SRINIVAS INSTITUTE OF MANAGEMENT STUDIES

PANDESHWAR, MANGALORE-575 001

BACKGROUND STUDY MATERIAL

Java Programming

B.C.A - V SEMESTER

SRINIVAS GROUP
SAMAGRA GNANA
ESTD: 1988

Compiled by

Mr. Krishna Prasad K

Faculty, SIMS

2015-2016

UNIT-1
Chapter-1
Java Fundamentals

1.1 The Origin of java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems in 1991. This language was initially called "Oak" but was renamed "Java" in 1995.

The original movement for Java was not the internet; instead, the primary motivation was the need for a platform-independent language that could be used to create software to be embedded in various consumer electronic devices, such as toasters, microwave ovens, and remote controls. The trouble was that (at that time) most computer languages were designed to be compiled for a specific target. The problem, however, is that compilers are expensive and time-consuming to create. In an attempt to find a better solution, Gosling and others worked on a portable, cross-platform language that could produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

The World Wide Web would play a crucial role in the future of Java. Java might have remained a useful but unclear language for programming consumer electronics. However, with the emergence of the Web, Java was drive to the forefront of computer language design, because the Web, too, demanded portable programs.

With the advent of the Internet and the Web, the old problem of portability returned with a revenge. After all, the Internet consists of a diverse, distributed universe populated with many types of computers, operating systems, and CPUs.

By 1993 it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while it was the desire for an architecture-neutral programming language that provided the initial spark, it was the Internet that ultimately led to Java's large-scale success.

How Java Relates to C and C++: Java is directly related to both C and C++. Java inherits its syntax from C. Its object model is adapted from C++. Java's relationship with C and C++ is important for several reasons. First, many programmers are familiar with the C/C++ syntax. This makes it easy for a C/C++ programmer to learn Java and, conversely, for a Java programmer to learn C/C++. Second, Java's designers did not start from the scratch. Instead, they further refined an already highly successful programming paradigm. The modern age of programming began with C. It moved to C++, and now to Java. By inheriting and building upon that rich heritage, Java provides a powerful, logically consistent programming environment that takes the best of the past and adds new features required by the online environment. Perhaps most important, because of their similarities, C, C++, and Java define a common, conceptual framework for the professional programmer. Programmers do not face major rifts when switching from one language to another.

One of the central design philosophies of both C and C++ is that the programmer is in charge, Java also inherits this philosophy. Java gives you, the programmer, and full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers. Java has one other attribute in common with C and C++: it was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and

experiences of the people who invented it. There is no better way to produce a top-flight professional programming language.

Because of the similarities between Java and C++, especially their support for object-oriented programming, it is tempting to think of Java as simply the "Internet version of C++." Java has significant practical and philosophical differences. Although Java was influenced by C++, it is not an enhanced version of C++. For example, it is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, you will feel right at home with Java. Another point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. They will coexist for many years to come.

How Java Relates to C# : A few years after the creation of Java, Microsoft developed the C# language. This is important because C# is closely related to Java. In fact, many of C#'s features directly parallel Java. Both Java and C# share the same general C++-style syntax, support distributed programming, and utilize the same object model. There are, of course, differences between Java and C#, but the overall "look and feel" of these languages is very similar. This means that if you already know C#, then learning Java will be especially easy. Conversely, if C# is in your future, then your knowledge of Java will come in handy. Java and C# are optimized for two different types of computing environments.

1.2 Java's contribution to the internet

The Internet helped throw Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the important issues associated with the Internet: portability and security.

Java Applets: An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server. As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems.

Security: As you are likely aware, every time that you download a "normal" program, you are taking a risk because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be safely downloaded and executed on the client computer, it was necessary to prevent an applet from launching such an attack. Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

Portability: Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of different CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The same code must work in all computers. Therefore, some means of generating portable executable code was needed.

1.3 The Bytecode

The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). In essence, the original JVM was designed as an interpreter for bytecode. This may come as a bit of a surprise because many modern languages are designed to be compiled into executable code due to performance concerns. However, the fact that a Java program is executed by the JVM helps to solve the major problems associated with web-based programs.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system. Safety is also enhanced by certain restrictions that exist in the Java language. When a program is interpreted, it generally runs slower than the same program would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java's initial release. HotSpot provides a just-in-time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time on a piece-by-piece, demand basis.

It is important to understand that it is not practical to compile an entire Java program into executable code all at once because Java performs various run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply because the JVM is still in charge of the execution environment.

1.4 The Java Buzzwords

Although the fundamental forces that necessitated the invention of Java are portability and security, other factors played an important role in molding the final form of the language. The key considerations were summed up by the Java design team in the following list of buzzwords.

Simple: Java is a small and simple language. Java does not use pointers, preprocessor header files, goto statement and many others. It also eliminates operator overloading and multiple inheritance. Java is modeled on C and C++ languages. So it is familiar to programmers.

Secure: Java provides a secure means of creating internet applications. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

Portable: Java ensures portability in two ways. First, Java compiler generates bytecode instructions that can be implemented on any machine. Secondly, the sizes of the primitive data types are machine-independent.

Robust: It has strict compile time and run time checking for data types. It is designed as a garbage-collected language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures series errors and eliminates any risk of crashing the system.

Multithreaded: Multithreaded means handling multiple task-simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another.. This feature greatly improves the interactive performance of graphical applications.

Architecture Neutral: Java is not tied to a specific machine or operating system architecture. Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs.

Interpreted: Java supports cross-platform code through the use of Java bytecode. Byte codes are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly executed by the machine that is running the Java program.

High Performance: Java performance is impressive for an interpreted language, mainly due to the use of intermediate bytecode. Java architecture is also designed to reduce overheads during runtime. Further, the incorporation of multithreading enhances the overall execution speed of Java programs.

Distributed: Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

Dynamic: Java Programs carry with them substantial amount of run-time type information that is used to verify and resolve access to objects at run time. Java is a dynamic language. Java is capable of dynamically linking in new class libraries, methods, and objects. Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response

1.5 Object Oriented Programming

At the center of Java is object-oriented programming (OOP). The object-oriented methodology is inseparable from Java, and all Java programs are, to at least some extent, object-oriented. Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (what is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data." Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data. To support the principles of object-oriented programming, all OOP languages, including Java, have three traits in common: Abstraction, encapsulation, polymorphism, and inheritance.

Abstraction: 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 methods to operate on these attributes.

Encapsulation: Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained black box is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation. Within an object, code, data, or both may be private to that object or public. Private code or data is known to and accessible by only another part of the object. That is, private code or data cannot be accessed by a piece of the program that exists outside the object.

When code or data is public, other parts of your program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object. Java's basic unit of encapsulation is the class. A class defines the form of an object. It specifies both the data and the code that will operate on that data. Java uses a class specification to construct objects. Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object. The code and data that constitute a class are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods.

Polymorphism: Polymorphism (from Greek, meaning "many forms") is the quality that allows one interface to access a general class of actions. The specific action is determined by the exact nature of the situation. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (i.e., the interface) is the same no matter what type of actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and- pinion steering. Therefore, once you know how to operate the steering wheel, you can drive any type of car. The same principle can also apply to programming. More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. Polymorphism helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the general interface.

Inheritance: Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of hierarchical classification. If you think about it, most knowledge is made manageable by hierarchical (i.e., top-down) classifications. For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. That is, the food class possesses certain qualities (edible, nutritious, etc.) which also, logically, apply to its subclass, fruit. In addition to these qualities, the fruit class has specific characteristics (juicy, sweet, etc.) that distinguish it from other food. The apple class defines those qualities specific to an apple (grows on trees, not tropical, etc.). A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes, and would define only those qualities that make it unique. Without the use of hierarchies, each object would have to explicitly define all of its characteristics. Using inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

1.6 Structure of a simple program

```
class SampleOne
{
  public static void main (String args[ ])
  {
    System.out.println("Java is better than C++.");
     }
}
```

Class Declaration

The first line "**class SampleOne"** declares a class, which is an object-oriented construct. Java is a true object-oriented language and therefore, *everything* must be placed inside a class. class is a keyword and declares that a new class definition follows. SampleOne is a Java *identifier* that specifies the name of the class to be defined.

Opening Brace

Every class definition in Java begins with an opening brace "{" and ends with a matching closing brace"}" appearing in the last line in the example.

The main Line, The third line, **public static void main (String args[ ])**

Defines a method named main. Every Java application program must include the main( ) method. This is the starting point for the interpreter to begin the execution of the program. A Java application can have any number of classes but *only* one of them must include a main method to initiate the execution. This line contains a number of keywords, public, static and void.

public: The keyword public is an access specifier that declares the main method as unprotected and therefore making it accessible to all other classes.

static: which declares this method as one that belongs to the entire class and not a part of any objects of the class. The main must always be declared as static since the interpreter uses this method before any objects are created.

void: The type modifier void states that the main method does not return any value (but simply prints some text to the screen). All parameters to a method are declared inside a pair of parentheses. Here, String args[ ] declares a parameter named args, which contains an array of objects of the class type String.

A Simple Java program may contain one or more sections.

Documentation Section: The documentation section comprises a set of comment lines giving the name of the program, the author and other details, which the programmer would like to refer to at a later stage. Comments must explain *why* and *what* of classes and how of algorithms. Java permits both the single-line comments and multi-line comments. The single-line comments begin with // and end at the end of the line. For longer comments, we can create long multi-line comments by starting with a /* and ending with */. Java also uses a third style of comment /**……*/ known as *documentation comment.* This form of comment is used for generating documentation automatically.

