Department of Computer Science,
University of Ibadan

CSC 235
(Object Oriented Programming)

PROPERTIES OF DLC UI, IBADAN

S. O. Akinola (PhD)

CSC 235: OBJECT ORIENTED PROGRAMMING

Course Contents

Basic Types and Expressions; Assignment Statements; Loops and Conditionals (Simple and Nested); Handling Simple I/O; Objects and Classes; Methods with and without parameters; Inheritance; Constructor Methods (and the use of 'new'); Method Overloading; Method Overriding; Arrays and simple sorting; Basic File Handling; Try and Catch (Simple Exception Handling); Implementing Simple Graphical User Interfaces; Incorporating Applets in a Web page; Simple builtin Dynamic Structures - Vectors; Types vs. Classes; Scope of Variables; Code Layout and Documentation.

PROPERTIES OF DLC UI, IBADAN

General Introduction to the Course

CSC 235 (Object Oriented Programming) is a course meant to introduce students to the general principles of Object Oriented Programming (OOP). In this course, you shall be introduced to the meaning of OOP paradigm, the principles of OOP and general programming styles in OOP.

Java is introduced in order for you to understand the basic principles of Object Oriented programming in a more concise manner. Various language constructs including variables declaration and usages, data types, arrays, Objects and Classes are considered in this aspect of the course.

At the end of this course, you should be able to write simple Object Oriented programs in Java.

PROPERTIES OF DLC UI, IBADAN

1 *Essential Object-Oriented Concepts And Terminologies*

1.1 Introduction
This chapter introduces you to the essential object-oriented concepts and terminology. We start by explaining the concepts of "Object" and "Object-oriented programming". The differences between procedural-oriented and object-oriented programming paradigms are not left out.

1.2 Objectives
At the end of this lecture, you should be able to:
1. explain the different concepts of object-oriented programming;
2. distinguish between procedural-oriented and object-oriented programming paradigms

1.3 Pre-Test
1. What is a language?
2. How many programming languages have you ever heard of? How many of them have you written before.
3. Explain the problems you encountered in the language(s) if any.

1.4 Main Content

1.4.1 Programming Paradigms
Object-oriented programming is one of several programming paradigms. Other programming paradigms include the imperative programming paradigm (as exemplified by languages such as Pascal or C), the logic programming paradigm (Prolog), and the functional programming paradigm (exemplified by languages such as ML, Haskell or Lisp). Logic and functional languages are said to be declarative languages.

We use the word paradigm to mean "any example or model". This usage of the word was popularised by the science historian Thomas Kuhn. He used the term to describe a set of theories, standards and methods that together represent a way of organising knowledge—a way of viewing the world. Thus a programming paradigm is a way of conceptualising what it means to perform computation and how tasks to be carried out on a computer should be structured and organised.

We can distinguish between two types of programming languages: Imperative languages and declarative languages. Imperative knowledge describes "*how-to knowledge*" while declarative knowledge is "*what-is knowledge*". A program is "declarative" if it describes what something is like, rather than how to create it. This is a different approach from traditional imperative programming languages such as Fortran, and C, which require the programmer to specify an algorithm to be run.

In short, imperative programs make the algorithm explicit and leave the goal implicit, while declarative programs make the goal explicit and leave the algorithm implicit. Imperative languages require you to write down a step-by-step recipe specifying how something is to be done. For example to calculate the factorial function in an imperative language we would write something like:

```
public int factorial(int n) {
        int ans=1;
        for (int i = 2; i <= n; i++){
                ans = ans * i;
        }
        return ans;
  }
```

PROPERTIES OF DLC UI, IBADAN

Here, we give a procedure (a set of steps) that when followed will produce the answer.

1.4.2 What is Object-oriented Programming?

Object-oriented programming (**OOP**) is a programming paradigm using "objects", which are data structures consisting of data fields and methods together with their interactions, to design applications and computer programs.

It is an *approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand."*

Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access the data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.
One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify and maintain.

To perform object-oriented programming, one needs an *object-oriented programming language (OOPL)*. Java, C++ and Smalltalk are three of the more popular languages, and there are also object-oriented versions of Pascal and FORTRAN.

1.4.3 A Look at Procedure-Oriented Programming

Conventional programming, using high-level languages such as COBOL, FORTRAN and C, is commonly known as procedure-oriented programming (POP). Simple, POP programs may be one "long" list of statements (or commands). More complex programs will often group smaller sections of these statements into functions or subroutines, each of which might perform a particular task. With designs of this sort, it is common for some of the program's data to be 'global', i.e. accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects. In essence, ***while we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?***

In a multi-function program, many important data items are placed as *global* so that they may be accessed by all the functions. Each function may have its own *local data*. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

Some characteristics exhibited by procedure-oriented programming are:

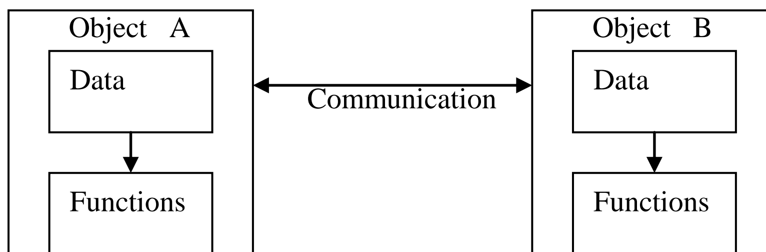
- Emphasis is on doing things (algorithms)
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.

PROPERTIES OF DLC UI, IBADAN

- Employs top-down approach in program design.

1.4.4 Object-Oriented Programming Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. ***OOP treats data as a critical element in the program development and does not allow it to flow freely around the system***. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects. The organization of data and functions in object-oriented programs is shown in Figure below. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.



Some of the striking features of object-oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

1.4.5 Basic Concepts of Object-Oriented Programming

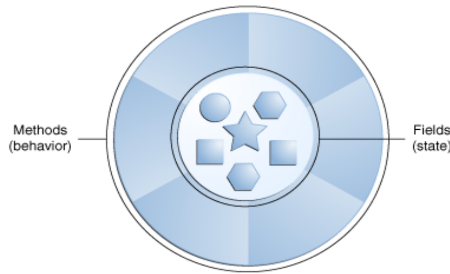
It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

1.4.5.1 Objects

An object is a bundle, a clump, a gathering together of items of information that belong together, and functions that work on those items of information. Software objects are modeled after real-world objects in that they too have state and behaviour. A software object maintains its state in one or more *variables*. A variable is an item of data named by an identifier. A software object implements its behaviour with *methods*. A method is a function (or subroutine or procedure) associated with an object.

Therefore, an object is a software bundle of variables and related methods.

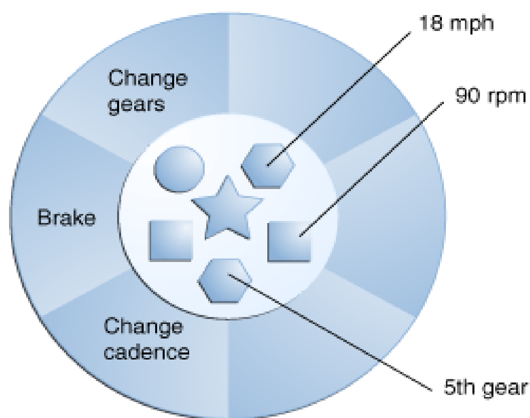
PROPERTIES OF DLC UI, IBADAN



A Software Object

Everything that the software object knows (state) and can do (behaviour) is expressed by the variables and the methods within that object.

Consider real life objects, dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching and wagging tail). Bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behaviour (braking, accelerating, slowing down, changing gears).



A bicycle modeled as a software object

A BankAccount object might gather up a customer number, account number, and current balance--these three pieces of information are required for all bank accounts. Many languages provide a way to group related information together into *structures* or *records* or whatever the language calls the feature. However, where an object differs from these is in including functions, or behavior, as well as information. Our BankAccount object will have Deposit( ), Withdraw( ), and GetBalance( ) functions, for example. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending ***messages*** to one another. For example, if 'customer' and 'account' are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. This information is passed along with the message as *Parameters.* Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects

1.4.5. 2 Classes

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint

PROPERTIES OF DLC UI, IBADAN

from which individual objects are created. **In essence*, "A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind".***

Objects vs. Classes: you probably noticed that all the illustrations of objects and classes look very similar. And indeed, the difference between them is often the source of some confusion. In the real world, it's obvious that classes are not themselves the objects they describe. A blueprint of a bicycle is not a bicycle. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it's also because the term "object" is sometimes used to refer to both classes and instances.

1.4.5.3 Data Abstraction And Encapsulation

The wrapping up of data [variables or state] and functions [methods] into a single unit (called class) is known as **encapsulation**. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions, which are wrapped in the class, can access it. These functions provide the interface between the object's data and the program. This insulation of the data from the direct access by the program is called **data hiding or information hiding.** Encapsulating related variables and methods [functions] into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

(1) **Modularity**: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system.
(2) **Information hiding:** an object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods/functions that can be changed at any time without affecting the other objects that depend on it. For instance, you don't need to understand the gear mechanism on your bike to use it.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called methods or member functions.

Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

1.4.5.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties of another class. It supports the concept of hierarchical classification. For example, the bird 'Robin' is a part of the class 'flying bird', which is again a part of the class ' bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

In **OOP**, the concept of inheritance provides the ideal of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only these features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

PROPERTIES OF DLC UI IBADAN

1.4.5.5 Polymorphism

Polymorphism is another important **OOP** concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition (+). For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by *concatenation*. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading.*

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

1.4.5.6 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run - time. It is associated with polymorphism and inheritance. A function call associated a polymorphic reference depends on the dynamic type of that reference.

1.4.5.7 Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts. Message passing involves specifying the name of the object, the name of the function /method and the information to be sent. E.g. employee.salary(name);

Bundling code into individual software objects provides a number of benefits, including:

- Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

The **promising areas for OOP application** include Real-time systems, Simulation and Modelling, Object-oriented Databases, Hypertext, Hypermedia, AI, Expert Systems, Neural Networks, Decision support systems CAM and CAD systems.

PROPERTIES OF DLC UI, IBADAN

Summary

This chapter briefly introduced you to the basic principles of Object Oriented Programming paradigms. The two main programming paradigms – Imperative and Declarative were discussed. You have also being introduced to the basic concepts of OOP.

In summary, the five basic concepts of object-oriented design are the implementation level features that are built into the programming language. These features are often referred to by these common names:

- **Object/Class**: A tight coupling or association of data structures with the methods or functions that act on the data. This is called a *class*, or *object* (an object is created based on a class). Each object serves a separate function. It is defined by its properties, what it is and what it can do. An object can be part of a class, which is a set of objects that are similar.
- **Information hiding**: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as *private* or *protected* to the owning *class*.
- **Inheritance**: The ability for a *class* to extend or override functionality of another *class*. The so-called *subclass* has a whole section that is derived (inherited) from the *superclass* and then it has its own set of functions and data.
- **Interface**: The ability to defer the implementation of a *method*. The ability to define the *functions* or *methods* signatures without implementing them.
- **Polymorphism**: The ability to replace an *object* with its *subobjects*. The ability of an *object-variable* to contain, not only that *object*, but also all of its *subobjects*.

Post-Test

1. Attempt to mention some common programming languages you are familiar with and put them into suitable language categories.
2. Differentiate between imperative and declarative programming languages.
3. Explain the concepts of object, class, inheritance and polymorphism.
4. What are the major advantages of OOP paradigm?

PROPERTIES OF D

2 *Java Programming Basics*

2.1 Introduction

In this chapter, the basic principles of writing and executing program codes in Java language are discussed. We begin introducing you to writing codes in the language

Java is an Object Oriented programming language with a relatively simple grammar. Java omits rarely used, poorly understood, confusing features of C++ such as header files, pointer arithmetic, structures, unions, operator overloading, and templates. Java also adds new features like automatic garbage collection. All methods, fields and constructors are local to classes—that is, there is no global data. Java supports static methods and fields, exception handling, inheritance, and control structures such as while loops, for loops and if/else statements.

2.2 Objective

At the end of this lecture, you should be able to:

1. write simple java programs;
2. understand the different variable types, operators and expressions in Java.
3. understand how to format outputs from your program

2.3 Pre-Test

1. In which language(s) have you programmed before?
2. Write a simple program to compute the sum of any two numbers.

2.4 Main Content

2.4.1 The Java Language

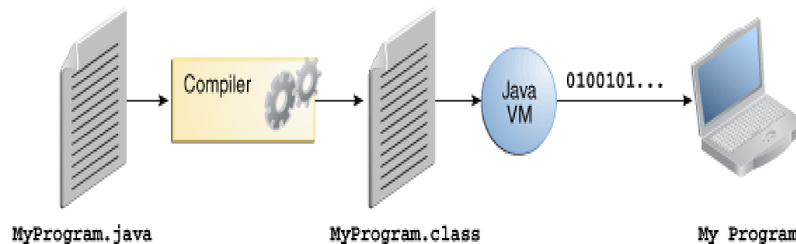
The following are the fundamental features of Java programming language:

1. **Simplicity:** the designers of the language were trying to develop a language that a programmer could learn quickly. They eliminated some constructs in C and C++ that are complex such as pointers.
2. **Object Oriented:** Everything is an object in java. The focus is on the data and the methods that operate on the data in the application. Java does not concentrate on procedures only.
3. **Platform independent:** Java has the ability to move from one computer to the other or from one operating system to another without any difficulty.
4. **Robust:** Java is strictly a strongly-typed language. It requires explicit declaration. Java has exception handling features, the programmer does not need to worry about memory allocation, and it does not have pointer and pointer arithmetic.
5. **Security:** Java is totally secured. It provides a controlled environment for the execution of the program. It never assumes that the code is safe for execution. It provides several layers of security control.
6. **Distributed:** Java can be used to develop applications that are portable across multiple platforms, operating systems and graphical user interfaces (GUI). Java is designed to support network applications.
7. **Multi-threaded:** Java programs use a process called multithreading to perform many tasks simultaneously.

In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class

PROPERTIES OF DLC UI, IBADAN

file does not contain code that is native to your processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.



Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS. Some virtual machines, such as the Java HotSpot virtual machine, perform additional steps at runtime to give your application a performance boost. This includes various tasks such as finding performance bottlenecks and recompiling (to native code) frequently used sections of code.

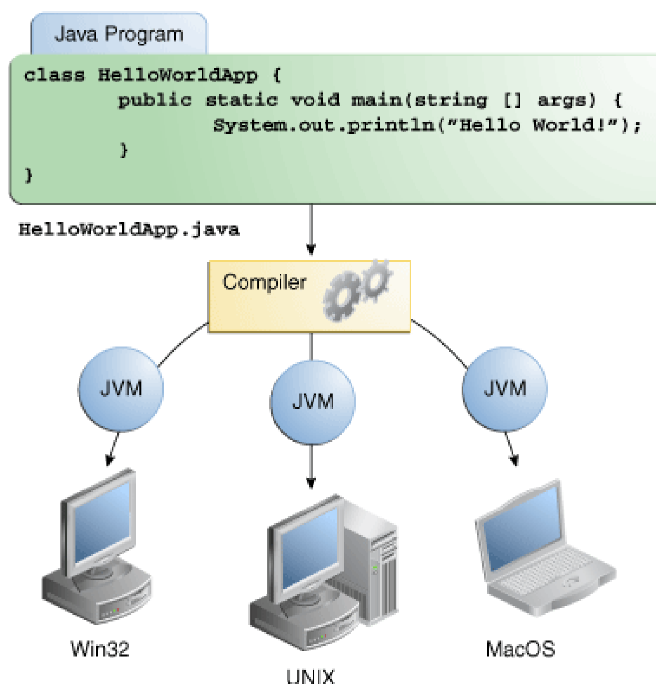
2.4.1.1 The Java Platform

A *platform* is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it's a software - only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

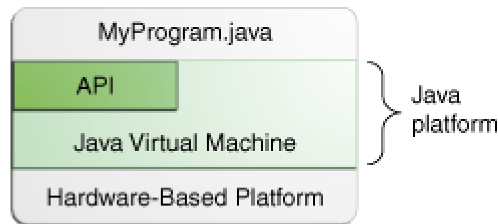
- The *Java Virtual Machine*
- The *Java Application Programming Interface* (API)

You've already been introduced to the Java Virtual Machine; it's the base for the Java platform and is ported onto various hardware-based platforms. The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.



Through the Java VM, the same application is capable of running on multiple platforms.

PROPERTY OF DLC UI IBADAN



The API and Java Virtual Machine insulate the program from the underlying hardware.

As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java platform.

2.4.2 Creating and Running a Java Program.

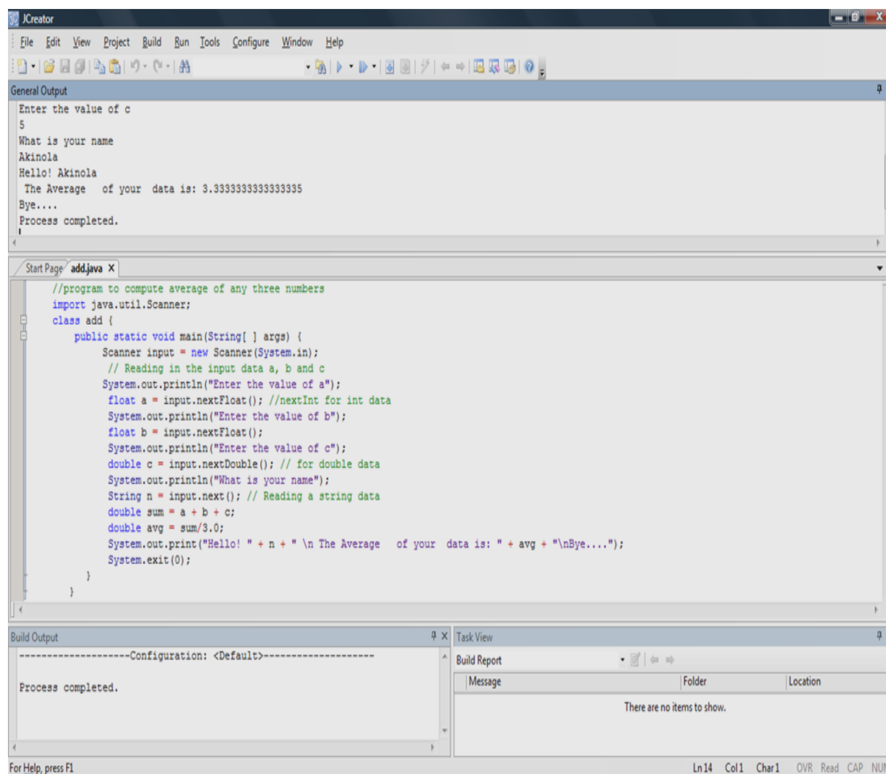
Nowadays, we have some integrated packages for editing and running java programs. For example, we have Java Net Beans, JCreator, JBuilder, Oracle JDeveloper, and many more. They are windows-like, just like Visual C++. Some of them contain both the editor and compiler together for editing and running java applications (e.g. Net Beans). Many of them are downloadable free from the Internet. However, the use of JCreator is recommended. First, the Java toolkit is downloaded and installed on the computer. You can download Java 1.6 and above free from the Internet. Just type Java 1.6, 1.7 etc on Google. JCreator is also free on the Net. Download the JCreator Pro and install it. Note that you need to link the JCreator with the Java toolkit after installation. This would be asked from you by the installer.

Follow the steps below to run programs with JCreator Pro.

1. Open the JCreator via the start menu or double click its icon on the desktop.
2. Click on File, New, then File.
3. Click Java Classes, then Main Class in the File wizard window provided.
4. Type the name of your file, e.g. demo.java, in the name box of file path box. Then click on finish. **Note that the name you will give to your class in the program must be the name given to your file, with the extension .java. for instance, the header of your program must read class demo, if you are saving your file with demo.java**
5. Key in your code in the code editor window, just below the public static void main( ) , where it is written // TODO code application logic here
6. Save the file by pressing Ctrl + S
7. To compile, click Build then Build File.
8. To execute the program, click Run, then Run File

JCreator Pro interface looks like the one below:

PROPERTIES OF DLC UI, IBADAN



2.4.3 Writing your first program

A simple Java code looks like this:

```
public class Welcome  {
  public static void main(String[ ] args)  {
        System.out.println(“Welcome to Java world”) ;
  }  //End main method
}  // End class Welcome
```

Note: Java is case–sensitive. Capital letters must be capitals e.g. String, System.

Let’s try to analyze the above simple java program.

(a) The first line declares a class named Welcome.

```
public class Welcome  {
```

This line could be regarded as the header of the program. Java sees a program as a class and the name of the class must be the same as the name of the file, i.e., the name you use to save the file. In this case we save the file as Welcome.java. Any nonempty string of letters and digits can be used for the class name as long as it begins with a letter and contains no blanks.

(b) The second line begins with the left brace character; just like BEGIN in Pascal. There must be a corresponding right brace at the last line of the program, i.e. representing END as in Pascal. The two braces form the program block, which encloses the program’s body.

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

PROPERTIES OF DLC © IBADAN

Every class must have a method or function that will be used to manipulate the data in the class. the default name given to that method is main. The method main has some descriptors associated with it – public, static, and void discussed below. A method or function could have zero or more list of arguments, enclosed in open and close brackets after the function's name. The default argument for method main is args, which is an array of strings.

- public means that the contents of the following block (the function/method) are accessible from all other classes.
- static means that the method being defined applies to the class itself rather than to objects of the class
- void means that the method being defined has no return value.
- main means this is the name of the method being defined, just as Welcome is the name of the class being defined. The parenthesized string following main forms the parameter list for the main method, which are local variables used to transmit information to the method from the outside world; (String[ ] args). It states that this method has one parameter, its name is args and it is an array of string objects.

(c) The third line contains the single executable statement.

```
System.out.println("Welcome to Java world") ;
```

The message put in quotes would be printed out as they are written. The word println is the name of the method that tells the system how to do the printing, which means, after the message is printed, the cursor should move to the next line. Note the parenthesis and the semi colon usage. **The semi colon is a terminator for each executable line.**

(d) The two closing braces, } mark the end of the program. The first closes the main method and the other closes the class.

2.4.4 print( ) and println( ) methods

Both print( ) and println( ) are standard output functions that print data to the monitor screen. The statement:

```
System.out. print (a, b, c)
```

will print the values of the data items a, b and c on a single line and the cursor will remain at the end of the printing.

However, if we had used System.out.println(a, b, c), the values of the data items would also be printed on a single line but after all the printings, the cursor will move to the next line for other printing commands. The two work like WRITE( ) and WRITELN( ) in PASCAL programming language. For example, given the following data and the subsequent code segment:

```
a = 2;
b = 4;
c = 6;
d= 8;

System.out.print(a, b);    // Line 1
System.out.print(c);    // Line 2
System.out.println( );    // Line 3
System.out.println(d);    // Line 4
```

The code will produce the following outputs:

PROPERTIES OF DLC UI, IBADAN

For Line 1, 2 and 4 will be printed on a single line and the blinking cursor will remain at the end of that line, waiting for another printing action. In Line 2, 6 will be printed after 4 on the same line with the previous printing. In Line 3, the cursor moves to next line without printing any value, since no data was given. In Line 4, 8 would be printed in the second line where the cursor was before in Line 3 and after the printing, it moves to next line. Println( ) is a post-active function. The final output will look like below:

```
2  4  6
8
```

2.4.5 Inserting Comments into Your Program

Comments are very good in programs. They enhances easy comprehension/readability of the codes, section by section and the are veritable tool for future program maintenance. Anybody can pick the code in the future and with the help of the comment lines, modify, upgrade or correct the program for some errors. Java embraces both C and C++ styles of comments.

The C comment style is a multi-line style. It is used when we have several lines of comments to be inserted into our programs. Take for example,

```
/* Program written by …..
     Matric No….   Version 1.0     Date  ……

*/
```

One major problem of the C comment style is that we must ensure that the closing sysmbol (*/) is inserted at the end of the entire comment lines; else the compiler will assume that all other lines below the opening symbol (/*) are all comments!

The C++ Style comment could be used as an in-line comment, inserted at the end of an executable statement like:

```
X = X – 4  // subtracting 4 from X
```

Or as a free-standing comment like:

```
// Program written by ……..
```

But, the comment style is only meant for one line. If there is need to extend comments to another line, we have to put another comment symbol (//) against that line.

Note: Adding comments to your programs is called documenting your code and comments are normally ignored during compilation. Comments promote readability, understand-ability and maintainability of programs.

2.4.6 Inputting Data into Java Programs

2.4.6.1 Using the Standard Input / Output (Keyboard and Monitor) on Command Line

This can be achieved by the use of Scanner facility provided by Java. The following example code computes the average of any three numbers. Note the reading of each of the data types – float, int, double and string in the code.

PROPERTIES OF DLC UI, IBADAN

```
1.  //program to compute average of any three numbers
2.  import java.util.Scanner;
3.  class add {
4.     public static void main(String[ ] args) {
5.         Scanner input = new Scanner(System.in);
6.          // Reading in the input data a, b and c
7.         System.out.println("Enter the value of a");
8.          float a = input.nextFloat( ); //nextInt for int data
9.          System.out.println("Enter the value of b");
10.         float b = input.nextFloat();
11.         System.out.println("Enter the value of c");
12.         double c = input.nextDouble(); // for double data
13.         System.out.println("What is your name");
14.         String n = input.next(); // Reading a string data
15.         double sum = a + b + c;
16.         double avg = sum/3.0;
17.         System.out.print("Hello! " + n + " \n The Average   of your
18.          data is: " + avg + "\nBye....");
19.         System.exit(0);
20.     }
21.  }
```

Sample Output

```
--------------------Configuration: <Default>--------------------
Enter the value of a
2
Enter the value of b
3
Enter the value of c
5
What is your name
Akinola
Hello! Akinola
 The Average   of your  data is: 3.3333333333333335
Bye....
Process completed.
```

Note:

1. The Scanner has to be imported into your program, similar to #include in C or C++ language. This is done in Line 2.
2. In line 5, an object of the Scanner class (input) is created. The variable/object input is user-defined. You can give any other meaningful identifier name for this object. This will serve as the anchor for receiving data from the keyboard buffer as you are entering data via the keyboard.
3. Each data type has its own format for receiving it in the program. For int data, we use input.nextInt( ), for long data, use input.Long( ), for float use input.nextFloat( ), for double, use input.nextDouble( ) and for string data, use input.next( ). The input.next( ) is only for one string at a time.

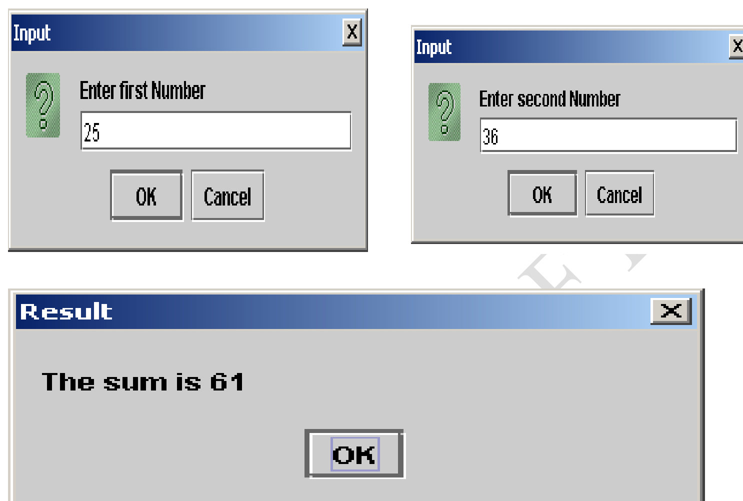
Using the Swing Facility

As another example, the program below illustrates the use of swings facility in java, a form of windows-based programming. Key in and run this program and report your observations:

PROPERTIES OF DLC UI, IBADAN

```
1.   import javax.swing.*;
2.   // program to add two numbers together
3.   public class Addition {
4.           public static void main(String args[ ])  {
5.                   // Declaring your variables …
6.                   String firstNumber, secondNumber;
7.                   int number1, number2, sum;
8.                   firstNumber = JOptionPane.showInputDialog("Enter  first
9.                                   Number");
10.                  secondNumber = JOptionPane.showInputDialog("Enter
11.                                  second Number" );
12.                  number1= Integer.parseInt(firstNumber);
13.                  number2= Integer.parseInt(secondNumber);
14.                  sum =  number1 +  number2;
15.                  JOptionPane.showMessageDialog(null,"The sum is"+
16.                     sum, "Result", JOptionPane.PLAIN_MESSAGE);
17.                  System.exit(0);
18.          } //end main method
19.   }  //end class Addition
```

Output:



Explanations:

(i) The JOptionPane is a subclass of the swing class and has some methods or functions associated with it. One of which is the showInputDialog used above. The purpose of this method is to draw an input textbox, in which the user will type in his / her data. What is typed in double quote as an argument in the method will serve as a prompt for the user to know what he is to do with the textbox when it comes onto the screen. However, any data captured by the showInputDialog( ) method is a string, even if you had entered a number! Any numeric data entered into the textbox has to be converted (parsed) to the proper numeric type. The next explanation gives this detail.

(ii) Lines 12 - 13

```
number1= Integer.parseInt(firstNumber);
number2= Integer.parseInt(secondNumber);
```

are the lines of code that converts (*parse*) the numeric data captured by the showInputDialog( ) into numeric data, either float, double or int or long. However, both the lines 6-11 for declarations and showInputDialog( ) and the parsing (Lines 12 – 13) can be combined into only one line to minimize space and time. Thus the lines:

PROPERTY OF UI, IBADAN

```
String firstNumber;
Int number 1;
firstNumber = JOptionPane.showInputDialog("Enter first  Number");
number1= Integer.parseInt(firstNumber);
```

can be written as follows:

```
int number1 =  Integer.parseInt(JOptionPane.showInputDialog("Enter first  Number"));
```

As java is highly case sensitive, note that int against number1 to declare it is with small i, while all the I's on right hand side of the assignment statement are all capitals. Other variations are

```
float a = Float.parseFloat(JOptionPane.showInputDialog( " "  ));
double b = Double.parseDouble(JOptionPane.showInputDialog( " " ));
long c = Long.parseLong(JOptionPane.showInputDialog( " " ));
```

(iii) The showMessageDialog method has four parameters to be passed into it: null, user's output, title of the message dialog and the type of icon to be attached to the message box whether error, information or any other. Each of these parameters is separated by commas. Check this with the example above. We are particular about the last three. All the output that the user wants the message box to print out including prompting messages are specified in the user's output parameter. The prompting messages are doubly quoted along with the variable values to be printed out. The title given to the message box as the third parameter must be typed in double quotes and should be relevant to the output to be brought out. The last parameter is the icon to be attached to the message box. This time, we used the JOptionPane.PLAIN_MESSAGE, meaning that no icon will show. We can also use JOptionPane.INFORMATION_MESSAGE or JOptionPane.ERROR_MESSAGE. Note the use of the underscore.

Example Codes:

E1: This code requests for your age now and the current year. It then computes the year you were born and reports back to you when you were born.

```
1.  //Computing your Year of Birth
2.  import java.util.Scanner;
3.  public class yearOfBirth {
4.       public static void main(String[ ] args)   {
5.              Scanner input = new Scanner(System.in);
6.              System.out.println("Enter your age: ");
7.              int age = input.nextInt();
8.              System.out.println("Enter this year's value e.g. 2013: ");
9.              int currentYr  = input.nextInt();
10.             int year = currentYr - age;
11.             System.out.println("You are " +  age + " years old now");
12.             System.out.println(" So you were probably born in  " +
13.                   year);
14.       }
15.  }
```

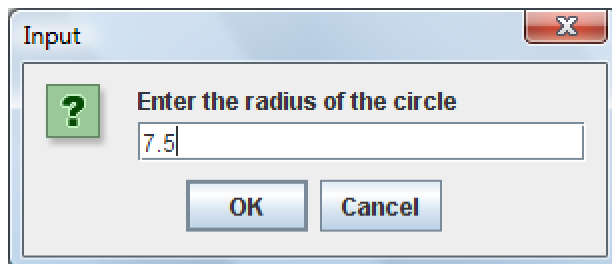
Output:

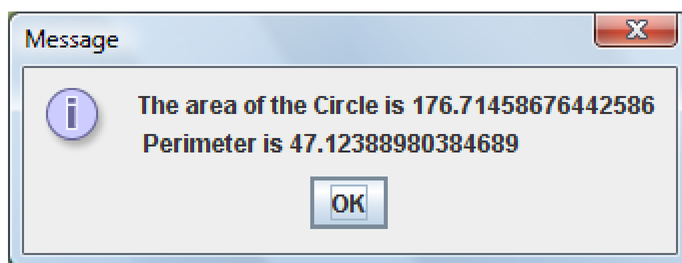
```
--------------------Configuration: <Default>--------------------
Enter your age:
45
Enter this year's value e.g. 2013:
2013
You are 45 years old now
 So you were probably born in  1968
```

PROPERTIES OF DLC UI, IBADAN

E2: Computing the Area of a circle

```
1.   //Computing the area and perimeter of a circle
2.   import javax.swing.JOptionPane;
3.   public class add {
4.          public static void main(String[ ] args)   {
5.              float r  = Float.parseFloat(JOptionPane.showInputDialog("Enter
6.                          the radius of the circle"));
7.              double area = Math.PI * r * r;
8.              double peri = 2.0 * Math.PI * r;
9.             JOptionPane.showMessageDialog(null, "The area of the Circle is
10.                 " + area + "\n Perimeter is "+ peri);
11.        } // end main
12.   } // end class
```





Summary
This chapter briefly introduced you to Java programming basics. You have been introduced to

- Java Virtual Machine (JVM)
- Java Toolkit
- How to edit your code in Java
- How Java execute a program
- The different ways of running your programs in Java

Post-Test

1. Write a program to compute the area and perimeter of a rectangular object
2. Write a program to calculate simple interest on a sum of money invested for some number of years at certain rate percent.

3 *Java Variables and Objects*

3.1 Introduction

A variable, as you may already know, is simply something that stores a value. Imagine that each variable is a box. Each box is labelled with a name and a category; its category is what type of item it stores, and its name is what that specific variable is called. Boxes can store many things, but only if they fit in that box. This is the same with variables. They all have **types, names,** and **sizes**, which govern what can and can't be stored inside them. An object on the other hand, is an instance of a class and may contain many variables You shall be introduced to variables and objects in this chapter.

3.2 Objective

At the end of this lecture, you should be able to:
1. identify the various variable types in Java language;
2. understand how variables are formed
3. understand the scope of a variable
4. identify the differences between variables and objects

3.3 Pre-Test
1. How many types of variables are you familiar with?
2. Give four of them

3.4 Main Content

3.4.1 What are Java Variables and Objects

These hold data in Java. A variable has a type and hold a single value. An object is an instance of a class and may contain many variables, the composite of whose values is called the "state" of the object. Whereas every variable has a unique name, on being declared, objects have references instead of names, and they need not be unique. An object is created by using the "**new**" operator to invoke a "**Constructor**" and it dies when it has no references. E.g. in the Circle program above, r and area are the 2 variables while reader, input, text, x and System.out are the five objects in the program. These objects are instances of the classes inputStreamReader, BufferedReader, String, Double and PrintStream respectively. In the Java programming language, the following must hold true for a simple name:

1. It must be a legal identifier. An identifier is an unlimited series of Unicode characters that begins with a letter.
2. It must not be a keyboard, a boolean literal (true or false), or the reserved word null.
3. It must be unique within its scope. A variable may have the same name as a variable whose declaration appears in a different scope. In some situations, a variable may share the same name as another variable if it is declared within a nested block of code. (We will cover this in the next section, Scope.)
4. It must not be a java reserved identifier such as swing, String, int, short etc. Reserved words are pre-defined in java, and so, cannot be re-defined.

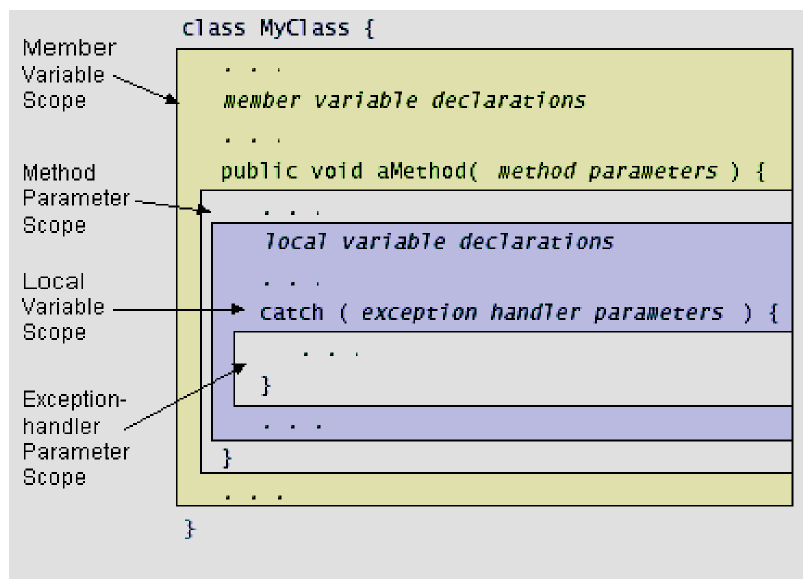
By Convention: Variable names begin with a lowercase letter, and class names begin with an uppercase letter. If a variable name consists of more than one word, the words are joined together, and each word after the first begins with an uppercase letter, like this: isVisible. The underscore character (_) is acceptable anywhere in a name, but by convention is used only to separate words in constants (because constants are all caps by convention and thus cannot be case-delimited).

PROPERTIES OF DLC UI, IBADAN

3.4.2 *Scope of Variables*

A variable's scope is the region of a program within which the variable can be referred to by its simple name. Secondarily, scope also determines when the system creates and destroys memory for the variable. Scope is distinct from visibility, which applies only to member variables and determines whether the variable can be used from outside of the class within which it is declared. Visibility is set with an access modifier.
The location of the variable declaration within your program establishes its scope and places it into one of these four categories:

- *member variable*
- *local variable*
- method parameter
- exception-handler parameter



A **member variable** (or **global variable**) is a member of a class or an object. It is declared within a class but outside of any method or constructor. A member variable's scope is the entire declaration of the class. However, the declaration of a member needs to appear before it is used when the use is in a member initialization expression.

