort(array a, int length) {
  int i = 0;
  while (i < length) {
    insert(a, i, a[i]);
    i = i + 1;
  }
}
```

RECURSIVE SORT – QUICKSORT
Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array, we choose an element, called a pivot, move all smaller elements before the pivot, and move all greater elements after it. Then one recursively sorts the lesser and greater sublists. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice.

```c
void quicksort (int[] a, int lo, int hi)
{
    // lo is the lower index, hi is the upper index
    // of the region of array a that is to be sorted
    int i=lo, j=hi, h;
    int x=a[(lo+hi)/2];

    // partition
    do
    {
        while (a[i]<x) i++;  // move all smaller elements before the pivot
        while (a[j]>x) j--;  // move all greater elements after it
        if (i<=j)  // recursion
        {
            h=a[i];  // exchange the elements
            a[i]=a[j];  // between the pivot
            a[j]=h;
            i++;  // move the partition
            j--;
        }
    } while (i<=j);

    // recursion
    if (lo<j) quicksort(a, lo, j);
    if (i<hi) quicksort(a, i, hi);
}
```

### VALIDATION CHECKS ALGORITHMS

Keeping in mind the GIGO (Garbage In, Garbage Out) principle, we must ensure that the data which is being fed to a program or system is valid. If a user enters inappropriate or erroneous data, the software must handle this situation gracefully. The software should present a descriptive error message that suggests appropriate action, and must avoid responses such as program exit, data loss, or lockup. Ideally, the software checks user input as soon as it is entered, at the input field level. If there are any problems with an entry, such as out-of-range values or alphabetic characters when numeric values are expected, an error message informs the user what type of entry is required.

As a second choice, the software checks user input at the page level. If a problem entry is found, the user is pointed to the field needing correction. This approach is less convenient for the user, but is acceptable. If the software waits to check the user input until a final calculation, after several screens have been completed and many fields of data have been entered, the result may be frustration and confusion for the user. Error checking at the field level and the page level is much preferred.

### RANGE CHECKING

This type of check ensures that the data inputted in a system is within the acceptable range. Example if one is entering the date, the month has to be between 1 and 12. If we turn this to pseudo code:
Read n;
Repeat
    If (n >= 1) & (n <= 12)
    then
        print("Month has to be between 1 and 12");
        x = true;
    else
        month = n;
        x = false;
Until x = false;

DATA TYPE CHECKING

While writing a program one must make sure that the correct data types are used. Usually these types are caught at compilation time, like assigning an integer to a string array. However at runtime the programmer must take care to validate the input or giving correct instructions to the user to avoid entering incorrect data. Alternatively one can use exception handling (try/catch).

FILE OPERATIONS

There are different types of file organisations. They can be divided into two major groups, according to the devices, which are Serial Access and Direct Access.

Serial Access Method (e.g. magnetic tapes):

- Serial Access: files are read/stored one after the other
- Sequential Access: similar to serial but the records are ordered on a particular field

Direct Access Method (random – e.g. disks)

- Direct Access: no particular sequence or sorting of the data, however the machine has to keep track of where each file is located

Serial/Direct Access Combination:

- Indexed-Sequential: this type of access method makes use of sequential access when handling large file batches and uses direct access properties when rapid/frequent access to records is required.

STREAMS AND EXCEPTION HANDLING

Streams are the channels used by Java through which data flows, e.g. keyboard to memory/program, memory/program to screen, text file to/from memory. Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Exception handling is utilized to cater for errors during data input or file operations (try/catch). Java defines two types of streams.
BYTE Streams

- used in earlier version of Java
- handles bytes (reading/writing of binary data)
- defined using two class hierarchies
- InputStream (abstract class)
- OutputStream (abstract class)

CHARACTER Streams

- introduced later
- make use of Unicode
- handles characters
- more efficient
- International
- defined using two class hierarchies
- Reader - input (abstract class)
- Writer – output (abstract class)

We are used to write System.out (or .in). The System is a class of the java.lang package and has three predefined stream variables: System.in (default = keyboard), System.out (default = console), System.err (default = console).

Example: Reading data from keyboard