Class Definitions: A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java program. These classes are used to map the objects of real-world problems. The number of classes used depends on the complexity of the problem.

Main Method Class: Since every Java stand-alone program requires a main method as its starting point, this class is the essential part of a Java program. A simple Java program may contain only this part. The main method creates objects of various classes and establishes communications between them. On reaching the end of main, the program terminates and the control passes back to the operating system.

1.7 The Java Keywords

Keywords are an essential part of a language definition. They implement specific features of the language. Java language has reserved 60 words as keywords. These keywords, combined with operators and separators according to syntax, form definition of the Java language.

abstract	boolean	break
case	cast*	catch
const*	continue	default
else	extends	false**
float	For	future*
if	implements	import
int	interface	Long
null**	operator*	outer*
protected	public	rest*
static	super	Switch
threadsafe*	throw	Throws
try	var*	Void

*Reserved for future use

**These are values defined by java

Table 1.1 Java Keywords

Since keywords have specific meaning in Java, we cannot use them as names for variables, classes, methods and so on. All keywords are to be written in lower-case letters. Since Java is case-sensitive, one can use these words as identifiers by changing one or more letters to upper case. However, it is a bad practice and should be avoided.

1.8 Identifiers in Java

Identifiers are programmer-designed tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow the following rules:

1. They can have alphabets, digits, and the underscore and dollar sign characters.
2. They must not begin with a digit.
3. Uppercase and lowercase letters are distinct.
4. They can be of any length.

Identifier must be meaningful, short enough to be quickly and easily typed and long enough to be descriptive and easily read. Java developers have followed some naming conventions.

1. Names of all public methods and instance variables start with a leading lowercase letter. Examples: average, sum
2. When more than one word are used in a name, the second and subsequent words are marked with a leading uppercase letters. Examples: dayTemperature, firstDayOfMonth, totalMarks
3. All private and local variables use only lowercase letters combined with underscores. Examples: length, batch_strength
4. All classes and interfaces start with a leading uppercase letter (and each subsequent word with a leading uppercase letter). Examples: Student, HelloJava, Vehicle, MotorCycle
5. Variables that represent constant values use all uppercase letters and underscores between words. Examples: TOTAL F _MAX, PRINCIPAL_AMOUNT

It should be remembered that all these are conventions and not rules. We may follow our own conventions as long as we do not break the basic rules of naming identifiers.

1.9 The Java Class Libraries

The println( ) and print( ) methods are members of the System class, which is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for windowed output. Thus, Java as a totality is a combination of the Java language itself, plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, part of becoming a Java programmer is learning to use the standard Java classes.

1.10 Assignment-1

Short Answer Questions (2 marks each)

1. List any four features of java.
2. How java more secure than other languages?
3. What was the original name of Java?
4. How is java byte code different from other low level computer languages? (M.U.Oct/Nov.2009)
5. What is byte code? Write the benefits of byte code. (M.U.Oct/Nov.2014)
6. What are keywords?
7. What is identifier?
8. What is Applet?
9. Define Data Abstraction.
10. What is data Encapsulation?
11. What is polymorphism?
12. Write the meaning of public static void main(String args[])
13. Mention two ways of writing comments in Java
14. How to create single-line and multi-line comments in Java?
15. What you men by static?
16. How platform independence is achieved in Java?

Long Answer Questions

1. Write a note on java history. (5 marks- M.U.Oct./Nov. 2009)
2. List and explain different buzzwords of Java.(7 marks- M.U.Oct./Nov. 2014)
3. Differentiate between C/C++ and Java. (4 marks)
4. Explain the java's contribution to the internet.(5 marks- M.U.Oct./Nov. 2014)
5. Explain the features of OOPs? (7 marks)
6. Write a short note on Java keywords (4 marks).
7. What are identifiers? Write the rules to be followed when creating an identifier. (4 marks)
8. Write a short note on java class libraries (4 marks)
9. Explain structure of simple java program (5 marks)
10. Why main( ) method in java is defined as public, static, void ? Explain. (5 Marks-M.U. Oct/Nov.2012)

UNIT-1
Chapter-2
Data Types and Operators

2.1 Java's Primitive Types

Java contains two general categories of built-in data types: object-oriented and non-object-oriented. Java's object-oriented types are defined by classes. However, at the core of Java are eight primitive (also called elemental or simple) types of data, which are shown in Table 2.1. The term primitive is used here to indicate that these types are not objects in an object-oriented sense, but rather, normal binary values. These primitive types are not objects because of efficiency concerns. All of Java's other data types are constructed from these primitive types.

Type	**Meaning**
boolean	Represents true/false values
Byte	8-bit integer
Char	Character
double	Double precision floating point
Float	Single precision floating point
Int	Integer
Long	Long Integer
Short	Short Integer

Table 2.1: Primitive Types

Java strictly specifies a range and behavior for each primitive type, which all implementations of the Java Virtual Machine must support. Because of Java's portability requirement, Java is uncompromising on this account. For example, an int is the same in all execution environments. This allows programs to be fully portable. There is no need to rewrite code to fit a specific platform. Although strictly specifying the range of the primitive types may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Integer Types: Integer types can hold whole numbers such as 123, -96, and 5639. The size of the values that can be stored depends on the integer data type we choose. Java supports four types of integers. They are byte, short, int, and long. Java does not support the concept of *unsigned* types and therefore all Java values are signed meaning they can be positive or negative.

Type	Size	Minimum Value	Maximum Value
byte	One byte	-128	127
short	Two Bytes	-32,768	+32,767
int	Four bytes	-2,147,483,648	2,147,483,647
long	Eight bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Table 2.2 Integer Type

It should be remembered that wider data types require more time for manipulation and therefore it is advisable to use smaller data types, wherever possible.

Floating Point Types: Integer types can hold only whole numbers and therefore we use another type known *as floating point* type to hold numbers containing fractional parts such as 27.59 and -

1. The float type values are *single-precision* numbers while the double types represent *double precision* numbers. Floating point numbers are treated as double-precision quantities. To force them to be in single-precision mode, we must append for F to the numbers. Example: 1. 23f, 7.56923e5F. Double-precision types are used when we need greater precision in storage of floating point numbers. All mathematical functions, such as sin, cos and sqrt returns double type values. Floating point data types support a special value known as Not-a-Number (NaN). NaN is used to represent the result of operations such as dividing zero by a number, where an actual number is not produced. Most operations that have NaN as an operand will produce NaN as a result.

Type	Size	Minimum Value	Maximum Value
Float	4 bytes	3.4e-038	3.4e+038
Double	8 Bytes	1.7e-308	1.7e+308

Table 2.3 Floating Point Types

Character Type: In order to store character constants in memory, Java provides a character data type called char. The char type assumes a size of 2 bytes but, basically, it can hold only a single character.

Boolean Type

Boolean type is used when we want to test a particular condition during the execution of the program. There are only two values that a boolean type can take: true or false. Remember, both these words have been declared as keywords. boolean type is denoted by the keyword Boolean and uses only one bit of storage. All comparison operators return boolean type values. Boolean values are often used in selection and iteration statements.

2.2 Literals

Literals: Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables. Java language specifies five major types of literals. They are:

- Integer literals
- Floating point literals
- Character literals
- String literals
- Boolean literals

Each of them has a type associated with it. The type describes how the values behave and how they are stored.

Integer literals: An *integer* literal refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer

Decimal integer literals consist of a set of digits, 0 through 9, preceded by an optional minus sign. Valid examples of decimal integer constants are: 123, -321, 0654321

An *octal* integer literal consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are: 037, 00435, 0551

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer (hex integer). They may also include alphabets A through F or a through f. A letter A through F represents the numbers 10 through 15. Valid examples of hex integers are: OX2 OX9F Oxbcd Ox. We rarely use octal and hexadecimal numbers in programming.

Floating point literals: Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real (or floating point)* constants. Further examples of real constants are: 0.0083, -0.75435.36.

These numbers are shown in *decimal notation,* having a whole number followed by a decimal point and the fractional part, which is an integer. A real number may also be expressed in *exponential* (or *scientific) notation.* For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 100.

The general form is: *mantissa* e *exponent.* The *mantissa* is either if real number expressed in *decimal notation* or an integer. The *exponent* is an integer with an optional *plus* or *minus* sign. The letter e separating the mantissa and the exponent can be written in either lowercase or uppercase. A floating point literal may thus comprise four parts:

- Ø A whole number
- Ø A decimal point
- Ø A fractional part
- Ø An exponent

Single Character literals: A single literal contains a single character enclosed within a pair of single quote marks. Examples of character constants are: '5', 'X', ';'

String literals: A string literal is a sequence of characters enclosed between double quotes. The characters may be alphabets, digits, special characters and blank spaces. Examples are: "Hello Java", "1997", "WELL DONE", "?..!", "5+3", "X".

Boolean literals: Boolean literals include true or false values.