You declare **local variables** within a block of code. In general, the scope of a local variable extends from its declaration to the end of the code block in which it was declared.

Parameters are formal arguments to methods or constructors and are used to pass values into methods and constructors. The scope of a parameter is the entire method or constructor for which it is a parameter.

Exception-handler parameters are similar to parameters but are arguments to an exception handler rather than to a method or a constructor. The scope of an exception-handler parameter is the code block between { and } that follow a catch statement. We shall deal with this topic later.

Consider the following code sample:

```
if (...) {
   int i = 17;
   ...
```

```
}
System.out.println("The value of i = " + i); // error
```

The final line won't compile because the local variable i is out of scope. The scope of i is the block of code between the { and }. The i variable does not exist anymore after the closing }. Either the variable declaration needs to be moved outside of the if statement block, or the println method call needs to be moved into the if statement block.

3.4.3 Variable Initialization

Local variables and member variables can be initialized with an assignment statement when they're declared. The data type of the variable must match the data type of the value assigned to it. Parameters and exception-handler parameters cannot be initialized in this way. The value for a parameter is set by the caller.

3.4.4 Final Variables / Java Constants

You can declare a variable in any scope to be *final.* The value of a final variable cannot change after it has been initialized. Such variables are similar to constants in other programming languages. To declare a final variable, use the final keyword in the variable declaration before the type:

```
final int aFinalVar = 0;
```

The previous statement declares a final variable and initializes it, all at once. Subsequent attempts to assign a value to aFinalVar result in a compiler error. You may, if necessary, defer initialization of a final *local* variable. Simply declare the local variable and initialize it later, like this:

```
final int blankfinal;
        . . .
    blankfinal = 0;
```

A final local variable that has been declared but not yet initialized is called a *blank final*. Again, once a final local variable has been initialized, it cannot be set, and any later attempt to assign a value to blankfinal is an error.

3.4.5 Data Types

Every variable must have a data type. A variable's data type determines the values that the variable can contain and the operations that can be performed on it. For example, declaring a variable to be int var means that var is an integer data type. Integers can contain only integral (whole numbers) values (both positive and negative) and we can perform arithmetic operations, such as addition, on integer variables.

Java programming has two categories of data types: *primitive* and *reference*. A variable of primitive type contains a single value of the appropriate size and format for its type: a number, a character, or a Boolean value. The table below lists all of the primitive data types along with their sizes and formats:

Keyword	Description	Size/Format
(Integers: numbers without decimal places, eg. 45, 456)		

PROPERTIES OF DLC UI, IBADAN

Byte	Byte-length integer	8-bit two's complement
Short	Short integer	16-bit two's complement
Int	Integer	32-bit two's complement
Long	Long integer	64-bit two's complement
(Real numbers: numbers with decimal places, e.g. 3.21)		
Float	Single-precision floating point	32-bit IEEE 754
Double	Double-precision floating point	64-bit IEEE 754
(other types)		
Char	A single character	16-bit Unicode character
Boolean	A boolean value (true or false)	true or false

PROPERTIES OF DLC UI, IBADAN

NB: In other languages, the format and size of primitive data types may depend on the platform on which a program is running. In contrast, the java programming language specifies the size and format of its primitive data types. Hence, we don't have to worry about system-dependencies.

We can put a literal primitive value directly in our codes. For example, if we need to assign the literal integer value 4 to an integer variable we can write this:

```
int anInt = 4;
```

int are integers in the range of 1 to a few thousands. However, if a variable is going to run into millions, we'd better declare it as long. For instance, when we are writing a factorial program, the factorial of big numbers like 20 may run into large values. short and byte are usually used in systems programming dealing with registers and memory addresses.

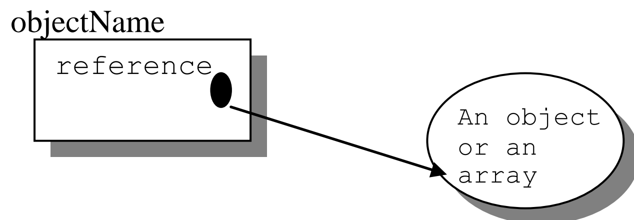
float is designed for real data whose number of decimal places may run not more than 5 places, e.g. 245.234. But double is used if we have a recurring decimal running up to say 10 decimals places like pi (Π). For instance, Java insists that whenever we are carrying out division, the variable to assign the result to must be declared as double, so as to avoid loss of precision.

Other examples of literal values are:

178	int
456L	Long
34.876	double
23.786D	double

56.89F	float
‘s’	char
true	Boolean

Arrays, classes and interfaces are **reference data types.** The value of a reference type variable, in contrast to that of primitive type, is a reference to (an address of) the value or set of values represented by the variable. A reference is called a pointer, or a memory address in other languages. The java programming does not support the explicit use of addresses like other languages do. We use the variable’s name instead.



Summary

- In this chapter you were introduced to the different types of variables that are suppoted by Java. We have 8 primitive types – int, short, byte, long, float, double, char and Boolean. Objects contain different variables.

Post-Test

1. Assuming you want to compute the standard deviation of some numbers, identify the input and out variables you will need and their types.
2. Write a program to compute the standard deviation of any 4 numbers

PROPERTIES OF DLC IBADAN

4 *Java Operators*

4.1 Introduction

An operator performs a function on one, two, or three operands. We have different types of operators in Java. Some are for arithmetic operations while others are for comparisons. In this chapter, you shall be introduced to the different operators supported by Java language.

4.2 Objective

At the end of this lecture, you should be able to:
3. identify the various operators in Java language;
4. understand how the operators are used

4.3 Pre-Test
1. How many types of operators are you familiar with?
2. Give four of them

4.4 Main Content

4.4.1 Java Operators

An operator performs a function on one, two, or three operands. An operator that requires one operand (**Op**) is called a **unary operator**. For example, ++ is a unary operator that increments the value of its operand by 1. An operator that requires two operands is a **binary operator**. For example, = is a binary operator that assigns the value from its right hand operand to its left-hand operand. And finally, a **ternary operator** is one that requires three operands. The java programming language has one ternary operator, ?:, which is a short-hand *if – else* statement. In addition to performing the operation, an operator returns a value. The return value and its type depend on the operator and the type of its operand. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and subtraction, return numbers – the result of the arithmetic operation. The data type returned by an arithmetic operator depends on the type of its operands: if we add two integers, we get an integer back. An operation is said to *evaluate to* its result. Below are the java-supported operators.

4.4.2 Arithmetic Operators

These operators are used in arithmetic computations in java programming environment. These are:

Operator	Use	Description
++	Op++ e.g i++;	Increments op by 1; evaluates to the value of op before it was incremented
++	++op e.g ++i;	Increments op by 1; evaluates to the value of op after it was incremented
--	Op— e.g i--;	Decrements op by 1; evaluates to the value of op before it was decremented
--	--Op	Decrements op by 1; evaluates to the

PROPERTIES OF DLC UI, IBADAN

	e.g –i;	value of op after it was decremented
+	a + b	Add a and b together
-	a – b	Subtract b from a
/	a / b	Divides a into b
*	a * b	Product of a and b
%	a % b	Remainder when a is divided by b

What happens when we have combinations of different data types in one single expression? It needs be noted that when an integer and a floating-point number are used as operands to a single arithmetic operation, the result is floating point. The integer is implicitly converted to a floating-point number before the operation takes place. This is called data coercion. For instance, if we have the following statement:

```
int y, x;
float b;
double a, c;
a = c + b + y + x;
```

The next table summarizes how java handles these situations. The data type returned by the arithmetic operators is based on the data type of the operands. The necessary conversions take place before the operation is performed.

Data Type of Result	Data Type of Operands
long	Neither operand is a float or a double (integer arithmetic); at least one operand is a long.
int	Neither operand is a float or a double (integer arithmetic); neither operand is a long.
double	At least one operand is a double.
float	At least one operand is a float; neither operand is a double.

In addition to the binary forms of + and -, each of these operators has unary versions that perform the following operations:

Operator	**Use**	**Description**
+	+op	Promotes op to int if it is a byte, short or a char.
-	-op	Arithmetically negates op.

PROPERTIES OF DLC UI, IBADAN

4.4.3 Assignment Operators

We use the basic assignment operator, =, to assign one value to another. In ordinary arithmetic, we are permitted to write a + b = c – d, but this is not allowed in programming. The expression at the left hand side (a + b) is regarded as a memory location to which we are assigning the result computed at the right hand side to. Java also provides several short-cut assignment operators that allow us to perform arithmetic, shift, or bitwise operation and an assignment operation, all with one operator. The table below gives the major assignment operators in java.

Oper-ator	**Use**	**Equivalent to**	**Example**
+=	Op1 += op2	op1 = op1 + op2	a = a + b ≡ a += b
-=	Op1 -= op2	op1 = op1 - op2	a = a + b ≡ a += b
*=	Op1 *= op2	op1 = op1 * op2	a = a * b ≡ a *= b
/=	Op1 /= op2	op1 = op1 / op2	a = a / b ≡ a /= b
%=	Op1 %= op2	op1 = op1 % op2	a = a % b ≡ a %= b
&=	Op1 &= op2	op1 = op1 & op2	**We shall study these operators in the latter sections**
|=	Op1 |= op2	op1 = op1 | op2	
^=	Op1 ^= op2	op1 = op1 ^ op2	
<<=	Op1 <<= op2	op1 = op1 << op2	
>>=	Op1 >>= op2	op1 = op1 >> op2	
>>>=	Op1 >>>= op2	op1 = op1 >>> op2	

Examples: Arithmetic and Assignment Operators

```
A = x;            // Assignment operator
n += 22;          // n = n + 22
++n;              // Increment Operator i.e. n = n + 1
n / a             // Quotient (Division) operator
int y = n % b ;   // Remainder Operator, divides n by b and
                  //assign the remainder to y
```

Note that +=, ++ and others are to be written together without any space in between them.

4.4.4 Relational and Conditional Operators:

A **relational operator** compares two values and determines the relationship between them. For example, != returns true if the two operands are unequal.

The table overleaf gives the examples of relational operators.

PROPERTIES OF DLC UI, IBADAN

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2 e.g. if (a > b)
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2 e.g. if (a < b)
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal e.g. if (a == b)
!=	op1 != op2	op1 and op2 are not equal

Relational operators often are used with **conditional / logical operators** to construct more complex decision-making expressions.

Java supports six conditional/logical operators – five binary and one unary – as shown in the following table:

<table>
<tr><th>Operator</th><th>Use</th><th>Returns true if</th></tr>
<tr><td>&&</td><td>op1 && op2</td><td>op1 AND op2 are both true, conditionally evaluates op2. E.g. if ((a < b) && (c > a)</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>either op1 OR op2 is true, conditionally evaluates op2. E.g if ((a < b) || (c > a)</td></tr>
<tr><td>!</td><td>! op</td><td>op is false . E.g. if !(a < b)</td></tr>
<tr><td>&</td><td>op1 & op2</td><td>op1 and op2 are both true, always evaluates op1 and op2</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>either op1 or op2 is true, always evaluates op1 and op2</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>if op1 and op2 are different - that is, if one or the other of the operands is true but not both</td></tr>
</table>

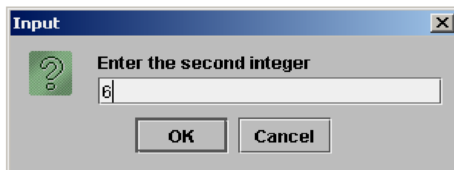
This program makes use of the different comparison operators

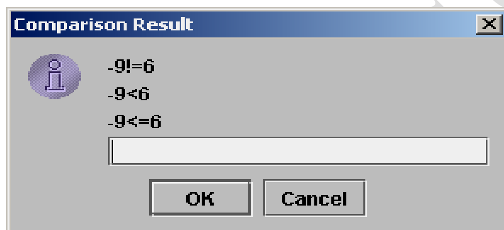
```
1.   import javax.swing.JOptionPane;
2.   // program for comparing values
3.   public class comparison1 {
4.        public static void main( String args[ ] ) {
5.               // read first  number from user as a string
6.               int number1 =  Integer.parseInt( JOptionPane.showInput
7.                     Dialog( "Enter the first integer" );
8.               // read second number from user as a string
9.               int number2 = Integer.parseInt(JOptionPane.showInput
10.                Dialog("Enter the second integer");
11.
12.              String result="";
13.
14.              if  ( number1 == number2)
15.                         result = result + number1 + "==" + number2;
```

PROPERTIES OF DLC UI, IBADAN

```
16.         if  ( number1 != number2 )
17.              result = result + number1 + "!="  + number2;
18.         if  ( number1 < number2 )
19.             result = result  + "\n"+  number1 + "<" + number2;
20.         if  ( number1 > number2 )
21.              result = result  + "\n" + number1  + ">"+ number2;
22.         if  (number1 <= number2 )
23.              result = result  + "\n" + number1  + "<=" + number2;
24.         if  (number1 >= number2 )
25.              result = result   + "\n" + number1  +"<=" + number2;
26.
27.         // Display results
28.         JOptionPane.showInputDialog(null, result, "Comparison
29.              Result", JOptionPane.INFORMATION_MESSAGE);
30.          System.exit( 0 ); //terminate application
31.      }
32.  }
```

Output:





4.4.5 The Math Class

The math class provides an extensive set of mathematical methods in the form of a static class library for manipulating the different mathematical expressions such as e, ∏, sin, cos, etc. Note that the trigonometric ratios, sin, cos and tan are implemented in radian measure and not in degree in Java. You need to write your own code to convert the angles given in degrees to radians. Also, note that ∏-rad = 180^0

```
/***This program prints the constants e and ∏, absolute of –1234, cos(∏/4), sin(∏/2), tan(∏/4), ln(1), e^∏ and
five random double numbers between 0.0 and 1.1 **/

public class MathApp {
  public static void main(String args [ ]) {
```


PROPERTY OF DLC UI, IBADAN

```
            System.out.println("Math.E = " + Math.E);
            System.out.println("Math.PI = "+Math.PI);
            System.out.println("Math.abs(-1234) =    "+Math.abs(- 1234));
            System.out.println("Math.cos(Math.PI/4) = +Math.cos(Math.PI/4));
            System.out.println("Math.sin(Math.PI/2) = " Math.sin(Math.PI/2));
            System.out.println("Math.tan(Math.PI/4) = +Math.tan(Math.PI/4));
            System.out.println("Math.log(1) = "+Math.log(1));
            System.out.println("Math.exp(Math.PI) = "+Math.exp(Math.PI));
            System.out.println("The first five Random numbers from 1 to 100 are .... ");
            // To generate random numbers …
            for (int i=0;i<5;++i)
              System.out.println(Math.floor(Math.random( )*100)  + " ");
            System.out.println();
  }
 }
```

Output:

```
Math.E = 2.718281828459045
Math.PI = 3.141592653589793
Math.abs(-1234) = 1234
Math.cos(Math.PI/4) = 0.7071067811865476
Math.sin(Math.PI/2) = 1.0
Math.tan(Math.PI/4) = 0.9999999999999999
Math.log(1) = 0.0
Math.exp(Math.PI) = 23.140692632779267
The first five Random numbers from 1 to 100 are ....
35.0
13.0
4.0
35.0
79.0
```

PROPERTIES OF DLC UI, IBADAN

Program to illustrate different arithmetic operators

```
public class Arith
{ public static void main(String[ ] args)
        { int m =25;
          int n = 7;
          System.out.println(“m = “ + m);
          System.out.println(“n = “ + n);
          int sum = m + n;
          System.out.println(“m + n = “ + sum);
          int difference = m-n;
          System.out.println(“m - n= “ + difference);
          int product = m*n;
          System.out.println(“m * n = “ + product);
          int quotient = m/n;
          System.out.println(“m/n = “ + quotient);
          int remainder = m%n;
          System.out.println(“m%n = “ + remainder);
        }
```

```
}

Output:
C:\j2sdk1.4.2_04\bin>java Arith
m = 25
n = 7
m + n = 32
m - n= 18
m * n = 175
m/n = 3
m%n = 4
```

4.4.6 Shift and Logical operators

A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left. The table overleaf summarizes the shift operators available in Java programming.

Operator	Use	Operation
>>	Op1 >> op2	Shifts bits of op1 right by distance op2
<<	Op1 << op2	Shifts bits op1 left by distance op2
>>>	Op1>>>op2	Shifts bits of op1 right by distance op2 (unsigned)

Each operator shifts the bits of the left-hand operand over by the number of positions by the right-hand operand. The shift occurs in the direction indicated by the operator itself. For example, The following statement shifts the bits of the integer 13 to the right by one position:

```
13 >> 1;
```

The binary representation of the number 13 is 1101. The result of the shift operation is 1101 shifted by one position –110, or 6 in decimal. The left-hand bits are filled with 0s as needed.

The following table shows the four operators the java programming language provides to perform bitwise functions on their operands:

Operator	Use	Operation
&	Op1 & op2	Bitwise AND
|	Op1 | op2	Bitwise OR
^	Op1 ^ op2	Bitwise XOR
~	~op2	Bitwise Complement

When its operands are numbers, the & operation performs the bitwise AND function on each parallel pair of bits in each operand.

The AND function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown below:

Op1	Op2	Result

PROPERTIES OF DLC UI, IBADAN

0	0	0
0	1	0
1	0	0
1	1	1

For instance, suppose that you were to AND the values 13 and 12, like this: 13 & 12. The result of this operation is 12 because the binary representation of 12 is 1100 and that of 13 is 1101.

```
   1101   // 13
&  1100   // 12
   ----
   1100   // 12
```

For the *inclusive OR* operation, if either of the two bits is 1, the result is 1. The following table shows the results of this operation:

Op1	**Op2**	**Result**
0	0	0
0	1	1
1	0	1
1	1	1

Exclusive OR means that if the two operand bits are different, the result is 1, otherwise the result is 1.

PROPERTIES OF DLC UI, IBADAN

Check the table below:

Op1	Op2	Result
0	0	0
0	1	1
1	0	1
1	1	0

Finally, the complement operator inverts the value of each bit of the operand bits is 1, the result is 0 and if the operand bit is 0, the result is 1.

4.4.7 Other Operators

4.4.7.1 The ternary operator / Shortcut for if - else statement ?:

op1 ? op2 : op3

The ?: operator returns op2 if op1 is true or returns op3 if op1 is false.

4.4.7.2 The [ ] Operator

We use square brackets to declare arrays, to create arrays and to access a particular element in an array. For example, to declare an array that can hold ten floating point numbers, we write:

```
float [ ] floatArray = new float [10];
```

To access the 7th element of the array, we write

```
floatArray[6];
```

Note that array indices begin at 0 in java.

4.4.7.3 The . (Dot) Operator

The dot (.) operator accesses instance members of an object or class members of a class. We shall more of this under the topic classes and inheritance.

4.4.7.4 The ( ) Operator

When declaring or calling a method, we list the method's arguments between (and). We can also specify an empty argument list by using ( ) with nothing between them.

4.4.7.5 The (type) Operator

Casts (or "converts") a value to the specified type. Example:

```
double sum = (double) (num/a);
```

PROPERTIES OF DLC UI, IBADAN

The statements converts the value evaluated from (num/a) to a double value.

4.4.7.6 The 'new' Operator

We use the operator to create a new object or a new array.

4.4.7.7 The 'instanceOf' Operator

The InstanceOf operator tests whether its first operand is an instance of its second.

```
Op1 instanceOf op2
```

Op1 must be the name of an object and op2 must be the name of a class. Ann object is considered to be an instance of a class if that object directly or indirectly descends from that class.

Examples

A. Given the following code snippet:

```
int  i = 10;
int n = i ++ %5;
```

Q1: What are the values of i and n after the code is executed?

Answer: i = 11, n = 0

Q2: What are the final values of i and n if instead of using postfix increment operator (i++), we use the prefix version (++i)?

Answer: i = 11, n = 1

Can you give reasons for these answers? Check relevant tables up.

B. What is the value of i after the following snippet executes?

```
int i = 8;
i >>= 2;
```

Answer: i = 2

c. What is the value of i after the following snippet executes?

```
int i = 17;
i >> = 1;
```

Answer: i = 8

4.4.8 Expressions

Variables and operators are the basic building blocks of programs. We combine literals, variables and operators to form expressions – segment of code that perform computations and return values. Certain expressions can be made into statements – complete units of execution. By grouping statements together with curly braces { and }, we create blocks of code.

Among other things, expressions are used to compute and to assign values to variables and to help control the execution flow of a program. The job of an expression is two-fold: to perform the

PROPERTIES OF DLC UI, IBADAN

computation indicated by the elements of the expression and to return a value that is the result of the computation.

Therefore, an expression is a series of variables, operators and method calls (constructed according to the syntax of the language) that evaluates to a single value. An expression can be arithmetic or logical. We have to specify how we want an arithmetic expression to be evaluated by using balanced parentheses (and). For example,

x + y / 100 is ambiguous but
(x + y) / 100 is unambiguous and recommended.

If we don't explicitly indicate the order in which we want the operations in a compound expression to be performed, the order is determined by the ***precedence*** assigned to the operators in use within the expression. Operators with a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Thus, the two following statements are equivalent:

x + y / 100
x + (y / 100) // Unambiguous, recommended

4.4.9 Operators' Order of Precedence

The following table shows the precedence assigned to the operators in Java. The operators in this table are listed in precedence order: the higher in the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with a relatively lower precedence. Operators on the same line have equal precedence.

Postfix operators	[ ], ., (params), expr++, expr--
Unary operators	++expr, --expr, +expr, -expr, ~, !
Creation or cast	new, (type)expr
Multiplicative	*, ?, %
Additive	+, -
Shift	<<, >>, >>>
Relational	<, >, <=, >=, instanceOf
Equality	==, !=
Bitwise AND	&
Bitwise exclusive OR	^
Bitwise inclusive OR	|
Logical AND	&&
Logical OR	||
Conditional	?:
Assignment	=, +=, -=, *=, /=, %=, ^=, |=, <<=, >>=, >>>=

PROPERTIES OF DLC UI, IBADAN

All binary operators except for the assignment operators are evaluated in left-to-right order. Assignment operators are evaluated right to left.

4.4.10 Statements

Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;):

- Assignment expressions
- Any use of ++ or --
- Method calls
- Object creation expressions

These kinds of statements are called ***expression statements***. Here are some examples of expression statements:

```
aValue = 23.89; // assignment statement
aValue++;  // increment statement
System.out.println(aValue);// method call statement
integer valueArray = new integer(4); // object creation
```

In addition to these kinds of expression statements, there are two kinds of statements. A ***declaration statement*** declares a variable. For example,

```
double aValue = 34.9;   // declaration statement
```

A ***control flow statement*** regulates the order in which statements get executed. The *for loop* and *if* statement are examples.

4.4.11 Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following program snippet shows two blocks each containing a single statement;

```
if (Character.isUpperCase(aChar)) {
 System.out.println ("The character " + aChar + " is
   upper case");
 } else {
   System.out.println("The character " + aChar + " is
    Lower case");
 }
```

4.4.12 Formatting Your Outputs

Sometimes we may want to format our outputs either to some number of decimal places or to display the output in a particular base value. Java borrows the C – Language way of formatting outputs. Method printf formats and outputs data to the standard output stream, System.out. Class **Formatter** formats and outputs data to a specified destination, such as a string or a file output stream

Every call to printf supplies as the first argument a **format string** that describes the output format. The format string may consist of fixed text and **format specifiers**. Fixed text is output by printf just as it would be output by System.out methods print or println. Each format specifier is a placeholder for a value and specifies the type of data to output. Format specifiers also may include optional formatting information.

PROPERTIES OF DLC UI, IBADAN

4.4.12.1 Integers

An integer is a whole number, such as 776, 0 or 52, that contains no decimal point. Integer values are displayed in one of several formats. The table below describes the **integral conversion characters**.

Integer conversion characters.	
Conversion character	**Description**
D	Display a decimal (base 10) integer.
O	Display an octal (base 8) integer.
x or X	Display a hexadecimal (base 16) integer. X causes the digits 09 and the letters AF to be displayed and x causes the digits 09 and af to be displayed.

The printf method has the form

```
printf( format-string, argument-list );
```

where format-string describes the output format, and argument-list contains the values that correspond to each format specifier in format-string. There can be many format specifiers in one format string.

Examine the code snippets below:

```
System.out.printf( "%d\n", 26 );
System.out.printf("%d\n", +26 );
System.out.printf("%d\n", -26 );
System.out.printf( "%o\n", 26 );
System.out.printf( "%x\n", 26 );
System.out.printf( "%X\n", 26 );
```

The Outputs below would be produced:

26
26
-26
32
1a
1A

4.4.12.2 Floating-Point Numbers

A floating-point value contains a decimal point, as in 33.5, 0.0 or -657.983. Floating-point values are displayed in one of several formats. Table below describes the floating-point conversions. The **conversion character e** and **E** displays floating-point values in computerized scientific notation (also called **exponential notation**). Exponential notation is the computer equivalent of the scientific notation used in mathematics. For example, the value 150.4582 is represented in scientific notation in mathematics as 1.504582×10^2

PROPERTIES OF DLC UI, IBADAN

Floating-point Conversion Characters.

Conversion character	Description
e or E	Display a floating-point value in exponential notation. When conversion character E is used, the output is displayed in uppercase letters.
f	Display a floating-point value in decimal format.
g or G	Display a floating-point value in either the floating-point format f or the exponential format e based on the magnitude of the value. If the magnitude is less than 10^{3}, or greater than or equal to 10^{7}, the floating-point value is printed with e (or E). Otherwise, the value is printed in format f. When conversion character G is used, the output is displayed in uppercase letters.
a or A	Display a floating-point number in hexadecimal format. When conversion character A is used, the output is displayed in uppercase letters.

Values printed with the conversion characters e, E and f are output with six digits of precision to the right of the decimal point by default (e.g., 1.045921). Other precisions must be specified explicitly. For values printed with the conversion character g, the precision represents the total number of digits displayed, excluding the exponent. The default is six digits (e.g., 12345678.9 is displayed as 1.23457e+07). Conversion character f always prints at least one digit to the left of the decimal point. Conversion character e and E print lowercase e and uppercase E preceding the exponent and always print exactly one digit to the left of the decimal point. Rounding occurs if the value being formatted has more significant digits than the precision.

Conversion character g (or **G**) prints in either e (E) or f format, depending on the floating-point value. For example, the values 0.0000875, 87500000.0, 8.75, 87.50 and 875.0 are printed as 8.750000e-05, 8.750000e+07, 8.750000, 87.500000 and 875.000000 with the conversion character g. The value 0.0000875 uses e notation because the magnitude is less than 10^{-3}. The value 87500000.0 uses e notation because the magnitude is greater than 107. Code snippets below illustrate the use of these conversion characters.

```
System.out.printf( “%e\n”, 12345678.9 );
System.out.printf(“%e\n”, +12345678.9 );
System.out.printf(“%e\n”, -12345678.9 );
System.out.printf(“%E\n”,12345678.9);
System.out.printf( “%f\n”, 12345678.9);
System.out.printf(“%g\n”, 12345678.9);
System.out.printf(“G\n”, 12345678.9);
```

Outputs

```
1.234568e+07
1.234568e+07
-1.234568e+07
1.234568E+07
12345678.900000
1.23457e+07
1.23457E+07
```

4.4.12.4 Printing Dates and Times

PROPERTIES OF DLC UI IBADAN

With the **conversion character t** or **T**, we can print dates and times in various formats. Conversion character t or T is always followed by a conversion suffix character that specifies the date and/or time format. When conversion character T is used, the output is displayed in uppercase letters. The table below lists the common conversion suffix characters for formatting date and time compositions that display both the date and the time. Next table 2 lists the common conversion suffix characters for formatting dates. Table 3 lists the common conversion suffix characters for formatting times. To view the complete list of conversion suffix characters, visit the Web site java.sun.com/j2se/5.0/docs/api/java/util/Formatter.html.

1. Date and time composition conversion suffix characters.

Conversion suffix character	Description
c	Display date and time formatted as day month date hour:minute:second time-zone year with three characters for day and month, two digits for date, hour, minute and second and four digits for yearfor example, Wed Mar 03 16:30:25 GMT-05:00 2004. The 24-hour clock is used. In this example, GMT-05:00 is the time zone.
F	Display date formatted as year-month-date with four digits for the year and two digits each for the month and the date (e.g., 2004-05-04).
D	Display date formatted as month/day/year with two digits each for the month, day and year (e.g., 03/03/04).
r	Display time formatted as hour:minute:second AM|PM with two digits each for the hour, minute and second (e.g., 04:30:25 PM). The 12-hour clock is used.
R	Display time formatted as hour:minute with two digits each for the hour and minute (e.g., 16:30). The 24-hour clock is used.
T	Display time formatted as hour:minute:second with two digits for the hour, minute and second (e.g., 16:30:25). The 24-hour clock is used.

2. Date formatting conversion suffix characters.

Conversion suffix character	Description
A	Display full name of the day of the week (e.g., Wednesday).
a	Display the three-character short name of the day of the week (e.g., Wed).
B	Display full name of the month (e.g., March).
b	Display the three-character short name of the month (e.g., Mar).
d	Display the day of the month with two digits, padding with leading zeros as necessary (e.g., 03).
m	Display the month with two digits, padding with leading zeros as necessary (e.g., 07).
e	Display the day of month without leading zeros (e.g., 3).

PROPERTIES OF DLCU, BADAN

Conversion suffix character	Description
Y	Display the year with four digits (e.g., 2004).
y	Display the last two digits of the year with leading zeros as necessary (e.g., 04).
j	Display the day of the year with three digits, padding with leading zeros as necessary (e.g., 016).

3. Time formatting conversion suffix characters.

Conversion suffix character	Description
H	Display hour in 24-hour clock with a leading zero as necessary (e.g., 16).
I	Display hour in 12-hour clock with a leading zero as necessary (e.g., 04).
k	Display hour in 24-hour clock without leading zeros (e.g., 16).
l	Display hour in 12-hour clock without leading zeros (e.g., 4).
M	Display minute with a leading zero as necessary (e.g., 06).
S	Display second with a leading zero as necessary (e.g., 05).
Z	Display the abbreviation for the time zone (e.g., GMT-05:00, stands for Eastern Standard Time, which is 5 hours behind Greenwich Mean Time).
P	Display morning or afternoon marker in lower case (e.g., pm).
p	Display morning or afternoon marker in upper case (e.g., PM).

Conversion character t requires the corresponding argument to be of type long, Long, Calendar or Date (both in package java.util) objects of each of these classes can represent dates and times. Class Calendar is the preferred class for this purpose because some constructors and methods in class Date are replaced by those in class Calendar. From the code below, Line 10 invokes static method getInstance of Calendar to obtain a calendar with the current date and time. Lines 13-17, 20-22 and 25-26 use this Calendar object in printf statements as the value to be formatted with conversion character t. Note that lines 20-22 and 25-26 use the optional argument index ("1$") to indicate that all format specifiers in the format string use the first argument after the format string in the argument list. Using the argument index eliminates the need to repeatedly list the same argument.

PROPERTIES OF DLC UI, IBADAN

Example Code

```
1.  import java.util.*;
2.  class cal {
3.      public static void main(String[] args) {
4.          Calendar dateTime = Calendar.getInstance( );
5.          System.out.printf("%tc\n", dateTime);
6.          System.out.printf("%tF\n", dateTime);
```

```
7.              System.out.printf("%tD\n", dateTime);
8.              System.out.printf("%tr\n", dateTime);
9.              System.out.printf("%tT\n", dateTime);
10.             // printing with conversion characters for date
11.             System.out.printf( "%1$tA, %1$tB%1$td, %1$tY\n",
12.                  dateTime);
13.             System.out.printf("%1$TA, %1$TB%1$Td, %1$TY\n",
14.                     dateTime);
15.             System.out.printf("%1$ta, %1$tb%1$te, %1$ty\n",
16.                   dateTime);
17.             // printing with conversion characters for time
18.             System.out.printf( "%1$tH:%1$tM:%1$tS\n", dateTime );
19.       }
20.   }
```

Outputs

```
Fri Mar 16 15:39:07 CET 2012
2012-03-16
03/16/12
03:39:07 PM
15:39:07
Friday, March16, 2012
FRIDAY, MARCH16, 2012
Fri, Mar16, 12
15:39:07
```

Summary

In this chapter you have been introduced to the different operators that Java supports: Arithmetic, relational and logical. The formatting styles for your outputs were also introduced

Post Test

2. Write Java equivalent expressions for the following arithmetic expressions.

   (i) $\frac{2(L + b)}{K^2} + e^{-h}$

   (ii) $\prod r^n - Log(\prod/2)$

   (iii) $\frac{Sin^2h - Cos^2h}{m^{xn}}$

2. Write a program to compute the area and perimeter of any plane shape and format your outputs to only two decimal places.

5 *Java Control Structures*

5.1 Introduction

When you write a program, you type statements into a file. Without control flow statements, the interpreter executes these statements in the order they appear in the file from left to right, top to bottom. Basically, there are three types of control structures in a programming language. The following table gives a summary of these structures.

Type of Control	Meaning	Control Structures
Sequential control	The program runs from line one to the last line without branching or testing for any condition	All the programs we have been writing before now are examples of this structure
Selection/Branching	One or more Conditions are tested. Statements are executed upon fulfilment of the conditions.	Simple if, If – else, else if, goto, nested if, arithmetic if, Switch, break and continue structures.
Looping/Repetition/Iteration	Statements are repeatedly executed either on fulfilling a condition or for some number of times.	For, while and do-while loops

You can use ***control flow statements*** in your programs to conditionally execute statements, to repeatedly execute a block of statements, and to otherwise change the normal, sequential flow of control. For example, in the following code snippet, the if statement conditionally executes the System.out.println statement within the braces, based on the return value of isUpperCase ( ).

```
Character.isUpperCase(aChar):
char c;
...
if (Character.isUpperCase(aChar)) {
   System.out.println("The character " + aChar + "
      is upper case.");
}
```

The Java programming language provides several control flow statements, which are listed in the following table.

Statement Type	Keyword
Looping / repetition	while, do-while , for
Decision making / selection	if-else, switch-case
Exception handling	try-catch-finally, throw
Branching	break, continue, label:, return

Note: Although goto is a reserved word, currently the Java programming language does not support the goto statement.

5.2 Objectives

PROPERTIES OF DLC UI, IBADAN

At the end of this chapter, you should be able to identify and use the different control structures that Java supports.

5.3 Pre Test

1. If you were to make a decision in your program, how would you achieve it?
2. Explain how you would find the average of any 20 numbers.

5.4 Main Content

5.4.1 The if/else Statements

The *if* statement enables your program to selectively execute other statements, based on some criteria.
Syntax:

Generally, the syntax of simple if statement can be written like this:

```
if (logical expression) {
   statement(s)
}
```

The *statement(s)* will only be executed only if the *logical expression* evaluates to true. If the expression is not rtue, the control simply passes to the next statement following the *statement(s)* block, { …. } and continues downwards from there.

As an example, suppose we have input a value for x previously in our program, we could write:

```
if (x < 50)
   System.out.println("The score is below cut off");
```

What if we want to perform a different set of statements if the expression is false? We use the *else* statement for that. The next program segment illustrates this:

```
System.out.println("Enter the grade: ");
int grade = input.nextInt();
if (grade >= 50)
  System.out.println("The grade is above cut off point ");
  else
        System.out.println("Grade below cut off point ");
```

Note that if the number of statements to be affected by the if is more than one, the statements must be enblocked thus:

```
if (logical expression) {
   Statement1;
   Statement2;
    .....
}
else {
     statement1;
     statements2;
    .....
}
```

Another form of the else statement, *else if*, executes a statement based on another expression. An if statement can have any number of companion else if statements but only one else.

PROPERTIES OF DLC UI, IBADAN

Following is a program, IfElse that assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on:

```
public class IfElse {
        public static void main(String[ ] args) {
            int testscore = 76;
            char grade;
            if (testscore >= 90) {
                grade = 'A';
            } else if (testscore >= 80) {
                grade = 'B';
            } else if (testscore >= 70) {
                grade = 'C';
            } else if (testscore >= 60) {
                grade = 'D';
            } else {
                grade = 'F';
            }
            System.out.println("Grade = " +   grade);
        }
    }
```

The output from this program is:

Grade = C

You may have noticed that the value of testscore can satisfy more than one of the expressions in the compound if statement: 76 >= 70 and 76 >= 60. However, as the runtime system processes a compound if statement such as this one, once a condition is satisfied, the appropriate statements are executed (grade = 'C';), and control passes out of the if statement without evaluating the remaining conditions

.