```java
import java.io.*;
class ReadBytes {
public static void main(String args[]) throws IOException {
    byte data[] = new byte[10];
    System.out.println("Enter some characters.");
    System.in.read(data);
    System.out.print("You entered: ");
    for(int i=0; i < data.length; i++)
        System.out.print((char) data[i]);
}
}
```

Here is a sample run:

Enter some characters.
Read Bytes (the user enters these)
You entered: Read Bytes

FILES AND STREAMS – TEXT FILES

The following example illustrates how to read from text files and output the content to console/screen. To try out this class one has to create a test text file using MS Notepad. You can save it as test.txt. Then refer to the following code:

```java
import java.io.*;
class ShowFile {
public static void main(String args[]) throws IOException {
    int i;
    FileInputStream fin;
    try {
        fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException exc) {
        System.out.println("File Not Found");
        return;
    } catch(ArrayIndexOutOfBoundsException exc) {
        System.out.println("Usage: ShowFile File");
        return;
    }
    try {
        while ((i = fin.read()) != -1) {
            System.out.print((char) i);
        }
    } finally {
        fin.close();
    }
}
```
After compiling the code, make sure that the text file is in the same directory as the class file and type the following (CLI):

C:\ java ShowFile test.txt

To demonstrate the ability to write to a file, refer to the following code which reads a file and writes a copy of this file. You can use the previous text file as the source.

```java
import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException {
        int i;
        FileInputStream fin;
        FileOutputStream fout;
        try {
            // open input file
            try {
                fin = new FileInputStream(args[0]);
            }
            catch (FileNotFoundException exc) {
                System.out.println("Input File Not Found");
                return;
            }
            // open output file
            try {
                fout = new FileOutputStream(args[1]);
            }
            catch (FileNotFoundException exc) {
                System.out.println("Error Opening Output File");
                return;
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println("Usage: CopyFile From To");
                return;
            }
            // Copy File
            try {
                do {
                    i = fin.read();
                    if (i != -1) fout.write(i);
                } while (i != -1); // if i = -1, it means EOF
                catch (IOException exc) {
                    System.out.println("File Error");
                }
                fin.close();
                fout.close();
            }
        }
    }
}
```
**Program which compares two files:

```java
import java.io.*;

class CompFiles {
    public static void main(String args[]) throws IOException {
        int i=0, j=0;
        FileInputStream f1;
        FileInputStream f2;
        try {
            // open first file
            try {
                f1 = new FileInputStream(args[0]);
            } catch (FileNotFoundException exc) {
                System.out.println(args[0] + " File Not Found");
                return;
            }
            // open second file
            try {
                f2 = new FileInputStream(args[1]);
            } catch (FileNotFoundException exc) {
                System.out.println(args[1] + " File Not Found");
                return;
            }
            // Compare files
            try {
                do {
                    i = f1.read();
                    j = f2.read();
                    if (i != j) break;
                } while (i != -1 && j != -1);
                catch (IOException exc) {
                    System.out.println("File Error");
                    if (i != j)
                        System.out.println("Files differ.");
                    else
                        System.out.println("Files are the same.");
                }
            } catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println("Usage: CompFile f1 f2");
                return;
            }
        } catch (ArrayIndexOutOfBoundsException exc) {
            System.out.println("Usage: CompFile f1 f2");
            return;
        }
    }
}
```

After compiling the above program you have to supply to text files to perform the comparison. Execution is as follows: `c:\ java CompFiles java1.txt java2.txt`

All the file operations listed so far are sequential. File operations can be also random (Random access). This time instead of Input/OutputStream two different interfaces are used DataInput/Output.
USING FILE WRITER/READER

FileWriter creates a Writer that can be used to write to a file. Its most commonly used constructors are shown here:

FileWriter(String fileName) throws IOException

FileWriter(String fileName, boolean append) throws IOException

Note:

- fileName is the full path name of a file.
- If append is true, then output is appended to the end of the file.
- If append is false, the file is overwritten
- Both throw an IOException on failure.

FileWriter is derived from OutputStreamWriter and Writer.

The following program is a simple key-to-disk utility that reads lines of text entered at the keyboard and writes them to a file called “test.txt.” Text is read until the user enters the word “stop.”