Character Escape Sequences: Java supports some special Character Escape Sequences that are used in output methods. For example, the symbol '\n' stands for newline character. Note that each one of them represents one character, although they consist of two characters. Constant Meaning

'\b'	back space
'\f'	form feed
'\n'	new line
'r'	carriage return
't'	horizontal tab
'\''	single quote
'\"'	double quote
'\\'	backslash

2.3 Variables

A variable may take different values at different times during the execution of the program. A variable name can be chosen by the programmer in a meaningful way so as to reflect what it

represents in the program. Some examples of variable names are*: average, height total_height.*
Variable names may consist of alphabets, digits, the underscore (-) and dollar characters, subject to the following conditions:
1. They must not begin with a digit.
2. Uppercase and lowercase are distinct. This means that the variable Total is not the same as total or TOTAL.
3. It should not be a keyword.
4. White space is not allowed.
5. Variable names can be of any length.

In Java, variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things:
1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. The place of declaration (in the program) decides the scope of the variable.
A variable must be declared before it is used in the program. A variable can be used to store a value of any data type. That is, the name has nothing to do with the type. Java allows any properly formed variable to have any declared data type. The declaration statement defines the type of variable. The general form of declaration of a variable is:

Type variable1, variable2……., variableN;

Variables are separated by commas, A declaration statement must end with a semicolon,
Some valid declarations are:

```
int    count;
float  x, y;
double pi;
```

Initializing a Variable: In general, you must give a variable a value prior to using it. One way to give a variable a value is through an assignment statement, as you have already seen. Another way is by giving it an initial value when it is declared. To do this, follow the variable's name with an equal sign and the value being assigned. The general form of initialization is shown here:

```
type var = value;
```

Here, value is the value that is given to var when var is created. The value must be compatible with the specified type. Here are some examples:

```
int count=10;  // give count an initial value of 10
char ch='X';   // initialize ch with the letter X
float f= 1.2F;  // f is initialized with 1.2
```

When declaring two or more variables of the same type using a comma-separated list, you can give one or more of those variables an initial value. For example

```
int a, b=8, c=19, d;  // b and c have initialization
```

In this case, only b and c are initialized.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the volume of a cylinder given the radius of its base and its height:

```
//Demonstrates dynamic initialization
class DynInit
{
        public static void main (String args[ ])  {
              double radius=4, height=5;
             // dynamically initialize volume
            double volume=3.1416 * radius * height;
            System.out.println( "Volume is" + volume);
        }
}
```

Here, three local variables—radius, height, and volume are declared. The first two, radius and height, are initialized by constants. However, volume is initialized dynamically to the volume of the cylinder. The key point here is that the initialization expression can use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

2.4 Scope and lifetime of variables

Usually in java all of the variables are declared at the start of the main( ) method. However, Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects. Many other computer languages define two general categories of scopes: global and local. Although supported by Java, these are not the best ways to categorize Java's scopes. The most important scopes in Java are those defined by a class and those defined by a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it. To understand the effect of nested scopes, consider the following program.

```
// demonstrates block scope.
class ScopeDemo {
    public static void main(String args[ ]) {
    int x; // Known to all code within main
    x=10;
    if (x = = 10) { //start new scope
       int y = 20; // known only to this block
     // x and y both known here.
```

```
        System.out.println("x and y:" + x + "  " + y);
        x= y * 2;
     }
    // y = 100; //Error! Y not known here
    // x is still known here.
    System.out.println("x is " +x);
}
```

As the comments indicate, the variable x is declared at the start of main( )'s scope and is accessible to all subsequent code within main( ). Within the if block, y is declared. Since a block defines a scope, y is visible only to other code within its block. This is why outside of its block, the line y = 100; is commented out. If you remove the leading comment symbol, a compile-time error will occur, because y is not visible outside of its block. Within the if block, x can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

If a variable declaration includes an initializer, that variable will be reinitialized each time the block in which it is declared is entered. Although blocks can be nested, no variable declared within an inner scope can have the same name as a variable declared by an enclosing scope.

2.5 Operators

Java supports a rich set of operators. Operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions. Java operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators.
7. Bitwise operators
8. Special operators

2.5.1Arithmetic operators

Java provides all the basic arithmetic operators. These can operate on any built-in numeric data type of Java. We cannot use these operators on boolean type. The unary minus operator, in effect, multiplies its single operand by 1. Therefore, a number preceded by a minus sign changes its sign.

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Table 2.4 Arithmetic Operators

Arithmetic operators are used as shown below:
a-b a*b a%b a+b a/b -a*b
Here a and b may be variables or constants and are known as *operands.*
Integer Arithmetic: When both the operands in a single arithmetic expression such as a+ b are integers, the expression is called an *integer expression,* and the operation is called *integer arithmetic.* Integer arithmetic always yields an integer value
Real Arithmetic: An arithmetic operation involving only real operands is called *real arithmetic.* A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. .
Mixed-mode Arithmetic: When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic e*xpression. If either operand is of the real type, then the other operand is converted to real and the real arithmetic is performed. The result will be a real.

2.5.2 Increment and Decrement Operators

Java has two very useful operators not generally found in many other languages. These are the increment and decrement operators: ++ and --
The operator + + adds 1 to the operand while - - subtracts 1. Both are unary operators and are used in the following form.
++m; or m++; is equivalent to m = m + 1;
--In; or m--; is equivalent to m = m - 1;

We use the increment and decrement operators extensively in for and while loops. While + +m and m+ + mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement.
Consider the following:
m = 5;
y = ++m;
In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as
m = 5;
Y = m++;
Then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.

2.5.3 Relational and Logical Operators

We often compare two quantities, and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators.*

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
= =	is equal to
!=	is not equal to

Table 2.5 Relational Operators

A simple relational expression contains only one relational operator and is of the following form:

ae-1 relational operator ae-2

ae-l and *ae-2* are arithmetic expressions, which may be simple constants, variables or combination of them.

Logical Operators: In addition to the relational operators, Java has three logical operators,

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical Not

Table 2.6 Logical Operators

The logical operators && and | | are used when we want to form compound conditions by combining two or more relations. An example is: a > b && x 10

An expression of this kind which combines two or more relational expressions is termed as a *logical expression* or a *compound relational expression.* Like the simple relational expressions. Logical expression also yields a value of true or false.

2.5.4 Short-Circuit Logical Operators

Java supplies special short-circuit versions of its AND and OR logical operators that can be used to produce more efficient code. In an AND operation, if the first operand is false, the outcome is false no matter what value the second operand has. In an OR operation, if the first operand is true, the outcome of the operation is true no matter what the value of the second operand. Thus, in these two cases there is no need to evaluate the second operand. By not evaluating the second operand, time is saved and more efficient code is produced. The short-circuit AND operator is &&, and the short-circuit OR operator is ||. Their normal counterparts are & and |. The only difference between the normal and short-circuit versions is that the normal operands will always evaluate each operand, but short-circuit versions will evaluate the second operand only when necessary.

Here is a program that demonstrates the short-circuit AND operator. The program determines whether the value in d is a factor of n. It does this by performing a modulus operation. If the remainder of n / d is zero, then d is a factor. However, since the modulus

operation involves a division, the short-circuit form of the AND is used to prevent a divide-by-zero error.

```
//Demonstrates the short-circuit operators.
Class Scops {
    public static void main (String args[ ]) {
        int n, d, q;
        n=10;
        d=2;
        if (d !=0 && (n % d)  = = 0)
            System.out.println(d  +  " is a factor of  " + n);
        d=0;   //now, set d to zero
        //Since d is zero, the second operand is not evaluated.
        if (d !=0 && ( n % d) = = 0)
          system.out.println(d + "is a factor of " + n);
        /* Now, try same thing without short-circuit operator.
            This will cause a divide-by-zero error. */
        if (d != 0 & (n % d) = = 0)
          System.out.println( d + " is a factor of " +n);
}
```

To prevent a divide-by-zero, the if statement first checks to see if d is equal to zero. If it is, the short-circuit AND stops at that point and does not perform the modulus division. Thus, in the first test d is 2 and the modulus operation is performed. The second test fails because d is set to zero, and the modulus operation is skipped, avoiding a divide-by-zero error. Finally, the normal AND operator is tried. This causes both operands to be evaluated, which leads to a runtime error when the division by zero occurs.

2.5.5 The Assignment Operators

Assignment operators are used to assign the value of an expression to a variable. We have seen the usual assignment operator = . In addition, Java has a set of 'shorthand' assignment operators which are used in the form

```
v  op=  exp;
```

Where v is a variable, *exp* is an expression and *op* is a Java binary operator. The operator op = is known as the shorthand assignment operator.

The assignment statement v op= exp; is equivalent to v = v op (exp) with v accessed only once.

Statement with simple assignment operator
a= a+1
a= a-1
a= a* (n+1)
a= a/(n+1)
a= a%b

Table 2.7 Assignment Operators

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. Use of shorthand operator results in a more efficient code

2.5.6 The Bitwise and Shift Operator

Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to float or double.

Operator	Meaning
&	Bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
~~	One's Complement
<<	Shift left
>>	Shift right
>>>	Shift right with zero fill

Table 2.8 Bitwise Operators

2.5.7 The ?: operator

The character pair ?: is a ternary operator available in Java. This operator is used to construct conditional expressions of the form

exp1 ? exp2 : exp3

Where *expl, exp2,* and *exp3* are expressions.

The operator?: works as follows: *expl* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the conditional expression. If *expl* is false, *exp3* is evaluated and its value becomes the value of the conditional expression. Note that only one of the expressions (either *exp2* or *exp3)* is evaluated. a = 10;

b = 15; x = (a > b) ? a : b; In this example, x will be assigned the value of b.