As another example, consider the following code on quadratic equation:

The general model for quadratic equations is $ax^2 + bx + c = 0$, where a, b and c are the variables that are used to determine the values of x; called the roots of the equation. Using the formula method

$$X = \frac{-b \pm \sqrt{d}}{2a}$$

d = discriminant = $b^2 - 4ac$

if $d = 0$, only one roots exists with value $-b/2a$

if $d < 0$, the roots are complex and if $d > 0$, there are two real roots and the formula stated above is used to compute the roots.

```
1.  // program to compute the roots of a quadratic equation
2.  import java.util.Scanner;
3.  class quadratic {
4.        public static void main(String[ ] args) {
5.              Scanner input = new Scanner(System.in);
6.              // getting the values of the coefficients a, b and c
7.              System.out.println("Enter the value of a");
8.              float a = input.nextFloat( );
9.              System.out.println("Enter the value of b");
10.             float b = input.nextFloat( );
11.             System.out.println("Enter the value of c");
12.             float c = input.nextFloat( );
13.
14.             //computing the discriminant d
```

PROPERTIES OF DLC UI, IBADAN

```
15.         double d = b*b - 4.0*a*c;
16.
17.         // testing for the solution path
18.         if (d<0) {
19.                 System.out.println("Complex roots pleas...");
20.         }
21.         else if (d = = 0) {
22.                 System.out.println("Only one real root exists
23.                       with  value....");
24.                 double x =( -b)/(2*a);
25.                 System.out.println(x);
26.         }
27.         else {
28.                 double x1 = ((-b) + Math.sqrt(d))/(2*a);
29.                 double x2 = ((-b) - Math.sqrt(d))/(2*a);
30.                 System.out.println("Two real roots exists with
31.                       values...");
32.                 System.out.println(x1 + "  and "  + x2);
33.         }
34.
35.         System.out.println("Bye - Bye to the user");
36.         System.exit(0);
37.     }
38. }
```

The sample output is shown in the next figure

```
--------------------Configuration: <Default>--------------------
Enter the value of a
2
Enter the value of b
3
Enter the value of c
1
Two real roots exists with values...
-0.5  and -1.0
Bye - Bye to the user
```

5.4.2 The switch Statement

We use the *switch* statement to conditionally perform statements based on an integer expression. The general syntax of a switch statement is:

```
switch (expression) {
      case L1:    statements; break;
      case L2:    statements; break;
      case L3:    statements; break;
      ……
      default:   statements
}
```

Note that:

1. The case labels L1 to Ln must be integers.
2. At the end of all statements for a case, a *break* statement is written. This is done to avoid the unwarranted flowing of control to other cases when a particular case has been executed.
3. The *default* statement may not be necessary, but may be part of the switch statements.

PROPERTIES OF DLC UI, IBADAN

4. The last *case* or *default* may not end with break statement since the control will automatically passed out of the switch block when they are executed.

Following is a sample program, SwitchDemo that declares an integer named month whose value supposedly represents the month in a date. The program displays the name of the month, based on the value of month, using the switch statement:

```
public class SwitchDemo {
  public static void main(String[ ] args) {
        int month = 8;
        switch (month) {
          case 1:  System.out.println("January"); break;
          case 2:  System.out.println("February");break;
          case 3:  System.out.println("March"); break;
          case 4: System.out.println("April"); break;
          case 5:  System.out.println("May"); break;
          case 6:  System.out.println("June"); break;
          case 7:  System.out.println("July"); break;
          case 8:  System.out.println("August"); break;
          case 9:  System.out.println("September"); break;
          case 10: System.out.println("October"); break;
          case 11: System.out.println("November"); break;
          case 12: System.out.println("December"); break;
    }
   }
  }
```

The switch statement evaluates its expression, in this case the value of month, and executes the appropriate *case* statement. Thus, the output of the program is: August. Of course, you could implement this by using an if statement:

```
int month = 8;
if (month == 1) {
   System.out.println("January");
} else if (month == 2) {
   System.out.println("February");
}
. . . // and so on
```

Deciding whether to use an if statement or a switch statement is a judgment call. You can decide which to use, based on readability and other factors. An if statement can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based only on a single integer value. Also, the value provided to each case statement must be unique. Another point of interest in the switch statement is the *break* statement after each case. As explained earlier, each break statement terminates the enclosing switch statement, and the flow of control continues with the first statement following the switch block. The break statements are necessary because without them, the case statements fall through. That is, without an explicit break, control will flow sequentially through subsequent case statements. Following is an example, SwitchDemo2, which illustrates why it might be useful to have case statements fall through:

```
public class SwitchDemo2 {
        public static void main(String[ ] args) {
        int month = 2;
        int year = 2000;
  int numDays = 0;
        switch (month) {
     case 1:
     case 3:
     case 5:
     case 7:
     case 8:
```


PROPERTIES OF DLC UI, IBADAN

```
        case 10:
        case 12:
            numDays = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
                        numDays = 30;
                        break;
                    case 2:
    if (((year % 4 == 0) && !(year%100 == 0))
                     || (year % 400 == 0) )
                         numDays = 29;
                            else
                               numDays = 28;
                               break;
               }
  System.out.println("Number of Days = " +      numDays);
              }
   }
```

The output from this program is:
Number of Days = 29

Technically, the final break is not required because flow would fall out of the switch statement anyway. However, we recommend using a break for the last case statement just in case you need to add more case statements at a later date. This makes modifying the code easier and less error-prone. You will see break used to terminate loops in Branching Statements. Finally, you can use the *default* statement at the end of the switch to handle all values that aren't explicitly handled by one of the case statements.

```
int month = 8;
. . .
switch (month) {
case 1:  System.out.println("January"); break;
case 2: System.out.println("February"); break;
case 3:  System.out.println("March"); break;
case 4:  System.out.println("April"); break;
case 5:  System.out.println("May"); break;
case 6:  System.out.println("June"); break;
case 7:  System.out.println("July"); break;
case 8:  System.out.println("August"); break;
case 9:  System.out.println("September"); break;
case 10: System.out.println("October"); break;
case 11: System.out.println("November"); break;
case 12: System.out.println("December"); break;
default: System.out.println("Hey, that's not a valid  month!"); break;
}
```

The following program gives a real-life application of the switch – case structure. The program computes the area and pperimeters of some common plane shapes.

```
1.  import java.util.Scanner;
2.  class shapes {
3.        public static void main(String[ ] args)  {
4.              Scanner input = new Scanner(System.in);
```

PROPERTIES OF DLC UI, IBADAN

```
5.          System.out.println("This program computes the area and
6.            perimeter of plane shapes" + " \n Enter 1 for Rectangle
7.            \n Enter 2 for Square \n Enter 3 for Circle \n Enter 4 for
8.             Parallelogram" + "\n Enter 5 for Triangle \n Enter 6 to
9.              exit the program \n\n Your  Option here...");
10.         int option = input.nextInt( );
11.         System.out.println( );
12.         switch (option) {
13.               case  1:  System.out.println("You have chosen
14.                              Rectangle");
15.                         System.out.println("Enter its length");
16.                          float l = input.nextFloat( );
17.                          System.out.println("Enter its breadth");
18.                          float b = input.nextFloat( );
19.                          double a = l * b;
20.                          double p = 2.0 *(l + b);
21.                          System.out.println("Area = "+ a +" Cm"+
22.                            "\nPerimeter =  "+ p + " Cm-square");
23.                          System.out.println( );
24.                          break;
25.
26.                case 2:   System.out.println("You have chosen
27.                                 Square");
28.                          System.out.println("Enter its length");
29.                          l = input.nextFloat( );
30.                          a = l * l;
31.                          p = 4.0 * l;
32.                          System.out.println("Area = "+ a +" Cm"+
33.                              "\nPerimeter = "+ p + " Cm-square");
34.                          System.out.println( );
35.                          break;
36.
37.             case 3:     System.out.println("You have chosen
38.                            Circle");
39.                         System.out.println("Enter its radius ");
40.                         float r = input.nextFloat( );
41.                         a = Math.PI * r * r;
42.                         p = 2.0 * Math.PI * r;
43.                         System.out.printf("Area = %.2f Cm
44.                            \nPerimeter = %.2f  Cm-square", a ,p);
45.                         System.out.println( );
46.                         break;
47.
48.             case 4:    System.out.println("You have chosen
49.                          Parallelogram");
50.                        System.out.println("Enter its Height");
51.                        float h = input.nextFloat( );
52.                        System.out.println("Enter one of its slanting
53.                             parallel sides ");
54.                        l = input.nextFloat( );
55.                        System.out.println("Enter its base");
56.                        b = input.nextFloat( );
57.                        a = b * h;
58.                        p = (2.0 * l) + (2.0 * b);
59.                        System.out.println("Area = "+ a +" Cm"+
60.                            "\nPerimeter = "+ p + " Cm-square");
61.                        System.out.println( );
62.                        break;
63.
64.            case 5:     System.out.println("You have chosen
```

PROPERTIES OF DLC UI, IBADAN

```
65.                              Triangle");
66.                             System.out.println("Enter its height");
67.                             h = input.nextFloat( );
68.                             System.out.println("Enter first slanting height
69.                                 ");
70.                             float l1 = input.nextFloat( );
71.                             System.out.println("Enter second slanting
72.                                    height");
73.                             float l2 = input.nextFloat( );
74.                             System.out.println("Enter its base");
75.                             b = input.nextFloat( );
76.                             a = 0.5 * b * h;
77.                            p = l1 + l2 + b;
78.                            System.out.println("Area = "+ a +" Cm"+
79.                                "\nPerimeter = "+ p + " Cm-square");
80.                             System.out.println( );
81.                              break;
82.
83.                   case 6:     System.out.println("You have chosen to exit,
84.                                     Thank you, Program stops...");
85.                               System.out.println( );
86.                               System.exit(0); break;
87.
88.                  default:     System.out.println("Wrong option chosen,
89.                               Program terminates... ");
90.                               System.out.println( );
91.                               System.exit(0);
92.               } //end switch
93.              System.out.println( );
94.              System.out.println("Thanks for using this program,
95.                     Bye...");
96.              System.out.println( );
97.              System.exit(0);
98.          }  // end method main
99.       }  // end class shapes
```

Sample Output:

```
-------------------Configuration: <Default>--------------------
This program computes the area and perimeter of plane shapes
 Enter 1 for Rectangle
 Enter 2 for Square
 Enter 3 for Circle
 Enter 4 for Parallelogram
 Enter 5 for Triangle
 Enter 6 to exit the program

 Your Option here...
2

You have chosen Square
Enter its length
6
Area = 36.0 Cm
Perimeter =   24.0 Cm-square

Thanks for using this program, Bye...
```

PROPERTIES OF DLC UI, IBADAN

5.4.3 The for Statement

The *for* statement provides a compact way to iterate over a range of values. It is used when we know ahead the number of times a section of code is to be repeated. It is a counter-controlled loop.

The general form of the for statement can be expressed like this:

for (*initialization*; *termination/continuation criterion*; *increment*) {
statements;
}

The *initialization* is an expression that initializes the loop. It is executed once at the beginning of the loop. The *termination/continuation* criterion determines when to terminate the loop. It in essence gives the conditions to continuing the loop until when the loop will terminate. This expression is evaluated at the top of each iteration of the loop. When the expression evaluates to false, the loop terminates. Finally, *increment* is an expression that gets invoked after each iteration through the loop. All these components are optional. In fact, to write an infinite loop, you omit all three expressions:

```
for ( ; ; ) {
 // infinite loop    ...
}
```

The Demo program that follows adds the numbers from 1 to 10 and displays the result.

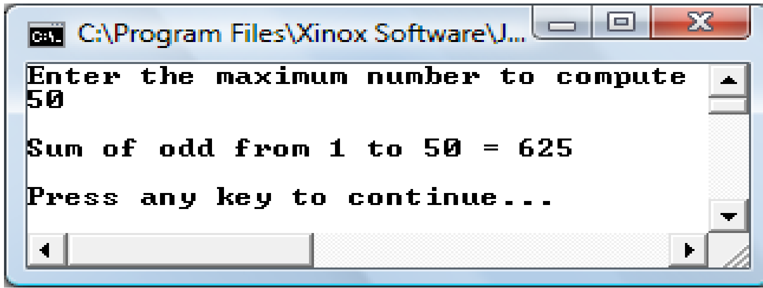
```
1.  public class Demo {
2.          public static void main(String[ ] args) {
3.                  int sum = 0;
4.                  for (int i = 1; i<= 10; i++) {
5.                          sum += i;
6.                    }// next i
7.                    System.out.println("Sum = " + sum);
8.              }
9.          }
```

The output from this program is: Sum = 55

Other Examples
To sum all odd numbers together from 1 to n

```
1.  import java.util.Scanner;
2.  public class number {
3.         public static void main(String[ ] args) {
4.                Scanner in = new Scanner(System.in);
5.                System.out.println("Enter the maximum number to
6.                  compute");
7.                int n = in.nextInt( );
8.                int sum = 0;
9.                for (int i = 1; i<= n; i+=2) {
10.                      sum += i;
11.               }  //next i
12.               System.out.println( );
13.               System.out.println("Sum of odd from 1 to " + n + " = " +
14.                 sum);
15.               System.out.println( );
16.       }
17.    }
```

PROPERTIES OF DLCUI, IBADAN



Note: The increment is i += 2 or i = i + 2 and not i +2!

The sum is being used as an accumulator here and has to be initialized to 0. It accumulates all the summations in the loop.

Note that you can declare a local variable within the initialization expression of a for loop. The scope of this variable extends from its declaration to the end of the block governed by the for statement so it can be used in the termination and increment expressions as well. If the variable that controls a for loop is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names j, k, and i are often used to control for loops; declaring them within the for loop initialization expression limits their life span and reduces errors.

Nested For-loops

In some cases, two or more for-loops can be nested with each other. In such cases, the innermost for-loop will run faster than the outer one. This means that the innermost loop will have to run into completion before the control is passed to the outermost loop.
The general syntax for two nested for-loops is:

```
for (  i loop)  {
   for (j loop) {
         S;
   } // next j
}  // next i
```

Consider the example code below, which computes the times table from 1 to 10:

```
1.  class times  {
2.  public static void main(String [ ] args) {
3.  for (int i = 1; i <= 10; i++) {  //outermost loop
4.      for (int j = 1; j <= 10; j++) { //innermost loop
5.           int Times = i * j;
6.           System.out.print("|" + Times + "\t");
7.      }   // Ending Innermost loop
8.     System.out.println("…………………………………………");
9.    }   //Ending outermost loop
10.  }    // End main
11.  }  //End class
```

Output next page ………………..

TIMES TABLE

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

PROPERTIES OF DLC UI, IBADAN

```
-----------------------------------------------------------------
| 2  | 4  | 6  | 8  | 10  | 12 | 14  | 16 | 18 | 20
------------------------------------------------------------------
| 3  | 6  | 9  | 12 | 15  | 18 | 21  | 24 | 27 | 30
------------------------------------------------------------------
| 4  | 8  | 12 | 16 | 20  | 24 | 28  | 32 | 36 | 40
------------------------------------------------------------------
| 5  | 10 | 15 | 20 | 25  | 30 | 35  | 40 | 45 | 50
------------------------------------------------------------------
| 6  | 12 | 18 | 24 | 30  | 36 | 42  | 48 | 54 | 60
------------------------------------------------------------------
| 7  | 14 | 21 | 28 | 35  | 42 | 49  | 56 | 63 | 70
------------------------------------------------------------------
| 8  | 16 | 24 | 32 | 40  | 48 | 56  | 64 | 72 | 80
------------------------------------------------------------------
| 9  | 18 | 27 | 36 | 45  | 54 | 63  | 72 | 81 | 90
------------------------------------------------------------------
| 10 | 20 | 30 | 40 | 50  | 60 | 70  | 80 | 90 | 100
```

Explanation:
At the first entry of the two for-loops (lines 3 – 4), each of the *loop invariants* i and j have initial values of 1 and 1 respectively. After the first iteration, the innermost j will have to increment to 2 while i still remain at 1. This is how the innermost loop will increment up to 10. At the end of the 10th iteration, then a line will be printed (Line 8) and then i now increment to 2, while j starts the iterations again as before. Note the use of System.out.print in the innermost loop (Line 6), just to print the results on a line; and the use of System.out.println (Line 8) in the outermost loop, just to move on to next line.

As an exercise, implement a program to compute means of five experiments' data with each experiment repeated 3 times each.

5.4.4 The while and do-while Statements

while loop

We use a *while* statement to continually execute a block of statements while a condition remains true. It is somehow regarded as a pre-conditional testing control. The general syntax of the while statement is:

```
while (expression) {
   Statements;
}
```

First, the while statement evaluates *expression*, which must return a boolean value. If the expression returns true, then the while statement executes the statement(s) associated with it. The while statement continues testing the expression and executing its block until the expression returns false. In this wise, we say a while loop is a pre-test looping mechanism, since it will first of all test the condition before executing the loop. The control moves to the next statement after the while statements' block when the *expression* is no more valid and execution continues from there downwards.

Consider the following program segment that sums all numbers from 1 to 10.

```
int sum = 0;
int i = 1;
while (i <= 10) {
```

PROPERTIES OF DLC UI, IBADAN

```
   sum += i;
   i++;
 }//end while
System.out.print("Total sum of numbers from 1 to 10 = " + sum);
```

The while first of all tests for value of i if it is less or equal to 10. If true, then the block of code for while is executed and i is incremented in the block. The while will continue like this until the condition i<=10 is no more valid. Then it will jump to the statement that follows the while block – System.out.print("Total sum of numbers from 1 to 10 = " + sum);

Consider the following program, which continuously accepts some positive numbers until when a negative number is entered. The negative number terminates the loop. Any data that is used to terminate a loop like the negative number in this case is called a **sentinel**.

```
1.   import javax.swing.*;
2.   class summation {
3.   public static void main(String[ ] args) {
4.         int count = 0;
5.         float sum = 0;
6.         float data = Float.parseFloat (JOptionPane.showInputDialog
7.              ("Enter first  data"));
8.         while (data > 0)  {
9.               count++;
10.              sum += data;
11.              data = Float.parseFloat(JOptionPane.showInputDialog(
12.                 "Enter next  data"));
13.        }//end while
14.       JOptionPane.showMessageDialog(null, "Sum of " + count + "
15.            data added is " + sum,"Results",
16.                 JOptionPane.INFORMATION_MESSAGE);
17.       System.exit(0);
18.     }
19.   }
```

Exercise: Type the program into an editor and run the program for the following sets of data. Record your observations.

(i) 5, 6, -5, 8, -9

(ii) –8, 8, -7

Answers: (i) 11 (ii) 0, Give reasons for these answers

do-while loop

The Java programming language provides another statement that is similar to the while statement-the *do-while* statement. It is often regarded as a post-conditional testing control.

The general syntax of the do-while is:

```
do {
   statement(s)
} while (expression);
```

Instead of evaluating the expression at the top of the loop, do-while evaluates the expression at the bottom. Thus the statements associated with a do-while are executed at least once. This structure is post-conditional testing technique, just like Repeat-Until in some languages like Pascal. Consider the program below which is the do – while version of the last program.

PROPERTIES OF DLC UI, IBADAN

```
import javax.swing.*;
1.  class summation1 {
2.      public static void main(String[ ] args) {
3.          int count = 0;
4.          float sum = 0;
5.          float data =   Float.parseFloat(JOptionPane.
6.                  showInputDialog ("Enter first data"));
7.          do  {
8.                  count++;
9.                  sum += data;
10.                 float data = Float.parseFloat (JOptionPane.
11.                     showInputDialog  ("Enter next data"));
12.         } while (data > 0);
13.         JOptionPane.showMessageDialog(null, "Sum of " + count
14.            + " data added is " + sum,"Results",
15.           JOptionPane.INFORMATION_MESSAGE);
16.         System.exit(0);
17.     }
18. }
```

Practical Exercise: Type the program into an editor and run the program for the following sets of data. Record your observations.

(i) 5, 6, -5, 8, -9
(ii) –3,6,7,8,-1,8
(iii) –8, 8, 7

Answers: (i) 11 (ii) 18 (iii) infinite looping. Explain the reasons for these answers

5.4.5 Exception Handling Statements

The Java programming language provides a mechanism known as *exceptions* to help programs report and handle errors. When an error occurs, the program throws an exception. What does this mean? It means that the normal flow of the program is interrupted and that the runtime environment attempts to find an *exception handler*--a block of code that can handle a particular type of error. The exception handler can attempt to recover from the error or, if it determines that the error is unrecoverable, provide a gentle exit from the program.
Three statements play a part in handling exceptions:

- The *try* statement identifies a block of statements within which an exception might be thrown.
- The *catch* statement must be associated with a try statement and identifies a block of statements that can handle a particular type of exception. The statements are executed if an exception of a particular type occurs within the try block.
- The *finally* statement must be associated with a try statement and identifies a block of statements that are executed regardless of whether or not an error occurs within the try block.

Here's the general form of these statements:

```
try {
    statement(s)
} catch (exceptiontype  name) {
    statement(s)
} finally {
    statement(s)
}
```

PROPERTIES OF DLC UI, IBADAN

This has been a brief overview of the statements provided by the Java programming language used in reporting and handling errors. However, other factors and considerations, such as the difference between runtime and checked exceptions and the hierarchy of exceptions classes, which represent various types of exceptions, play a role in using the exception mechanism.

5.4.6 Your First Encounter with Java Exceptions

If you have done any amount of Java programming at all, you have undoubtedly already encountered exceptions. Your first encounter with Java exceptions was probably in the form of an error message from the compiler like this one:

```
InputFile.java:11: Exception java.io.FileNotFoundException must be caught, or it must be declared in the
throws clause of this method.

in = new FileReader(filename);
           ^
```

This message indicates that the compiler found an exception that is not being handled. The Java language requires that a method either catch all "checked" exceptions (those that are checked by the runtime system) or specify that it can throw that type of exception.

5.4.7 Branching Statements

The Java programming language supports three branching statements:

- The *break* statement
- The *continue* statement
- The *return* statement

The *break* statement and the continue statement, which are covered next, can be used with or without a label. A *label* is an identifier placed before a statement. The label is followed by a colon (:):

statementName: someJavaStatement;

You'll see an example of a label within the context of a program in the next section.

- **The *break* Statement**

The *break* statement has two forms: unlabeled and labeled. You saw the unlabeled form of the *break* statement used with switch earlier. As noted there, an unlabeled *break* terminates the enclosing switch statement, and flow of control transfers to the statement immediately following the switch. You can also use the unlabeled form of the *break* statement to terminate a *for, while*, or *do-while* loop. The following sample program, Breakdemo, contains a *for* loop that searches for a particular value within an array:

```
1.  public class BreakDemo {
2.      public static void main(String[ ] args) {
3.          int[ ] arrayOfInts = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622,
4.                                 127  };
5.          int searchfor = 12;
6.          int i = 0;
7.          boolean foundIt = false;
8.          for ( ; i < arrayOfInts.length; i++) {
9.              if (arrayOfInts[i] == searchfor) {
10.                 foundIt = true;
11.                 break;
12.             }
13.         } //end for
14.         if (foundIt) {
```


PROPERTIES OF DLC UI, IBADAN

```
15.            System.out.println("Found " + searchfor + " at index " + i);
16.            } else {    System.out.println(searchfor + "not in the array");
17.            }
18.          }
19.      }
```

The *break* statement terminates the for loop when the value is found. The flow of control transfers to the statement following the enclosing *for*, which is the print statement at the end of the program. The output of this program is: Found 12 at index 4

The unlabeled form of the *break* statement is used to terminate the innermost *switch, for, while*, or *do-while*; the labeled form terminates an outer statement, which is identified by the label specified in the break statement. The following program, BreakWithlabelDemo, is similar to the previous one, but it searches for a value in a two-dimensional array. Two nested for loops traverse the array. When the value is found, a labeled break terminates the statement labeled search, which is the outer *for* loop:

```
1.      public class BreakWithLabelDemo {
2.          public static void main(String[ ] args) {
3.              int[ ][ ] arrayOfInts = { { 32, 87, 3, 589 },
4.                                        { 12, 1076, 2000, 8 },
5.                                        { 622, 127, 77, 955 }
6.                                      };
7.                  int searchfor = 12;
8.              int i = 0;
9.              int j = 0;
10.             boolean foundIt = false;
11.             search:
12.             for ( ; i < arrayOfInts.length; i++) {
13.                 for (j = 0; j < arrayOfInts[i].length; j++) {
14.                     if (arrayOfInts[i][j] == searchfor) {
15.                           foundIt = true;
16.                           break search;
17.                      } // end if
18.                  } // next j
19.              }   // next i
20.             if (foundIt) {
21.                 System.out.println("Found " + searchfor + " at " + i +
22.                    ", " + j);
23.             } else {
24.                     System.out.println(searchfor + "not in the
25.                       array");
26.                 }
27.         }
28.     }
```

The output of this program is: Found 12 at 1, 0

This syntax can be a little confusing. The break statement terminates the labeled statement; it does not transfer the flow of control to the label. The flow of control transfers to the statement immediately following the labeled (terminated) statement.

- **The Continue Statement**

You use the *continue* statement to skip the current iteration of a *for, while*, or *do-while* loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, basically skipping the remainder of this iteration of the loop. The following program, ContinueDemo, steps through a string buffer checking each letter. If the current character is

PROPERTIES OF DLC UI, IBADAN

not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it is a p, the program increments a counter, and converts the p to an uppercase letter.

```
1.  public class ContinueDemo {
2.        public static void main(String[ ] args) {
3.            StringBuffer searchMe = new StringBuffer( "peter
4.                        piper picked a peck of pickled
5.                        peppers");
6.            int max = searchMe.length();
7.            int numPs = 0;
8.            for (int i = 0; i < max; i++) {
9.                  //interested only in p's
10.                 if (searchMe.charAt(i) != 'p')
11.                      continue;
12.                 //process p's
13.                 numPs++;
14.                 searchMe.setCharAt(i, 'P');
15.            } //next i
16.            System.out.println("Found " +  numPs + " p's in
17.              the string.");
18.            System.out.println(searchMe);
19.       }
20.   }
```

Here is the output of this program:

```
Found 9 p's in the string.
Peter PiPer Picked a Peck of Pickled
PePPers
```

The labeled form of the continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. This program uses the labeled form of continue to skip an iteration in the outer loop:

```
1.  public class ContinueWithLabelDemo {
2.     public static void main(String[ ] args) {
3.           String searchMe = "Look for a substring in me";
4.           String substring = "sub";
5.           boolean foundIt = false;
6.           int max = searchMe.length() - substring.length();
7.           test:
8.           for (int i = 0; i <= max; i++) {
9.                  int n = substring.length();
10.                 int j = i;
11.                 int k = 0;
12.                 while (n-- != 0) {
13.                        if (searchMe.charAt(j++) !=
14.                            substring.charAt(k++)) {
15.                           continue test;
16.                         } // end if
17.                  } // end while
18.                 foundIt = true;
19.                 break test;
20.              } // end for
21.          System.out.println(foundIt ? "Found it" :  "Didn't find
22.             it");
23.       }
```

PROPERTIES OF DLC UI, IBADAN

```
24.         }
```

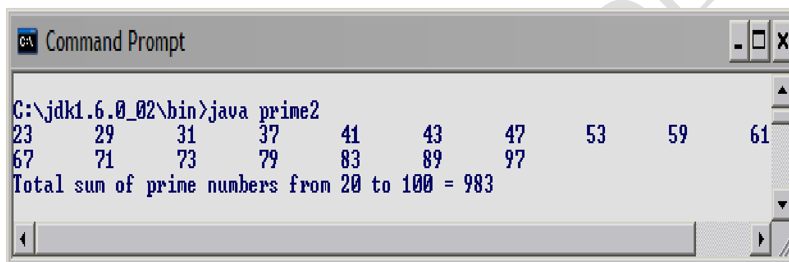
Here is the output from this program: Found it

Another example

Consider the following code, which writes and computes the sum of all prime numbers from 20 to 100.

```
1.    public class prime {
2.        public static void main(String[ ] args) {
3.                int sum = 0, i, j;
4.                L1: for (i = 20; i <= 100; i++) {  // numbers from 20 to 100
5.                        for (j = 2; j < i; j++)  {
6.                            if (i%j = = 0) // remainder of i by j from 2 to i – 1
7.                            continue L1;
8.                         }// Next j
9.                        System.out.print(i + "\t");
10.                       sum += i;
11.                    }// Next i
12.                   System.out.println("\n" + "Total sum of prime
13.                     numbers from  20 to 100 = " + sum);
14.             }
15.         }
```

Output:



If i is divided by j and the remainder equals zero (0) (Line 6), this means i is not prime. The continue L1; statement (Line 4) makes the control to jump out of j – loop and go to for-loop labeled L1. The current iteration of i is skipped; i is then incremented to the next value. This construct works like GOTO statement found in other languages like FORTRAN. But it only works with nested loops.

5.4.8 The return Statement

The last of Java's branching statements is the *return* statement. You use return to exit from the current method. The flow of control returns to the statement that follows the original method call. The return statement has two forms: one that returns a value and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword:

return ++count;

The data type of the value returned by return must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value: *return;*

Summary

In this chapter you have been introduced to the control structures in Java programming. The ball is in your court. Start writing simple programs now so that you become a Java 'guru'

PROPERTY OF DLC UI, IBADAN

Post Test

1. A man borrows N3000 from a bank at an interest rate of 1.5% per month. He pays N250 at the end of each month. The amount he owes (AM) at the end of the each month is calculated thus:

   $$AM = P + (1.5 / 100 * P) - 250$$

   Where P = the Principal amount left to be paid.

   Write a program that will print the amount he owes each month and the number of months it will take him to pay all the debt.

2. At Lever Brothers Nigeria Limited, each staff member is given a monthly basic salary commensurate with his/her salary grade level. In addition, each staff is given 10% of basic salary as Transport allowance, 15.5% of basic as Housing Allowance and N500 flat as meal subsidy. Also, if a staff has spent over 10 years in the company, he/she is given 2% of his basic salary as 'long serving staff allowance'. Write a java program to implement the above pay-roll policy, assuming 50 staff numbers in all.

3. There were twenty experimental set-ups in a Chemistry Laboratory. Each of the experiments was repeated four times. Implement a program that captures the results from the experiments and reports the mean values for each of the experiments. Hint: Use Nested *for* loops.

4. The are n stores in a big marketing company. Each of the stores has m departments. Each department has daily sales per week. Write a program to compute the (i) total sales per week for the company, (ii) total sales per week for each each of the stores and (iii) total sales per week for each of the departments.

PROPERTIES OF DLC UI, IBADAN

6 *Java Arrays and Vectors*

6.1 Introduction

In this chapter, you shall be introduced to the use use of arrays and how Java handles them. You are advised to review the concept of loops especially *for*-loops in the previous chapter.

6.2 Objectives

At the end of this chapter, you should be able to
(i) know the meaning, types and uses of arrays in Java
(ii) declare and insert data into arrays in Java

6.3 Pre Test

(i) Briefly explain the algorithm to calculate the standard deviation of any three numbers.
(ii) How many variables would you need if you were to calculate the same standard deviation for hundred numbers?

6.4 Main Content

6.4.1 What is an Array?

An array is a group of variables, all of the same data type that is referred to by a single name. The values in the group occupy contiguous locations in the computer memory, i.e., the data are stored one next to each other in the computer memory. Each element in the array is identified by the array name and a subscript pointing to the particular location within the array.

We have two types of array – one (single) and multi-dimensional arrays. Let us explain the term dimension before we go on to the lecture properly. Dimension has to do with the number of objects or entities a data is related to. For instance, if we say the score of a particular student. In this case, the score data is related to one entity - student. This is one dimensional array, usually depicted on a single row vector. But if we now say the score of a particular student in a particular course. Then, the same score data is related to a student and course. This is two-dimension in this case, usually depicted with a table having rows and columns. We could go ahead to relate the same score to student, course and session, making a three dimension of the score data. Another example is the rain volume in a particular day. This is one dimension. Rain volume in a particular day in a particular week. This is two-dimension. Now, rain volume in a particular day, in a particular week, in a particular month, (making three-dimension), in a particular year, making four-dimension.

The following is a logical representation of a single dimensional array in memory.

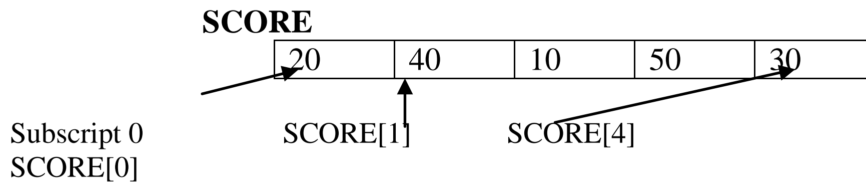
A(1)	A(2)	A(3)	A(4)	A(5)

A(1) to A(5) are subscripts or locations or indexes of the array A and they can contain any valid values of the same data type. The individual cell or location in the array is also called a subscript. Subscript A(1) could contain 56 as its data. So, we reference this 56 with the subscript number, A(1).

In Java, the elements are numbered with subscripts starting from 0, and can be referenced by their number using the subscript operator [ ].

For example, the following depicts the scores of 5 students in a test.

PROPERTIES OF DLC UI, IBADAN



The name of the Array is SCORE, 20 is stored at subscript/index 0 and 30 is stored at subscript 4

6.4.2 Types of Arrays

1. One dimensional array – with a single row or column
2. Two dimensional Array – with more than one row and column
3. Character Arrays
4. String arrays

6.4.3 Creating a One-dimensional Array

The general syntax for creating an array is:

element-type [ ] arrayName; // declares the array
arrayName = new element type[n] // allocates storage for n elements

As with single objects, both the declaration and allocation can be combined in a single declaration with initialization as shown below:

element-type[ ] arrayName = new element-type[n]

element-type could be any of the primitive data type such as int, float, double, char etc.

Examples:

- float[ ] x; // declares x to be a reference to an array of floats
- x = new float[8]; // allocates an array of 8 floats, referenced by x
- boolean [ ] flags = new boolean[8];
- int [ ] score = new score[20];

6.4.4 Reading in Data into a One-dimensional Array

for-loop is commonly used to achieve this. The following code snippet reads in data into a one dimensional array:

```
for (int i = 0; i < arrayName.length; i++) {
   float arrayName[i] = Float.parseFloat
    (JOptionPane.showInputDialog("Enter data for subscript " + (i+1) ));
}
```

Comments: when i is 0, the pointer is at location 0 of the array. What would be printed on screen is Enter data for subscript 1. Ideally it should be subscript 0, but we have done an arithmetic operation in the string parameter in the JOptionPane, (i + 1). This is only done at the level of the user of the program, who does not know anything about java zero-based indexing. Data entered at this point will be assigned to location 0 of the array. The *for* loop will iterate until the length of the array has been filled up. Note that the termination point of the loop should be 1 less than the total array length because we are starting from subscript zero (i < arrayName.length). If we had used i <=

PROPERTIES OF DLC UI, IBADAN

arrayName.length, then we would have commited an arrayOutOfBoundException error, i.e., we go beyond the length of the array. The *length* function should not contain parenthesis, ( ) as in the String *length* function.

6.4.5 Summing all the Data in a One-dimensional Array

The following code snippet sums all the data in a one-dimensional array;

```
float sum = 0; //sum serves as an accumulator
for (int i = 0; i < arrayName.length; i++) {
        sum += arrayName[i];
}
```

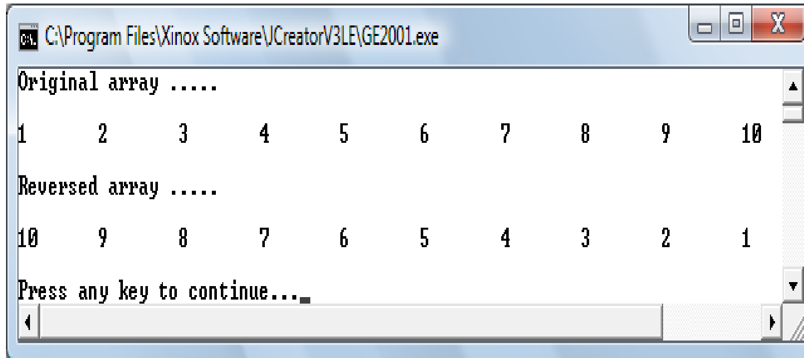
6.4.6 Reversing the elements of an array

Study the code below and explain the algorithm for reversing the elements of a one-dimensional array.

```
1.   import javax.swing.*;
2.   class arrayReverse {
3.      public static void main (String[] args) {
4.           int n = Integer.parseInt(JOptionPane.showInputDialog
5.                ("Enter the length of the array"));
6.           int[ ] a = new int[n];
7.
8.           // Reading Data into the Array a
9.           for (int m = 0; m < n; m++)
10.              a[m] = Integer.parseInt(JOptionPane.showInputDialog
11.                     ("Enter data"+ (m+1)));
12.
13.          // printing the original array a
14.          System.out.println("Original array .....\n");
15.          for (int l = 0; l < n; l++ )
16.               System.out.print(a[l] + "\t");
17.
18.          System.out.println("\n");
19.
20.          // reversing the array a
21.          int i = 0, j = n - 1;
22.          while ((i != j)) {
23.                int temp = a[i];   //swapping
24.                a[i] = a[j];
25.                a[j] = temp;
26.                i++;
27.                j--;
28.                if (j < i)
29.                   break;
30.            }  // end while
31.
32.          // Printing the array in reverse order
33.          System.out.println("Reversed array ..... \n");
34.          for (int k = 0; k < n; k++ )
35.                System.out.print(a[k] + "\t");
36.
37.          //Closing the program
38.          System.out.println("\n");
39.          System.exit(0);
40.       }
41.   }
```

PROPERTIES OF DLC UI, IBADAN

Sample Output



6.4.7 Character Arrays

The element-type of character Array is char. The strings are nearly the same as the character array but with little difference. The program below compares a string object with a char array.

Example:

```
class   TestCharArrays {
  public static void main(String args[ ]) {
    String s = new String( "ABCDEFG");
    char[ ] a = s.toCharArray();
    System.out.print("S = "+  s +" \t a =" + a + "\' " );
    System.out.print("s.length( )=" + s.length( )+"\t a.length = " + a.length );
    for (int i = 0; i < s.length(); i++)
    System.out.print("S. charAt("+i +") = " + s.charAt(i)  + "\t a [ " + i + "] ="
            + a[i]);
  }
}
```

The output is

```
s =  “ABCDEFG”              a =”ABCDEFG”
s.length() = 7              a. length = 7
s.charAt(0) = A             a[0] = A
s.chartAt(1) = B            a[1] = B
.                                    .
.                                    .
.                                    .
s.chartAt(6) = G            a[6] = G
```

Note that

- Arrays are zero-based indexing, i.e., the first element has index 0.
- s and a are reference variables.
- s refers to a string object while a refers to a char[ ] object
- Array objects have a public field named length which stores the number of elements in the array. So the expression a.length is analogous to the invocation of s.length( ).
- An array of length n has index numbers from 0 to n-1.
- We can initialize an array explicitly with an initialization list : like this:
  int [ ] c = {44, 88, 55, 33};

This single line is equivalent to the following six lines

PROPERTIES OF DLC UI, IBADAN

```
int [ ] c;
c = new int [4];
c[0] = 44;
c[1] =  88;
c[2] = 55;
c[3] =  33;
```

6.4.8 Two-dimensional Arrays

A two-dimensional array looks like a matrix as shown below:

34	23	24
24	56	13
15	10	19

The following are therefore the features of a two-dimensional array

- It uses 2 subscripts; [row][column]
- It has grid of rows and columns, with the 1st subscript locating the row and the 2nd subscript locating the column.
- Java sees a two-dimensional array as a big array containing small arrays. Each row forms an array of the whole array.