```java
import java.io.*;
class KtoD {
    public static void main(String args[]) throws IOException {
        String str;
        FileWriter fw;
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        try {
            fw = new FileWriter("test.txt"); // file creation
            System.out.println("Enter text ('stop' to quit).");
            do {
                System.out.print(":");
                str = br.readLine();
                if(str.compareTo("stop") == 0) break;
                str = str + "\n"; // add newline
                fw.write(str); // strings written to file
            } while(str.compareTo("stop") != 0);
            fw.close();
        } catch(IOException exc) {
            System.out.println("Cannot open file.");
            return;
        }
    }
}
```

The FileReader class creates a Reader that can be used to read the contents of a file. Its most commonly used constructor is shown here:

FileReader(String fileName) throws FileNotFoundException

fileName is the full path name of a file. It throws a FileNotFoundException if the file does not exist. FileReader is derived from InputStreamReader and Reader.
The following program reads content from file and displays them on screen (complementary to the previous example)

```java
import java.io.*;
class DtoS {
    public static void main(String args[]) throws Exception {
        FileReader fr = new FileReader("test.txt");
        BufferedReader br = new BufferedReader(fr);
        String s;
        while((s = br.readLine()) != null) {
            System.out.println(s);
        }
        fr.close();
    }
}
```

**FILES AND STREAMS – OBJECT FILES/STREAMS**

The process of reading or writing Objects is referred to as Object Serialization. The principle is the same as for byte or character; however the difference is that now we have objects which are converted to a sequence of bytes (serialization). The following example illustrates object serialization:

Class Student:

```java
import java.io.*;

public class Student implements Serializable {
    private String name;
    private int year;
    private double gpa;

    public Student() {}

    public Student (String nameIn, int yr, double gpaIn) {
        name = nameIn;
        year = yr;
        gpa = gpaIn;
    }

    public void writeToFile(FileOutputStream outStream) throws IOException {
        ObjectOutputStream ooStream = new ObjectOutputStream(outStream);
        ooStream.writeObject(this);
        ooStream.flush();
    } // writeToFile()

    public void readFromFile(FileInputStream inStream) throws IOException, ClassNotFoundException {
        ObjectInputStream oiStream = new ObjectInputStream(inStream);
        Student s = (Student)oiStream.readObject();
        this.name = s.name;
        this.year = s.year;
        this.gpa = s.gpa;
    } // readFromFile()

    public String toString() {
        return name + "\t" + year + "\t" + gpa;
    }
} // Student class
```
Class ObjectIO:

```java
import javax.swing.*; // Swing components
import java.awt.*;
import java.io.*;
import java.awt.event.*;

public class ObjectIO extends JFrame implements ActionListener{
    private JTextArea display = new JTextArea();
    private JButton read = new JButton("Read From File");
        write = new JButton("Write to File");
    private JTextField nameField = new JTextField(10);
    private JLabel prompt = new JLabel("Filename:",JLabel.RIGHT);
    private JPanel commands = new JPanel();

    public ObjectIO () {
        super("ObjectIO Demo"); // Set window title
        read.addActionListener(this);
        write.addActionListener(this);
        commands.setLayout(new GridLayout(2,2,1,1));
        commands.add(prompt); // Control panel
        commands.add(nameField);
        commands.add(read);
        commands.add(write);
        display.setLineWrap(true);
        this.getContentPane().setLayout(new BorderLayout()); // and control panel
       (commands); // Central
        this.getContentPane().add("Center", display);
    } // ObjectIO()