2.5.8 Shorthand Assignment

Java provides special shorthand assignment operators that simplify the coding of certain assignment statements. Let's begin with an example. The assignment statement shown here

x= x + 10

can be written using java short-hand as

x+=10

Statement with shorthand operator
a +=1
a -=1
a*= n+1
a/= n=1
a%=b

Table 2.9 Shorthand Assignment

2.6 Type Conversion in Assignments

In programming, it is common to assign one type of variable to another. For example, you might want to assign an int value to a float variable, as shown here:

```
int i;
float f;
i=10;
```

f=i //assigns an int to a float

When compatible types are mixed in an assignment, the value of the right side is automatically converted to the type of the left side. Thus, in the preceding fragment, the value int i is converted into a float and then assigned to f. However, because of Java's strict type checking, not all types are compatible, and thus, not all type conversions are implicitly allowed. For example, boolean and int are not compatible. When one type of data is assigned to another type of variable, an automatic type conversion will take place if the two types are compatible. The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, and both int and byte are integer types, so an automatic conversion from byte to int can be applied. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. There are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other. However, an integer literal can be assigned to char. The process of assigning a larger type to a smaller one is known as *narrowing*. Note that narrowing may result in loss of information.

```
// demonstrates automatic conversion from long to double.
class LtoD
{
      public static void main(String args[ ])  {
        long L;
       double D;
       L=100123285L;
       D= L     //Automatic conversion from long to double
       System. out.println(" L and D " + L + "  "  +D);
     }
}
```

2.7 Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all programming needs because they apply only to widening conversions between compatible types. For all other cases you must employ a cast. A cast is an instruction to the compiler to convert one type into another. Thus, it requests an explicit type conversion. A cast has this general form:

```
(target-type) expression
```

Here, target-type specifies the desired type to convert the specified expression to. For example, if you want to convert the type of the expression x/y to int, you can write

```
double x, y;
//. . .
(int) (x / y);
```

Here, even though x and y are of type double, the cast converts the outcome of the expression to int. The parentheses surrounding x / y are necessary. Otherwise, the cast to int would apply only to the x and not to the outcome of the division. The cast is necessary here because there is no automatic conversion from double to int.

When a cast involves a narrowing conversion, information might be lost. For example, when casting a long into a short, information will be lost if the long's value is greater than the range of a short because its high-order bits are removed. When a floating-point value is cast to an integer type, the fractional component will also be lost due to truncation. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 is lost. The following program demonstrates some type conversions that require casts:

```
// demonstrate casting.
class CastDemo {
    public static void main(String args[ ] ) {
        double x, y;
        byte b;
        int i;
       char ch;
       x= 10.0;
       y= 3.0;
       i = (int) (x / y );   // cast double to int
       System.out.println( "Integer outcome of x / y : " + i);
       i = 100;
       b =  (byte) i;
       System.out.println("Value of b: " + b);
       i = 257;
       b = (byte ) i;
       System.out.println("Value of b: " + b);
       b = 88;   //ASCII code for X
       ch = (char) b;
       System.out.println("ch: " +ch);
     }
}
```

The output from the program

```
Integer outcome of x / y: 3
Value of b: 100
Value of b: 1
```

In the program, the cast of (x / y) to int results in the truncation of the fractional component, and information is lost. Next, no loss of information occurs when b is assigned the value 100 because a byte can hold the value 100. However, when the attempt is made to assign b the value 257, information loss occurs because 257 exceeds a byte's maximum value. Finally, no information is lost, but a cast is needed when assigning a byte value to a char.

2.8 Operator Precedence

Each operator in Java has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of the levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the associativity property of an operator.

Operator	Description	Associativity	Rank
. ( ) [ ]	Member selection Function call Array element reference	Left to right	1
- ++ -- ! ~ (type)	Unary Minus Increment Decrement Logical negation Ones complement Casting	Right to Left	2
* / %	Multiplication Division Modulus	Left to right	3
+ -	Addition Subtraction	Left to right	4
<< >> >>>	Left shift Right shift Right shift with zero fill	Left to right	5
< <= > >= instance of	Less than Less than or equal to Greater than Greater than or equal to Type comparison	Left to right	6
== !=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional Operator	Right to left	13
= op=	Assignment Operator Shorthand assignment	Right to left	14

Table 2.10 Summary of Java Operators

2.9 Expressions

Operators, variables, and literals are constituents of expressions. Expressions are evaluated using an assignment statement of the form

variable=expression;

Variable is any valid Java variable name. When the statement is encountered, the *expression* is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

x= a*b-c; y = b/c*a; z= a-b/c+d;

The blank space around an operator is optional and is added only to improve readability. When these statements are used in program, the variables a,b,c and d must be defined before they are used in the expressions.

Type Conversion in Expressions: Within an expression, it is possible to mix two or more different types of data as long as they are compatible with each other. For example, you can mix short and long within an expression because they are both numeric types. When different types of data are mixed within an expression, they are all converted to the same type. This is accomplished through the use of Java's type promotion rules.

First, all char, byte, and short values are promoted to int. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float operand, the entire expression is promoted to float. If any of the operands is double, the result is double.

It is important to understand that type promotions apply only to the values operated upon when an expression is evaluated. For example, if the value of a byte variable is promoted to int inside an expression, outside the expression, the variable is still a byte. Type promotion only affects the evaluation of an expression.

Type promotion can, however, lead to somewhat unexpected results. For example, when an arithmetic operation involves two byte values, the following sequence occurs: First, the byte operands are promoted to int. Then the operation takes place, yielding an int result. Thus, the outcome of an operation involving two byte values will be an int. This is not what you might intuitively expect.

```
// A promotion surprise!
 class PromDemo  {
      public static void main(String args[ ] )  {
          byte b;
          int i;
          b = 10;
          i =  b * b;   // OK, no cast needed
          b = 10;
          b =  (byte) (b * b) ; // cast needed!!
          System.out.println( " i and b : " +  i  + "  " + b);
      }
}
```

No cast is needed when assigning b*b to i, because b is promoted to int when the expression is evaluated. However, when you try to assign b * b to b, you do need a cast back to byte.

Arithmetic Expressions: An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. Java can handle any complex mathematical expressions.

Algebraic expression	Java expression
a b-c	a*b-c
(m+n) (x+y)	(m+n) * (x+y)
—	a*b/c
3 +2x+1	3*x*x+2*x+1
– +c	x/y+c

Table 2.10 Expressions

2.10 Assignment-2

Short Answer questions (2 marks each)

1. What is the difference between 10.3 and 10.3f?
2. What are the two ways of initializing variables?
3. List different integer data types available in Java.
4. What is dynamic initialization of variables? Give example.
5. What are block-level variables? Give example.
6. What are the standard default values used for the various data types in java (M.U. Oct/Nov-2012)
7. List the eight basic data types used in Java. Give examples. (M.U. Oct/Nov 2008)
8. What is precedence and associativity of operators?
9. List different arithmetic operators available in Java
10. List different relational operators available in Java
11. What is a short circuit logical operator?
12. What do you mean by literal?
13. List out any four java primitive data types.
14. What is scope and lifetime of variables?
15. What is type conversion? What are its types?
16. What is automatic (implicit) type conversion? Give example.
17. When does automatic type conversion take place?
18. Write the meaning of System.out.println()
19. Write a single print statement to produce the following output

    ONE
    TWO
    THREE
20. How to create a block of code? Give example.

Long Answer Questions

1. Enumerate rules for identifiers? (5 Marks)
2. Write a simple Java program to display the message(5 marks-M.U. Oct/Nov. 2008)
3. What are different conventions used in java identifiers? (5 Marks)
4. What is Type casting? Why is it required in programming? How it is done in Java(5 Marks-M.U. Oct/Nov.2012, 2013)
5. What are variables? How to declare an assign values to variable? Explain with an example? (6 Marks)
6. List any five data types used in Java? Give Examples. (6 Marks-M.U. Oct/Nov.2012)
7. Write a java code to generate first 'N' Fibonacci series (5 Marks-M.U. Oct/Nov.2012)
8. List the primitive data types avialble in Java. Give examples. (5 Marks-M.U. Oct/Nov. 2010, 2014)
9. Write a program to find the number and sum of all integers greater than 100 and less than 200 that are divisible by 13. (8 Marks-M.U. Oct/Nov. 2008)
10. With example explain (5 Marks-M.U. Oct/Nov.2011, 2013)
    (i) Conditional operator (ii) increment and decrement operator
11. What are operators? Explain different types of operators? (10 Marks)

12. What is an expression? Explain arithmetic expression? (4 marks)
13. Explain different types of variables with their scope and examples. (6 Marks-M.U. Nov/Dec 2009)
14. Write a note on scope and life time of variables (4 or 5 Marks-M.U. Oct/Nov 2010, 2008)
15. Explain the relational and logical operators in Java. (4 Marks-M.U. Oct/Nov.2010)
16. Write the output of the following (4 Marks- M.U. Oct/Nov.2011)
    i. 10%3*2
    ii. 50/10 %-4
    iii. 15.75 %3.5+4.25
    iv. 60-14/7 % 30
17. Explain automatic type conversion with suitable code example. (4 marks)
18. Explain explicit type conversion with example. (4 Marks)
19. Write a note on :
    i) Increment and Decrement operator
    ii) Type conversion in Expression.