For example,

```
int [ ][ ] a = new int [7][9];
```

The above declares a two-dimensional array a of 7 rows and 9 columns. The row number is specified first followed by the column number. To assign a value to a location in the array:

```
a[0][2] = 50;
```

assigns 50 to the element in row 0 column 2, (1st row, 3rd column)

6.4.8.1 Reading Data into a Two-dimensional Array

The following program snippet would achieve this:

```
for (int i = 0; i < a.length; i++) {
  for (int j = 0; j < a[i].length; j++) {
        System.out.println("Enter data for subscript [ " +  i + "," + j + "]");
        float a[i][j] = input.nextFloat();
  }
}
```

The snippet uses two nested *for* loops. The first *i-loop* moves round the rows while the inner *j-loop* moves round the columns of the array. As you have been taught while learning nested *for* loop control structure, the *j*-loop moves faster than the *i*-loop. This means that for each row i, the columns would heve to be filled first before going to the next row. The row length is specified as *a.length* while the column length is specified as *a[i].length.* Why the *a[i].length* for the column number? This is because each row of a two-dimensional array is an array itself with its own length. As i increases, it denotes the next row to operate on.

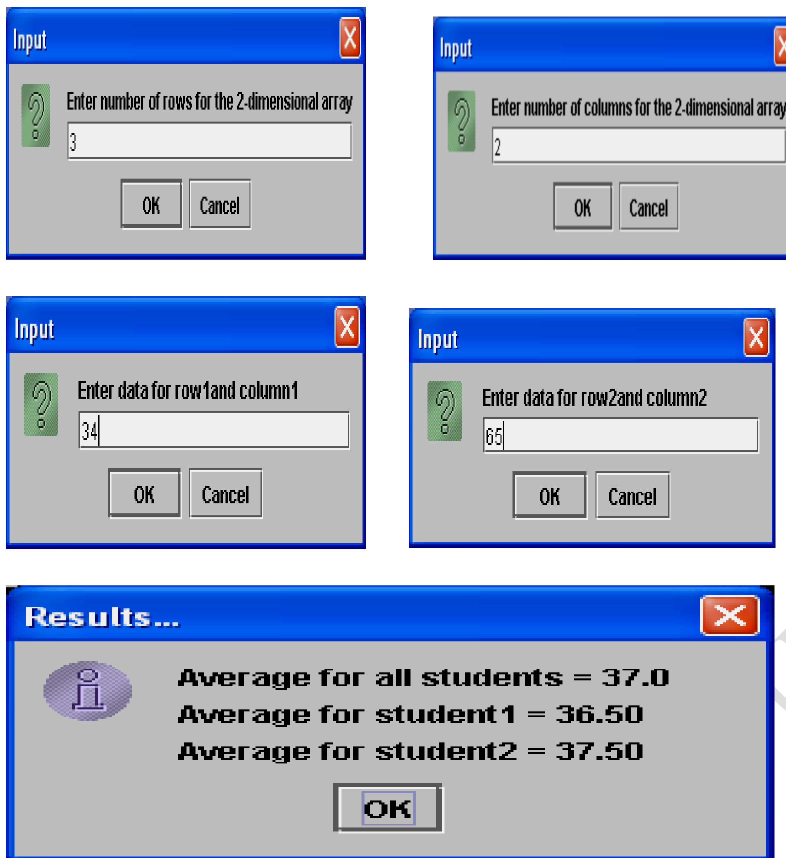
PROPERTIES OF DLC UI, IBADAN

As an illustration, consider the following program

```
1.  import javax.swing.*;
2.  import java.text.DecimalFormat; // to format our output
3.  class array2D {
4.      public static void main(String[ ] args)   {
5.              float[ ][ ] score;  //declaring a 2-dimensional array
6.
7.              //Getting the number of rows and columns for the array
8.              int r = Integer.parseInt(JOptionPane.showInputDialog
                        ("Enter number of rows for the 2-dimensional
                        array"));
9.              int c= Integer.parseInt(JOptionPane.showInputDialog
                       ("Enter number of columns for the 2-dimensional
                         array"));

10.             // Giving the array the number of rows and columns
11.             score = new float[r][c];
12.             float sum = 0;
13.
14.             // Data capturing into the array
15.             for (int i = 0; i < r; i++){ //Outer loop for rows
16.                   for (int j = 0; j < c; j++) { // Outer loop for columns
17.                         score[i][j] = Float.parseFloat(JOptionPane.
18.                                 showInputDialog("Enter data for row" +
                                            (i + 1) + "and column"  + (j + 1)));
19.                         sum += score[i][j]; //Accumulating data into sum
20.                    } //Ending innermost for loop
21.             }//Ending outermost for loop

22.             // Computing the average score for the whole data
23.             double mean = sum/(r*c);
24.             String output = "Average for all students = " + mean +
25.                                 "\n";

26.             // Computing the average score for each student
27.             //Note the position of the initialized accumulator sum
28.             for (int i = 0; i < r; i++)   {
29.                   sum = 0;
30.                   for (int j = 0; j < c; j++) {
31.                         sum += score[i][j];
32.                   }  //Ending innermost for loop for a row
33.                   double avg = sum/c;
34.
35.                    // rounding up avg to 2 places of decimals
36.                   DecimalFormat twoDigits = new DecimalFormat
37.                                                         ("0.00");
38.                   AvgRnd = twoDigits.format(avg);
39.                   output += "Average for student" +(i+1)+ " = " +
40.                               AvgRnd + "\n";
41.               } //Ending outermost loop to process another row of  data

42.             //Printing the final result
43.             JOptionPane.showMessageDialog(null, output,
44.                  "Results…",   JOptionPane.INFORMATION_
45.                      MESSAGE);
46.             System.exit(0);
47.          } //Ending the main method
48.      } // Ending the class array.
```

PROPERTIES OF DLCUI, IBADAN

Sample Output Screen Shots:



Note:

(i) We use i<r and i<c in the for loops, instead of <= to avoid array out-of-bound-exception, i.e. We don't want to go beyond the array's dimension, since java array indexing starts from zero. Again, we try to use the conventional means of accessing array, not using the array.length method.

(ii) To do any arithmetic computation inside a string, we need to put the expression inside a bracket within the string. For instance, we have (i+1) inside the string of showInputDialog and that of the output. Can you guess the reason why we use (i + 1) and (j + 1) instead of ordinary i and j in the showInputDialog?

(iii) Note the effect of the DecimalFormat method. It is a powerful method from the Text class which can be used to format our output to some number of decimal places in Java.

(iv) The two-dimensional array used in this program is assumed to contain scores of some students in rows in some courses done in columns.

	Course1	Course2	Course3	Course4	Course5
Student1	45	67	90	76	45
Student2	60	69	59	40	47
Student3	79	58	60	32	57
Student4	56	34	70	36	49

(v) In computing the average score of each student, the accumulator sum is initialized to 0 and placed in between the two for loops. Can you guess a reason for this? This is done so that when the innermost for-loop is completed and average for that row of data computed and reported, then the accumulator will be re-initialized back to zero for the next set of data for

PROPERTIES OF UI, IBADAN

another student. Otherwise, the previous value in the accumulator sum will be carried over to the next set of iteration of the outermost for-loop.

Practical Exercise:

(i) Try to copy and execute the program and state your observations.

(ii) How would you modify the program above to compute the average for each column, that is, average score per course?

A two dimensional array is actually an array of arrays. Each row is a separate array. Consider the program below, which illustrates this point.

Example:

```
class test {
 public Static void main (String args[]) {
   int [ ][ ] a =  new int [7][9];
   System.out.println (“a. length  = ” + a.length);
   System.out.println (“a[0].lenth = ” + a[0].length;
 }
}
```

Output

a.length = 7
a[0].length = 9

As an array, the object ‘a’ has length 7. That’s because it is really an array of 7 rows. The first of these row arrays is a[0]; having length 9. Each row has 9 elements. We can also initialize a 2-dimensional array like a 1-dimensional array. The only difference is that since it is an array of arrays; its initialization list has to be a list of initialization lists. For example, consider the case of a **ragged 2-dimensional array** below. Ragged because the lengths of its rows vary.

```
1.  class test2 {
2.  public static void main (String args[]) {
3.              int  [ ][ ] a =  { { 77, 33, 88},
                                   {11, 55, 22, 99},
                                   {66, 44}
                                 };
4.              for (int i = 0; i < a.length; i++) {
5.                  for (int j = 0; j< a[i].length; j++)
6.                      System.out.print (“\t” + a[i][j]);
7.                 System.out.println( );
8.              }
9.          }
10.     }
```

The output is

```
77      33      88
11      55      22      99
66      44
```

Note: The use of nested for loops, which is the standard way to process 2-dimensional arrays. The outside loop is controlled by the row index i, and the inside loop is controlled by the column index j. The row index i increments until it reaches a.length, which is 3 in this example. For each value of i, the column index j increments until it reaches a[i].length, which in this example is 3 when i=0, 4 when i = 1 and 2 when i = 2.

PROPERTIES OF DLC UI, IBADAN

Processing 3-dimensional array is similar. Three nested for loops are used:

```
for (int i = 0; i < a.length; i++){  // plane i
 for (int j = 0; j < a[i].length; j++) { // row j
  for (int k = 0; k < a[i][j].length; k++) { // column K.
               // process a[i][j][k] element
```

The element is analogous to a single letter on a line on a page in a book: a[i][j][k] would represent character number k on line number j on page number i, a[i][j] would represent line number j on page number i and a[i] would represent page number i. The number of characters on line j on page i would be a[i][j].length and the number of lines on page i would be a[I].length. So, iteration of the first loop would process page [i], iteration j of the second loop would process line a[i][j] and the iteration k of the third loop would process character a[i][j][k].

Exercise: Multiplication of Matrices

Two matrices A_{mn} and B_{np} could be multiplied to produce the third matrix C_{mp}. The ijth element of C is computed as follows:

$$C_{ij} = \sum^{n}_{k=1} A_{ik} B_{kj} \quad \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, p$$

The following algorithm may be used to solve this problem:

```
for (i =1; i <= m; i++) {
    for (j =1; j < = p; j++) {
                    C[i][j] = 0;
                    for (k =1; k < = n; k++) {
                              C[i][j] = C[i][j] + A[i][k] + C[k][j];
                    }
           }
 }
```

Exercise: Translate this algorithm into a java code.

6.4.9 Sorting and Searching an Array – Intrinsic functions

Java provides a predefined sort method in its Application Programming Interface (API). The Arrays.sort(arrayName) method could be invoked on a one-dimensional array in order to sort it in ascending order. Note that java.util.* must be imported to your program before the method could be used. Example program below illustrates the use of the method.

```
1.  import javax.swing.*;
2.  import java.util.*;
3.  class array {
4.     public static void main(String[ ] args) {
5.            int n = Integer.parseInt(JOptionPane.show
6.                 InputDialog("Enter the size of the array"));
7.            int [ ] a = new int[n];
8.            for (int i = 0; i < a.length; i++)
9.                  a[i] = Integer.parseInt(JOptionPane.show
10.                    InputDialog("Enter data for column"  + (i+1)));
11.
12.           //printing original array
13.           System.out.println("Original array data ..\n");
14.           for (int i = 0; i < a.length; i++)
15.                System.out.print("\t" + a[i]);
16.           System.out.println( );
17.
18.           // Sorting the array using a predefined sort procedure
19.           Arrays.sort(a);
```

PROPERTIES OF DLC UI, IBADAN

```
20.
21.            //Printing sorted array ..
22.            System.out.println("Sorted array data ..\n");
23.            for (int i = 0; i < a.length; i++)
24.                System.out.print("\t" + a[i]);
25.
26.            System.out.println( );
27.            System.exit(0);
28.        }
29.    }
```

Sample Output

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Original array data ..

        32      90      12      87      56
Sorted array data ..

        12      32      56      87      90
Press any key to continue...
```

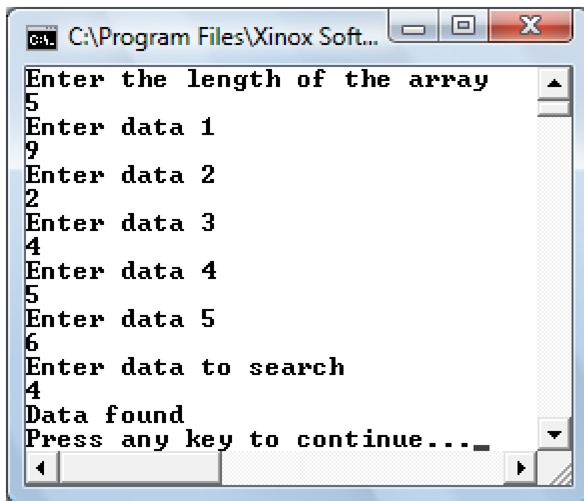
Practical Exercise

The method, ***Arrays.sort(arrayName)*** can only sort an array in ascending order, implement a program that will sort an array in descending order using bubble sorting algorithm or any other you know.

Java also provides an intrinsic function to do a binary search on a linear array, using the **Arrays.binarySearch(ArryaName, x)** function. The function can only work on sorted array and that is why we need to firstly apply Arrays.sort(ArrayName) function to the array. The ArrayName is the name of the array we are working with while x is the data we want to search on the array. The Arrays.binarySearch(A, x) function returns 1 if the data (x) is found in the array and zero (0) otherwise. Examine this aspect critically in Lines 17 – 20 in the code below.

```
1.  import java.util.*;
2.  class binasearch {
3.      public static void main(String args[ ]) {
4.          Scanner inp = new Scanner(System.in);
5.          System.out.println("Enter the length of the array");
6.          int n = inp.nextInt( );
7.          int A[ ] = new int[n];
8.          for (int i = 0; i < n; i++) {
9.                  System.out.println("Enter data "+ (i+1));
10.                 A[i] = inp.nextInt( );
11.         } // next i
12.
13.         System.out.println("Enter data to search ");
14.         int x = inp.nextInt( );
15.
16.         Arrays.sort(A);
17.         if(Arrays.binarySearch(A, x) == 1)
18.                 System.out.println("Data found") ;
19.         else
20.                 System.out.println("Data not found");
21.     }
22. }
```

PROPERTIES OF DLC UI, IBADAN



6.4.10 Java Vectors

Vectors are expandable and contractible array of objects. The size of a vector can change at will by the programmer. Below are some of the operations that can be performed on vectors:

6.4.10.1 Declaring a Vector

The following simple line of code would create a vector for you:

Vector v = new Vector();

Note that V in Vector is upper-case. This constructor creates a default vector containing no elements. Two technical terms commonly used with vectors are size and capacity. The size of a vector is the number of elements currently stored into it. Whereas the capacity of a vector is the amount of memory allocated to hold elements, and it is always greater than or equal to the size.

To specify the capacity of a vector, we declare the vector as:

Vector v = new Vector (25);

This means the vector will support 25 elements. But vectors can be expanded! Therefore, for the vector to accept more data, we will specify the factor of expansion. So we could declare the vector like this:

Vector v = new Vector(25,5);

This vector has an initial size of 25 elements and expands in increments of 5 elements when more than 25 elements are added to it. That means the vector jumps to 30 elements in size, and then 35, and so on.

6.4.10.2 Accessing Elements in a Vector

Just like in arrays, we use the square brackets (“[ ]”) to achieve this operation in vectors.

PROPERTIES OF DLC UI, IBADAN

6.4.10.3 Adding Elements to the Vector

We use the add( ) method to add an element to a vector. For example, the codes below shows how to add some strings to the vector v:

```
v.add("Akinola");
v.add("S. O.");
v.add("Ibadan");
```

6.4.10.4 Retrieving the Last Element From a Vector

The following code will retrieve the last element added to the vector v:

```
String s = (String)v.lastElement();
```

So, s will be assigned 'Ibadan' at the end of the operation. Note the casting into String (the keyword String is put into bracket) for the vector. This is because a vector is designed to work with the object class.

6.4.10.5 Retrieving elements at a particular index from a Vector

The get( ) method helps us to do this:

```
String s1 = (String)v.get(0);
String s2 = (String)v.get(2);
```

Because vectors are zero-based in indexing, the first statement retrieves "Akinola" and the second retrieves **"**Ibadan**".**

6.4.10.6 Add and remove elements at a particular index

```
v.add(1, "Solomon");
v.add(0, "Olalekan");
v.remove (3);
```

The first **statement** inserts "Solomon" at index 1, between "Akinola" and "S. O." strings. The second statement inserts "Olalekan" at index 0, which happens to be the beginning of the vector. All existing elements will be moved up to accommodate the new elements inserted in the appropriate positions.

The resulting vector will look like below:

- Olalekan
- Solomon
- Akinola
- S. O.
- Ibadan

The call to remove( ) at index 3 in line 3 will cause "S. O." to be removed from the vector. Therefore, the final list in the vector will now be:

- Olalekan
- Solomon
- Akinola

PROPERTIES OF DLC UI, IBADAN

- Ibadan

Removing the element based on the element itself rather than on an index can be achieved via the statement below:

```
v.removeElement(”S. O.”);
```

6.4.10.7 Overwriting elements at a particular index in a Vector

The following statement will change “Solomon” to “Olasunkanmi” in the vector v using the set() method.

```
v.set(1, “Olasunkanmi”);
```

6.4.10.8 Deleting all elements from a Vector

Use the following code to clear all the elements from the vector v.

```
v.clear();
```

6.4.10.9 Searching for a Particular Element in a Vector

For instance, to check if “Ade” is in the vector list, we write

```
boolean isThere = v.contains(“Ade”);
```

To find the index of a particular element, we write

```
int i = v.indexOf(“Akinola”);
```

The indexOf returns the position of “Akinola” in the vector if it is actually there, else it returns -1.

Summary

In this Chapter, you have been introduced to the declarations and use of arrays in Java. We were able to see that an array can be used to manipulate a large set of related data that are of same type and format.

Post Test

1. Write a program segment to compute the average value of all data stored in even positions (2, 4, 6, …) inside a one-dimensional array, assuming data has been stored into the array initially.

2. (a) *A linear search compares each element of an array with a search key.* Write a java code snippet that will return “found” with the index of an element if found in the array and “Not found” otherwise.
   - (a) Extend the program in 5(a) to compute the sum of all data which are in even indexes (2, 4, 6…) in an array.
   - (b) Extend the program in 5(a) to reverse all the elements in the array; such that element in the last index will now occupy the first index in that order.

PROF... S OF DLC UI, IBADAN

7 *Java Strings Manipulations*

7.1 Introduction

A string is a sequence of characters. Words, sentences and names are strings. This chapter introduces you manipulations of strings in Java programming language.

7.2 Objectives

At the end of the chapter, you should be able to
(i) Identify string objects and their functions
(ii) manipulate strings in your program

7.3 Pre Test
1. Give two examples of string.
2. How would you determine the length of the strings.
3. How would you reverse the strings

7.4 Main Content

7.4.1 The String Class

The simplest type of string in Java is an object of the string class and cannot be changed.

7.4.2 Operations on Strings

(1) **Length of Strings:** gives the total number of characters in a string. Consider the program segment below:

```
{
  String str = “I am now a student”;
  int strLength = str.length( );
   System.out.println(“Length of the string given is is ” +     strLength);
 }
```

Output: Length of the string given is is 18

Note that
- (i) space is a character in a string.
- (ii) the length method used in this case as different from that of the array. Length method in strings has the opening and closing brackets as suffix, length( ), i.e., it is a method without any argument passed into it.

(2) **Getting the Character at a Particular Point or Index**. Note: Java starts its counting from 0. Using the example above

```
String strPosition5 = str.charAt(5);
```

Solution here is letter n

(3) **Getting the Index of a Character:** The index of a character is the position of the character in a string. Zero indexing must be observed. There are 18 characters in the string.

```
String stringCharIndex = str.indexOf(‘m’); //solution, 3
```

PROPERTIES OF DLC UI, IBADAN

(4) Getting the Hash code for a String

The hash code gives the unique integer value for a string. The number or code has no meaning other than serving as a numeric label for the object. Hash values are used as storage locators when the objects are stored in tables. Although each string object has one and only one hash code, the same number may be the hash code for more than one object. This operation is commonly used in compiler construction.

```
String stringHashCode = str.hashCode();
```

(5) Getting Substrings

A substring is a string whose characters form a contiguous part of another string. The Substring() method extracts substrings from strings. Two arguments are normally passed into this method, the lower and upper indices. To get this method clear, we have to find the difference between the two indices, say 8 – 4 = 4 as in the example below. This now means that we are extracting a total of 4 characters starting from the lower index (4) upwards.

For example:

```
String str = “I am now a student”;
String substringFrom4to8 = str.substring(4,8);
Solution:  now   // note that a space is at the prefix of the string return

String substringFrom4to4 = str.substring(4,4);
Solution: nil
```

Note: The substring keyword must be in small case letters throughout.

(6) Changing Case of a String: Case has to do with small or bigletters such as a, A, b, B. We can change all the characters in a string to either big or small letters.

```
String country = “NIGEria”;
String lower = country.toLowerCase( );// solution: nigeria
String upper = country.toUpperCase( ); //solution: NIGERIA
```

(7) Concatenation: We have dealt with this in our past examples. We use + symbol to concatenate or join strings together. Java also use the *concat* method for this operation. For example:

```
String a1 = “I am coming ”;
String b = “to you soon”;
String c1 = a1 + b;
String c2 = a1.concat(b);
```

solution: both c1 and c2 contain *I am coming to you soon*

(8) Searching for Characters in a String: We can search for a particular character in a string. Also, we can look for other positions or indexes of the character in the string. Consider the example below:

```
String str = “Arise to sing a song”;
// To get the first index of s
int i = str.indexOf(‘s’);
system.out.println(“First index of ‘s’: ” + i);
//To get the next index of s
int j = str.indexOf(‘s’, i  + 1);
```


PROPERTIES OF DLC UI, IBADAN

```
system.out.println("Next index of 's': " + j);
int k = str.indexOf('s', j + 1);
system.out.println("Next index of 's':" + k);
//To get the last index of s
k = str.lastIndexOf('s');
system.out.println("Last index of 's':" + k);
```

(9) Replacing Characters in a String. Using the replace() method,we can substitute a character with another character in a string. The example below illustrates this:

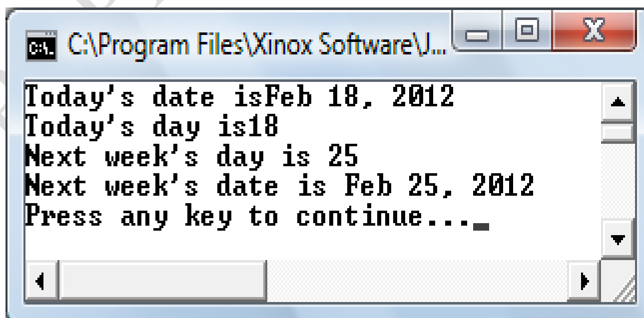
```
public class Replacing {
     public static void main (String[ ] args) {
         String inventor = "Charles Babbage";
                         system.out.println(inventor.replace('B','C'));
              }
}    //      Output: Charles Cabbage
```

(10) Converting Strings into Primitive Types:

The program below shows how to do arithmetic on numerical values that are embedded within a string. Study, compile and execute this program and report your observations.

```
public class TextConversion {
 public static void main (String[ ] args) {
     String today = "Feb 18, 2009";
     String todaysDayString = today.substring(4,6);
    int todaysDayInt = Integer.parseInt(todaysDayString);
    int nextWeeksDayInt = todaysDayInt + 7;
    String nextWeek = today.substring(0,4) + nextWeeksDayInt +
     today.substring(6);
    System.out.println("Today's date is" + today);
    System.out.println("Today's day is" + todaysDayInt);
    System.out.println("Next week's day is " +
     nextWeeksDayInt);
    System.out.println("Next week's date is " + nextWeek);
  }
}
```

The Output:



- The parseInt() method (defined in the integer class) reads the two characters '1' and '8' from the todaysDayString string, converts them to their equivalent numerical values 1 and 8, combines them to form 18 and then returns that int value.

PROPERTIES OF DLC UI, IBADAN

11. Comparing Strings Using the Methods equals, equalIgnoreCase, compareTo and regionMatches

Consider the following segment

```
String s1 = new String("Akinola");
String s2 = "good day";
String s3 = "Happy Day";
String s4 = "happy day";
```

If we wanted to test for equality, we could use the method ***equals***. Method equals tests any two objects for equality, that is, if the strings contained in the two objects are identical. The method returns true if they were and false otherwise. This method uses a lexicographical comparison. The integer Unicode values that represent each character in each string are compared. Thus the statement;

```
if (s1.equals("akinola"))
```

will return false. Note that java is case sensitive.

We use "==" operator to compare object references to determine whether they refer to the same object; not whether two objects have the same contents. However, if we write the following code:

if (s1 == "Akinola"), the condition will be true.

Method ***IgnoreCase*** ignores the case of the letters in each string. Thus, if we write the following code:

```
if (s3.equalsIgnoreCase(s4))
```

the condition will be true since the cases of the letters does not matter in "Happy Day" and "happy day" strings. This method is ideal for sorting strings.

Method ***compareTo*** returns zero (0) if the strings are equal, a negative number if the string that invokes compareTo is less than the string that is passed as an argument and a positive number if otherwise. That is, the invoking string is greater than the argument. The method uses a lexicographical comparison to compare the numeric values of corresponding characters in each string. For instance, the code:

```
String output = s3.compareTo(s4);
```

returns a negative number to output. But s4.compareTo(s3) will return a positive number. Note that method compareTo is case-sensitive. Consider the following code, which sorts words in a string in an ascending order using a bubble sort technique. Method split( ) is used in the code to split a string into tokens and converts the string into a string array. This is explained further in section 17 of this chapter.

Sorting Words in a String

```
1.  import javax.swing.*;
2.  class str {
3.      public static void main(String[ ] args) {

4.          // The program accepts a string, converts the string into a string
5.          //array Using the split method
6.          // Sorts the string in ascending order
7.
```

PROPERTIES OF DLC UI, IBADAN

```
8.          // reading the string
9.          String s =JOptionPane.showInputDialog("Enter a string of
10.               any length");
11.         System.out.println("\n The original string is \n\n" + s);
12.
13.         // Splitting the string into a string array.
14.         // Space is used as the token-separator in the string
15.         String[ ] sArray = s.split(" ");
16.
17.         // sorting with bubble sort technique, using the compareTo( ) method
18.         for (int i = 0; i< sArray.length; i++) {
19.              for (int j = (i + 1); j < sArray.length; j++) {
20.                   int k = sArray[i].compareTo(sArray[j]);
21.                   // compareTo( ) returns a negative, zero or positive integer
22.                   // if compareTo( ) returns a positive integer, swap the elements position
23.                   if (k > 0) {
24.                        String temp = sArray[i];
25.                        sArray[i] = sArray[j];
26.                        sArray[j] = temp;
27.                   } //end if
28.              } // next j
29.         } //next i
30.
31.         //Printing sorted string
32.         System.out.println("\n The sorted words in the string are\n");
33.         for (int i = 0; i < sArray.length; i++)
34.              System.out.println(sArray[i]);
35.         System.exit(0);
36.     }
37. }
```

```
C:\Program Files\Xinox Software\JCreat...

 The original string is

olu sayo john usman lucky lekan akinola

 The sorted words in the string are

akinola
john
lekan
lucky
olu
sayo
usman
Press any key to continue..._
```

Practical Exercise: Extend the above program to search for a particular word in a string.

Method ***regionMatches*** compares portions of two strings for equality. The first argument is the starting index in the string that invokes the method. The second argument is a comparison string. The third is the starting index in the comparison string and the last is the number of characters to compare between the two strings. The method only returns true only if the specified number of characters are lexicographically equal. For example,

```
String s3 = “Happy Day”;
String s4 = “happy day”;

if (s3.regionMatches(0, s4, 0, 5))
  output += “First 5 characters of s3 and s4 match\n”;
        else
           output += “First 5 characters of s3 and s4 do not match\n”;
```

PROPERTY OF DLC UI, IBADAN

Note that this method is case sensitive. In this case, the else path will be followed. However, if we want to ignore cases, then the code would be written as follows:

```
if (s3.regionMatches(true, 0, s4, 0, 5))
  output += "First 5 characters of s3 and s4 match\n";
        else
          output += "First 5 characters of s3 and s4 do not match\n";
```

In this case, the first five characters in the two strings matches.

12. Other Comparison Methods – startsWith and endsWith

Just as the words indicate, the methods test if a string starts or ends with some sets of characters. We could specify the position or index where we want to start the comparison on the strings. Consider the following code segments:

```
1.  public class str{
2.      public static void main(String[ ] args) {
3.          String strings[ ] = {"Akin", "Akinola", "Akintola", "Akinsola"};
4.          String output = " ";
5.
6.          for (int k = 0; k< strings.length; k++)
7.              if (strings[k].startsWith("Ak"))
8.                  output += strings[k] + " starts with Ak\n";
9.
10.         // test method startsWith starting from a position
11.         for (int k = 0; k < strings.length; k++)
12.             if (strings[k].startsWith("ol", 5 ))
13.                 output += strings[k] + " starts with ol at position 5 \n";
14.
15.         // test method endsWith
16.         for (int k = 0; k < strings.length; k++)
17.             if (strings[k].endsWith("la"))
18.                 output += strings[k] + " ends with la\n" ;
19.
20.         System.out.print(output);
21.     }
22. }
```

Output

```
Akin starts with Ak
Akinola starts with Ak
Akintola starts with Ak
Akinsola starts with Ak
Akintola starts with ol at position 5
Akinsola starts with ol at position 5
Akinola ends with la
Akintola ends with la
Akinsola ends with la
```

13. Reversing a String

Characters in a string can be reversed and printed in the reverse order. This is a way of data encryption. A counting down for-loop could be used to achieve this. For example,

```
public class sR {
        public static void main(String[ ] args) {
```

PROPERTIES OF DLC UI, IBADAN

```
        String string = "My name is Akinola";
        for (int i = string.length( )-1; i>=0; i--)
            System.out.print(string.charAt(i));
    }
}
```

Output

alonikA si eman yM

14. **Trimming a String:** The trim() method is normally used to remove leading or trailing white spaces before and after a string. For example:

```
String s1 = "   Akinola  ";
String s2 = s1.trim()
// no more spaces before and after Akinola
```

15. Converting a String into an Array using toCharArray() method

The code below creates a new character array ontaining a copy of the characters in string s1 and assigns a reference to variable charArray.

```
String s1 = new String("welcome");
//conversion to character array
char charArray[ ] = s1.toCharArray();

// to access characters in the charArray, we need to go by the way of array.

for(int j = 0; j < charArray.length; ++j)
  System.out.println(charArray[j]);
```

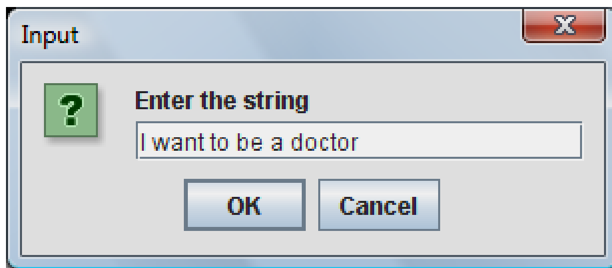
16. **String Tokenizer:** StringTokenizer class is a class from package java.util, which breaks a string into its component tokens. Tokens are individual words and punctuations, each of which conveys meaning to a reader of a sentence. Tokens are separated from one another by *delimiters*, such as space, tab, new line and carriage return. This class would be useful in compiler's tokenization. Compilers breaks up statements into individual pieces like keywords, identifiers, operators and other elements of a programming language. The following program illustrates string tokenization in Java.

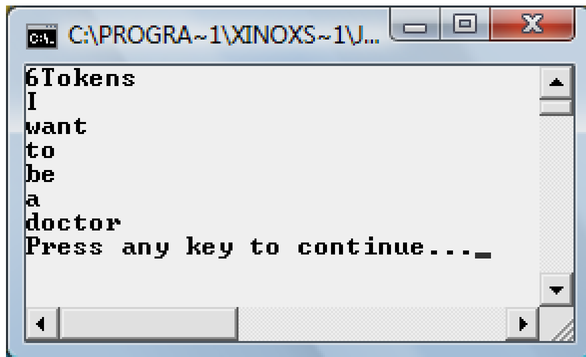
Consider this small code:

```
1.  import java.util.*;
2.  import javax.swing.*;
3.  class Tokenizer  {
4.      public static void main (String[ ] args){
5.          String str = JOptionPane.showInputDialog("Enter the string");
6.          StringTokenizer tokens = new StringTokenizer(str);
7.          System.out.println(tokens.countTokens( )+"Tokens");
8.          String [ ] words = new String[tokens.countTokens( )];
9.          while (tokens.hasMoreTokens( )){     // Print the tokens
10.             System.out.println(tokens.nextToken( ));
11.         } // end while
12.     }
13. }
```

Output

PROPERTIES OF DLC UI, IBADAN





17. String Splitting

Split( ) is a method that can be used to split a string into its constituent words and then stored them into a string array. The parameter to be passed into the split( ) will be the delimiter used to separate the words in the string, for example, space. Examine the code below, which is another way of implementing a tokenizer. The program also counts the number of occurrence of the word 'the' in the string given.

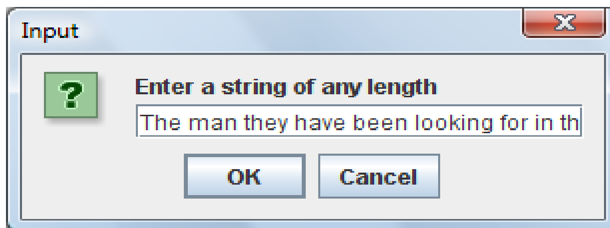
Using Split( ) method

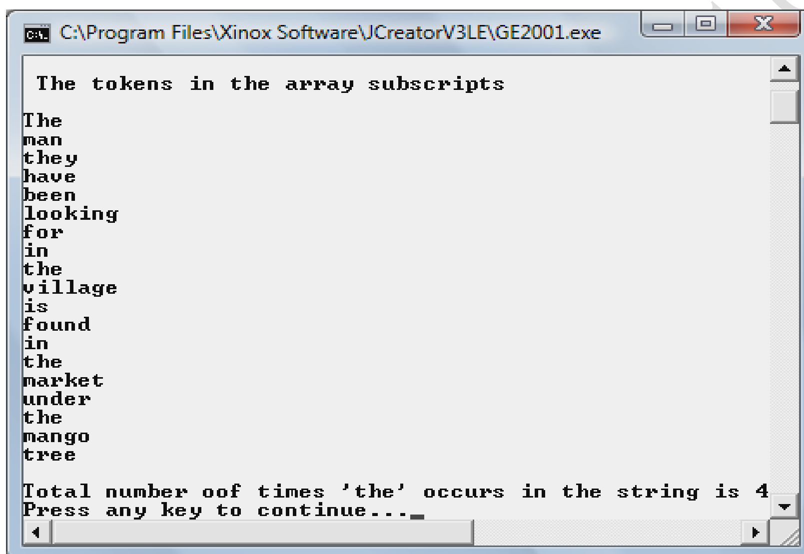
```
1.  import javax.swing.*;
2.  class str {
3.      public static void main(String[ ] args) {
4.              // The program accepts a string, converts the string into a
5.                  //string array Using the split method
6.          //prints the words / tokens in the string and finally checks for
7.          //how many times 'the' occurs in the  string
8.
9.          // reading the string
10.         String s =JOptionPane.showInputDialog("Enter a string of
11.                 any length");
12.         System.out.println("\n The original string is \n\n" + s);
13.
14.         // Splitting the string into a string array.
15.         // Space is used as the token-separator in the string
16.         String[ ] sArray = s.split(" ");
17.
18.         //Printing out the length of the string array
19.         System.out.println("Number of words in the array is "+
20.             sArray.length);
21.         //Printing out the tokens
22.         System.out.println("\n The tokens in the array
23.                 subscripts\n");
24.         for (int i = 0; i < sArray.length; i++)
25.             System.out.println(sArray[i]);
26.
27.         //Checking for 'the' occurrence
```

PROPERTIES OF DLC UI, IBADAN

```
28.           int count = 0;
29.           for (int i = 0; i < sArray.length; i++) {
30.                 if (sArray[i].equalsIgnoreCase("the"))
31.                     count++;
32.             } // next i
33.
34.             System.out.println("\nTotal number oof times 'the'
35.               occurs in the string is " + count);
36.
37.              System.exit(0);
38.           }
39.         }
```

Sample Output



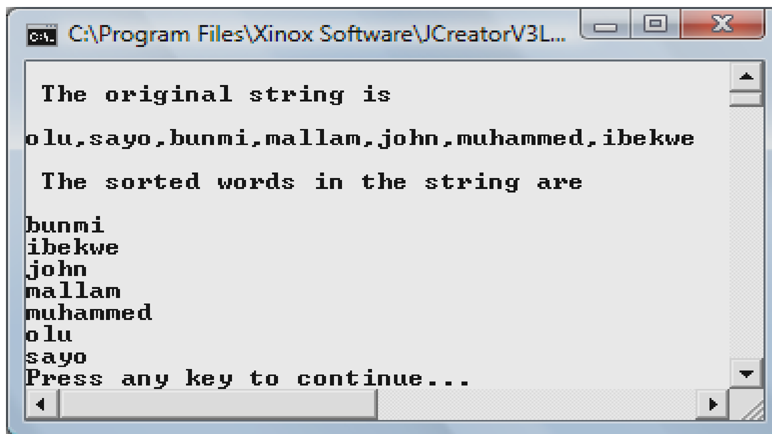


In the above code, space was used to separate the tokens in the string. Sometimes we may want to use a comma separated string in which comma (,) will be used to separate the tokens in the string. We only need to change the split( ) argument to comma, such as