    public void actionPerformed(ActionEvent evt) {
        String fileName = nameField.getText();
        if (evt.getSource() == read)
            readRecords(fileName);
        else
            writeRecords(fileName);
    } // actionPerformed()

    private void readRecords(String fileName) {
        try {
            FileInputStream inStream = new FileInputStream(fileName); // Open a stream
            display.setText("Name\tYear\tGPA\n");
            try {
                while (true) { // Infinite loop
                    Student student = new Student(); // Create a student instance
                    student.readFromFile(inStream); // and have it read an object
                    display.append(student.toString() + "\n"); // and display it
                } catch (IOException e) { // Until IOException
                    inStream.close(); // Close the stream
                } catch (FileNotFoundException e) {
                    display.append("IOERROR: File NOT Found: " + fileName + "\n");
                } catch (IOException e) {
                    display.append("IOERROR: " + e.getMessage() + "\n");
                } catch (ClassNotFoundException e) {
                }
        }
```
private void writeRecords(String fileName) {
    try {
        FileOutputStream outStream = new FileOutputStream(fileName);// Open stream
        for (int k = 0; k < 5; k++) { // Generate 5 random objects
            String name = "name" + k;
            int year = (int)(2000 + Math.random() * 4);
            double gpa = Math.random() * 12;
            Student student = new Student(name, year, gpa);
            display.append("Output:", student.toString(), "\n");
            student.writeToFile(outStream);
        }
        outStream.close();
    } catch (IOException e) {
        display.append("IOERROR: " + e.getMessage() + "\n");
    }
} // writeRecords()

public static void main(String args[]) {
    ObjectIO io = new ObjectIO();
    io.setSize(400, 200);
    io.setVisible(true);
    io.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0) // Quit the application
        }
    });
} // main()

The aim of the above example is for demonstration purposes only and you are only required to understand what is going on rather than remember the entire code. The above class, ObjectIO uses swing components to draw a window which implements ActionListener and other GUI components which we did not discuss yet. However you are encouraged to try it out.
Errors are referred to bugs. The ‘bug’ term was initiated by Grace Hopper when she managed to diagnose and trace out a fault in a US Navy Mainframe. The fault was cause by a moth which got squashed in a relay switch thus preventing flow of current!

**PROGRAM ERROR TYPES**

- **Syntax**: These types of errors are also known as Compile time errors as they are caught up by the compiler. Syntax error occur when one mistypes a word or a command, forgets to capitalize or insert a curly-bracket or semicolon, e.g.
  - int num;
  - nums = 3; (num/nums)
  - System.out.println (System.out.println)
- **Logical**: this type of error occurs when all the syntax is correct, i.e. compiler returns no syntax errors, however the program does not execute as predicted. Such error could lead to infinite loops. E.g.
  - area = length + breadth; (length * breadth)
  - for (i = 0; i < 9; i++) (loop runs for nine times not ten)
- **Run-time**: also known as execution errors. These errors occur when an unexpected event occurs during program execution, e.g.
  - Trying to access a file which does not exist
  - Exceeding the size of an array
  - Division by zero error

**DEBUGGING UTILITIES**

Software suites and IDE’s (integrated development environment) offer the programmer facilities to debug, i.e. test and remove errors from programs. Typical debugging utilities include:

- **Trace facility**: allows the program to control the speed of program execution (stepping mode) in order to view how the contents stored inside the program’s variables are being manipulated (during execution)
- **Identifier evaluation**: these are like watches which can be used to monitor state or content of any variables used in a program
- **Breakpoint**: this feature allows the programmer to halt execution at a particular point in a programme. This is useful while trying to trace out an error in execution. You break program up to a point where one is sure there are no errors. At that point usually the programmers can also check the state of any variables or possible fault logs.