UNIT-I
Chapter 3
Using I/O

3.1 Byte streams and character streams

Modern versions of Java define two types of streams: byte and character. (The original version of Java defined only the byte stream, but character streams were quickly added.) Byte streams provide a convenient means for handling input and output of bytes. They are used, for example, when reading or writing binary data. They are especially helpful when working with files. Character streams are designed for handling the input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The fact that Java defines two different types of streams makes the I/O system quite large because two separate sets of class hierarchies (one for bytes, one for characters) are needed. The sheer number of I/O classes can make the I/O system. For the most part, the functionality of byte streams is paralleled by that of the character streams. At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

The Byte Stream Classes: Byte streams are defined by using two class hierarchies. At the top of these are two abstract classes: InputStream and OutputStream. InputStream defines the characteristics common to byte input streams and OutputStream describes the behavior of byte output streams. From InputStream and OutputStream are created several concrete subclasses that offer varying functionality and handle the details of reading and writing to various devices, such as disk files. The byte stream classes are shown in Table 3.1.

Byte Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputtStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the java standard data types
DataOutputStream	An output stream that contains methods for writing the java standard data types
FileInputStream	Input stream that reads from a file
FIleOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilerOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
ObjectInputStream	Input Stream for objects
ObjectOutputStream	Output Stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output Stream that contains print( ) and println( )

PushbackInputStream	Input stream that allows byte to be returned to the stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after other.

Table 3.1: The Byte Stream Class

Character Stream Class: Character streams are defined by using two class hierarchies topped by these two abstract classes: Reader and Writer. Reader is used for input, and Writer is used for output. Concrete classes derived from Reader and Writer operates on Unicode character streams. From Reader and Writer are derived several concrete subclasses that handle various I/O situations. In general, the character-based classes parallel the byte-based classes. The character stream classes are shown in Table 3.2

Character Stream Class	**Meaning**
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a chracter array
FileReader	Input stream that reads from a file
FIleWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilerWriter	Filtered writer
InputStreamReader	Input stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output Stream that contains print( ) and println( )
PushbackReader	Input stream that allows character to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after other.

Table 3.2: The Character Stream Class

3.3 The Predefined Streams

All Java programs automatically import the java.lang package. This package defines a class called System, which encapsulates several aspects of the run-time environment. Among other things, it contains three predefined stream variables, called in, out, and err. These fields are

declared as public, final, and static within System. This means that they can be used by any other part of program and without reference to a specific System object.

System.out refers to the standard output stream. By default, this is the console. System.in refers to standard input, which is by default the keyboard. System.err refers to the standard error stream, which is also the console by default. However, these streams can be redirected to any compatible I/O device. System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream. These are byte streams, even though they are typically used to read and write characters from and to the console. The reason they are byte and not character streams is that the predefined streams were part of the original specification for Java, which did not include the character streams.

3.4 Reading Console Input

Originally, the only way to perform console input was to use a byte stream, and much Java code still uses the byte streams exclusively. Today, you can use byte or character streams. For commercial code, the preferred method of reading console input is to use a character-oriented stream. Doing so makes your program easier to internationalize and easier to maintain. It is also more convenient to operate directly on characters rather than converting back and forth between characters and bytes. However, for sample programs, simple utility programs for your own use, and applications that deal with raw keyboard input, using the byte streams is acceptable. For this reason, a console I/O using byte stream is examined here.

Because System.in is an instance of InputStream, you automatically have access to the methods defined by InputStream. Unfortunately, InputStream defines only one input method, read( ), which reads bytes. There are three versions of read( ), which are shown here:

```
int read( ) throws IOException
int read(byte data[ ]) throws IOException
int read(byte data[ ], int start, int max) throws IOException
```

first version of read( ) to read a single character from the keyboard (from System.in). It returns −1 when the end of the stream is encountered. The second version reads bytes from the input stream and puts them into data until the array is full, the end of stream is reached, or an error occurs. It returns the number of bytes read, or −1 when the end of the stream is encountered. The third version reads input into data beginning at the location specified by start. Up to max bytes are stored. It returns the number of bytes read, or −1 when the end of the stream is reached. All throw an IOException when an error occurs. When reading from System.in, pressing ENTER generates an end-of-stream condition.

Here is a program that demonstrates reading an array of bytes from System.in. Notice that any I/O exceptions that might be generated are simply thrown out of main( ). Such an approach is common when reading from the console.

```
//Read an array of bytes from the keyboard.
import java. io. *;
class ReadBytes  {
  public static void main ( String args[ ])
    throws IOException {
      byte data[ ] = new byte[10];
```

```
        System.out.println( "Enter some characters. " );
        System.in.read (data);
        System.out.println("You entered: ");
        for(int i=0; i< data. length; i++)
            System.out.println( (char) data [i]);
      }
 }
```

Here is a sample run:

```
Enter some characters.
Read Bytes
You entered: Read Bytes
```

3.5Reading Characters

Characters can be read from System.in using the read( ) method defined by BufferedReader in much the same way as they were read using byte streams. There are three versions of read( ) supported by BufferedReader.

```
int read( ) throws IOException
int read(char data[ ]) throws IOException
int read(char data[ ], int start, int max) throws IOException
```

The first version of read( ) reads a single Unicode character. It returns −1 when the end of the stream is reached. The second version reads characters from the input stream and puts them into data until either the array is full, the end of file is reached, or an error occurs. It returns the number of characters read or −1 at the end of the stream. The third version reads input into data beginning at the location specified by start. Up to max characters are stored. It returns the number of characters read or −1 when the end of the stream is encountered. All throw an IOException on error. When reading from System.in, pressing ENTER generates an end-of-stream condition.

The following program demonstrates read ( ) by reading characters from the console until the user types a period. Notice that any I/O exceptions that might be generated are simply thrown out of main ( ).

```
// Use a BufferedReader to read characters from the console.
import  java.io.*;
class ReadChars  {
    public static void main (String args[ ])  throws IOException
     {
        char c;
        BufferedReader br = new BufferedReader (new InputStreamReader (System. in);
        System.out.println ("Enter characters, period to quit. " );
        // read characters
        do {
                c = (char) br.read ( );
                 System.out. println( c );
             } while (c!= ' . ');
     }
```

```
  }
```

Output:

```
 Enter characters, period to quit.
 One Two.
 O
 n
 e
 T
 w
 o
 .
```

3.5 Reading Strings

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

String readLine( ) throws IOException

It returns a String object that contains the characters read. It returns null if an attempt is made to read when at the end of the stream. The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until you enter the word "stop".

```
//Read a string from console using BufferedReader.
import java.io.*;

class ReadLines  {
    public static void main (String args [ ]) throws IOException
    {
        // create a BufferedReader  using System.in
        BufferedReader br= new BufferedReader (new InputStreamReader (System.in));
        String str;
        System.out.println("Enter lines of text . ");
        System.out.println("Enter 'stop' to quit. ");
        do {
              str  = br.readLine ( ) ;
              System.out.println(str);
            } while (! Str.equals("stop"));
     }
}
```

3.6 Writing console output

As is the case with console input, Java originally provided only byte streams for console output. Java 1.1 added character streams. For the most portable code, character streams are recommended. Because System.out is a byte stream, however, byte-based console output is still widely used. Console output is most easily accomplished with print( ) and println( ). These methods are defined by the class PrintStream (which is the type of the object referenced by System.out). Even though System.out is a byte stream, it is still acceptable to use this stream for simple console output. Since PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, it is possible to write to the console by using write( ). The simplest form of write( ) defined by PrintStream is shown here:

void write(int byteval)

This method writes the byte specified by byteval to the file. Although byteval is declared as an integer, only the low-order 8 bits are written. Here is a short example that uses write( ) to output the character X followed by a new line:

```
//Demonstrate System.out.write ( );
 class WriteDemo  {
      public static void main (String args [ ]) {
         int b;
         b = 'X';
         System.out.write(b);
         System.out.write ('\n');
    }
}
```

We will not often use write ( ) to perform console output (although it might be useful in some situations), since print( ) and println( ) are substantially easier to use. PrintStream supplies two additional output methods: printf( ) and format( ). Both give you detailed control over the precise format of data that you output. For example, you can specify the number of decimal places displayed, a minimum field width, or the format of a negative value.

3.7 Assignment-3

Short Answer questions (2 marks each)

1. What is stream? List different types of streams in Java.(M.U. Oct/Nov.2014)
2. What is byte and character stream?
3. What is character stream?
4. What are the top two abstract classes of byte stream classes?
5. List out any four byte stream classes with their meaning.
6. What are the top two abstract classes of character stream classes?
7. List out any four character stream classes with their meaning.
8. List the predefined streams of system object
9. What are the three versions of read ( )?
10. Write the simplest form of write ( )?