```
String[ ] sArray = s.split(",");
```

The sorting code is re-run here but with a comma separated tokens. Only the output is shown here.

UI, IBADAN



18. StringBuffer Objects: normally used for string objects that need to be changed. Consider the example below, which creates only one object, buf, which is then modified several times using concatenation operator and the append() method:

```
public class TextAppending {
  public static void main (String[ ] args) {
    StringBuffer buf = new StringBuffer(10);
    buf.append("It was");
    System.out.println("Initial string is:" + buf);
    buf.append("the best");
    System.out.println("Final string is: " + buf);
  }
}
```

Output:
Initial string is: It was
Final string is: It was the best

19. Replacing StringBuffer Objects. Using the simple program above, write another line of code after the last line as:

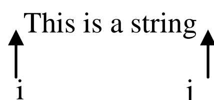
```
buf.SetCharAt(1,"s");
System.out.println("Replaced statement is:" + buf);
```

Output: Is was the best

Practical Example: Palindrome

A palindrome is a string that can be spelt same way forwards and backwards. The algorithm for a palindrome is stated here.

1. Two indexes i, and j are set at the extreme ends of the string respectively, Line 7 in the code below.



5. Index i is incremented by 1 to the right while index j is decremented by 1 to the left (Lines 9-13).
6. The Characters they are pointing to are compared. If they are the same, step number 2 is repeated to other set of characters in the string (Line 9).

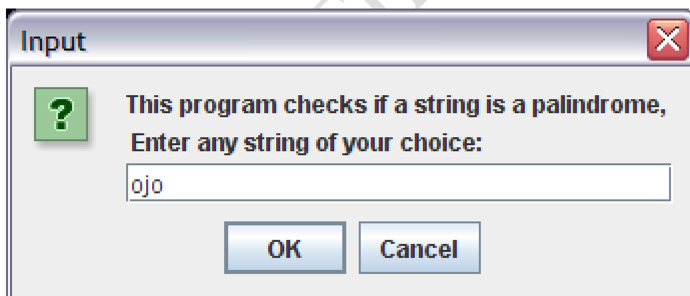
PROPERTIES OF DLC UI, IBADAN

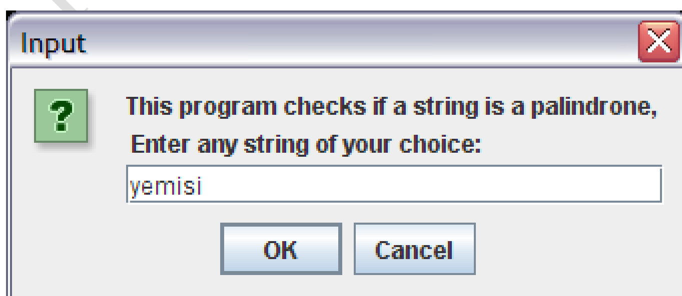
7. If at any point, the characters being pointed to by the indexes differs, the program reports that the string is not a palindrome (Lines 15-18). But when the two indexes meet on a character, the string is a palindrome (Line 22).

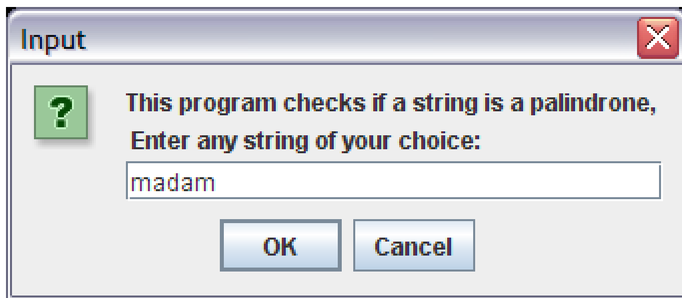
Here is a palindrome program that checks if a string is spelt same way forwards and backwards:

```
1.  import javax.swing.JOptionPane;
2.  class  palind {
3.          public static void main(String[ ] args) {
4.              String a =  JOptionPane.showInputDialog("This program
5.                     checks if a string is a palindrome, \n Enter any
6.                     string of your choice: ");
7.              boolean stop = false;
8.              int i = 0, j = a.length( )-1;
9.              while (i < a.length( ))   {
10.                 if (a.charAt(i) == a.charAt(j))  {
11.                     i++;
12.                     j--;
13.                     stop = true;
14.                     continue;
15.                   } // end if
16.                   else {
17.                          System.out.println(a + " is not a
18.                                 palindrome");
19.                          break;
20.                        } // end else
21.
22.                 }// end while
23.                if (stop = = true)  {
24.                    System.out.println(a + " is a palindrome");
25.                 } //end if
26.                System.out.println("Good bye to palindrome user");
27.             }// end main
28.         }//end class
```

Sample Output








```
Command Prompt
C:\jdk1.6.0_02\bin>java palind
ogunseye is not a palindrone
Good bye to palindrone user

C:\jdk1.6.0_02\bin>java palind
madam is a palindrone
Good bye to palindrone user

C:\jdk1.6.0_02\bin>java palind
bello is not a palindrone
Good bye to palindrone user

C:\jdk1.6.0_02\bin>java palind
dejosalawu is not a palindrone
Good bye to palindrone user

C:\jdk1.6.0_02\bin>java palind
aaajjjaaa is a palindrone
Good bye to palindrone user

C:\jdk1.6.0_02\bin>java palind
bbacxouity is not a palindrone
Good bye to palindrone user

C:\jdk1.6.0_02\bin>
```

A more efficient code is shown below

```
1.  import javax.swing.*;
2.  class palindrome{
3.        public static void main(String[ ]args){
4.              String str = JOptionPane.showInputDialog("Enter sting to
5.                      check");
6.              int lengthstr = str.length( );
7.
8.              String reverse = "";
9.              for (int i=lengthstr-1;i >= 0 ;i-- )
10.                     reverse += str.charAt(i);
11.
12.             if (str.equalsIgnoreCase(reverse))
13.                     JOptionPane.showMessageDialog(null,"String
14.                         "+str+" is a palindrome");
15.             else
16.                     JOptionPane.showMessageDialog(null,"String
17.                         "+str+" is not palindrome");
18.         }
19.     }
```

UI, IBADAN

Explain how this above code achieves the principle of palindrome.

Summary
The chapter has introduced you to strings and their manipulations in Java. Strings are very useful in Text mining and especially in Bioinformatics. Find out about these applications.

Post Test

1. Consider the following program segment/snippet:

   String univer = "My School is very good";
   String Uni = "my school is very Good";

   Write Java snippets to:
   a. Get the total length of the string univer.
   b. Compare if the two strings are equal, not minding letter cases.
   c. Print out only "University is good" from univer
   d. Reverse the string Uni
   e. Get the last index of 's' from the string univer

2. Write a program in Java that reads a line of text from the keyboard and displays it on the screen one word per line as well as writing on the screen the number of words in the text. The text contains letters and blanks only.

PROPERTIES OF DLC UI, IBADAN

8 *Java Methods*

8.1 Introduction

In this chapter, you will be introduced to the concept of modularity in Java, i.e. how Java handles subprograms or program modules. This is usually achieved by the use of “methods”; referred to as *functions* and *procedures* in some other languages.

8.2 Objectives

At the end of this chapter, you should be able to
(i) understand how programs could be broken down into modules or functions
(ii) know how to program functions (methods) in Java

8.3 Pre Test

Assuming you are interested in capturing some number of data, compute their sum, mean and standard deviation, state all the possible modules you may break the program into.

8.4 Main Content

8.4.1 What is a Method?

Methods are techniques employed for breaking large programs into small functional modules. It is popularly called functions in some languages. These modules in java are called methods and classes. There are some built-in methods and classes such as mathematical calculations, string manipulation, error checking, etc. Methods are written to define specific tasks that may be used at any points in a program.

A method is invoked by a method call. A method call is characterized by its name and the information (arguments) that the called method need to do its task. When the method call completes, the method either returns a result to the *calling method* (or caller) or simply returns control to the calling method. We can then infer from this fact that the caller is not interested in the way the called method will perform the job given to it and the called method can also call another method to do some job for it. In the following assignment statements;

```
Y = Math.sqrt(x);
Y = Integer.parseInt(x);
```

Math.sqrt and Integer.parseInt are the method names and x is the argument in the above arithmetic expressions.

Methods in Java could be programmer-declared or prepackaged. The prepackaged methods are available in the *Java Application Programming Interface Java API* or *Java Class Library.* The java API provides a rich collection of classes that contain methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, error checking and many other useful operations. Methods allow the programmer to modularize a program by separating its task into self-contained functional units. The actual statements implementing the methods are written only once and are hidden from other methods.

PROPERTIES OF DLC UI, IBADAN

8.4.2 Advantages of Using Methods in Programs

(1) **Program manageability:** Methods and classes in java make program development more manageable.

(2) **Software reusability:** Using existing methods and classes as building blocks to create new programs is one of the today's programming paradigms. We create programs from standardized methods and classes rather than by building customized codes. For example, in earlier programs, we did not have to define how to convert strings to integers and floating-point numbers**;** Java provides these capabilities in class Integer (static method parseInt) and class Double (static method parseDouble), respectively.

(3) **Reduction in code repetition:** Repeating codes in programs is highly reduced using methods. Packaging code as a method allows a program to execute that code from several locations in a program simply by calling the method.

(4) **Ease of program maintenance and debugging:** Methods make programs easier to debug and maintain.

Note:
(i) To promote software reusability, each method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.
(ii) A method should usually be no longer than one printed page. Better yet, a method should usually be no longer than half a printed page. Regardless of how long a method is, it should perform one task well. Small methods promote software reusability.

8.4.3 The Math-Class Methods

The table below summarizes some of the predefined methods in the Math class:

Method	Description	Example
abs(x)	Absolute value of x	abs(23.7) = 23.7, abs(-20.5) = 20.5
ceil(x)	Rounds x to the smallest integer not less than x	ceil(9.2) = 10.0 ceil(-9.8) = -9.0
cos(x)	Cosine of x (x is in radians)	cos(0.0) = 1.0
exp(x)	Exponential method e^x	exp(1.0) = 2.71828
floor(x)	Rounds x to the largest integer not greater than x	floor(9.2) = 9.0 floor(-9.8) = -10.0
log(x)	Natural logarithm of x, (base e)	log(Math.E) = 1.0
max(x,y)	Larger value of x and y	max(2.3, 12.7) = 12.7 max(-2.3, -12.3 = -2.3
min(x,y)	Smaller value of x and y	min(2.3, 12.7) = 2.3
pow(x,y)	X raised to the power of y (X^y)	pow(2.0, 3.0)= 2^3 = 8
sin(x)	Sine of x (x is in radians)	sin(0.0) = 0
sqrt(x)	Square root of x	sqrt(900.0) = 30
tan(x)	Tangent of x, (x is in radians)	tan(0.0) = 0

To use any of the above-predefined methods, we need to append Math as subscript to them. For instance, we could write xSqrt = Math.sqrt(x-y);

Class Math also declares two commonly used mathematical constants: Math.PI and Math.E. The constant Math.PI (3.14159…) of class Math is the ratio of a circle's circumference to its diameter. The constant Math.E (2.71828…) is the base value for natural logarithms.

PROPERTIES OF DLC UI, IBADAN

8.4.4 Writing Your Own Methods

The general format of a method declaration is that we firstly write the *method header*, which is the first line. Following the method header, *declarations* and *statements* in braces form the *method body*, which is a block. Variables can be declared in any block and blocks can be nested, but a method cannot be declared inside another method. The basic format of a method declaration is

```
return-value-type method-name(para1, para2,…. paraN) {
  Declarations and statements
 }
```

Note *para* means parameter.

The *method-name* is any valid identifier. The *return-value-type* is the type of the result returned by the method to the caller (int, char, double, …). Methods can return at most one value.

The *parameters* are declared in a comma-separated list enclosed in parentheses that declares each parameter's type and name. There must be one argument in the method call for each parameter in the method declaration. Also, each argument must be compatible with the type of the corresponding parameter. For example, a parameter of type double can receive values of 8.56, 67 or –0.89567, but not "year" (because a string cannot be implicitly converted to a double variable). If a method does not accept any arguments, the parameter list is empty (i.e., the name of the method is followed by an empty set of parentheses). Parameters are sometimes called *formal parameters.*

There must be a one-on-one correspondence between the arguments in the method call and the parameter list in the method. For example, if we have the following in the method call avg(a,b,c), then x, y, and z will represent a, b, c, respectively in the following method declaration

```
static double avg(float x, float y, float z)  {
```

Parameters x, y and z in the method avg are sometimes called the dummy arguments or formal parameters, simply because they are placeholders for the real and / or actual parameters or arguments that are coming from the method call. There is no hard and fast rule about whether the parameters in the method call and that of the method are the same. But for good programming principle, we always use different nomenclatures (names) for them. What this means is that float x, float y, float z in the method could as well be written as float a, float b, float c.

There are two major ways of passing parameters into methods/functions in a program. When the actual values of variables are passed into a method, this is called *parameter passing by value.* Any changes on this data do not affect the original data. *Parameter passing by reference* is a two-way communication process. Usually the address of a variable is passed into a method and any changes made on the variable affects the original value of the variable in wherever it is located in memory. Array address passing is an example.

There are three ways of returning control to the statement that calls a method.

- ✓ If the method does not return a result, control returns when the program flow reaches the method's ending right brace;
- ✓ Or when the statement

  return;

  is executed; and
- ✓ If the method returns a result, the statement

  return *expression;*

  evaluates the *expression*, then returns the resulting value to the caller.

PROPERTIES OF DLC UI, IBADAN

When a return statement executes, control returns immediately to the statement that called the method. Below is an example to illustrate programmer-defined methods.

Example 1: A Method that Returns the Cube of an Integer

```
public class TestCube {
  public static void main(String[ ] args) {
          for (int i = 0; i < 6; i++)
             System.out.println(i + "\t" + Cube(i));
          }
          static int Cube(int n) {
                  return n * n * n;
          }
 }
```

Output

```
0	0
1	1
2	8
3	27
4	64
5	125
```

Note: Method Cube is invoked from the println method, with argument i, which represents the current value of i in the for-loop. Method call is Cube(i).

8.4.5 Where Can We Place Our Methods?

Methods could be placed at the end of the main method as was done in Example 1 above. That is, after we might have written all the codes for the main(String[ ] args) method.

Alternatively, we could write all our methods just after the declaration of the class header. Example 2 below illustrates this.

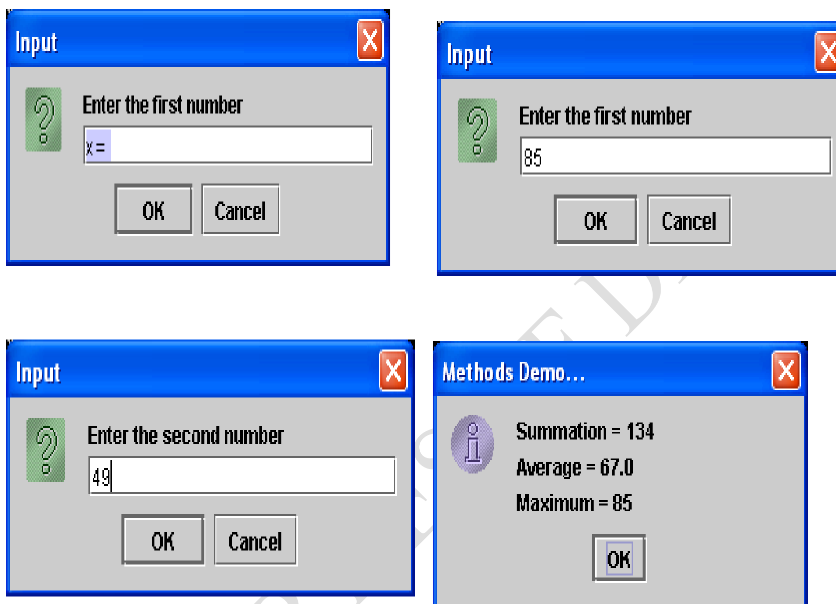
Example 2: Methods that return the sum, mean and maximum of 2 numbers

```
1.   import javax.swing.*;
2.   class sumMeanMax {
3.         // method to sum the two numbers
4.         static int sum(int a, int b) {
5.               return a + b;
6.          }
7.
8.          // method to find the average of the two numbers
9.         static double mean(int a, int b) {
10.        // add, declared below is a local variable, local to this method
11.        int add = sum(a,b);  // calling the sum method
12.        double avg = add/2.0;
13.        return avg;
14.       }
15.
16.      // method that returns the maximum of the two numbers
17.       static int maxim(int a, int b) {
18.              if (a > b)
19.                  return a;
20.              else
```

PROPERTIES OF DLC UI, IBADAN

```
21.                 return b;
22.       }// end method maxim
23.
24.       // The main method
25.       public static void main(String[ ] args) {
26.              int x =  Integer.parseInt(JOptionPane.showInputDialog("
27.                    Enter the first number", "x = "));
28.              int y =  Integer.parseInt(JOptionPane.showInputDialog
29.                    ("Enter the second number", "y = "));
30.              String output = "Summation = " + sum(x,y) + "\n" +
31.                "Average = " + mean(x,y) + "\n" + "Maximum = " +
32.                    maxim(x,y);
33.              JOptionPane.showMessageDialog(null,output,"Methods
34.                Demo...",  JOptionPane.INFORMATION_MESSAGE);
35.              System.exit(0);
36.          } // End main
37.       } // End class
```

Sample Output:



Note: Methods can call another method if it will need the method. An example is illustrated in the method mean above, repeated below:

```
8.        // method to find the average of the two numbers
9.       static double mean(int a, int b) {
10.      // add, declared below is a local variable, local to this method
11.      int add = sum(a,b);  // calling the sum method
12.      double avg = add/2.0;
13.      return avg;
14.      }
```

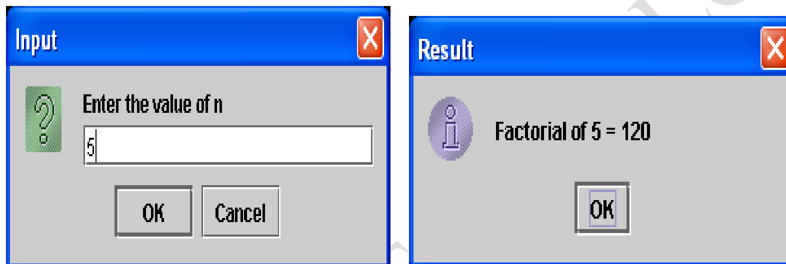
Also, a local variable has its scope only within the method that declares it. An attempt to reference add, which is declared in the mean method in another method will produce an error. Refer to Section 2.9 for further knowledge on this.

Exercise: Rewrite the Maxim method for any three numbers using the Math.max method.

PROPERTY OF ..., IBADAN

Example: Program that computes the factorial of n, method fac assumes that the factorial of any numbers less or equal to 1 is zero.

```
1.   import javax.swing.JOptionPane;
2.   class factorial {
3.       // The Factorial Method
4.       static long fac(int n) {
5.               if (n <= 1)
6.                    return 1;
7.               long f = 1;
8.               for (int i = n; i >= 2; i--)
9.                   f = f*i;
10.              return f;
11.        }
12.
13.        // Method Main
14.        public static void main(String[ ] args) {
15.              int n =   Integer.parseInt(JOptionPane.showInputDialog
16.              ("Enter  the value of n"));
17.              long fact = fac(n);  // fact declared as long precision data
18.              JOptionPane.showMessageDialog(null, "Factorial of " + n
19.                   + " = " + fact, "Result",
20.              JOptionPane.INFORMATION_MESSAGE);
21.              System.exit(0);
22.         }
23.    }
```



Exercise: Extend the above program to compute the combination of n and r, (nCr).

8.4.6 Passing Arrays into Methods

To pass an array argument to a method, specify the name of the array without any brackets. For example, if array score is declared as

```
int [ ] score = new int[5];
```

then the method call

```
swap(score);
```

passes a reference to array score to method swap. In Java, every array object "knows" its own length (via the length field). Thus, when we pass an array object into a method, we do not need to pass the length of the array as an additional argument.

Although entire arrays and objects referred to by individual elements of reference-type are passed by reference, individual array elements of primitive types are passed by value exactly as simple variables

PROPERTIES OF KDLC UI, IBADAN

are. To pass an array element to a method, use the indexed name of the array as an argument in the method call.

For a method to receive an array through a method call, the method's parameter list must specify an array parameter (or several if more than one array is to be received). For example, the method header for method arrayMean below is written as

```
static double arrayMean(int a[ ])
```

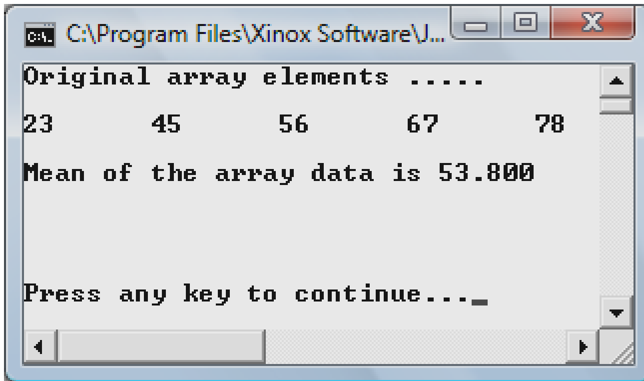
indicating that arrayMean expects to receive an integer array in parameter a. Since arrays are passed by reference, when the called method uses the array name a, it refers to the actual array (named a) in the calling method.

Example: The program below passes an array into a method and computes the mean value in the array

```
1.   import javax.swing.*;
2.   class arrayMethod {
3.          //method to compute the mean value in the array
4.          static double arrayMean(int a[ ]) {  // Entire array passed
5.               double sum = 0;
6.               for(int i = 0; i < a.length; i++){
7.                sum += a[i];
8.          } // next i
9.
10.           double mean = (double)sum/a.length;
11.            return mean;
12.       } // end method arrayMean
13.
14.      //Main program
15.      public static void main (String[ ] args) {
16.           // declaring the length of the array
17.           int n = Integer.parseInt(JOptionPane.showInputDialog
18.                     ("Enter the length of the array"));
19.           int[ ] a = new int[n];
20.           // Reading Data into the Array a
21.           for (int m = 0; m < n; m++)
22.               a[m] = Integer.parseInt(JOptionPane.showInputDialog
23.                     ("Enter data"+ (m+1)));
24.
25.           // printing the original array a
26.            System.out.println("Original array elements .....\n");
27.             for (int l = 0; l < n; l++ )
28.                  System.out.print(a[l] + "\t");
29.
30.               System.out.println("\n");
31.
32.               //Calling the sum method to compute the mean in the array
33.               System.out.printf("Mean of the array data is %.3f  \n\n\n",
34.                       arrayMean(a) );
35.
36.               //Closing the program
37.               System.exit(0);
38.           }
39.       }
```

Sample Output:

PROPERTIES OF DLC UI, IBADAN



8.4.7 Random Number Generation

Random numbers are popular with simulation experiment and game playing applications. Random method is part of the Math Class, which could be invoked and used in programs.

Math method random generates a random double value in the range from 0.0 up to, but not including, 1.0. The values returned by random are actually pseudo-random numbers – a sequence of values produced by a complex mathematical calculation. The calculation uses the current system time to "seed" the random number generator such that each execution of a program yields a different sequence of random values. For example, the code:

```
double randomVal = Math.random( );
```

will assign a double random number to randomVal.

Since the values of random numbers we need in most applications would be in integers, then, we need to *type-cast* the values returned by the generator. We also need to specify the initial and final values of numbers we want.

This manipulation is called *scaling the range of values produced by Math method random.* For example, to produce integers from 0 to 5, we could write:

```
(int)(Math.random( ) * 6)
```

The number 6 in the expression is called the scaling factor. The integer cast operator (int), truncates the floating point part (the part after the decimal point) of each value produced by the expression. The *shifting value* that specifies the initial value in the desired set of random numbers should also be stated; else, zero (0) is assumed as in the above code. Thus we have the following formula holding for random numbers:

```
number = shiftingValue + (int)(Math.random( ) * scaling factor);
```

For example, for a die rolling, we could write

```
int dieFace = 1 +(int)(Math.random() * 6);
```

which will assign an integer random value to dieFace.

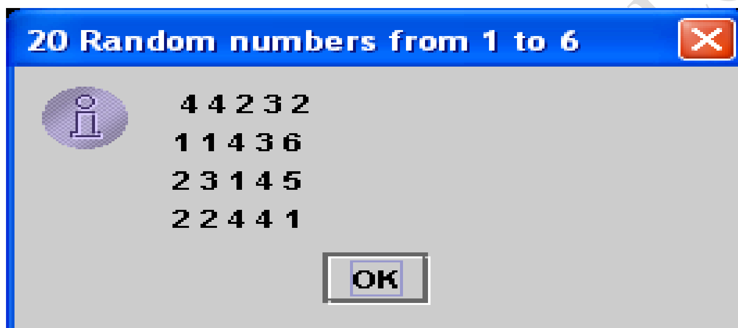
A complete program is illustrated below, which generates 20 random numbers from 1 to 6.

PROPERTIES OF DLC UI. IBADAN

Example: 20 random numbers from 1 to 6, printing 5 data per line.

```
1.  import javax.swing.*;
2.  public class RandomDie {
3.      public static void main(String[ ] args) {
4.          String output = " "; // for appending results
5.          for (int i = 1; i <= 20; i++) {
6.
7.          //Generates random integers from 1 to 6
8.          int value =  1 +(int)(Math.random( ) * 6);
9.
10.         output += value + " ";  //append value to output
11.
12.         // if i is divisible by 5, append new line to output
13.         if (i % 5 = = 0)
14.             output += "\n";
15.         }  //end for-loop
16.
17.         JOptionPane.showMessageDialog(null, output, "20 Random
18.             numbers from 1 to 6", JOptionPane.INFORMATION_
19.             MESSAGE);
20.         System.exit(0);
21.     }  // end main method
22. } //end class
```

Output:



Practical Exercise: Copy this code into your editor, compile and execute this program.

8.4.8 Void Methods

If a method would only receive data but will not return any value to any other method, that method is void in nature. And if a method has a void return type, control returns at the method-ending right brace or by executing the statement:

return;

Consider the Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, ….
The series begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers. The ratio of successive Fibonacci numbers converges on a constant value 1.618, a number called the *golden ratio* or *golden mean*. The Fibonacci model equation is given as:

Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2)

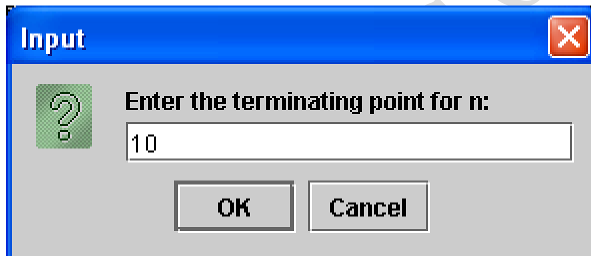
PROPERTY OF UCUI, IBADAN

The implementation below uses an array. It assumes that the Fibonacci numbers are kept in an array with the 0 and 1 in the first two indexes or locations representing the Fibonacci values of 0 and 1 respectively. The subsequent Fibonacci numbers are computed based on the general formula presented above. That is, the Fibonacci of 3 will be the sum of Fibonacci at location [3 – 1] and location [3 – 2], which are locations 2 and 1 respectively.

The Fibonacci program

```
1.   import javax.swing.*;
2.   class fibonacci {

3.        // the void method

4.        static void fibn( int a) {
5.             int[ ] fib = new int[a];
6.             fib[0] = 0;
7.             fib[1] = 1;
8.             for (int i = 2; i<a; i++)
9.                 fib[i] = fib[i-1] + fib[i-2];
10.
11.            for (int i = 0; i < fib.length; i++)
12.                 System.out.println("Fib[" + i + "] = " + fib[i]);
13.       }// end method fibn

14.       // the main method …
15.        public static void main(String[ ] args)  {
16.            int n = Integer.parseInt(JOptionPane.showInputDialog
17.                 ("Enter the terminating point for n: "));
18.            fibn(n); // calling the void method fibn
19.        }
20.   }
```

Output:



Fib[0] = 0
Fib[1] = 1
Fib[2] = 1
Fib[3] = 2
Fib[4] = 3
Fib[5] = 5
Fib[6] = 8
Fib[7] = 13
Fib[8] = 21
Fib[9] = 34

Note that the method fibn is just called at the main method without assigning it to any variable or object; since it is not going to return a value to anywhere.

PROPERTY OF DLC UI, IBADAN

8.4.9 Recursive Methods

The example methods we have discussed so far are structured as methods that can call each other in a disciplined, hierarchical manner. However, a method can call itself in a number of times. A recursive method is one that calls itself either directly or indirectly through another method.

In recursion, the problems being solved are similar in nature and their solutions too are similar. When a recursive method is called to solve a problem, the method could actually solve the simplest case or *base case*. If the method is called with a base case, the method returns a result. However, if the method is called with a complex problem, it divides the problem into two conceptual pieces: a piece that the method knows how to solve (the base case) and a piece that the method does not know how to solve. The latter piece must resemble the original problem but be a slightly simpler or slightly smaller version of it. Because this new problem looks like the original problem, the method calls a fresh copy of itself to work on the small problem. This procedure is called a *recursive call* or *recursion step.* The recursion step must include a return statement because its result will be combined with the portion of the problem the method knew how to solve to form a result that will passed back to the original caller.

The recursion step executes while the original call has not finished executing. As a matter of fact, there could be many recursion calls, as the method divides each new subproblem into two conceptual pieces.

Example 1: Recursive Factorial

```
public long fac (int n) {
   // Base case
    if (n <= 1)
            return 1;
     //Recursive step
          else
             return n * fac(n – 1);
 } // End method fac
```

Example 2: Recursive Fibonacci Series

```
public long fib (int n) {
          if (n == 0 || n == 1)
                    return n;      // Base case
          else
                    return fib(n – 1) + fib(n – 2);     // Recursive step
  } // End method fib
```

Comparing Recursion and Iteration

- Both iteration and recursion are based on a control statement: iteration uses a repetition control (for, while or do- while); recursion uses a selection control (if, if – else or switch).
- Both involve repetition: Iteration explicitly uses a repetition statement, recursion achieves repetition through repeated method calls.
- Both involve a termination test: Iteration terminates when the loop continuation condition fails. Recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and recursion, each gradually approach termination: iteration keeps modifying a counter until the counter assumes a value that makes the loop continuation condition fail, recursion keeps producing simpler versions of the original problem until the base case is reached.
- Both can occur infinitely: an infinite loop occurs with iteration if the loop continuation test never becomes false. Infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

PROPERTIES OF DLC UI, IBADAN

- **Demerits of recursion**: it repeatedly invokes the mechanism, and consequently, the overhead, of method calls. This repetition can be expensive in terms of both processor time and memory space. Each recursive call causes another copy of the method (actually, only the method's variables) to be created; this set of copies can consume considerable memory space. Iteration occurs within a method, so repeated method calls and extra memory assignment are avoided. Therefore, there is no need of choosing recursion (Deitel and Deitel, 2007).

Summary
This chapter has introduced you to the use of methods in a Java program. Methods are like functions and procedures in the procedural languges. They are used to break a program into functional modules. Methods enhances program manageability and usability.

Post Test

1. Write a function in Java that could accept an integer denoting a year and returns true or false according to whether the year is a leap year or not.

2. Write a function in Java that accepts two dates in a year and computes the number of days **between** them. State the conditions under which the function may work properly (regarding the ordering of input to the function).

3. Write a recursive Java method to return the factorial of a number, n.

4. Using the concept of java methods, write a java program that solves the combination problem, i.e. nCr. What advantages can you infer from using method in this program?

PROPERTIES OF DLC UI, IBADAN

9 *Principles of Object-Oriented Programming with Java*

9.1 Introduction

This chapter is an introduction to object-based programming. In this chapter, we shall be introduced to the techniques involved in developing our own objects and classes in Java programming language.

9.2 Objectives

At the end of this chapter, you should be able to
- (i) understand the key principles of object-oriented programming
- (ii) design and implement object-based programs in Java

9.3 Pre Test

1. What are the major components of an object?
2. Explain what you understand by the term classes and objects
3. Draw a UML class diagram for Class Customer having name, address, and quantity-needed as attributes

9.4 Main Content

9.4.1 Concept of Object Oriented Programming

An object-oriented program will usually contain different types of objects, each type corresponding to a particular kind of complex data to be managed or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might well contain multiple copies of each type of object, one for each of the real-world objects the program is dealing with. For instance, there could be one bank account object for each real-world account at a particular bank. Each copy of the bank account object would be alike in the methods it offers for manipulating or reading its data, but the data inside each object would differ reflecting the different history of each account.

Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. The object's methods will typically include checks and safeguards that are specific to the types of data the object contains. An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified. This approach can also be used to offer standardized methods across different types of objects. As an example, several different types of objects might offer print methods. Each type of object might implement that print method in a different way, reflecting the different kinds of data each contains, but all the different print methods might be called in the same standardized manner from elsewhere in the program. These features become especially useful when more than one programmer is contributing code to a project or when the goal is to reuse code between projects.

An object-oriented program may thus be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to "carry their own operators around with them" (or at least "inherit" them from a similar object or class) - except when they have to be serialized.

PROPERTIES OF DLC UI, IBADAN

9.4.2 OOP languages

Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making simulation programs, in which what came to be called objects were the most important information representation. Smalltalk (1972 to 1980) is arguably the canonical example, and the one with which much of the theory of object-oriented programming was developed. Concerning the degree of object orientation, following distinction can be made:

- Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: Scala, Smalltalk, Eiffel, JADE, Emerald.
- Languages designed mainly for OO programming, but with some procedural elements. Examples: C++, Java, C#, VB.NET, Python.
- Languages that are historically procedural languages, but have been extended with some OO features. Examples: Visual Basic (derived from BASIC), FORTRAN 2003, Perl, COBOL 2002, PHP, ABAP.
- Languages with most of the features of objects (classes, methods, inheritance, reusability), but in a distinctly original form. Examples: Oberon (Oberon-1 or Oberon-2) and Common Lisp.
- Languages with abstract data type support, but not all features of object-orientation, sometimes called object-*based* languages. Examples: Modula-2 (with excellent encapsulation and information hiding), Pliant, CLU.

9.4.3 Java Classes

A java program has one or more files that contain java classes, one of which is public, containing a method named main( ). For instance,

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

        Statements;
}
```

A class is usually described as the template or blueprint from which the object is actually made. When you create an object from a class, you "*create an instance*" of the class. Consider the example below:

```
fruits mango = new fruits( );
```

The new operator creates a new instance of the fruits class.
An object is characterized by its 'state' and 'behaviour'. The object's data is its state, and the method used in manipulating those that data is its behaviour.

In essence, Object Oriented Programming (OOP) means ***writing programs that define classes whose method carry out the program's instructions.*** OOP programs are designed by deciding first what classes will be needed and then defining methods for those classes that solve the problem. Java classes are similar to types (int, short), but three essential characteristics of classes that distinguish them from Java types are:

- They are or can be user/programmer's defined
- Class objects can contain variables, including references to other objects.
- Classes can contain methods that give their objects the ability to act.

Just as Java type specifies the range of values e.g. (-32767 to 32767 for short) that variables of that type can have, a java class specifies the range of values that objects of that class can have.

PROPERTIES OF DLC UI IBADAN

Classes can be (and in Java always are) built on other classes. Therefore, a class that builds on another class '*extends'* it. Java comes with '*base class*' from which all other classes are built.

When a base class is extended, the new class has all the properties and methods of its parent. We then choose whether to modify or simply keep any method of the parents or supply new methods that apply to the child class only. This concept of extending a base class is called '*inheritance'*.

9.4.4 Relationships between classes:

The relationships between different classes could be

- Use
- Containment ("has –a")
- Inheritance ("Is-a")

A class uses another class if it manipulates objects of that class. Generally class A uses class B if

- a method of A sends a message to an objects of class B
  or
- a method of A creates, receives or returns objects of class B

Containment is a special case of use; if object A contains objects B, then at least one method of class A will make use of that object of class B.

If class A extends class B, class A inherits methods from class B, but has more capabilities.

9.4.5 Building Your Own Classes

The syntax for a class in Java is:

```
accessSpecifier class NameOfClass {

        // definitions of the class's features
        // includes methods and instance fields
}
```

The classes do not (and often cannot) stand alone: they are the building blocks for constructing or building stand-alone programs.

Essentially, there are three aspects to consider when we are writing our own classes.

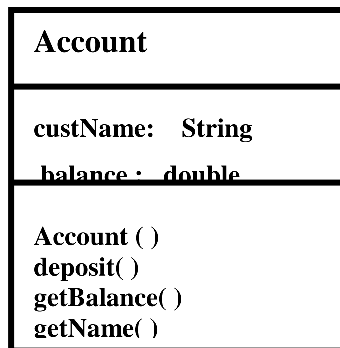
(i) The class fields, which are the private variables to be declared and used in the class,

(ii) The class constructor, which is a special method/function that will be used to *instantiate* the class, and

(iii) Other methods that will be used to manipulate the private data in the class.