Long Answer questions

1. Explain byte stream classes. (5 Marks)
2. Explain any five byte stream classes. (5 Marks)
3. Write a short note on character stream classes. (5Marks)
4. Explain any five character stream classes. (5 Marks)
5. Write a short note on Predefined Streams. (5 Marks)
6. Explain any eight byte stream classes. (8 Marks)
7. Explain any eight character stream classes. (8 Marks)
8. Explain reading console input with an example. (6 Marks)
9. Explain the process of reading characters with an example code. (5 Marks)
10. With syntax and example explain the process of reading a string from the keyboard. (4 marks-M.U.Oct/Nov.2014)

UNIT-I

Chapter-4

Control Statements

4.1 Input Characters from the Keyboard

In Java one type of console input is reading a character from the keyboard. To read a character from the keyboard we will use System.in.read( ). System.in is the complement to System.out. It is the input object attached to the keyboard. The read( ) method waits until the user presses a key and then returns the result. The character is returned as an integer, so it must be cast into a char to assign it to a char variable. By default, console input is line buffered. Here, the term buffer refers to a small portion of memory that is used to hold the characters before they are read by program. In this case, the buffer holds a complete line of text. As a result, you must press ENTER before any character that you type will be sent to your program. Here is a program that reads a character from the keyboard.

```
//Read a character from the keyboard.
Class KbIn {
    public static void main (String args[ ]) throws java.io.IOException {
      char ch;
      System.out.println("Press a key followed by Enter: " );
      ch = (char) System.in.read( );  // get a char
      System.out.println("Your key is:  " + ch);
  }
}
```

In the program, notice that main( ) begins like this:

```
public static void main (String args[ ]) throws java.io.IOException {
```

Because System.in.read( ) is being used, the program must specify the throws java.io.IOException clause. This line is necessary to handle input errors. It is part of Java's exception handling mechanism. When you press ENTER, a carriage return, line feed sequence is entered into the input stream. Furthermore, these characters are left pending in the input buffer until you read them. Thus, for some applications, you may need to remove them (by reading them) before the next input operation.

4.2 The if statement

The 'if' statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a *two-way* decision statement and is used in conjunction with an expression. It takes the following form:

```
if  (test expression)
```

It allows the computer to evaluate the *expression* first and then, depending on whether the value of the *expression* (relation or condition) is 'true' or 'false', it transfers the control to a particular statement. This point of program has two paths to follow, one for the *true* condition and the other for the false condition

Simple if: The general form of a simple if statement is

```
If (test expression )
    {
        Statement-block;
    }
```

```
        Statement-x;
```

The 'statement-block' may be a single statement or a group of statements. If the *test expression* is true, the *statement-block* will be executed; otherwise the statement-block will be skipped and the execution will jump to the *statement-x.* Consider a case having two test conditions, one for weight and another for height. This is done using the compound relation

```
if (weight < 50 && height> 170) count = count +1;
```

This would have been equivalently done using two if statements as follows:

```
if (weight<50)
if (height>170)
count = count+l;
```

If the value of weight is less than 50, then the following statement is executed. Which in turn is another if statement. This if statement tests height and if the height is greater than **170,** then the count is incremented by 1. It should be remembered that when the condition is true both the statement-block and the statement-x are executed in sequence.

The if...else statement: The if else statement is an extension of the simple if statement. The general form is

```
    if (test expression)
      {
            True-block statement (s)
      }
      else
       {
            False-block statement (s)
       }
    statement-x
```

If the *test expression* is true, then *the true-block statement(s)* immediately following the if statement, are executed; otherwise, the *false-block statement(s)* are executed. In either case, either *true-block* or *false-block* will be executed, not both. In both the cases, the control is transferred subsequently to the *statement-x.*

```
if (code == 1)
boy = boy + 1;
else
girl = girl + 1;
XXX;
```

if the code is equal to 1, the statement boy = boy + 1; is executed and the control is transferred to the statement xxx, after skipp,ing the else part. If the code is not equal to 1, the statement boy = boy + 1; is skipped and the statement in the else part girl = girl + 1; is executed before the control reaches the statement xxx

4.3 Nested ifs

When a series of decisions are involved, we may have to use more than one if else statement in *nested* form. If the *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 per cent of the *balance* held on 31st December is given to every one, irrespective of their balances, and 5 per cent is given to female account holders if their balance is more than Rs 5000. This logic can be coded as follows

```
if (sex is female)
{
    if (balance> 5000)
       bonus = 0.05 * balance;
    else
         bonus = 0.02 * balance;
}
else
{
      bonus = 0.02 * balance;
}
     balance = balance + bonus;
```

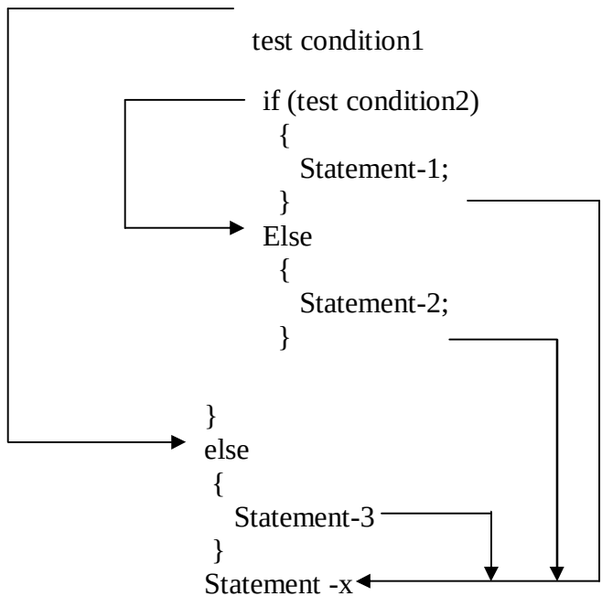

Figure 4.1 Nesting of if.. else statement

4.4 The if..else..if Ladder

There is another way of putting ifs together when multipath decisions are involved. A multipath decision is a chain of ifs in which the statement associated with each else is an if. It takes the following general form

The conditions are evaluated from the top *(of* the ladder), downwards. As *soon* as the true condition is found, the statement associated with it is executed and the control is transferred *to* the *statement-x* (skipping the rest *of* the ladder).when all the n conditions become false, then the final else containing the *default-statement* will be executed.

Consider an example *of* grading the students in an academic institution. The grading is done according to the following rules:

Average marks Grade
80 to 100 Honours
60 to 79 First Division
SOto 59 Second Division
40 to 49 Third Division
0 to 39 Fail
This grading can be done using the else ***if*** ladder as follows~

```
if(marks>-79)
        grade = "Honours";
else if (marks > 59)
        grade = "First Division";
else if (marks > 49)
        grade = "Second Division";
else if (marks > 39)
        grade = "Third Division";
else
        grade = "Fail";
System.out.println("Grade: " + grade);
```

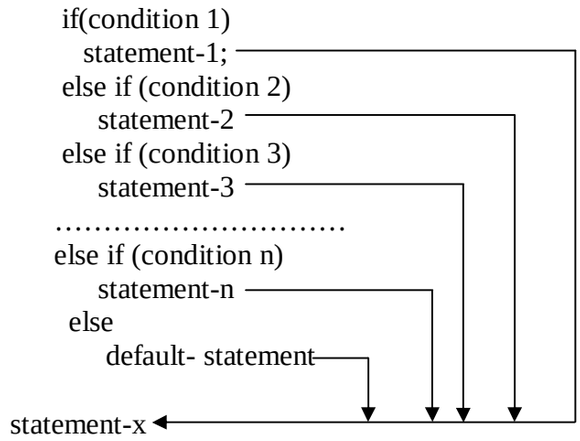

Figure4.2 General form of else if ladder

4.5 The switch statement

When one of the many alternatives is to be selected, we can design a program using if statements to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases. The program becomes difficult to read and follow. At times, it may confuse even the designer of the program. Fortunately, Java has a built-in multiway decision statement known as a switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed.

The *expression* is an integer expression or characters. *value-l, value-2* .". are constants or constant expressions (evaluable to an integral constant) and are known as *case labels* Each of these values should be unique within a switch statement. *block-l, block-2* are statement lists and

may contain zero or more statements. There is no need to put braces around these blocks but it is important to note that case labels end with a colon (:). When the switch is executed, the value of the expression is successively compared against the values *value-l, value-2,* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed. The *break* statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the *statement-x* following the switch. The *expression* is an integer expression or characters. *value-l, value-2* .". are constants or constant expressions (evaluable to an integral constant) and are known as *case labels* Each of these values should be unique within a switch statement. *block-l, block-2* are statement lists and may contain zero or more statements. There is no need to put braces around these blocks but it is important to note that case labels end with a colon (:).
The general form of the switch statement is as shown below:

```
switch (expression)
    {
        case value-1:
                block-l
                break;
        case value-2:
                block-2
                break;
        ……..
      ……
        default:
                default-block
                break;
    }
statement-x
```

When the switch is executed, the value of the expression is successively compared against the values *value-l, value-2,* If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed. The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the *statement-x* following the switch.
The default is an optional case. When present, it will be execute a if the value of the expression does not match with any of the case values. If not present, no action takes place when all matches fail and the control goes to the *statement-x.*

The switch statement can be used to grade the students as

```
…….
……...
index = marks/10;
switch (index)
  {
     case 10:
     case  9:
     case  8:
```

```
                grade = "Honours";
                        break;
        case 7:
        case 6:
                grade = "First Division";
                        break;
        case 5:
                grade = "Second Division";
                        break;
        case 4:
                grade =  "Third Division";
                        break;
               default:
                       grade = "Fail";
                        break;
  }
 System.out.println(grade);
```

4.6 Nested switch statement

It is possible to have a switch as part of the statement sequence of an outer switch. This is called a nested switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise. For example, the following code fragment is perfectly acceptable:

```
switch (ch1) {
      case 'A': System.out.println("This A is part of outer switch. ");
         switch (ch2)  {
             case 'A' :
                System.out.println (" This A is part of inner switch");
                Break;
             case 'B':  // …
         } // end of inner switch
        break;
       case 'B':   // . . .
```

4.7 The for loop

The for loop is an *entry-controlled* loop that provides a more concise loop control structure. The general form of the for loop is

```
for (intialization ; test condition; increment)
{
      Body of the loop
}
```

The execution of the for statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = o. The variables i and count are known as loop-control variables.

2. The value of the control variable is tested using the *test condition. The* test condition is a relational expression, such as i <10 that determines when the loop will exit. If the condition is *true,* the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i+ 1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.

Consider the following segment of a program

```
for (x = 0 ; x < = 9 ; x = x+1)
{
      System.out.println(x);
}
```

This for loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that there is no semicolon at the end of the *increment* section, x = x+ 1. The lor statement allows for negative *increments. For* example, the loop discussed above can be written as follows

```
for ( x = 9 ; x > = 0 ; x = x-1)
     System.out.println(x);
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Braces are optional when the body of the loop contains only one statement. Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for (x = 9; x < 9;x = x-1)
{
 …….
 …….
}
```

Will never be executed because the test condition fails at the very beginning itself.
One of the important points about for loop is that all the three actions, namely *initialization, testing* and *incrementing,* are placed in the for statement itself, thus making them visible to the programmers and users, in one place.
Additional Features of for Loop: The for loop has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the for statement. The statements

```
p = 1;
for (n=O; n<17; ++n)
```

initialization section has two parts p = 1 and n = 1 separated by a *comma.* Like the initialization section, the increment section may also have more than one part. For example, the loop

```
for (n=l, m=50; n<=m; n=n+1, m=m-1
```

```
 {
      …….
      …….
  }
```

is perfectly valid. The multiple arguments in the increment section are separated by *commas*. The third feature is that the test condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example that follows:

```
sum = 0 ;
for (i = 1, i < 20 && sum < 100; ++i)
  {
        ……..
        ……..
  }
```

The loop uses a compound test condition with the control variable i and external variable sum. The loop is executed as long as both the conditions i < 20 and sum < 100 are true. The sum is evaluated inside the loop. It is also permissible to use expressions in the assignment statements of initialization and increment sections.

For example, a statement of the type

for (x = (m+n)/2; x > 0; x = x/2) is perfectly valid.

Another unique aspect of for loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
m = 5;
for ( ;m ! = 100 ; )
 {
      System.out.println(m);
      m = m+5;
 }
```

Both the initialization and increment sections are omitted in for statement. The initialization has been done before t for statement and the control variable is incremented inside the loop. In such cases, the sections are left blank. However, the semicolons separating the sections must remain.

Notice that the body of the loop contains only a semicolon, known as a *empty* statement. This can also be written as

for (j=1000; j > 0; j = j-1);

This implies that the compiler will not give an error message if we place a semicolon by mistake at the end of for statement. The semicolon will be considered as an *empty* statement and the program may produce some nonsense.

4.8 The while Loop

The simplest of all the looping structures in Java is the while statement. The basic format of the while statement is

```
initialization;
 While (test condition)
   {
      Body of the loop
```

```
}
```

The while is an *entry-controlled* loop statement. The *test condition* is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the 1oop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

Consider the following code segment:

The body of the loop is executed 10 times for-n = 1,2, , 10 each time adding the square of the value of n, which.js incremented inside the loop. The test condition may also be written as n < 11; the result would be the same.

```
sum = 0;
n = 1;
while (n <= 10)
 {
      sum = sum + n * n;
      n = n+l;
 }
System.out.println("Sum = "+ sum);
```

4.9 The do..while loop

The while loop construct that we have discussed in the previous section makes a test condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. This takes the form:

```
initialization;
do
{
    Body of the loop
}
while (test condition)
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test condition* in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement. Since the *test condition* is evaluated at the bottom of the loop, the do while construct provides an *exit-controlled* loop and therefore the body of the loop is always executed at least once.

Consider an example:

```
i=1
sum =0;
do
{
      sum = sum + i;
      i = i+2;
}
while (sum < 40 I I i < 10);
```

The loop will be executed as long as one of the two relations is true.

4.10 break

An early exit from a loop can be accomplished by using the break statement. This statement can also be used within while, do or for loops. When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.

Use break as a form of goto: In Java, we can give a label to a block of statements. A label is any valid Java variable name. To give a label to a loop, place it before the loop with a colon at the end. Example:

```
Loop1:   for(……..)
                {
                    …….
                    …….
                }
                ……..
```

Simple break statement causes the control to jump outside the nearest loop. If we want to jump outside a nested loop, then we may have to use the labeled break statements.

```
class SpecialBreak
{
   public static void main (String args[ ])
   {
      LOOP1:   for (int i = 0; i < 10; i++) {
                        for(int j=0; j<10;j++) {
                           for(int k=0;k<10;k++)   {
                                System.out.println ( k + “  “);
                                if ( k = = 5 ) break Loop; // jump to done
                           }
                          System.out.println ( “After k loop”);  //won’t execute
                        }
                       System.out.println ( “After j loop”);  //won’t execute
                      }
                     System.out.println ( “After i loop”);
     }
}
```

The output from the program is shown here

```
0
1
2
3
4
5
After i loop
```

4.11 Continue

Like the break statement, Java supports another similar statement called the continue statement. However, unlike the break which causes the loop to be terminated, the continue, as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler. "skip the following statementsand continue with the next iteration". The format of the continue statement is:

```
continue;
```

In while and do loops, continue causes the control to go directly to the *test condition* and then to continue the iteration process. In the case of for loop, the *increment* section of the loop is executed before the *test condition* is evaluated.

```
// Use continue.
Class ContDemo  {
    public static void main(String args[ ]) {
        int i;
        // print even numbers between 0 and 100\
         for (i=0; i<=100;  i++)  {
            if (i % 2) != 0) continue;  // iterate
            System.out.println(i);
         }
      }
    }
```

Only even numbers are printed, because an odd one will cause the loop to iterate early, bypassing the call to println( ). As with the break statement, continue may specify a label to describe which enclosing loop to continue. Here is an example program that uses continue with a label:

```
class ContinueBreak
{
   public static void main (String args[ ])
   {
      LOOP1:   for (int i = 1; i < 100; i++)
                  {
                     System,out.println("  “);
                     if (i >= 10) break;
                     for (int j = 1; j < 100; j++)
                       {
                          System.out.print(" * ");
                          if ( j == i)
                            continue LOOP1;
```

```
                    }
                }
            System.out.println("Termination by BREAK");
    }
}
```

Program produces following output

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
```

Termination by BREAK

4.12 Nested Loops

Nesting of loops, that is, one loop can be nested inside of another. The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each loop statement. Nested loops are used to solve a wide variety of programming problems and are an essential part of programming.

A program segment to print a multiplication table using for loops is shown below

```
for (row= 1; row <= ROWMAX; ++row)
{
    for (column = 1; column <= COLMAX; ++ column)
      {
          y = row * column
          System.out.print(“  “+ y);
      }
    System.out.println(" " );
}
```

```
for (i = 1; i < 10; ++i)
  {
    ……..
    ……..
    for (j = 1; j != 5; ++j)
      {                                  Inner Loop
        ……..
        …….
      }
    …….                                  Outer Loop
    …….
  }
```

4.13 Assignment-4

Short Answer questions (2 Marks each)

1. Which is the method used to input characters from the keyboard?
2. Write the general syntax of if statement.
3. What is difference between break and continue statement?
4. What are labeled loops?
5. What is the value of count and sum after executing following segment? (M.U. Oct/Nov. 2011)