PROPERTIES OF DLC UI, IBADAN

Consider the following example, which is an Account class, depicted in the class diagram below, that might be used by a business in writing a simple customer accounting system:



Example Class code: The Account Class

```
1.  class Account  {
2.         // instance variables or class fields
3.        String custName;
4.        double balance;
5.
6.        // constructor
7.        public Account(String name, double bal) {
8.             custName = name;
9.             balance = bal;
10.       }
11.
12.       // method to compute new balance after deposit
13.       public void deposit(double dep) {
14.            balance += dep;
15.       }
16.
17.       // method to get balance
18.       public double getBalance( ) {
19.            return balance;
20.       }
21.
22.       // method to get the customer's name
23.       public String getName( ) {
24.            return custName;
25.       }
26.       // method to compute new balance after withdrawal
27.       public void withdraw(double out) {
28.            balance -= out;
29.       }
30.   } // end class Account
```

```
1.  // Program to test the class
2.  public class AccountTest {
3.        public static void main(String[ ] args) {
4.             //Instantiating the Account class
5.             Account acc = new Account("Akinola S.O.", 500);
6.             String name;
7.             double bal;
8.             name = acc.getName( );
9.             bal = acc.getBalance( );
10.            System.out.println("Account for " + name + "has balance
```

PROPERTIES OF DLC UI, IBADAN

```
11.                 =N=" + bal);
12.         acc.deposit(800);
13.         bal = acc.getBalance( );
14.         System.out.println("New Account balance after deposit
15.             for "  + name + "is =N=" + bal);
16.         acc.withdraw(1000);
17.         bal = acc.getBalance( );
18.         System.out.println("New Account balance after withdraw
19.            for  " + name + "is =N=" + bal);
20.     }
21.  }
```

Sample Output from this Program is shown next:

```
Command Prompt

C:\j2sdk1.4.2_04\bin>java AccountTest
Account for Akinola S.O.has balance =N=500.0
New Account balance after deposit for Akinola S.O.is =N=1300.0
New Account balance after withdraw for  Akinola S.O.is =N=300.0

C:\j2sdk1.4.2_04\bin>
```

Note: To run this program,

(i) The class Account and its tester AccountTest.Java need to be in one file and saved as AccountTest.Java since this is the one having the main method. Actually, both the two might also be compiled separately with their class names.

(ii) On compilation, two classes will be created; AccountTest.class and Account.class

(iii) On Executing or running the code, java byte code interpreter starts running the code in the main method in the AccountTest class. This code in turn creates some new Account objects and shows us their states.

Now, let us analyze this Account class.

1. The class has five **methods** with headers:

```
public Account(String name, double bal)
public void deposit(double dep)
public double getBalance( )
public String getName( )
public void withdraw(double out)
```

The word public here is called Access modifier discussed soon. These modifiers always specify who can use the method or the class. In this case public means that any method in any class that has access to an instance of the Account class can call the method.

2. There are two **instance fields**, holding data in the Account class:

```
String custName;
double balance;
```

Normally, private keyword is used as access modifier for instance fields. That is, we could declare the instance filed like:

```
private String custName;
```

private keyword means that no outside agency can access the instance fields except through the methods of our class. Using public keyword or not specifying anything as in the Account class for the instance fields does not matter but ruins encapsulation.

3. The Class Constructor

Classes can have three kinds of members: fields, methods and constructors.

- A **field** is a variable that is declared as a member of a class. Its type may be any of the eight primitive data types - boolean, char, byte, short, int, long, float, double) or a reference to an object.
- A **Method** is a function that is used to perform some action for instances of the class.
- A **constructor** is a special function/method whose only purpose is to create the class's objects. This is called ***instantiating*** the class, and the objects are called instances of the class.

The constructor in our Account class is:

```
7.      // constructor
8.      public Account(String name, double bal) {
9.          custName = name;
10.         balance = bal;
11.     }
```

This constructor is used to initialize objects of the class -giving the instance variables the initial state you want them to have.

For example, in creating an instance of the class with code

```
Account acc = new Account("Akinola S.O.", 500);
```

We set the instance fields, as follows

```
custName = "Akinola S.O";
balance  = 500;
```

We could see that the **new** method is always used together with a constructor to create an object of the class. This forces us to set the initial state of our objects either explicitly or implicitly. An object created without a correct initialization is useless and occasionally dangerous. It can cause a general memory protection fault (GPF), i.e. memory corruption.

PROPERTIES OF DLC UI, IBADAN

9.4.6 Properties of Constructors

The following are the general properties of constructors

1. A constructor has the same name as the class itself.
2. A constructor may take one or more (or none) parameters.
3. A constructor is always called with the **new** keyword.
4. A constructor returns no value.

Q. Differentiate constructors from ordinary methods

Every class has at least one constructor to instantiate it. For a class without one, the compiler will automatically declare one with no arguments. This is called a default constructor. A default constructor has no arguments. Java provides this type of constructor if we design a class with no constructor. The default constructor sets all the instance variables to their default values. All numeric data contained in the instance fields would be zeroed out, all booleans would be false and all object variables would point to null.

We can overload a constructor or a method. That is, different constructors/methods with the same signature. We shall see example of this later. Also, it is possible to have more than one constructor in a class.

Summarily, a ***constructor*** is a member function of a class that is used to create objects of that class. This has the same name as the class itself, has no return type and is invoked using the new operator. A ***method*** is an ordinary member function of a class. It has its own name, a return type (which may be void) and is invoked using the dot operator. For example, double y = Math.pow(a,b)

9.4.7 Caution About Constructor

We must not introduce local variables with the same names as the instance fields. For example, the following constructor will not set the balance.

```
public Account (String name, double bal)  {
  custName = name;
  double balance = bal; // Error, variable balance redeclared
}
```

The last line declares a local variable ‘balance’ (with double appended as data type beside it) that is only accessible inside the constructor. This shadows the instance field ‘balance’.

Again we must be careful in all of our methods that we don’t use variable names that equal the names for instance fields.

9.4.8 The Methods of the Account Class

These four methods are just like ordinary methods we know. However, all these methods have access to the private instance fields by name. This is because instance fields are always accessible by the methods of their own class.

PROPERTIES OF DLC UI, IBADAN

For example,

```
public void deposit (double dep) {
        balance += dep;
}
```

sets a new value for the balance instance field in the object that executes this method. That is, increasing it by value of dep.

Note: This method does not return a value. For example, the call

```
Acc.deposit(1000);
```

will make initial deposit to be increased by 1000.

Every method declared within a class has two arguments passed to it: one is implicit and will not appear in the method's parenthesis and the other(s) is/are explicitly declared in the parenthesis. For instance, the implicit argument to the deposit method is the object of type Account, which does not appear in the method declaration. The explicit argument -double dep, is listed in the method declaration.

Other methods in the account class are like ordinary methods we know. Some of the methods may have empty argument list, that is, no data is passed into them. The idea is that they make use of the instantiating field values already passed into the constructor of the class. These type of methods are normally used to return values to the object's references.

Any method that will get argument data from outside object referencing the class will be declared as void, meaning that it will not return a value outside. The data values passed into these type of methods are normally used to manipulate the instantiating fields, or class variables.

9.4.9 Using the Class

In order to make use of this class in programs, the Account class will be called or referenced like we have in arrays:

```
Account acc = new Account ("Akinola S. O.", 500)
```

Compared to an array:

```
int array[ ] = new int(5);
```

acc is a new object we are creating from the Account class. The parameters ("Akinola S. O.", 500) are the initial data we are sending to the class to instantiate the class.

We could reference any of the methods of the class via the object we create from it. For example, acc.getName( ) means we are referencing the method, getName from the class, which will in turn return the name of the account holder, Akinola S. O., passed as instantiating data to the class. The next examples illustrate further on creating objects and classes

```
1.   //The Rectangle class
2.   class Rectangle {
3.        // instance variables
4.        private double width;
5.        private double height;
6.        // constructor
7.
8.         public Rectangle (double h, double w) {
9.                height = h;
10.               width = w;
11.         }
```

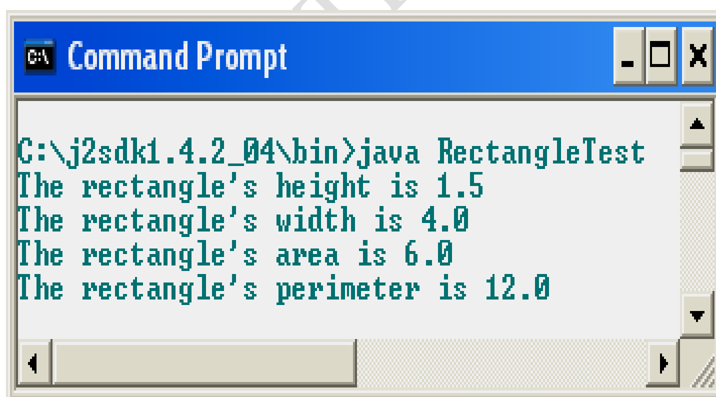

PROPERTIES OF DLC UI, IBADAN

```
12.          // method to return the area of the rectangle
13.         public double area( ) {
14.              return height * width;
15.            }
16.          // method to return the height of the rectangle
17.         public double getHeight( ) {
18.              return height;
19.         }
20.         // method to return the width of the rectangle
21.         public double getWidth( ) {
22.             return width;
23.          }
24.         // method to return the perimeter of the rectangle
25.        public double perimeter() {
26.             return 2 * (height * width);
27.        }
28.    }
```

// Main program to test the Rectangle class

```
1.   public class RectangleTest {
2.        public static void main(String [ ] args) {
3.                Rectangle myRectangle = new Rectangle(1.5, 4);
4.                double height, width, area, perimeter;
5.                height = myRectangle.getHeight( );
6.                width = myRectangle.getWidth( );
7.                area = myRectangle.area( );
8.                perimeter = myRectangle.perimeter( );
9.                System.out.println("The rectangle's height is " +   height);
10.               System.out.println("The rectangle's width is " +  width);
11.               System.out.println("The rectangle's area is " + area);
12.               System.out.println("The rectangle's perimeter is " +   perimeter);
13.      }
14.   }
```

Sample Output:



9.4.10 Classes without Explicit Constructor

We could define a class without an explicit constructor. Just as was mentioned earlier, in this type of class, the compiler will automatically declare one with no arguments. This is called a default constructor. A default constructor has no arguments. The default constructor sets all the instance

variables to their default values. All numeric data contained in the instance fields would be set to zero, all booleans would be false and all object variables would point to null. Consider the example below:

Example: The person Class, with default constructor

```
3.   class Person {
4.
5.        //instance variables
6.        private String familyName = "I don't have a name yet";
7.        private String givenName = "No name here either!";
8.        private int yearOfBirth;
9.
10.       //method to return the family name
11.       public String getFamilyName( ) {
12.            return familyName;
13.       }
14.
15.       //method to return the given name
16.       public String getGivenName( ) {
17.            return givenName;
18.       }
19.
20.       //method to return the year of Birth
21.       public int getYearOfBirth() {
22.            return yearOfBirth;
23.       }
24.
25.       // method to give a value to the family name
26.       public void setFamilyName(String name) {
27.            familyName = name;
28.       }
29.
30.       // method to give a value to the given name
31.       public void setGivenName(String name) {
32.            givenName = name;
33.       }
34.
35.       // method to give a value to the year of birth
36.       public void setYearOfBirth(int year) {
37.            yearOfBirth = year;
38.       }
39.    }   // end class
1.     // main program to test the person class
2.     public class PersonTest {
3.          public static void main(String [ ] args) {
4.               Person theLecturer = new Person( );
5.               String temp;
6.               int year;
7.               temp = theLecturer.getFamilyName( );
8.               System.out.println("The Lecturer's family name is "
9.                    + temp);
10.              theLecturer.setFamilyName("Akinola");
11.              temp = theLecturer.getFamilyName( );
12.              System.out.println("The Lecturer's family name is "
13.                  + temp);
14.              temp = theLecturer.getGivenName( );
15.              System.out.println("The Lecturer's given name is " +
16.                  temp);
17.              theLecturer.setGivenName("Olalekan");
18.              temp = theLecturer.getGivenName( );
```

PROPERTIES OF DLC UI, IBADAN

```
19.                 System.out.println("The Lecturer's given name is " +
20.                     temp);
21.                 year = theLecturer.getYearOfBirth();
22.                 System.out.println("The Lecturer's year of birth is " +
23.                     year);
24.                 theLecturer.setYearOfBirth(1990);
25.                 year = theLecturer.getYearOfBirth();
26.                 System.out.println("The Lecturer's year of birth is " +
27.                     year);
28.         }
29.     }
```

```
Command Prompt

C:\j2sdk1.4.2_04\bin>javac PersonTest.java

C:\j2sdk1.4.2_04\bin>java PersonTest
The Lecturer's family name is I don't have a name yet
The Lecturer's family name is Akinola
The Lecturer's given name is No name here either!
The Lecturer's given name is Olalekan
The Lecturer's year of birth is 0
The Lecturer's year of birth is 1990

C:\j2sdk1.4.2_04\bin>_
```

Note that in this type of class with default constructor:

(i) When we are creating the new object theLecturer from the class Person, we did not pass any instantiating value into the invocation.

```
Person theLecturer = new Person();
```

This shows that we want to use a default constructor for our class. We could initialize the class instantiating fields to whatever we want, or we just declare them without initialization.

```
6.      //instance variables
7.      private String familyName = "I don't have a name yet";
8.      private String givenName = "No name here either!";
9.      private int yearOfBirth;
```

The first two fields are initialized while the third is not. But by default, yearOfBirth being of integer data type will be automatically set to zero as we mentioned before that all numeric fields will be set zero by default constructor.

(ii) The class has some other methods, which are used to give initial values to the class fields. For example,

```
  // method to give a value to the year of birth
 public void setYearOfBirth(int year) {
        yearOfBirth = year;
```

PROPERTIES OF ... IBADAN

}

(iv) Any attempt for us to reference the class methods without first of all set values to the class fields would mean that the values returned by the methods will be default values as set by the default constructor itself. For instance, consider the code:

```
temp = theLecturer.getFamilyName( );
System.out.println("The Lecturer's family name is " + temp);
```

This method *getFamilyName( )* returns: The Lecturer's family name is I don't have a name yet

This happens to be the value given to the familyName field initially.

```
private String familyName = "I don't have a name yet";
```

But when we set the field with a value:

```
theLecturer.setFamilyName("Akinola");
temp = theLecturer.getFamilyName( );
System.out.println("The Lecturer's family name is " + temp);
```

Then the method returns:

The Lecturer's family name is Akinola.

9.4.11 The Life Cycle of an Object

Once an object has completed the work for which it was created, it is garbage-collected and its resources are recycled for use by other objects. This may be likened to garbage collection with pointers in C, C^{++} and Pascal languages.

9.4.12 The Scope of a Class

All instance variables and methods in a class belong to the class scope. Within a class's scope, class members are immediately accessible to all of that class's methods and can be referenced simply by name. Variables defined in a method belong to that method and are 'local' to it. Such variables are said to have block scope.

The class members that could be accessed by other objects (i.e., public members) can be accessed off a "handle" (i.e. primitive data type members can be referred to by

objectReferenceName.primitiveVariableName

and object members can be referenced by

objectReferenceName.objectMemberName.

If a method defines a variable with the same name as a variable with class scope (i.e. an instance variable), the class-scope variable is hidden by the method-scope variable in the method scope. A hidden instance variable can be accessed in the method by preceding its name with the keyword 'this' and the dot operator, as in ***this.x***. The 'this' reference is implicitly used to refer to both the instance variables and methods of an object. Java uses '**this'** keyword that the proper object is referenced when a method of a class references another member of that class for a specific object of that class. With this, each object has access to a reference to itself - called the "**this reference**".

PROPERTIES OF DLC UI, IBADAN

For example,

```
class  Arith {
        private int a, b;
        // constructor
        public Arith (int a, int b) {
           this.a = a;
          this.b = b;
  }
```

Note: In a method in which a method parameter has the same name as one of the class members, use '**this'** explicitly if you want to access the class member, otherwise you will incorrectly reference the method parameters.

Essentially, in a method, the keyword 'this' refers to the object on which the method operates. For example, many java classes have a method called "**toString()**" that returns a string describing the object. If you pass any object to the System.out.println method, that method invokes the 'toString()' method on the object and prints the resulting string. Therefore, we can print out the current state of the implicit argument of any method as System.out.println(this). This is a useful strategy for debugging.

The '**this'** keyword also has another meaning. If the first statement of a constructor has the form **this(…)**, then the constructor calls another constructor of the same class. For example,

```
 class customer (String n ) {
   this(n, Account.getNewNumber( );
{

public customer (String n, int a) {
   name = n,
   accountNumber = a;
}
…
…
…
}
```

A call, like **new customer("Akinola S.O")** invokes the **customer(String n)**, which in turn calls the customer**(String n, int a)** constructor.

As an example, consider the following example with default constructor. The program demonstrates that methods in the same class can call each other and that both the class and the tester program for the class could be in one monolithic file.

The class firstly computes the customer's payment before discount. Then it computes the discount to be given to a customer based on the number of quantities bought from the store. If the quantity bought is 20 and above then a 20% discount is given to the customer, if it is between 10 to 19, a 5% discount is given. If the quantity is between 6 to 9, discount is 2% and if it is between 1 to 5, 0.5% discount is given on the initial payment. Finally, the program computes the net pay after discount.

Example:

```
1.  import javax.swing.JOptionPane;
2.  class sales {
3.     // class fields
4.     private int q;
5.     private double p;
6.
7.     // using default constructor
8.     // method to return the Initial pay before discount
9.     public double ip( ) {       // ip means initial pay value
10.            return p * q;
```

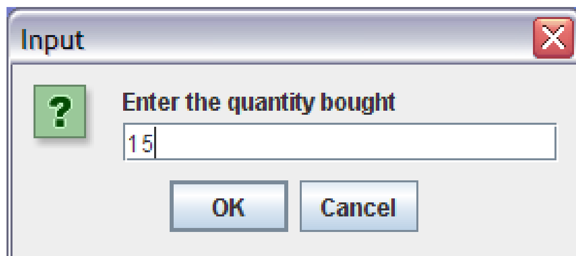
PROPERTIES OF DLC UI, IBADAN

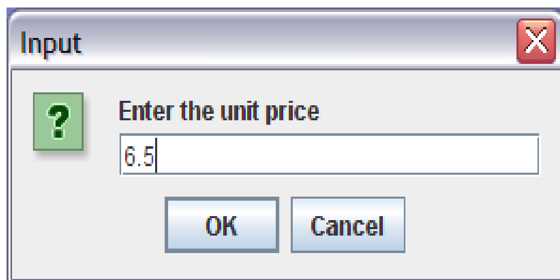
```
11.     }
12.
13.     // method to return the discount
14.     public double d( ) {
15.             double disc; // A local variable declared
16.             if (q >= 20)
17.                     disc = 10.0/100.0 * ip(); //ip() method called
18.             else if (q>= 10)
19.                     disc = 5.0/100.0 * ip( );
20.             else if (q >=6)
21.                     disc = 2.0/100.0 * ip( );
22.             else if (q >= 1)
23.                     disc = 0.5/100.0 * ip( );
24.             else
25.                     disc = 0;
26.             return disc;
27.       } // end method d( )
28.
29.     // Method to return the net pay after discount
30.     public double np( ) {
31.             return ip( ) - d( );  // the two methods up are called
32.     }
33.
34.     // method to set the unit price p
35.     public void setP(double x) {
36.             this.p = x; //keyword 'this' explained recently
37.     }
38.
39.     // method to set the quantity q
40.     public void setQ(int y) {
41.             this.q = y;
42.     }
43.
44.     // The main program begins…
45.     public static void main(String[ ] args) {
46.             sales market = new sales(); // calling the default
47.             //constructor with no arguments
48.
49.             int q = Integer.parseInt (JOptionPane.showInputDialog
50.                 ("Enter the quantity bought"));
51.             double p = Float.parseFloat(JOptionPane.showInput
52.                   Dialog ("Enter the unit price"));
53.
54.             //Setting the unit price and quantity class instance fields
55.                     using their methods in the class
56.             market.setP(p);
57.             market.setQ(q);
58.
59.             // using a string variable  'out' to capture the outputs
60.             String out = "Deatails of your transactions with us \n ";
61.             out += "Quantity bought:" + q + "\nUnit price:"+p + "\n";
62.             out += "Purchase Price Before Discount: " + market.ip( );
63.             out += "\nDiscount Given: " + market.d( );
64.             out += "\nPurchase Price After Discount: " + market.np( );
65.             out += "\n\n\n Thanks for patronizing us, call again";
66.
67.             // printing out values
68.             JOptionPane.showMessageDialog(null, out, "Akinola
69.                    Supermarket and Stores",  JOptionPane.
70.                     INFORMATION_MESSAGE);
```

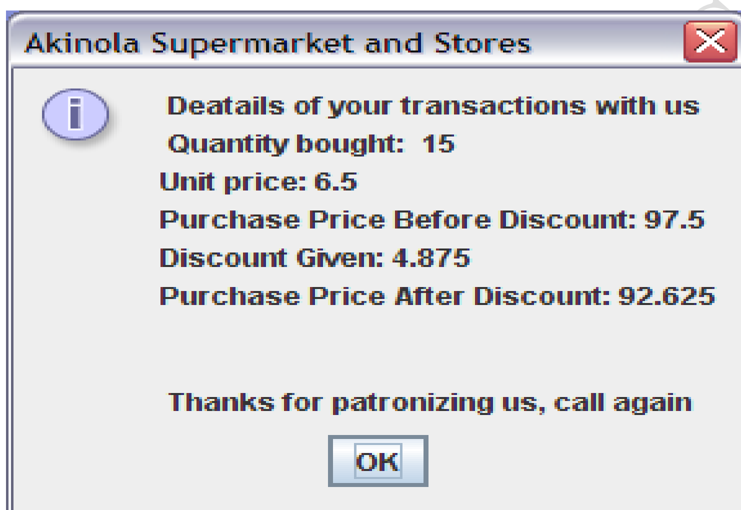
PROPERTIES OF DLC UI, IBADAN

```
71.             System.exit(0);
72.        } // end method main
73.    }  // end class
```

Output Screen Shots







The example below gives the general quadratic equation class

Quadratic_Equation
a: float **b : float**
c: float **quadratic ( )** **discriminant( )** **root1( )** **rroot2( )** **real ( )**

//The general class for solving quadratic equations

```
1.  import javax.swing.JOptionPane;
2.  class quadratic {
3.           private float a, b,c;
4.
5.       //Constructor
6.       public quadratic(float a, float b, float c) {
7.            this.a = a;
8.            this.b = b;
9.            this.c = c;
10.      }
11.
12.      //Method to compute discrimant, d
13.      public double d( ) {
14.           return (b*b - 4.0 *a * c);
15.      }
16.
17.      //Method to compute first root
18.      public double r1( ) {
19.           return (-b + Math.sqrt(d( )));
20.      }
21.
22.      //Method to compute second root
23.      public double r2( ) {
24.           return (-b - Math.sqrt(d( )));
25.      }
26.      //Method to compute real part
27.      public double real( ) {
28.           return (-b /(2.0*a));
29.      }
30.
31.      //Method to compute imaginary part
32.      public double imag( ) {
33.           return (Math.sqrt(Math.abs(d( )))/(2.0*a));
34.      }
35.
36.      //The main method
37.
38.      public static void main(String[] args)   {
39.           //Data entry
40.           float a = Float.parseFloat(JOptionPane.showInputDialog
41.              ("Enter  the coefficient of x-square, a"));
42.           float b = Float.parseFloat(JOptionPane.showInputDialog
43.               ("Enter the coefficient of x, b"));
44.           float c = Float.parseFloat(JOptionPane.showInputDialog
45.                ("Enter the constant, c"));
46.
47.           //Instantiating a new object quad from the quadratic class
48.           quadratic quad = new quadratic(a, b, c);
49.
50.           //Reporting the roots
51.           System.out.println("a = "+ a + " b = "+ b + " c = " + c +
52.                                "\n\n");
53.           if (quad.d( ) < 0)  // complex roots
54.               System.out.printf("Complex roots please \n Complex
55.                      Root1 =  i%.2f \n Complex Root2 = i%.2f",
56.                      quad.imag(), quad.real( ));
57.           else if (quad.d( ) == 0) // One real root
58.               System.out.printf("One Real Root with value %.2f",
59.                quad.real( ));
60.           else  //Two real roots
```

PROPERTIES OF DLC UI, IBADAN

```
61.                 System.out.printf("Two Real Roots with values \n
62.                 Root1 = %.2f   \nRoot2 %.2f= ", quad.r1( ) , quad.r2( ));
63.
64.             System.out.println( );
65.             System.out.exit(0);
66.         }// end method main
67.     }  // end class quadratic
```

Sample outputs

```
C:\Program Files\Xinox Software\JC...
a = 1.0 b = 2.0 c = 3.0

Complex roots please
 Complex Root1 = i1.41
 Complex Root2 = i-1.00
Press any key to continue..._
```

```
C:\Program Files\Xinox S...
a = -1.0 b = 3.0 c = 2.0

Two Real Roots with values
 Root1 = 1.12
Root2 -7.12=
Press any key to continue..._
```

```
C:\Program Files\Xinox Softwar...
a = 1.0 b = -4.0 c = 4.0

One Real Root with value 2.00
Press any key to continue..._
```

9.4.13 Access Modifiers

These specify where the declared entity can be used. The entity can be a class, field, constructor, or method.

The following table summarizes the modifiers that can appear in the declarations of class fields, local variables, constructors and methods.

If none of the three access modifiers, (public, protected and private) is specified for a class, field, constructor or method, then they have "package access", which means that it can be accessed from any class in the same package.

The modifier 'static' is used to specify that a method is a class method, else without it, the method is an instance method, which can be invoked only when bound to an object of the class.

For example, Add( ) is an instance method which can be invoked as y.Add( ) in the main( ) program. In this invocation, method Add( ) is bound to the object y, so y is an implicit argument.

A ‘class method’ is a method that is invoked without being bound to any specific object of the class.

Modifier	Class	Constructor	Field	Local Variables	Method
public	The class is accessible from all other class.	Accessible from all classes	Accessible from all classes	--	Accessible from all class
abstract	The class cannot be instantiated	Not Applicable	--	--	It has no body and belong to an abstract class
final	No subclasses be can declared	--	It must be initialized and cannot be changed e.g. final int x = 2;	It must be initialized and cannot be changed	No subclass can override it.
protected	Not applicable	It is accessible only from within its own class and its subclasses	It is accessible only from within its own class and its subclasses	___	It is accessible only from within its own class and its subclasses
private	Not applicable	It is accessible only from within its own class	It is accessible only from within its own class	___	It is accessible only from within its own class
static	___	___	Only one value of the field exist for all instances of the class	___	It is bound to the class itself instead of an instance of that class
native	___	___	__	___	Its body is implemented in another programming language

9.4.14 Constructor Overloading

We can have more than one constructor in a class with the same signature or name. Since all constructors in the same class must have the same name (viz the name of the class itself), there must be some other way for the compiler to be able to distinguish them. The only other way is for them to have distinct parameter list. The rule is that the sequence of parameter types must be different for each overloaded constructor or method. The example below illustrates further.

Example: The Rectangle Class, having four Constructors

```
1.  public class Rectangle {
2.      public int width = 0;
3.      public int height = 0;
4.      public Point origin;    // Point is another class
5.
6.      // The four Constructors..
7.      public Rectangle( ) {
8.          origin = new Point(0, 0);
9.      }
10.
11.     public Rectangle(Point p) {
12.         origin = p;
13.     }
```

PROPERTIES OF DLC UI, IBADAN

```
14.
15.       public Rectangle(int w, int h) {
16.             this(new Point(0, 0), w, h);
17.       }
18.
19.       public Rectangle(Point p, int w, int h) {
20.             origin = p;
21.             width = w;
22.             height = h;
23.       }
24.
25.       // A method for moving the rectangle
26.       public void move(int x, int y) {
27.             origin.x = x;
28.             origin.y = y;
29.       }
30.
31.       //A method for computing the area of the rectangle
32.       public int area( ) {
33.             return width * width;
34.       }
35.     }
36.
37.       // Here's the code for the Point class
38.       public class Point {
39.             public int x = 0;
40.             public int y = 0;
41.
42.             // The only constructor for the class
43.             public Point(int x, int y) {
44.                         this.x = x;
45.                         this.y = y;
46.             }
47.   }
```

48. /* Here is a small program, called RectangleTest, which creates three objects: one Point object and two Rectangle objects. You will need all three source files to compile this program */

```
49.       public class RectangleTest {
50.             public static void main(String[ ] args) {
51.                // creates a point object and two rectangle objects
52.                Point origin_one = new Point(23, 94);
53.                Rectangle rect_one = new Rectangle(50, 100);
54.                Rectangle rect_two = new Rectangle( );
55.
56.                // display rect_one's width, height and area
57.                System.out.println("Width of rect_one: " +
58.                     rect_one.width);
59.                System.out.println("Height of rect_one: " +
60.                       rect_one.height);
61.                System.out.println("Area of rect_one: " + rect_one.area(
62.                           ));
63.
64.                // Set rect_two's position
65.                rect_two.origin = origin_one;
66.
67.                // Display rect_two's position
68.                System.out.println("X position of rect_two: " +
69.                      rect_two.origin.x);
```

PROPERTIES OF DLC UI, IBADAN

```
70.                 System.out.println("Y position of rect_two: " +
71.                     rect_two.origin.y);
72.
73.                 // Move rect_two and display its new position
74.                 rect_two.move(40,72);
75.                 System.out.println("New X position of rect_two: " +
76.                       rect_two.origin.x);
77.                 System.out.println("New Y position of rect_two: " +
78.                         rect_two.origin.y);
79.             }
80.     }
```

After creating the objects, the program manipulates the objects and displays some information about them. Here's the output from the program:

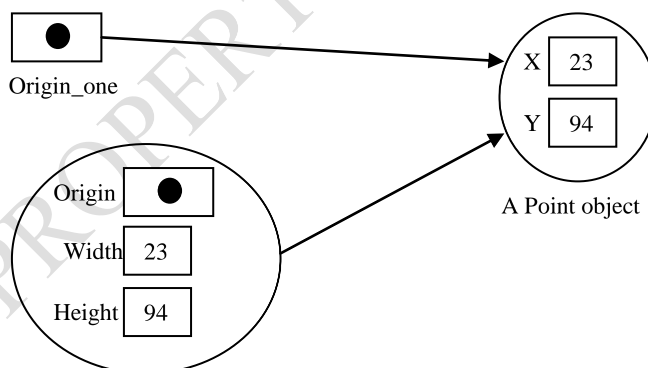
```
Width of rect_one: 100
Height of rect_one: 200
Area of rect_one: 20000
X position of rect_two: 23
Y position of rect_two: 94
New X position of rect_two: 40
New Y position of rect_two: 72
```

From the above program, you realize that each constructor let's you provide initial values for different aspects of the rectangle: the origin, width and the height, all three or none. The java platform differentiates the constructors, based on the number and the type of arguments. For instance when the java compiler encounters the following code:

```
Rectangle rect-one = new Rectangle(origin_one, 100, 200);
```

It knows to call the constructor in the rectangle class that requires a point argument followed by two integer arguments.

The call initializes the rectangle's origin variable to the point object referred to by origin_one. The code also sets width to 100 and height to 200. Multiple references can refer to the same object as in this case of the point object.



9.4.15 Writing Classes without Constructors and private fields

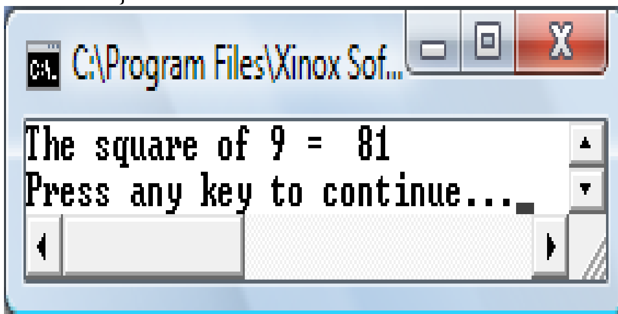
We have used the Math class methods without instantiating a new object of the class before we used them. For example, Math.pow, Math.PI, Math.E, etc. We can also write a class that may not have a constructor and without any private data field. The only thing is that the method of the class should be declared as static and must not refer to any of the fields declared for the class. To call the method from the class, we only need to write *className.method(argument),* like in the example below:

PROPERTIES OF DLCUI, IBADAN

demo.value(k). dem is the name of the class and value is a method we are calling from the class. k is the argument passed to the method from the main program.

Example:

```
1.  import javax.swing.*;
2.  class demo {
3.       static int value(int a) {
4.            return a*a;
5.        }
6.        public static void main(String[ ] args) {
7.       // computing the square of a number
8.        int k = Integer.parseInt(JOptionPane.showInputDialog ("Enter
9.              a number"));
10.       System.out.println("The square of " + k + " = "+ " " +
11.             demo.value(k));
12.       System.exit(0);
13.     }
14.  }
```



Exercise: Extend this class to compute the cube and cube-root of numbers.

Summary

In this chapter, you have been introduced to object oriented programming in Java. The concept of constructors, access modifiers and instantiation of classes to create objects have been learnt. It was seen that classes may be constructor-less while some may not even constructor and private fields

Post Test

1. Write a java program, **using a class** to compute the roots of any quadratic equation whose model equation is $ax^2 + bx + c = 0$.

2. Write a java class to search for x in a 1-dimensional array of integer data, x belongs to the class of integers.

3. Matrix has been a wonderful mathematical tool for solving real – life computational problems. Write Java class program that

- (a) Adds three sum-able matrices together.
- (b) Computes the product of any two multipliable matrices.
- (c) Add all the elements in the right diagonal of a square matrix A, with those in the left of another square matrix B, and prints out the summation.

(All conditional rules of matrix operations should be included in your codes).

PROPERTY OF DLC UI, IBADAN

10 *Programming Inheritance with Java*

10.1 Introduction

A class represents a set of objects which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. This is a powerful idea. However, something like this can be done in most programming languages. The central new idea in object-oriented programming–the idea that really distinguishes it from traditional programming–is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can be expressed using inheritance and polymorphism.

The idea behind inheritance in Object Oriented Programming (OOP) is that we can re-use or change the methods of existing classes, as well as add new instance fields and new methods in order to adapt them to new situations. For instance, we need to use inheritance to show text or graphics in a window.

10.2 Learning Objectives

At the end of this chapter, it is expected that you know how to implement inheritance in Java

10.3 Pre Test

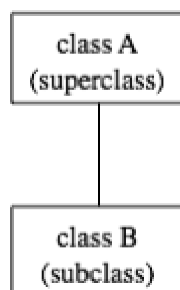
1. Explain how children inherit genes from their parents as you learned in O' Level Biology (Genetics).
2. Give a suitable definition to genetic inheritance as was learnt then.

10.4 Main Content

10.4.1 What is Inheritance?

Essentially, inheritance is a form of software re-usability in which new classes are created from existing classes by absorbing their attributes and behaviours and embellishing these with capabilities the new classes require. Software re-usability saves time in program development. It encourages reuse of proven and debugged high quality software, thus reducing problems after a system becomes operational. Polymorphism makes it easy to add new capabilities to a system. Inheritance and polymorphism are effective techniques for dealing with software complexity.

10.4.2 Inheritance and Class Hierarchy



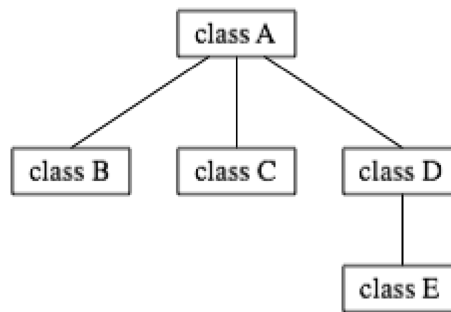
The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it

PROPERTIES OF DLC UI, IBADAN

inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived **class** and base **class** are used instead of subclass and superclass; this is the common terminology in C++.) A subclass can add to the structure and behaviour that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass.

In Java, to create a class named "B" as a subclass of a class named "A", you would write

```
class B extends A {
..
/ / additions to, and modification s of ,
. / / stuff inherited from class A
.
}
```



Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviours – namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations"of classes. This is shown in the diagram, where class E is a subclass of

When creating a new class, instead of writing completely new instance variables and instance methods, the programmer can designate that the new class is to inherit the instance variables and methods of a previously defined "**Superclass**". The new class is the "**Subclass**", which can become a superclass for some future subclass.

The direct superclass of a subclass is the superclass from which the subclass explicitly inherits (via the keyword 'extends'). An indirect superclass is inherited from two or more levels up the class hierarchy.

Illustrative Example

Consider books and magazines - both specific types of publication. We can show classes to represent these on a UML class diagram. In doing so we can see some of the instance variables and methods these classes may have.

Book
title author price copies
sellCopy() orderCopies()

Magazine
title price orderQty currIssue copies
sellCopy() adjustQty() recvNewIssue()

Attributes 'title', 'author' and 'price' are obvious. Less obvious is 'copies' this is how many are currently in stock. For books, orderCopies( ) takes a parameter specifying how many copies are added to stock. For magazines, orderQty is the number of copies received of each new issue and currIssue is the date/period of the current issue (e.g. "January 2013", "Fri 6 Jan", "Spring 2009" etc.) When a newIssue is received the old are discarded and orderQty copies are placed in stock. Therefore recvNewIssue( ) sets currIssue to date of new issue and restores copies to orderQty. adjustQty( ) modifies orderQty to alter how many copies of subsequent issues will be stocked.

These classes have three instance variables in common: title, price, copies.
They also have in common the method sellCopy( ).

The differences are as follows:

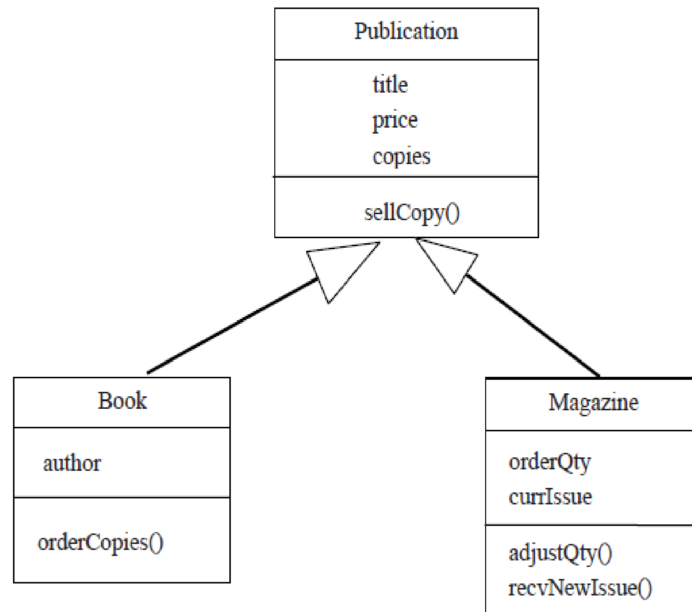
Book additionally has author, and orderCopies( ).
Magazine additionally has orderQty, currIssue, adjustQty( ) and recvNewIssue( ).

We can separate out ('factor out') these common members of the classes into a superclass called Publication.

Publication
title price copies
sellCopy()

The differences will need to be specified as additional members for the 'subclasses' Book and Magazine.

PROPERTY OF DLC UI IBADAN



In this UML Class Diagram, The hollow-centred arrow denotes inheritance.

Note the Subclass has the generalized superclass characteristics plus additional specialized characteristics. Thus the Book class has four instance variables (title, price, copies and author) it also has two methods (sellCopy( ) and orderCopies( )). The inherited characteristics are NOT listed in subclasses. The arrow shows they are acquired from superclass.

10.4.3 Inheritance in Java

Java only supports single inheritance whereby a class is derived from one Superclass. This is opposed to C^{++}, which supports multiple inheritance. This is archived in Java through "interfaces".

The superclass is not superior to its subclass or contains more functionality. In fact, the opposite is true; subclasses have more functionality than their superclasses. This is because a subclass normally adds instance variables and instance methods of its own, making it more robust. Every object of a subclass is also an object of the subclass's superclass. The converse is not true. Subclass methods and methods of other classes in the same package as the superclass can access 'protected' superclass members.

Recall our 'is a' and 'has a' relationships. 'Is a' is inheritance and 'has a' is composition. In a "has a" relationship, a class object has one or more objects of other classes as members.

A subclass's methods may need to access certain of its superclass instance variables and methods. If a subclass could access the superclass's 'private' members, this would violate information hiding in the superclass. A subclass can however access the 'public', 'protected' and 'package' access members of its superclass provided it is in the same package as the superclass. Superclass members that should not be accessible to a subclass via inheritance are declared 'private' in the superclass. A subclass can effect state changes in superclass 'private' members only through 'public', 'protected and package access methods provided in the superclass and inherited into the subclass.

One problem with inheritance is that a sub class can inherit methods that it does not need or should not have; but this could be 'overridden' (redefined) in the subclass with an appropriate implementation. However, someday, most software may be constructed from "standardized reusable components" just as hardware is often constructed today. This will help meet the challenges of developing the ever more powerful software we will need in the future.

PROPERTIES OF DLC UI, IBADAN

Examples of Superclass and Subclasses

Superclass	Subclasses
Quadrilateral	Rectangle, square, parallelograms, Rhombus, Trapezium. (For instance a rectangle is a quadrilateral").
Shape	Circle, Triangle, Rectangle
Student	Graduate, Undergraduate
Loan	Car loan, mortgage loan, Agric Loan
Staff	Academic, Non-academic, Junior, Senior.
Account	Current, Savings, Special.

10.4.4 Implementing Inheritance in Java

No special features are required to create a superclass. Thus any class can be a superclass unless specifically prevented. A subclass specifies it is inheriting features from a superclass using the keyword **extends**. For example

```
class MySubclass extends MySuperclass
{
   // additional instance variables and
   // additional methods
}
```

10.4.4.1 Constructors

Each class (whether sub or super) should encapsulate its own initialization, usually relating to setting the initial state of its instance variables. A constructor for a superclass should deal with general initialization. Each subclass can have its own constructor for specialised initialization but it must often invoke the behaviour of the superclass constructor. It does this using the keyword **super**.

```
class MySubClass extends MySuperClass
{
   public MySubClass (sub-parameters)
   {
      super(super-parameters);
      // other initialization
}
```

If **super** is called, ie. the superclass constructor, then this must be the first statement in the onstructor.

Usually some of the parameters passed to MySubClass will be initialized values for superclass instance variables, and these will simply be passed on to the superclass constructor as parameters. In other words *super-parameters* will be some (or all) of *sub-parameters*.

Shown below are two constructors, one for the Publication class and one for Book. The book constructor requires four parameters three of which are immediately passed on to the superclass constructor to initialize its instance variables.

```
public Publication (String pTitle, double pPrice, int pCopies)
{
   title = pTitle;
   // etc.
}
```

PROPERTIES OF DLC UI, IBADAN

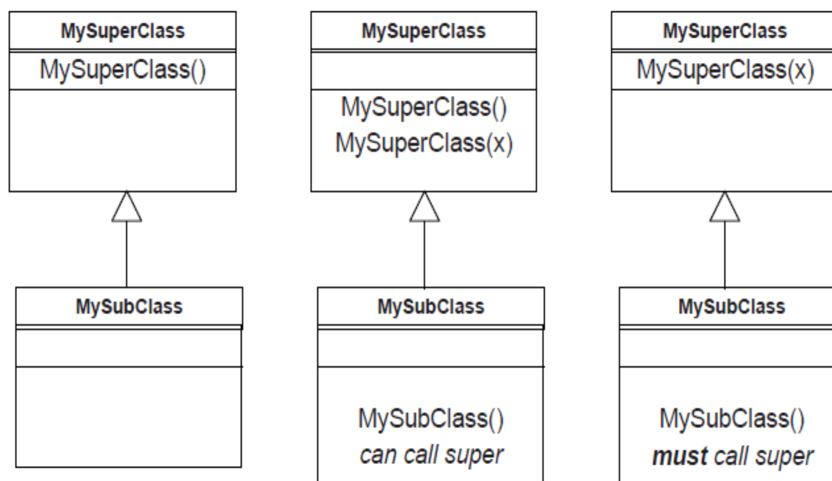
```
public Book (String pTitle, String pAuthor, double pPrice,int pCopies)
{
   super(pTitle, pPrice, pCopies);
   author = pAuthor;
   //etc.
}
```

10.4.4.2 Constructor Rules

Rules exist that govern the invocation of a superconstructor.

If the superclass has a parameterless (or default) constructor this will be called automatically if no explicit call to super is made in the subclass constructor though an explicit call is still better style for reasons of clarity.

However if the superclass has no parameterless constructor but does have a parameterized one, this **must** be called explicitly using super.
To illustrate this….



On the left above:- it is legal, though bad practice, to have a subclass with no constructor because superclass has a parameterless constructor.

In the centre:- if subclass constructor doesn't call super, the parameterless superclass constructor will be called.

On the right:- because superclass has no paramterless constructor, subclass **must** have a constructor and it **must** call super. This is simply because a (super) class with only a parameterized constructor can only be initialized by providing the required parameter(s).

10.4.4.3 Access Control

To enforce encapsulation we normally make instance variables **private** and provide accessor/mutator methods as necessary. The sellCopy( ) method of Publication needs to alter the value of the variable 'copies' it can do this even if 'copies' is a private variable. However Book and Magazine both need to alter 'copies'. There are two ways we can do this …

- make 'copies' 'protected' rather than 'private' – this makes it visible to subclasses, **or**
- create accessor and mutator methods.

PROPERTY … IBADAN

For variables we generally prefer to create accessors/mutators rather than compromise encapsulation though **protected** may be useful to allow subclasses to use methods (e.g. accessors and mutators) which we would not want generally available to other classes.

Thus in the superclass Publication we define 'copies' as a variable private but create two methods that can set and access the value 'copies'. As these accessor methods are public or protected they can be used within a subclass when access to 'copies' is required.

In the superclass Publication we would therefore have….

```
private int copies;

public int getCopies () {
   return copies;
}

public void setCopies(int pCopies) {
   copies = pCopies;
}
```

These methods allow superclass to control access to private instance variables.

As currently written they don't actually impose any restrictions, but suppose for example we wanted to make sure 'copies' is not set to a negative value.

(a) If 'copies' is **private**, we can put the validation (i.e. an if statement) within the setCopies method here and know for sure that the rule can never be compromised.

(b) If 'copies' is partially exposed as **protected**, we would have to look at every occasion where a subclass method changed the instance variable and do the validation at each separate place.

We might even consider making these *methods* **protected** rather than **public** themselves so their use is restricted to subclasses only and other classes cannot interfere with the value of 'copies'.

Making use of these methods in the sublasses Book and Magazine we have ..

```
// in Book
public void orderCopies(int pCopies)
{
   setCopies(getCopies( ) + pCopies);
}

// and in Magazine
public void recvNewIssue(String pNewIssue)
{
    setCopies(orderQty);
    currIssue = pNewIssue;
}
```

These statements are equivalent to
mCopies = mCopies + pCopies
and
mCopies = mOrderQty

PROPERTIES OF DLC UI, IBADAN

10.4.4.4 Abstract Classes

The idea of a Publication which is not a Book or a Magazine is meaningless, just like the idea of a Person who is neither a MalePerson nor a FemalePerson. Thus while we are happy to create Book or Magazine objects we may want to prevent the creation of objects of type Publication.

If we want to deal with a new type of Publication which is genuinely neither Book nor Magazine – e.g. a Calendar – it would naturally become another new subclass of Publication. As Publication will never be instantiated ie. we will never create objects of this type the only purpose of the class exists is to gather together the generalized features of its subclasses in one place for them to inherit.

We can enforce the fact that Publication is non-instantiable by declaring it 'abstract':-

```
abstract class Publication
{
  // etc.
```

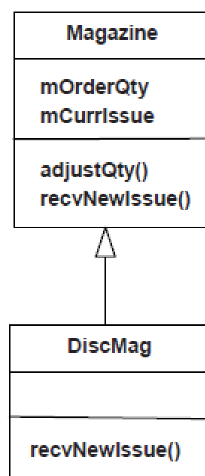
10.4.4.5 Overriding Methods

A subclass inherits the methods of its superclass and must therefore always provide at least that set of methods, and often more. However, the implementation of a method can be changed in a subclass. This is overriding the method.

To do this we write a new version in the subclass which replaces the inherited one. The new method should essentially perform the same functionality as the method that it is replacing however by changing the functionality we can improve the method and make its function more appropriate to a specific subclass.

For example, imagine a special category of magazine which has a disc attached to each copy – we can call this a DiscMag and we would create a subclass of Magazine to deal with DiscMags. When a new issue of a DiscMag arrives not only do we want to update the current stock but we want to check that the discs are correctly attached. Therefore we want some additional functionality in the recvNewIssue( ) method to remind us to do this. We achieve this by redefining recvNewIssue( ) in the DiscMag subclass.

Note: when a new issue of Magazine arrives, as these don't have a disc we want to invoke the original recNewIssue( ) method defined in the Magazine class.



- The definition of **recvNewIssue()** in DiscMag overrides the inherited one.
- Magazine is not affected – it retains its original definition of **recvNewIssue()**
- By showing **recvNewIssue()** in DiscMag we are stating that the inherited method is being overridden (ie. replaced) as we do not show in inherited methods in subclasses.

PROPERTIES OF DLC UI, IBADAN

When we call the **recvNewIssue( )** method on a DiscMag object Java automatically selects the new overriding version – the caller doesn't need to specify this, or even know that it is an overriden method at all. When we call the **recvNewIssue( )** method on a Magazine it is the method in the superclass that is invoked.

10.4.4.6 Implementing DiscMag

To implement DiscMag we must create a subclass of Magazine using extends. No additional instance variables or methods are required though it is possible to create some if there was a need. The constructor for DiscMag simply passes ALL its parameters directly on to the superclass and a version of newIssue( ) is defined in discMag to overrides the one inherited from Magazine (see next code).

```
public class DiscMag extends Magazine {
  // the constructor
        public DiscMag (String pTitle, double pPrice, int pOrderQt,String
                  pCurrIssue, int pCopies)  {

            super(pTitle, pPrice, pOrderQty, pCurrIssue, pCopies);
      } // end constructor

     // the overridden method
     public void recvNewIssue(String pNewIssue) {
         super.recvNewIssue(pNewIssue);
         System.out.println("Check discs attached to this magazine");
      } // end method
} // end class
```

Note the user of the **super** keyword to call a method of the superclass, thus re-using the existing functionality as part of the replacement, just as we do with constructors. It then additionally displays the required message for the user.

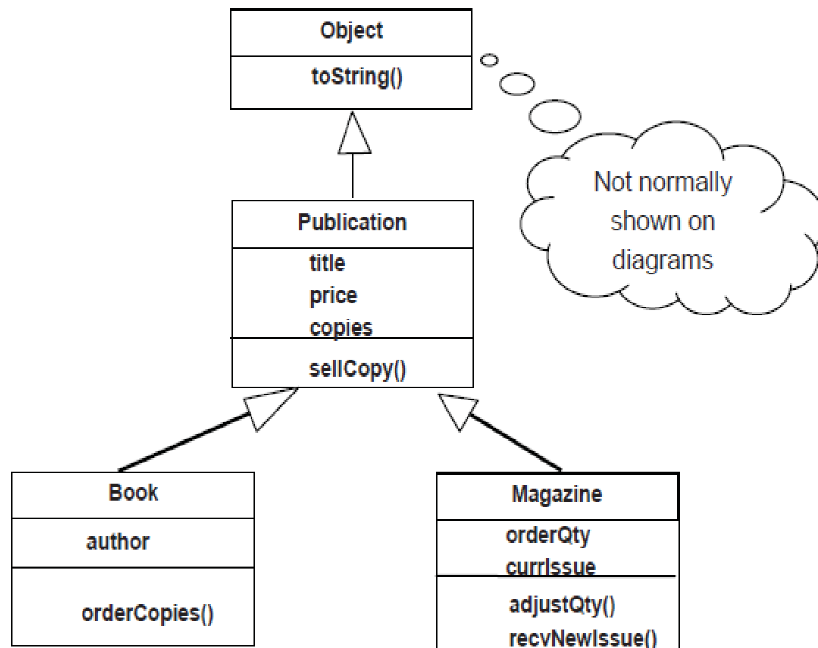
10.4.5 The 'Object' Class

In Java all objects are (direct or indirect) subclasses of a class called 'Object'. Object is the 'root' of the inheritance hierarchy in Java. Thus this class exists in every Java program ever created. If a class is not declared to extend another then it implicitly extends Object.

Object defines no instance variables but several methods. Generally these methods will be overridden by new classes to make them useful. An example is the **toString()** method. Thus when we define our own classes, by default they are direct subclasses of Object.

If our classes are organised into a hierarchy then the topmost superclass in the hierarchy is a direct subclass of object, and all others are indirect subclasses. Thus directly, or indirectly, all classes created in Java inherit toString( ).

PROPERTIES OF DLC UI, IBADAN



10.4.5.1 Overriding toString( ) defined in 'Object'

The Object class defines a toString( ) method, one of several useful methods.

toString( ) has the signature:
```
public String toString( )
```

Its purpose is to return a string value that represents the current object. The version of toString( ) defined by Object produces output like: "Book@11671b2". This is the class name and the "hash code" of the object. However to be generally useful we need to override this to give a more meaningful string.

In Publication
```
public String toString( )
{
   return mTitle;
}
```

In Book
```
public String toString( )
{
  return super.toString( ) + " by " + mAuthor;
}
```

In Magazine
```
public String toString( )
{
  return super.toString( ) + " (" + mCurrIssue + ")";
}
```

In the code above toString( ) originally defined in Object has been completely replaced, i.e. overridden, so that Publication.toString( ) returns just the title.

PROPERTIES OF DLC UI, DAN

The toString( ) method has been overridden again in Book such that Book.toString( ) returns title (via superclass toString( ) method) and author. Ie. this overridden version uses the version defined in Publication. Thus if Publication.toString( ) was rewritten to return the title and ISBN number then Book.toString( ) would automatically return the title, ISBN number and author. Magazine.toString( ) returns title (via superclass toString() method) and issue

We will not further override the method in DiscMag because the version it inherits from Magazine is OK. We could choose to provide more data (i.e. more, or even all, of the instance variable values) in these strings. The design judgement here is that these will be the most generally useful printable representation of objects of these classes. In this case title and author for a book, or title and current issue for a magazine, serve well to uniquely identify a particular publication.

Simple Example:

```
1.  class Rectangle {
2.      float l;
3.      float b;
4.       //constructor
5.       public Rectangle(float length,float breadth) {
6.           l = length;
7.           b = breadth;
8.         }
9.
10.     public float area( ) {
11.         return l*b;
12.     }
13.     public double perimeter( ) {
14.         return 2.0*(l+b);
15.      }
16.  }
```

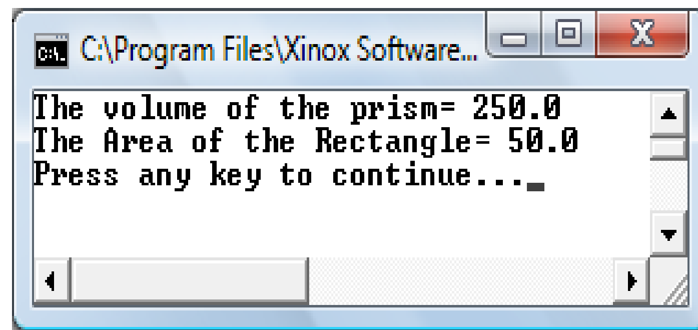
```
1.  //The following class inherits from the Rectangle class.
2.  class Prism extends Rectangle {
3.        float h;
4.        //constructor
5.        public Prism(float l,float b,float height){
6.              super(l, b);
7.               h = height;
8.        }
9.             //method to compute volume
10.     public float volume( ){
11.          return h * super.area( );
12.      }
13.  }
```

The following is a program to test the two clases.

```
1.  public class Tester {
2.        public static void main(String[ ] args){
3.              Prism newPrism = new Prism(10,5,5);
4.              System.out.println("The volume of the prism= "+
5.                   newPrism.volume( ));
6.              Rectangle r = new Rectangle(10,5);
7.              System.out.println("The Area of the Rectangle =
8.                 "+r.area( ));
9.        }
10.  }
```

PROPERTIES OF DLC UI, IBADAN

The Output:



Note: Each of the above programs is compiled separately in different files.

Summary

Inheritance allows us to factor out common attributes and behaviour. We model the commonalities in a superclass. Subclasses are used to model specialized attributes and behaviour. Code in a superclass is inherited to all subclasses. If we amend or improve code for a superclass it impacts on all subclasses. This reduces the code we need to write in our programs.

Special rules apply to constructors for subclasses.

A superclass can be declared **abstract** to prevent it being instantiated (i.e. objects created). We can 'override' inherited methods so a subclass implements an operation differently from its superclass.

In Java all classes descend from the class 'Object'. 'Object' defines some universal operations which can usefully be overriden in our own classes.

Post Test

Design and implement a program involving inheritance that will make use of banking accounts, deposits and withdrawals.

11 *File Handling with Java*

11.1 Introduction

All along, we have been using the keyboard and monitor to input data into our programs and output the results of computations respectively from our programs. We now turn our attention to the use of files. A *file* stores data permanently unlike variables and arrays, which loose data as soon as the programs that use them terminates. Java programs can read data as input from files kept in the secondary storage of computers such as harddisk or flash, and directs output results to another file in the storage.

Data items collected about a particular entity form the *fields*. For instance, data items about a student could be name, matric, sex, age, ume_score, etc. The combination of all these fields forms a *record*. Thus, a record is a group of related fields. Each of these fields has a *type* associated with it. Name is a *string* data type, while matric and ume_score are of type *int.*

Records are usually arranged in form of table as shown below:

Rec_no	Name	Matric	Age	Sex	Ume_Score
o1	Akinola s. O.	68888	25	m	222
02	Zacheus M.G.	133564	17	f	257
….	….	….		….	

The combination of all these records forms a *file*. Thus, a file is a group of related records.

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record *key*. Usually, the key is unique to every record to avoid duplication of records in the file. In the example table above, matric is a good candidate for the record key. The key is also used for searching and sorting records in a file.
Records in a file could be organized *sequentially*, in which records are stored in order by the record key, or in *random* order, in which case, there is no particular ordering of the records in the file, but the records are indexed.

Usually, in big organizations, data are stored in many files. For example we could have customer file, payroll file, recruitment file, supplies file, etc. These files are kept in a database. A group of related files is called a *database.* We introduce this concept in the next chapter.

11.2 Objectives

At the end of this chapter, you should be able to use sequential files in your programs.

11.3 Pre Test

(i) What are fields, records and files
(ii) Mention one business application of file.

PROPERTIES OF DLC UI, IBADAN

11.4 Main Contents

11.4.1 Java Perception of Files

Java views each file as a sequential stream of bytes. The end of file marker is usually provided by the Operating System. In some cases, the end-of-file indication occurs as an exception, and in other cases, the indication is a return value from a method invoked on a stream-processing object.

When java opens a file, an object in which a stream of bytes is associated with, is created. In some cases, java associates streams of bytes with any of the three different devices: *System.in, System.out* and *System.err.* Object System.in allows the input of data via the keyboard, System.out allows output of result to the screen and System.err allows output of error messages to the screen. All the three form the standard files, otherwise called standard stream objects in Java.

Each of these standard stream objects could be redirected to different location. The System.in enables the program to read from a different source while System.out and System.err enable output to be sent into some other locations, such as a file on disk. The *SetIn, SetOut* and *SetErr* methods from the Class System provide these capabilities.

Java.io package must be imported to a java program before files can be used in that program. The java.io package provides definitions for stream classes and methods such as

- FileInputStream – for byte-based input from a file
- FileOutputStream – for byte-based output to a file
- FileReader – for character-based input from a file
- FileWriter – for characterbased output to a file.

Files are opened by creating objects of these stream classes that inherit from classes InputStream, OutputStream, Reader and Writer respectively.to perform input and output of data types, objects of classes ObjectInputStream, DataInputStream, ObjectOutputStream and DataOutputStream will be used togther with the byte-based file stream classes FileInputStream and FileOutputStream.

Files that are created using byte-based streams are referred to as binary files, while files created using character-based streams are referred to as **text files**. Text files can be read by text editors, while binary files are read by a program that converts the data to a human-readable format.

Java provides many classes for performing input/output operations.

- InputStream and OutputStream subclasses of object are abstract classes that declare methods for performing byte-based input and output respectively.
- FileInputStream subclass of InputStream and FileOutputStream subclass of OutputStream are used to manipulate files.
- PipedOutputStream, PipedInputStream, are used in connection with threads in java.
- A FilterInputStream *filters* an InputStream as FilterOutputStream filters OutputStream. Filtering in this case could be *buffering, monitoring line numbers or aggregating data bytes into meaningful primitive type units.*
- A printStream subclass of FilterOutputStream performs text output to the specified stream. Systemout and System.err are PrintStream objects.

Reading data as raw bytes is fast but crude. Usually, programs read data as aggregates of bytes in form of an int, a float, a double, etc. Classes DataInputStream and RandomAccessFile from the interface DataInput provides methods readLine (for byte arrays), readBoolean, readByte, readChar, readInt, readDouble, readFloat, readFully (for byte arrays), readLong, readShort, readUnsignedByte, readUnsignedShort, readUTF (for strings) and SkipBytes for reading the primitive types. Classes DataOutputStream (a subclass of FilterOutputStream) and RandomAccessFile each implements the Interface DataOutput to write primitive type values such as byte or int. interface DataOutput contins

PROPERTIES OF DLC UI, IBADAN

methods such as write (for byte and byte arrays), writeBoolean, WriteDouble, writeFloat, writeChars (for Unicode strings), writeLong, writeSjhort, writeUTF, writeBytes, and writeChar.

In addition, large chunks of data may be transferred to a temporary storage before being read or written to enhance speed of data transfer. This is known as buffering technique. BufferedInputStream, a subclass of FileInputStream and BufferedOutputStream, a subclass of FileOutputStream are used for this purpose. Typical input/output operations are extremely slow compared with the speed of accessing computer memory. Buffered inputs and outputs normally yield significant performance improvements over unbuffered inputs and outputs.

In addition to the byte-based streams, we have Reader and Writer classes, which are Unicode two-byte, character-based streams. Most of the byte-based streams have corresponding character-based Reader and Writer classes.

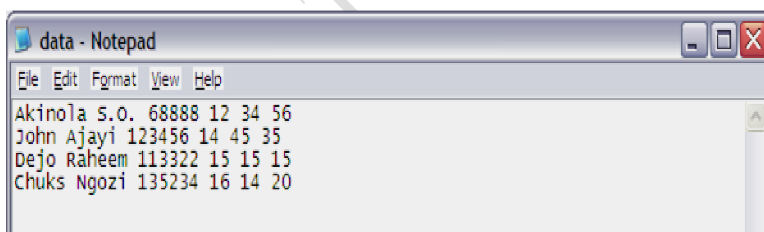
- Classes BufferedReader and BufferedWriter enable efficient buffering for character-based streams.
- Classes CharArrayReader and CharArrayWriter read and write respectively, a stream of characters to a character array.
- A LineNumberReader is a buffered character stream that keeps track of line numbers (i.e. a new line, a return or a carriage-return-line-feed combination).
- Class FileReader and FileWriter read characters from and write characters to a file.
- A PrintWriter writes characters to a stream.
- StringReader and StrigWriter read and write characters respectively.

In this text, we introduce the reader to sequential file processing in java.

11.4.2 Processing Sequential Access Data File

In this section, we implement a program to process a sequential data file containing both string and numeric data. This program shows how to read data sequentially from a text file.

We firstly create an input data file named "data.dat" and saved in the same folder as our program, i.e. the bin subfolder of the java toolkit. Each record in the file contains the following fields: Surname, Othernames, Matric No., and three scores in some courses. Each field is separated by a space. Figure below shows the screen shot of the input file.



The program listing on next page shows the program that uses the input file while the figure below it shows the output file created by the program itself in the same bin folder.

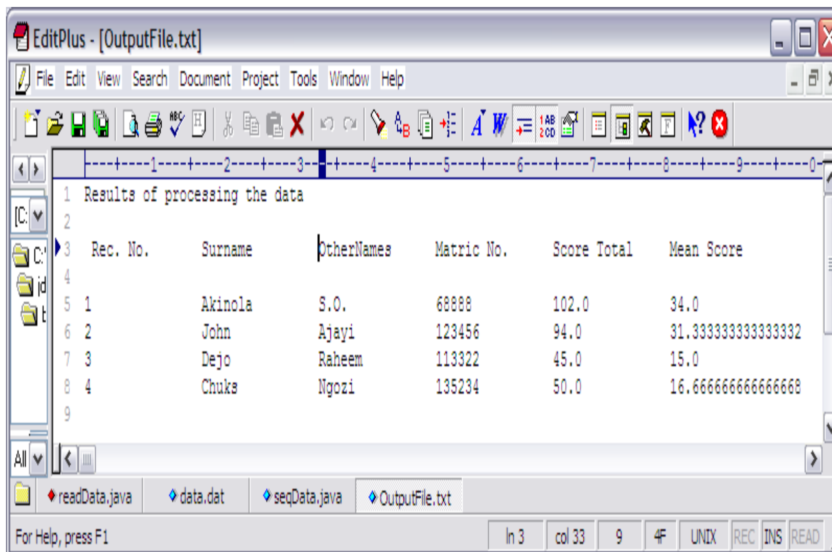
PROPERTIES OF DLC UI, IBADAN

Example Program:

```
1.  import java.io.*;
2.  import java.util.*;
3.  import javax.swing.*;
4.  // sequential file demo
5.   public class seqData {
6.     public static void main(String[ ] arguments)  {
7.       try {  // try1
8.           Scanner input = new Scanner(new File( "data.dat" ));
9.          int count = 0; // for counting records
10.         String out = "Results of processing the data \n\n ";
11.          out += "Rec. No. \tSurname \tOtherNames \tMatric
12.              No.\tScore Total\tMean Score\n\n";
13.         // read all the records
14.           while(input.hasNext( )) {
15.               count++;
16.         // Assign all the records' fields to their identifiers
17.               String name1 = input.next( );
18.                String name2 = input.next( );
19.                 int matric = input.nextInt( );
20.                 int score1 = input.nextInt( );
21.                 int score2 = input.nextInt( );
22.                  double score3 = input.nextDouble( );

                      //summing data
23.               double sum = score1 + score2 + score3;
24.                  double average = sum/3.0; // the mean
25.              out += count +"\t\t"+ name1 +"\t\t" +   name2 +"\t\t"
26.               + matric +"\t\t"+   sum +"\t\t"+  average  +"\t\t"
27.               + "\n";
28.                   // writing to an output file
29.            try {  // try2 to operate output file
30.                 //Creating the output file
31.               File file = new File("OutputFile.txt");
32.               PrintWriter output = new PrintWriter(file);
33.               output.write(out);     // write all the output
34.                 output.close();     //Close the output file
35.          }//end try2
36.
37.          catch(Exception exception){    //catch for try2
38.    JOptionPane.showMessageDialog(null,"Cannot write to
39.     file!","Error",JOptionPane.ERROR_MESSAGE);
40.           }//end catch for try2
41.     }//end while
42.   } catch(Exception exception){ //catch for try1
43. JOptionPane.showMessageDialog(null,"Cannot read from
44.          file!","Error",JOptionPane.ERROR_MESSAGE);
45.      }//end catch for try1
46.   //Notifying the user of the end of processing…
47.  JOptionPane.showMessageDialog(null,"Data has been
             written to file!","Writing Complete",
48.            JOptionPane.INFORMATION_MESSAGE);
49. } // end method main( )
50.}  //end class seqData
```

Output: Use WordPad to open the output if you don't have EditPlus.

PROPERTIES OF DLC UI, IBADAN



You need to study this program carefully line by line in order to understand how to use file for your subsequent programs. Below is the explanation for the program.

The program demonstrates how class Scanner can be used to input data from a file rather than the keyboard.

The Scanner is a Sub class of Java.util class (util means utility). Therefore, both java.io, javax.swing and java.util APIs are imported into the program in lines 1 to 3. Swing is for the JOptionPane. Line 14 uses Scanner method hasNext( ) to determine whether the end-of-file key combination has been entered. The loop executes until hasNext( ) encounters the end-of-file marker.

Line 7 contains a try statement. All processes in a file must be put in a try – catch block. This is a necessity as java believes that an error may occur in reading from or writing to a file. For instance, there may be file input error, in which the file name specified is not found.

```
8.   Scanner input = new Scanner(new File( "data.dat" ));
```

This line combines two functions: creating the new file object. We pass a File object to the constructor, which specifies that the Scanner object will read from the file "data.dat" located in the directory from which the application executes (bin). If the file cannot be found, a FileNotFoundException occurs. The exception is handled in lines 42 and “Cannot read from file” is generated in line 43-44. We could specify the complete path of the file to specify where it could be located; but it is advisable to save the file in the bin sub-folder of the java toolkit to avoid File input error.

Lines 10 – 12 are for generating the headings in the output as shown in the output screen shot. We generated the output in tabulated format with appropriate headings.

Line 17 – 22.

```
16.   // Assign all the records’ fields to their identifiers
17.         String name1 = input.next( );
18.         String name2 = input.next( );
19.         int matric = input.nextInt( );
20.         int score1 = input.nextInt( );
21.         int score2 = input.nextInt( );
22.         double score3 = input.nextDouble( );
```

PROPERTIES OF DLC UI IBADAN

In order to read a String data by the Scanner, input.next( ) method (or function) is used. To read an integer data, we use the input.nextInt( ) method. This goes for float, input.nextFloat( ) and double input.nextDouble( ). Note that the Scanner reads a record at a time as tokens (i.e. field by field). Therefore, we assign each field to their individual field names. It is possible to read the three scores into a one-dimensional array, and then use for-loop to read the data into the array, especially if we have more than three scores. However, this is left as an exercise for the reader.

Line 25 – 27:

```
25.  out += count +"\t\t"+ name1 +"\t\t" +   name2 +"\t\t"
26.   + matric +"\t\t"+   sum +"\t\t"+  average   +"\t\t"
27.    + "\n";
```

These lines are used to ‘append’ the results of summation and mean to the out string, which is going to be printed later.

Lines 29 – 40:

```
29.           try {  // try2 to operate output file
30.                  //Creating the output file
31.                File file = new File("OutputFile.txt");
32.                PrintWriter output = new PrintWriter(file);
33.                output.write(out);     // write all the output
34.                  output.close( );     //Close the output file
35.        }//end try2
36.
37.        catch(Exception exception){    //catch for try2
38.    JOptionPane.showMessageDialog(null,"Cannot write to
39.     file!","Error",JOptionPane.ERROR_MESSAGE);
40.          }//end catch for try2
```

These lines are for opening and closing the output file. However, the program will automatically open the output file and gives it the name we specify as parameter for the File constructor, "OutputFile.txt". All files opened must be closed, why? Can you suggest an answer for this? In the program, we did not close the input file. Can you suggest an appropriate position where the close should be put?

The catch statements are the necessary error routines that should be appended to the try statements in case there are errors in reading and writing to files. Note in particular how these statements are arranged to match the two try blocks.

Line 47 and 48:

```
47.  JOptionPane.showMessageDialog(null,"Data has been
             written to file!","Writing Complete",
48.          JOptionPane.INFORMATION_MESSAGE);
```

These statements inform the user of the program that the data has been processed.

11.4.3 Other File Methods

Method	Description
boolean canRead( )	Returns true if a file is readable, false otherwise
Boolean canWrite( )	Returns true if a file is writable, false otherwise
boolean exists( )	Returns true if the name specified as the argument to the File constructor is a file or directory in the specified path; false otherwise
boolean isFile( )	Returns true if the name specified as the argument to the File Constructor is a file, false otherwise

PROPERTIES OF DLC UI, IBADAN

boolean isDirectory( )	Returns true if the name specified as the argument to the File Constructor is a directory, false otherwise
boolean isAbsolute( )	Returns true if the arguments specified to the File Constructor indicate an absolute path to a file or directory; false otherwise.
String getAbsolutePath( )	Returns a string with the absolute path of the file or directory.
String getName()	Returns a string with the name of the file or directory
String getPath( )	Returns a string with the path of the file or directory
Long length( )	Returns the length of the file, in bytes. Returns 0 if the File object is a directory
Long lastModified( )	Returns a platform representation of the time at which the file or directory was last modified.
String[ ] list( )	Returns an array of strings representing the contents of a directory. Returns null if the file object is not a directory.

Summary

In this chapter, you have been briefly introduced to the manipulating of sequential files in Java. Move ahead to learn the manipulation of random files in Java. Consult text books or tutorials online.

PROPERTIES OF DLC UI, IBADAN

Post Test

1. Copy the code line by line into your editor. Compile and run the code and report your observations. You must have created the input data file before you execute the program. The line numbers in the program are not part of the program. They are there for explanation purpose only.

Modify the input data to contain 10 or more scores per record. Modify the program to use array for the scores and compile and run the program. Record your observations and errors incurred in the exercise.

12 *Database Handling with Java*

12.1 Introduction

The Java Database Connectivity Application Programming Interface (JDBC API) is a Java API that can access any kind of tabular data, especially data stored in a relational database. In this chapter you shall be introduced to writing business applications involving databases.

12.2. Objectives

At the end of this chapter, you should be able to know how to write and even implement business applications that make use of databases.

12.3 Pre Test

Explain the differences between a file and a database.

12.4 Main Contents

12.4.1 JDBC: An Introduction

The Java Database Connectivity (JDBC), helps you to write java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

The following simple code fragment gives a simple example of these three steps:

```
1.  Connection con =  DriverManager.getConnection( "jdbc:myDriver:wombat",
2.                              "myLogin","myPassword");
3.  Statement stmt =  con.createStatement( );
4.  ResultSet rs =  stmt.executeQuery("SELECT a, b, c FROM Table1");
5.  while (rs.next( )) {
6.      int x = rs.getInt("a");
7.      String s = rs.getString("b");
8.      float f = rs.getFloat("c");
9.      }
```

This short code fragment instantiates a DriverManager object to connect to a database driver and log into the database, instantiates a Statement object that carries your Structured Query Language (SQL) query to the database; instantiates a ResultSet object that retrieves the results of your query, and executes a simple while loop, which retrieves and displays those results. It's that simple.

12.4.2 JDBC Product Components

JDBC includes four components:

1. The JDBC API

The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and

PROPERTIES OF DLC UI, IBADAN

propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

The JDBC API is part of the Java platform, which includes the *Java™ Standard Edition* (Java™ SE) and the *Java™ Enterprise Edition* (Java™ EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.

2. JDBC Driver Manager

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

The Standard Extension packages javax.naming and javax.sql let you use a DataSource object registered with a *Java Naming and Directory Interface*™ (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a DataSource object is recommended whenever possible.

3. JDBC Test Suite

The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

4. JDBC-ODBC Bridge

The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture. A driver is a java interface that includes a method that is used to obtain database connections.

There are many possible implementations of JDBC drivers. These implementations are categorized as follows:

- **Type 1**: Drivers that implement the JDBC API as a mapping to another data access API, such as ODBC (Open Database Connectivity). Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a Type 1 driver.
- **Type 2**: Drivers that are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited. Oracle's OCI (Oracle Call Interface) client-side driver is an example of a Type 2 driver.
- **Type 3**: Drivers that use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client's requests to the data source.
- **Type 4**: Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

Check which driver type comes with your DBMS. Java DB comes with two Type 4 drivers, an Embedded driver and a Network Client Driver. MySQL Connector/J is a Type 4 driver.

Installing a JDBC driver generally consists of copying the driver to your computer, then adding the location of it to your class path. In addition, many JDBC drivers other than Type 4 drivers require you to install a client-side API. No other special configuration is usually needed.

PROPERTIES OF DLC UI, IBADAN

12.4.3 JDBC Architecture

Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

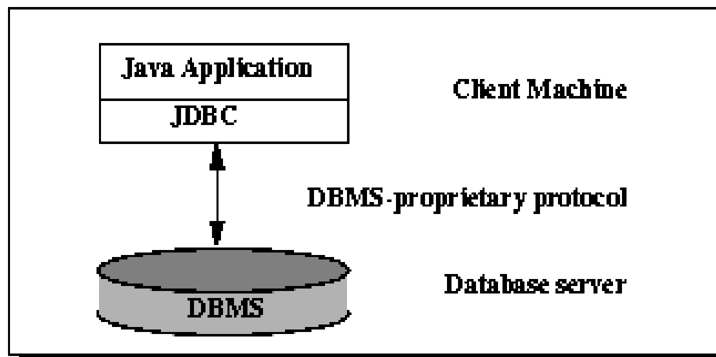


Figure 12.1: Two-tier Architecture for Data Access.

In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

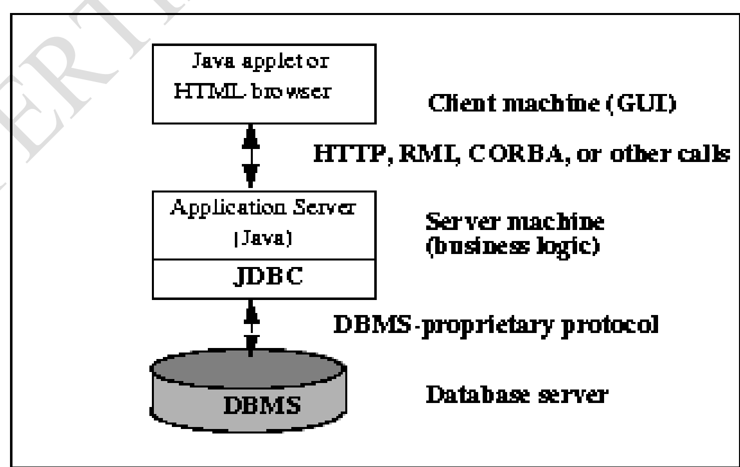


Figure 12.2: Three-tier Architecture for Data Access.

Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java

PROPERTIES OF DLC UI, IBADAN

platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

12.4.4 A Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

- **Integrity Rules**

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

Table 12.1 illustrates some of these relational database concepts. It has five columns and four rows, with each row representing a different student.

Table 12.1: Table Students

Matric_Number	Car_Number	First_name	Last_Name	Score
10001	5	Akinola	Solomon	45
10083		Abel	John	56
10005	12	Hammed	F.	67
10035		John	Solomon	46

The primary key for this table would generally be the Matric_number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also

PROPERTIES OF DLC UI, IBADAN

be possible to use First_Name and Last_Name because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Solomon." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If John Ibekwe gets an admission at this school and the primary key is First_Name, the RDBMS will not allow his name to be added (if it has been specified that no duplicates are permitted). Because there is already a John in the table, adding a second one would make the primary key useless as a way of identifying just one row. Note that although using First_Name and Last_Name is a unique composite key for this example, it might not be unique in a larger database.

Manipulating a database involves the use of *Structured Query Language* (SQL) to create tables, retrieve data from the database and edit data in the database.

SQL query keywords.	
SQL keyword	**Description**
SELECT	Retrieves data from one or more tables.
FROM	Tables involved in the query. Required in every SELECT.
WHERE	Criteria for selection that determine the rows to be retrieved, deleted or updated. Optional in a SQL query or a SQL statement.
GROUP BY	Criteria for grouping rows. Optional in a SELECT query.
ORDER BY	Criteria for ordering rows. Optional in a SELECT query.
INNER JOIN	Merge rows from multiple tables.
INSERT	Insert rows into a specified table.
UPDATE	Update rows in a specified table.
DELETE	Delete rows from a specified table.

- **SELECT Statements**

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the SELECT statement.

A SELECT statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A SELECT statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Students
WHERE Matric_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the Matric_Number column) follows. The first name and last name are printed for each row that satisfies the requirement

PROPERTIES OF DLC UI, IBADAN

because the SELECT statement (the first line) specifies the columns First_Name and Last_Name. The FROM clause (the second line) gives the table from which the columns will be selected.

First_name	Last_Name
Akinola	Solomon
Abel	John
Hammed	F.
John	Solomon

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no WHERE clause). Note that SELECT * means

"SELECT all columns."

```
SELECT *
FROM Students
```

- **WHERE Clauses**

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Jo'.

```
SELECT First_Name, Last_Name
FROM Students
WHERE Last_Name LIKE Jo%'
```

The keyword LIKE is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Jo', which signifies that any value containing the string 'Jo' plus zero or more additional characters will satisfy this selection criterion. So 'John' or 'Johnson' would be matches, but 'Jeoson' would not be. The other wildcard used in LIKE clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.
The code fragment below has a WHERE clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the student who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Students
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of students whose Matric_number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Students
WHERE Matric_Number > 10005
```

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated WHERE clauses, but the following code fragment has a WHERE clause with two conditions; this query selects the first and last names of students whose Matric_number is less than 10100 and who do not have a car.

```
SELECT First_Name, Last_Name
```

PROPERTIES OF DLC UI, IBADAN

```
FROM Students
WHERE Matric_Number < 10100 and Car_Number IS NULL
```

A special type of WHERE clause involves a join, which is explained next.

- **Joins**

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, Cars, shown in Table 12.2.

Table 12.2. Cars

Car_Number	**Make**	**Model**	**Year**
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is Car_Number, which is the primary key for the table Cars and the foreign key in the table Students. If the 1996 Honda Civic were wrecked and deleted from the Cars table, then Car_Number 5 would also have to be removed from the Students table in order to maintain what is called referential integrity. Otherwise, the foreign key column (Car_Number) in Students would contain an entry that did not refer to anything in Cars. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the Car_Number column in the table Students because it is possible for a student not to have a car.

The following code asks for the first and last names of Students who have cars and for the make, model, and year of those cars. Note that the FROM clause lists both Students and Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Students.First_Name, Students.Last_Name, Cars.Make, Cars.Model, Cars.Year
FROM Students, Cars
WHERE Students.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

```
FIRST_NAME  LAST_NAME   MAKE     MODEL    YEAR
----------  ----------  -------- -------- -------
Akinola     Solomon     Honda    CivicDX  1996
Hammed      F.          Toyota   Corolla  1999
```

- **Common SQL Commands**

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.
A list of the more common DML commands follows:

PROPERTIES OF DLC, UI, IBADAN

- **SELECT** — used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.
- **INSERT** — adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- **DELETE** — removes a specified row or set of rows from a table
- **UPDATE** — changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- **CREATE TABLE** — creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. CREATE TABLE is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- **DROP TABLE** — deletes all rows and removes the table definition from the database. A JDBC API implementation is required to support the DROP TABLE command as specified by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options of DROP TABLE is optional. In addition, the behavior of DROP TABLE is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.
- **ALTER TABLE** — adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

- **Result Sets and Cursors**

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

- **Transactions**

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes

PROPERTIES OF DLC UI, IBADAN

resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

- **Stored Procedures**

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static Java method that contains normal JDBC code. It accepts two input parameters and uses them to change a student's car number. Do not worry if you do not understand the example at this point. The code example below is presented only to illustrate what a stored procedure looks like. You will learn how to write the code in this example in the tutorials that follow.

```
import java.sql.*;
public class UpdateCar {
   public static void UpdateCarNum(int carNo, int empNo)  throws SQLException {
       Connection con = null;
       PreparedStatement pstmt = null;
       try {
         con = DriverManager.getConnection( "jdbc:default:connection");

           pstmt = con.prepareStatement( "UPDATE STUDENTS  SET
               CAR_NUMBER = ? " +  "WHERE  MATRIC_NUMBER  = ?");
           pstmt.setInt(1, carNo);
           pstmt.setInt(2, Matric_number);
           pstmt.executeUpdate( );
       }
       finally {
           if (pstmt != null) pstmt.close( );
       }
   }
}
```

- **Metadata**

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface DatabaseMetaData, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of

PROPERTIES OF DLC UI IBADAN

methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.

Example Code 1: Retrieving Data from a Database Table

Step 1: Create a Microsoft Access 2007 (or later) database having the following table schemas and add some hypothetical records to the tables:

Student(Matric, Surname, FirstName, Middlename, Age, Gendar)

Course(CosCode, Title, Unit, Status)

Student_Course(Matric, CosCode, Session, Score)

Note: The primary keys are underlined for the tables.

Save the database name as Stud in the BIN subfolder of the Java toolkit.

Step 2: Set up the JDBC-ODBC bridge driver via the following steps:

(i) From the Start Menu, Click on Control Panel
(ii) Double-Click on Administrative tools
(iii) Double-Click on Data Sources (ODBC)
(iv) Click on Add
(v) Select Microsoft Access Driver (*.mdb, *.accdb) on the next pop-up window, then click on Finish
(vi) Type Studs as the datasource name
(vii) Select the database via the Select button and then click on OK.

Step 3: Writing the Java Code

The first task in a JDBC program is to load the driver (or drivers) that will be used to connect to a data source. A driver is loaded with the *class.forName(String)* method. Class, part of the java.lang package, can be used to load classes into the Java interpreter. The *forName(String)* method loads the class named by the specified string. A *ClassNotFoundException* can be thrown by this method.

All programs that use an ODBC data sources use *sun.jdbc.odbc.jdbcodbcDriver*, the JDBC-ODBC bridge driver included with java. Loading this class into a Java Interpreter requires the following statement:

```
Class.forName("sun.jdbc.odbc.jdbcOdbcDriver");
```

After this driver has been loaded, you can establish a connection to the data source by using the *DriverManager* class in the *java.sql* package.

The *getConnection(String, String, String)* method of *DriverManager* can be used to set up the connection. It returns a reference to a connection object representing an active data connection. The three arguments of this method are as follows:

- A name identifying the data source and the type of database connectivity used to reach it
- A username
- A password

PROPERTIES OF DLC UI, IBADAN

The last two items are needed only if the data source is secured with a username and a password. If not, these arguments can be null strings (“”). The name of the data source is preceded by the text *jdbc:odbc:* when using the JDBC-ODBC bridge, which indicates the type of database connectivity in use.

The following statement could be used to connect to a data source called *Studs* with a username of “Lekan” and a password of “ola”:

```
Connection con = DriverManager.getConnection( “jdbc:odbc:Studs”, “Lekan”, “ola”);
```

After a connection is made, one can reuse it each time he wants to retrieve or store information from that connection’s data source.

The *getConnection*( ) Method and all others called on a data source throw *SQLException* errors if something goes wrong as the data source is being used. SQL has its own error messages, and they are passed along as part of *SQLException* objects.

An SQL statement is represented in Java by a *Statement* object. *Statement* is an interface, so it can’t be installed directly. However, an object that implements the interface is returned by the *CreateStatement*( ) method of a connection object, as in the following example:

```
Statement st = con.createStatement();
```

This *Statement* object, st, would now be used to conduct an SQL query by calling the object’s *executeQuery(String)* method as in the following example:

```
ResultSet rs = st.executeQuery("SELECT * " + "FROM Student  " + "WHERE Matno < 400 "  + “ORDER BY
        Gendar”);
        System.out.println("Matno \tSurname \tFirst Name \tMiddle  Name \tAge \tGendar\n");
```

Note that the *String* argument should be an SQL query that follows the syntax of that language.

The above query retrieves all the fields in the Table Student from the Stud database and the records are sorted according to the Gendar field because of the ORDER BY clause in the query. So all the females would come before the males in the query result. You need to acquaint yourself with SQL queries before you can effectively use them in your program.

If the SQL query has been phrased correctly, the *executeQuery( )* method returns a *ResultSet* object holding all the records that have been retrieved from the data source.

When a *ResultSet* is returned from *executeQuery( ),* it is positioned at the first record that has been retrieved. The following methods of *ResultSet* can be used to pull information from the current record:

- **getDate(String)** – Returns the Date value stored in the specified field name (Using the Date class in the java.sql package, not java.util.Date).
- **getDouble(String)** – Returns the double value stored in the specified field name.
- **getFloat(String)** – Returns the float value stored in the specified field name.
- **getInt(String)** – Returns the int value stored in the specified field name.
- **getLong(String)** – Returns the long value stored in the specified field name.
- **getString(String)** - Returns the string value stored in the specified field name.

These are just the simplest methods available in the *Resultset* interface. The methods used depend on the form that the field data takes in the database, although methods such as *getString*() and *getInt*() can be more flexible in the information they retrieve from a record.

PROPERTIES OF DLC UI, IBADAN

We can also use an integer as the argument to any of these methods, such as *getString*(5), instead of a string. The integer indicates which field to retrieve (1 for the first field, 2 for the second field and so on).

After we have pulled the information needed from a record, we can move to the next record by calling the *next( )* method of the *ResultSet* object. This method returns a false Boolean value when it tries to move past the end of a *resultset.* Normally, we move through a resultset once from start to finish, after which its contents cannot be retrieved again.

When we are finished using a connection to a data source, we close it by calling the connection's *close( )* method with no arguments.
An example program to illustrate all the above features is shown below.

The database table 'Student' used for the following program is shown below:

Student					
Matno	**Surname**	**FirstName**	**MiddleName**	**Age**	**Gender**
100	Chukwu	Ann	Mary	23	F
104	Michael	Janet	Tommy	21	F
123	Akinola	Olalekan	Solomon	20	M
345	John	Josephine	Eunice	23	F
456	Samson	Jonathan	Azikwe	50	M
789	Usman	Momodu	Karim	30	M

Example code: Retrieving Data from a Table in a Database

```
1.  import java.sql.*;
2.  public class database {
3.     public static void main(String[ ] args)  {
4.         try {
5.               Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
6.               Connection con = DriverManager.getConnection
7.                      ("jdbc:odbc:Studs","","");
8.               Statement st = con.createStatement();
9.                        ResultSet rs = st.executeQuery("SELECT * " + "FROM Student
10.                   " + "WHERE Matno < 400 "  + “ORDER BY Gendar”);
11.               System.out.println("Matno \tSurname \tFirst Name \tMiddle
12.                      Name \tAge \tGendar\n");
13.               while (rs.next( )) {
14.                     System.out.println(rs.getInt(1) + "\t" + rs.getString(2) +
15.                       "\t\t" + rs.getString(3)  + "\t" + rs.getString(4) + "\t\t" +
16.                         rs.getInt(5) + "\t" + rs.getString(6));
17.                } // end while
18.                st.close( );
19.           } catch (SQLException s) {
20.   System.out.println("SQL error: " + s.toString() + s.getErrorCode( )  +
21.          " "   + s.getSQLState( ));
22.   } catch (Exception e) {
23.   System.out.println("Error: " + e.toString() + e.getMessage( ));
24.   }
25.  }
26. }
```

PROPERTIES OF DLC UI, IBADAN

```
C:\Program Files\Xinox Software\JCreatorV3LE\GE2001.exe
Matno   Surname         First Name      Middle Name     Age     Gendar

100     Chukwu          Ann     Mary            23      F
104     Michael         Janet   Tommy           21      F
345     John            Josephine       Eunice          23      F
456     Samson          Jonathan        Azikwe          50      M
123     Akinola         Olalekan        Solomon         20      M
Press any key to continue...
```

Practical Exercise

Copy the above code into an editor, modify the code such that the output is directed to a file, compile and finally execute with a Microsoft Access Database set up initially.

12.4.5 ResultSet

An instance of ResultSet is returned from executeQuery( ), and one or more instances may be returned from execute( ). A ResultSet is a representation of the data returned by your query, and it allows you to process the results one row at a time. Before you can process a row, you must move the ResultSet's cursor (pointer) to that row, and the row that's pointed to by the cursor is called the current row. When a ResultSet is created, the cursor is initially positioned before the first row.

It's helpful to review some ResultSet properties before describing the methods defined in that interface, because its properties determine which of a ResultSet's methods you're able to use for a particular instance and how they function.

- **Forward-Only vs. Scrollable (Scrollability Type)**

Scrollability describes the type of cursor movement that's allowed, and a forward-only ResultSet allows the cursor to be moved forward only one row at a time using the next( ) method. However, with a scrollable ResultSet, you can use a variety of methods to position the cursor. It can be moved forward or backward, and it can be moved in those directions by any number of rows. In addition, it's possible to move the cursor to a specific row (in other words, to use absolute instead of relative positioning), including the first and last rows in the ResultSet.

- **Read-Only vs. Updatable (Concurrency Mode)**

ResultSet defines a large number of getXXX() methods that allow you to read column values from the current row (for example, getString(), getFloat(), and so on), and it includes a corresponding updateXXX() method for each getXXX(). While it's always possible to call the read/get methods, a ResultSet's concurrency mode determines whether you can use the write/update methods. As its name implies, a read-only ResultSet allows you only to read the data, while an updatable ResultSet allows you both to read the data and to modify it through the ResultSet.

- **Update Sensitivity**

While you're using a ResultSet to process the results of a query, it's usually possible for other users/applications to modify the rows in the database that were returned by your query. Update sensitivity indicates whether the ResultSet will reflect changes that are made to the underlying data after the ResultSet is created. Those updates are known as "changes by others" to distinguish them from changes made to the data using an updatable ResultSet's updateXXX() methods.

PROPERTIES OF DLC UI, IBADAN

If you call a getXXX() method to read data from the current row, a sensitive ResultSet will return the data stored in the underlying database even if the data was changed by another user after the ResultSet was created. However, an insensitive ResultSet doesn't detect such changes and may return outdated information.

Update sensitivity doesn't imply that a ResultSet is sensitive to all types of changes. For example, a ResultSet might be sensitive to row deletions but not to row updates or insertions.
In addition, a ResultSet's sensitivity to "changes by others" can be different from its sensitivity to its own changes (modifications to the data made through the updateXXX() methods). However, DatabaseMetaData provides methods that allow you to determine which types of changes are visible for a given ResultSet type.

- **Selecting ResultSet Properties**

To set the scrollability, concurrency, and sensitivity properties, you must specify the appropriate values when creating a Statement. The code segments shown earlier used the createStatement( ) method that doesn't accept any parameter values, but another version of createStatement( ) allows you to specify two integer values representing ResultSet properties:

```
int resultSetType, resultSetConcurrency;
// ...
Statement stmt = con.createStatement(resultSetType,
resultSetConcurrency);
```

The resultSetType parameter represents a combination of the scrollability and sensitivity properties, and it should be assigned one of the following constants defined in ResultSet:

TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE.

The resultSetConcurrency value represents the concurrency mode for ResultSet instances created by this statement and should be assigned the value of either CONCUR_READ_ONLY or CONCUR_UPDATABLE.

You can use these constants and the createStatement( ) method shown previously to create a Statement that will produce ResultSet instances with the desired properties. For example, you can use code similar to the following to create a Statement and request that the ResultSet instances it creates be scrollable, sensitive to others' changes, and updatable:

```
Statement stmt = connect.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

Note that if you specify a type of ResultSet that's not supported by the driver, it won't generate an error when createStatement() is called. Instead, the Statement will produce ResultSet instances that match the type you requested as closely as possible. In this case, for example, if the driver supports updatable ResultSet instances but not scrolling, it will create forward-only instances that are updatable.

12.4.6 PreparedStatement

When you call one of Statement's execute( ) methods, the SQL statement specified is "compiled" by the JDBC driver before being sent to the DBMS. In many cases, you'll want to execute multiple statements that are similar and may differ only by a single parameter value. For example, you might execute SQL statements like these:

PROPERTIES OF DLC UI, IBADAN

```
Statement stmt = connect.createStatement();
stmt.executeUpdate(
"UPDATE MYTABLE SET FNAME = 'Jacob' WHERE CUSTID = 123");

stmt.executeUpdate(
"UPDATE MYTABLE SET FNAME = 'Jordan' WHERE CUSTID = 456");

stmt.executeUpdate(
"UPDATE MYTABLE SET FNAME = 'Jeffery' WHERE CUSTID = 789");
```

Compiling each SQL statement can result in poor performance if a large number of statements are executed. However, this example illustrates the usefulness of PreparedStatement, which is a subclass of Statement. PreparedStatement allows you to compile a statement one time and use substitution parameters to modify the final SQL statement that's executed. In this case, for example, you might create a PreparedStatement using code like this:

```
PreparedStatement pstmt = connect.prepareStatement(
"UPDATE MYTABLE SET FNAME = ? WHERE CUSTID = ?");
```

The two question marks (?) in the statement represent substitution parameters, and you can use the setXXX() methods defined in PreparedStatement to specify values for those fields.

For example, the following code is functionally equivalent to the group of statements used earlier:

```
PreparedStatement pstmt = connect.prepareStatement(
"UPDATE MYTABLE SET FNAME = ? WHERE CUSTID = ?");
pstmt.setString(1, "Jacob");
pstmt.setInt(2, 123);
pstmt.executeUpdate( );
pstmt.setString(1, "Jordan");
pstmt.setInt(2, 456);
pstmt.executeUpdate( );
pstmt.setString(1, "Jeffery");
pstmt.setInt(2, 789);
pstmt.executeUpdate( );
```

This approach is much more efficient because the statement is compiled only once, but it's executed several times. Note that the substitution field index values are one-based instead of zero-based, meaning that the first question mark corresponds to field 1, the second to field 2, and so on.

Another advantage of using a PreparedStatement instead of a Statement is that it partially insulates your application from the details of creating a valid SQL statement.

The next example program add more data into the Student table:

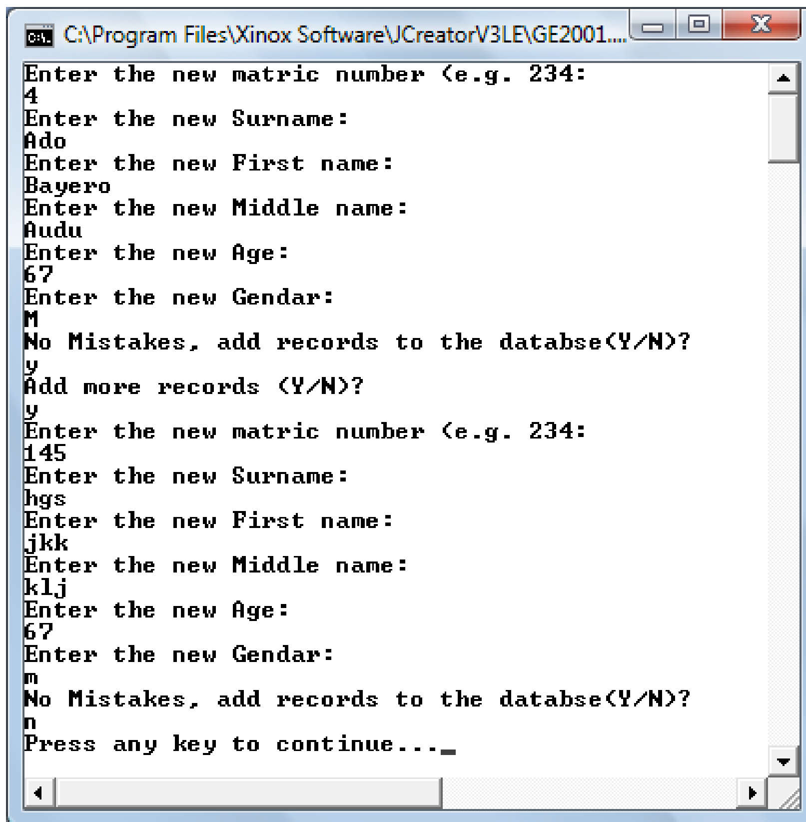
Example Program: Inserting Data into a Database Table

```
1. import java.sql.*;
2. import java.util.*;
3. class database2 {
4.   public static void main(String[ ] args)  {
5.     Scanner input = new Scanner(System.in);
6.      try {
7.          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
8.          Connection con = DriverManager.get
 9.                             Connection ("jdbc:odbc:Studs","","");
10.
11.         PreparedStatement st = con.prepareStatement("INSERT INTO " +
```

PROPERTIES OF DLC UI, IBADAN

```
12.          "Student(Matno, Surname, FirstName, MiddleName, Age,
13.                 Gendar )" +  "VALUES(?, ?, ?, ?, ?, ?)");
14.            L1:      //Label for a case when data is not correct
15.            while (true) {    // to continuously add records
16.
17.             System.out.println("Enter the new matric number (e.g. 234: " );
18.             String a = input.next( );
19.             st.setString(1, a);
20.
21.           System.out.println("Enter the new Surname: " );
22.           String b = input.next( );
23.           st.setString(2, b);
24.
25.          System.out.println("Enter the new First name: " );
26.          String c = input.next( );
27.          st.setString(3, c);
28.
29.          System.out.println("Enter the new Middle name: " );
30.          String d = input.next( );
31.          st.setString(4, d);
32.
33.          System.out.println("Enter the new Age: " );
34.          String e = input.next( );
35.          st.setString(5, e);
36.
37.          System.out.println("Enter the new Gendar: " );
38.          String f = input.next();
39.          st.setString(6, f);
40
41.                    // Check if there is no error in the data captured
42.                    System.out.println("No Mistakes, add records to the
43.                        databse (Y/N)?");
44.                    String reply1 = input.next( );
45.                    if  (reply1.equalsIgnoreCase("n"))
46.                        break L1; //Move to label L1 if  no problem
47.
48.                    //update the database table
49.
50.                      st.executeUpdate( );
51.
52.                      System.out.println("Add more records (Y/N)?");
53.                      String reply = input.next( );
54.                      if (reply.equalsIgnoreCase("n"))
55.                                  break;  // go out of loop to close database
56.                       }  // loop for more record update
57.                      con.close( ); // close database connection
58.                    } catch (SQLException s) {
59.                         System.out.println("SQL error: " + s.toString( ) +
60.                                  s.getErrorCode( )  + " " + s.getSQLState( ));
61.                    } catch (ClassNotFoundException cnfe) {
62.                           System.out.println("Error: " + cnfe.getMessage( ));
63.                    }
64.        } // end method main
65.    } // end class database2
```

PROPERTIES OF DLC UI, IBADAN



The structure of the Table Student after several data insertions is shown below.

Student					
Matno	**Surname**	**FirstName**	**MiddleName**	**Age**	**Gendar**
1	Bamgbade	Joseph	Adeolu	41	M
2	Mohammed	Abdulkareem	Sheu	27	M
4	Ado	Bayero	Audu	67	M
56	Ajala	Olaoluwa	Omooluwa	45	m
100	Chukwu	Ann	Mary	23	F
104	Michael	Janet	Tommy	21	F
106	Balogun	Adeolu	Ibukun	21	M
111	Augustine	Nnena	Abigael	34	F
123	Akinola	Olalekan	Solomon	20	M
124	Michael	Sandra	mary	24	F
333	Amodu	Usman	Muhammed	25	M
345	John	Josephine	Eunice	23	F
456	Samson	Jonathan	Azikwe	50	M
675	Kazeem	Bright	James	54	M
789	Usman	Momodu	Karim	30	M

An attempt to insert a record with the same matric number to the table would result into an error. This is because duplication of records is not allowed with the use of primary key in a table. The sample output obtained for this observation is shown below:

```
C:\Program Files\Xinox Software\JCreatorV3LE\GE2001.exe
Enter the new matric number (e.g. 234:
123
Enter the new Surname:
asgt
Enter the new First name:
jhd
Enter the new Middle name:
nmb'
Enter the new Age:
78
Enter the new Gendar:
m
No Mistakes, add records to the databse(Y/N)?
y
SQL error: java.sql.SQLException: General error0 S1000
Press any key to continue...
```

It is easier to send Microsoft Access strings and let the database program converts them automatically into the correct format as in Matno and Age which are integer data in the database, but string are used for them in the program above (Lines 17 – 19 and 33 – 35).

```
17.     System.out.println("Enter the new matric number (e.g. 234: " );
18.     String a = input.next( );
19.     st.setString(1, a);

33.     System.out.println("Enter the new Age: " );
34.     String e = input.next( );
35.     st.setString(5, e);
```

However, level of SQL support varies based on the product and ODBC driver involved.

After the prepared statement has been prepared and all the placeholders are filled, the statement's *executeUpdate*( ) method is called (Line 50). This either adds the quote data to the database or throws an SQL error.

```
50.     st.executeUpdate( );
```

The while (true) loop assists the user of the program to add more records into the database at a go. Note lines 41 – 46, the lines check the data to be inserted into the database for correctness.

```
41.     // Check if there is no error in the data captured
42.     System.out.println("No Mistakes, add records to the
43.         databse (Y/N)?");
44.     String reply1 = input.next( );
45.     if  (reply1.equalsIgnoreCase("n"))
46.         break L1; //Move to label L1 if  no problem
```

If the user has made a mistake in entering one of the data, the entire record will not be inserted into the database. This however could be painful especially if a large data entry is involved. How can you validate the data as they are being entered so that not all the data will be discarded at once?
After a successful insertion of a record or not, the program still asks the user if he wants to insert more records. This is achived in Lines 52 – 56. if the reply is yes, the user is asked to enter his new record fields.

```
52.             System.out.println("Add more records (Y/N)?");
53.             String reply = input.next( );
54.             if (reply.equalsIgnoreCase("n"))
55.                 break;  // go out of loop to close database
```

PROPERTIES OF DLC UI, IBADAN

```
56.                 } // loop for more record update
```

12.4.7 Moving Through Resultsets

The default behaviour of resultsets permits one trip through the set using its *next*( ) method to retrieve each record. By changing how statements and prepared statements are created, one can produce resultsets that support these additional methods:

- **afterLast( ) -** Moves to a place immediately after the last record in the set
- **beforeFirst( ) -** Moves to a place immediately before the last record in the set
- **first( ) -** Moves to the first record in the set
- **last( ) –** Moves to the last record in the set
- **previous( ) –** Moves to the previous record in the set.

These actions are possible when the resultset's policies have been specified as arguments to a database connection's *createStatement*( ) and *prepareStatement*( ) methods.

For a more flexible resultset, call *createStatement* ( ) with three integer arguments that set up how it can be used. Here is an example:

```
Statement st = con.CreateStatement (
   ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY,
   ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

The same three arguments can be used in the *prepareStatement(String, int, int, int)* method after the text of the statement.

12.4.8 Dealing With Multiple Tables (Joins)

Sometimes you need to use two or more tables to get the data you want. A join is a database operation that relates two or more tables by means of values that they share in common. In our example database, the tables STUDENT and STUDENT_COUSE both have the column Matno, which can be used to join them.

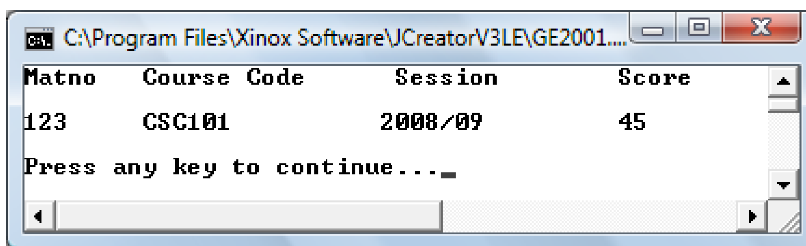
You need some way to distinguish to which Matno column you are referring. This is done by preceding the column name with the table name, as in "Student.Matno" to indicate that you mean the column Matno in the table Student. The example program below illustrates the concept of Join. Learn more of SQL in a database textbook in order to get the logic of joins in databases.

Example Program: Retrieving Data from Two Tables

```
1.  import java.sql.*;
2.  public class database3 {
3.      public static void main(String[ ] args)  {
4.              try {
5.                      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
6.                        Connection con = DriverManager.getConnection
7.                        ("jdbc:odbc:Studs","","");
8.                        Statement st = con.createStatement( );
9.                        ResultSet rs = st.executeQuery("SELECT
10.                        Student_Course.Matno,
11.                        Student_Course.CosCode, Session, Score " +
12.                          "FROM  Student, Course,
13.                           Student_Course " +  "WHERE Student.Matno
14.                            = Student_Course.matno
15.                             and Course.CosCode =
```


PROPERTIES OF DLC UI, IBADAN

```
16.                              Student_Course.CosCode");
17.                         System.out.println("Matno \tCourse Code \t
18.                              Session \tScore\n");
19.                         while (rs.next( )) {
20.                            System.out.println(rs.getInt(1) + "\t" +
21.                                  rs.getString(2) +   "\t\t" + rs.getString(3)  +
22.                                  "\t\t" + rs.getString(4) );
23.                         } // end while
24.                         st.close( );
25.                     } catch (SQLException s) {
26.                     System.out.println("SQL error: " + s.toString( ) +
27.                           s.getErrorCode( )  + " " + s.getSQLState( ));
28.                     } catch (Exception e) {
29.                 System.out.println("Error: " + e.toString( ) +
30.                                                        e.getMessage( ));
31.                 }
32.           }
33.     }
```



12.4.9 Database Search

Sometimes we may be interested in searching a record or group of records from a table in a database. Prepared statement is useful in this case for handling the query. The example below is a program that searches for a particular student in the Student table in our database. The program firstly asks for the Matric number of the student being searched for. If the Matric number is found, the program prints out his/her Matric number, Surname, Age and Gender. But if the number is not found in the database table, an empty record is printed.

Example Program: Searching for a record

```
1.  import java.sql.*;
2. import java.util.*;
3. public class database4 {
4.     public static void main(String[ ] args)  {
5.         // searching for a record
6.         Scanner input = new Scanner(System.in);
7.         boolean found = true;
8.         try {
9.                 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
10.                Connection con = DriverManager.getConnection
11.                      ("jdbc:odbc:Studs","","");
12.                L1: while (true){
13.               System.out.println("Enter the Student's matric number: ");
14.                String mat = input.next( ); // get matric  no of the student
15.
16.                // prepare the query statement
17                 PreparedStatement st = con.prepareStatement("SELECT
18.                      * " +  "FROM  Student " + "WHERE Matno = ? ");
19.
20.                // Set the place holder ? with a value
21.                 st.setString(1, mat);
```

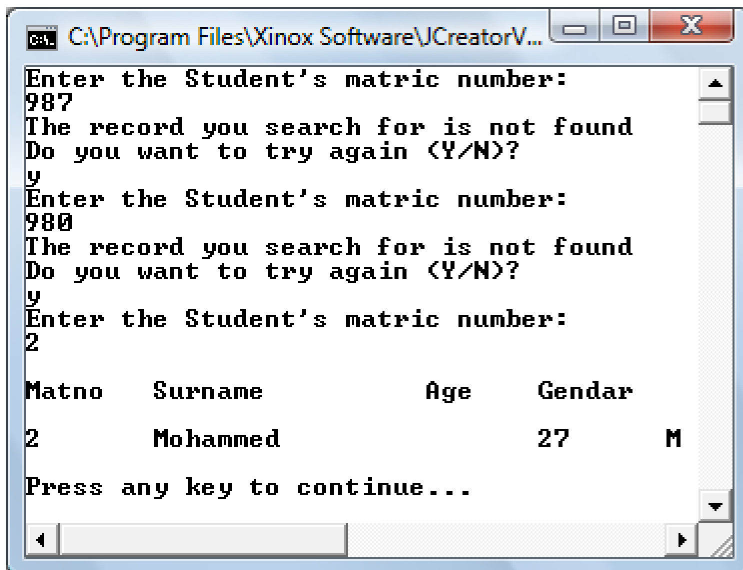
PROPERTIES OF DLC UI, IBADAN

```
22.
23.                     // Execute the query
24.                     ResultSet rs = st.executeQuery();
25.
26.                     boolean g = rs.next( ); // Set next record pointer to g
27.                     while (true) {
28.                             if (g != false) {
29.
30.                                 System.out.println("\nMatno \tSurname \t Age
31.                                   \tGendar\n");  // Heading Titles
32.
33.                               // Printing  record found
34.                               System.out.println(rs.getString("Matno") + "\t"
35.                                 +   rs.getString("Surname") + "\t\t" + rs.get
36.                                 String("Age") + "\t" + rs.getString("Gendar") );
37.
38.                               break L1;
39.
40.                            } else {
41.                               System.out.println("The record you search for is
42.                       not found \nDo you want to try again (Y/N)?");
43.                       String reply = input.next();
44.                       if (reply.equalsIgnoreCase("y"))
45.                               continue L1;
46.                      else
47.                         System.exit(0);
48.                         System.out.println("Good Bye");
49.                 }// end outer else
50.                 st.close( );
51.               }// end inner while
52.
53.           } // end L1 while
54.
55.           } catch (SQLException s) {
56.                 System.out.println("SQL error: " + s.toString( ) +
57.                     s.getErrorCode()  + " " + s.getSQLState( ));
58.           } catch (Exception e) {
59.                 System.out.println("Error: " + e.toString() + e.getMessage( ));
60.           }
61.           System.out.println( );
62.         } // End method main
63.     }  // End class database4
```

Sample Outputs:

```
C:\Program Files\Xinox Software\JCreatorV...
Enter the Student's matric number:
908
The record you search for is not found
Do you want to try again (Y/N)?
y
Enter the Student's matric number:
98
The record you search for is not found
Do you want to try again (Y/N)?
n
Good Bye
Press any key to continue..._
```

PROPERTIES OF DLC UI, IBADAN



The query for searching for a record in the database is implemented with the use of *PreparedStatement* (Lines 17 – 18). In Line 21, the place holder (?) in the query is assigned with the Matno (mat) of the student earlier captured in Lines 13 – 14. The query is finally executed in Line 24. At the point, the record set pointer rs would be assigned to the first record in the table.

Note that rs.next( ) specified in Line 26 only tests if there is any 'next record' in the table and so it returns a true or false boolean value.

```
27.     L2: while (true) {
```

Line 27 while loop is labeled as L2 for the program to be able to repeat the code again in case the record being searched for is not found and the user of the program still wants to search for another matric number. This is achieved with lines Lines 44 – 45 with the use of continue with label statement.

```
44.     if (reply.equalsIgnoreCase("y"))
45.         continue L1;
```

However, Line 27 is also used to terminate the entire program if what we looked for has been found and reported. This is achieved with the use of break with label in Line 38.

```
38.     break L1;
```

With this introduction to database handling in Java, powerful database-driven programs could be developed by you with no stress.

PROPERTIES OF DLC UI IBADAN

Summary
In this Chapter, you have been introduced to Java Database Connectivity (JDBC). You were taken through a database concept as seen with Microsoft Access. Codes implementations with Java were demonstrated with JDBC. JDBC can also be used on any other database engine like Oracle and MySQL. Find out!

Post Test

1. Think of an information system for a school library, implement a Java program that can be used for monitoring borrowing and returning of books in the library.

Perform a system analysis study of how your results are computed in your school. Write a java-based information system for computing final year results in your department.

PROPERTIES OF DLC UI, IBADAN