```
int count=0, sum=1, x=100;
while(count!=10);
{
   count++;
   sum+=x;
}
```

6. Write the general syntax of for loop.
7. Differentiate between while loop and do-while loop.
8. What do you mean by nested for loop.
9. Write the for statement for a loop that counts from 1000 to 0 by -2
10. Write a for loop to display even numbers between 1 and 100

Long Answer Questions

1. How to input characters from the keyboard? Explain with an example. (5 Marks)
2. Explain different type of if statements? (6 Marks)
3. What is entry controlled statement? Explain any one with its syntax and example. (5 Marks-M.U. Nov/Dec 2009)
4. Explain the if…else statement in java with a code example (6 Marks-M.U. Oct/Nov. 2010)
5. Explain the syntax of switch statement with an example. (7 Marks-M.U. Oct/Nov 2008)
6. Explain While loop and do while loop with a suitable code example. (6 Marks-M.U. Oct/Nov.2010, 2014)
7. Explain do while statement with syntax and examples. (5 Marks-M.U. Oct/Nov.2012)
8. Explain for statement with syntax and example (5 Marks- M.U. Oct/Nov.2013)
9. Explain different features of for loop? (5 Marks)
10. Explain switch statement with syntax and example. (5 Marks- M.U. Oct/Nov.2013)
11. Write a short note on nested switch statement. (5 Marks- M.U. Oct/Nov.2014)
12. Explain break statement with an example (5 Marks)
13. Explain continue statement with an example (5 Marks)
14. Explain break as a form of goto statement with an example (5 Marks)
15. What is infinite loop? Explain how to construct an infinite loop in Java.
16. Explain any three forms of for loop with suitable example.

UNIT-II
Chapter 5
Arrays

5.1 One-Dimensional Arrays

An array is a group of contiguous or related data items that share a common name. For instance, we can define an array name salary to represent a set of salaries of a group of employees.
A list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted* variable or a *one-dimensional* array. In mathematics, we often deal with variables that are single-subscripted.
In Java, single-subscripted variable x can be expressed as
x[l], x[2], x[3] x[n] The subscript can begin with number 0. That is x[0] is allowed. For example, if we want to represent a set of five numbers, say (35, 40, 20, 57, 19), by an array variable number, then we may create the variable number as follows
int number[ ] = **new** int[5]; and the computer reserves five storage location

The values to the array elements can be assigned as follows:
number[0]=35;
number[1]=40;
number[2]=20;
number[3]=57;
number[4]=19;

These elements may be used in programs just like any other Java variable. For example, the following are valid statements:
aNumber = number [0] +10;
number [4] = number [0] + number [2];
number [2] = x [5] + y[10];
value [6] = number [i] * 3;

The subscript of an array can be integer constants, integer variables like i , or expressions that yield integers.

5.2 Multidimensional Arrays

Although the one-dimensional array is the most commonly used array in programming, multidimensional arrays (arrays of two or more dimensions) are certainly not rare. In Java, a multidimensional array is an array of arrays.

5.2.1 Two-dimensional arrays

In mathematics, we represent a particular value in a matrix by using two subscripts such as V_{ij} Here v denotes the entire matrix and v_{ij} refers to the value in the i^{th} row and j^{th} column. Java allows us to define such tables of items by using *two-dimensional* arrays.
As with the single dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row. For creating two-dimensional arrays, we must follow the same steps as that of simple arrays. We may create a two-dimensional array like this

```
int myArray[ ] [ ];
myArray = new int [3] [4];
```

Or

```
int myArray[ ] [ ] = new int [3] [4];
```

This creates a table that can store 12 integer values, four across and three down. Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example, int table[2] [3] = {0, 0, 0, l, l, l}; initializes the elements of the first row to zero and the second row to one.

```
class MulTable
{
        final static int ROWS = 20;
        final static int COLUMNS = 20;
        public static void main (String args[ ])
          {
              int product[ ] [ ] = new int[ROWS] [COLUMNS];
              int row, column;
              System.out.println("MULTIPLICATION TABLE");
              System.out.println("  ");
              int i,j;
              for (i=10; i<ROWS; i++)
                {
              for (j=10; j<COLUMNS; J++)
                {
                   product [i] [j] = i*j;
                   System.out.print(" "+product[i] [j]);
               }
                 System.out.println("  ");
          }
```

Program Output

```
MULTIPLICATION TABLE
100 110  120  130 140 150 160 170 180 190
110 121  132  143 154 165 176 187 198 209
120 132  144  156 168 180 192 204 216 228
130 143  156  169 182 195 208 221 234 247
140 154  168  182 196 210 224 238 252 266
150 165  180  195 210 225 240 255 270 285
160 176  192  208 224 240 256 272 288 304
170 187  204  221 238 255 272 289 306 323
180 198  216  234 252 270 288 306 324 342
190 209  228  247 266 285 304 323 342 361
```

5.2.2 Irregular Arrays

Java treats multidimensional arrays as "arrays of arrays". When you allocate memory for a multidimensional array, you need to specify only the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, the following code allocates memory for the first dimension of table when it is declared.

It allocates the second dimension manually. For example-

```
int    x [ ] [ ]   = new int [3] [ ];
x [0]    = new   int [2];
```

```
x [1]  = new  int [4];
x [2]  = new  int [3];
```

These statements create a two-dimensional array as having different lengths for each row. Although there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions separately, you do not need to allocate the same number of elements for each index. Since multidimensional arrays are implemented as arrays of arrays, the length of each array is under your control.

5.2.3 Initializing Multidimensional Arrays

A multidimensional array can be initialized by enclosing each dimension's initializer list within its own set of curly braces. For example, the general form of array initialization for a two-dimensional array is shown here:

```
type-specifier array-name [ ] [ ] = {
     { val, val, val, …., val},
     { val, val, val, …., val},
.
.
.
  { val, val, val, …., val}
};
```

Here, val indicates an initialization value. Each inner block designates a row. Within each row, the first value will be stored in the first position of the sub array, the second value in the second position, and so on. Notice that commas separate the initializer blocks and that a semicolon follows the closing}.

For example,

int table[2] [3] = {0, 0, 0, l, l, l}; initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as int table [ ] [ ] = {{0, 0, 0}, {l, l, l}}; by surrounding the elements of each row by braces. We can also initializes a two-dimensional array in the form of a matrix as shown below:

```
int table [ ]  [ ]  =  {
                          { 0, 0, 0 },
                          { 0, 0, 0 }
                      }
```

We can refer to a value stored in a two-dimensional array by using subscripts for both the column and row of the corresponding element. Example: int value = table[1][2]; This retrieves the value stored in the second row and third column of table matrix.

5.3 Alternative Array Declaration Syntax

Arrays in Java may be declared as

```
type arrayname[ ];
```

Alternatively as

```
type [ ] arrayname;
```

Examples:

```
int          number [   ] ;
```

```
float          average [   ] ;
int [   ]      counter;
float [   ]    marks;
```

We do not enter the size of the arrays in the declaration.
After declaring an array, we need to create it in the memory. Java allows us to create arrays using new operator only, as shown below

```
arrayname  =  new   type [size];
```

Examples:

```
number  =  new int [5 ] ;
average  =  new float [10 ] ;
```

These lines create necessary memory locations for the arrays number and average and designate them as int and float respectively. Now, the variable number refers to an array of 5 integers and average refers to an array of 10 floating point values. It is also possible to combine the two steps-declaration and creation-into one as shown below:

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

5.4 Assigning Array References

As with other objects, when you assign one array reference variable to another, you are simply changing what object that variable refers to. You are not causing a copy of the array to be made, nor are you causing the contents of one array to be copied to the other. For example, consider this program:

```
//Assigning array references variables.
 class AssignAref  {
   public static void main (String args [ ] ) {
        int i;
        int num1 [ ] = new int [10];
        int num2 [ ] = new int [10];
        for (i=0; i<10; i++)
            num1[i] =  i;
        for (i=0; i<10; i++)
            num2[i] = - i;
        System.out.println ("Here is num1: ");
        for (i=0; i< 10; i++)
            System.out.println (nums1[i]  +  "  " );
         System.out.println( );
        System.out.println ("Here is num2: ");
        for (i=0; i< 10; i++)
            System.out.println (nums2[i]  +  "  " );
         System.out.println( );
         num2 = num1;   // now num2 refers to num1
        System.out.println( "Here is num2 after assignment: " );
        for (i=0; i< 10; i++)
            System.out.println (nums2[i]  +  "  " );
```

```
        System.out.println( );
      // now operate on num1 array through num2
      num2 [3] =99;
      System.out.println( “Here is num1 after change through num2: “);
        for (i=0; i< 10; i++)
            System.out.println (nums2[i]  +  “  “ );
        System.out.println( );
    }
}
```

The output of the program shown here
Here is num1: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Here is num2: 0, -1, -2, -3, -4, -5, -6, -7, -8, -9
Here is num2 after assignment: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Here is num1 after change through num2: 0, 1, 2, 99, 4, 5, 6, 7, 8, 9

As the output shows, after the assignment of nums1 to nums2, both array reference variables refer to the same object.

5.5 Using the length member

In Java, all arrays store the allocated size in a variable named length. We can access the length of the array a using a.length. Example: int aSize = a.length;
This information will be useful in the manipulation of arrays when their sizes are not known.

Sorting a *list* of numbers

```
class NumberSorting
{
      public static void main (String args[ ])
       {
            int number [ ] = { 55, 40, 80, 65, 71 };
            int n = number. length;
            System.out.print("Given list:   “);
            for (int i = 0; i < n; i++)
             {
                  System.out.print("   “ + number[i]);
             }
            System.out.println("\n”);
            / / Sorting begins
            for (int j = 0; j < n; j++)
               {
                    for (int j = i+l; j < n; j++)
                      {
                           if (number[i] < number[j])
                              {
                                  / / Interchange values
                                    int temp = number[i];
                                    number[i] = number[j];
                                    number[j] = temp;
```

```
                        }
                    }
                }
            System.out.print("Sorted list:    “);
            for (int i = 0; i < n; i++)
            {
                System.out.println(“   " + number[i]);
            }
            System.out.println(“  ");
        }
    }
```

Output
Given list: 55 40 80 65 71
Sorted list: 80 71 65 55 40

5.6 The For..Each Style for loop

When working with arrays, it is common to encounter situations in which each element in an array must be examined, from start to finish. For example, to compute the sum of the values held in an array, each element in the array must be examined. The second form of the for implements a “for-each” style loop. A for-each loop cycles through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. The for-each style of for is also referred to as the enhanced for loop. The general form of the for-each style for is shown here.

for(type itr-var: collection) statement-block

Here, type specifies the type, and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by collection. There are various types of collections that can be used with the for, out of which one is array. With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained. Thus, when iterating over an array of size N, the enhanced for obtains the elements in the array in index order, from 0 to N–1

Because the iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, type must be compatible with the element type of the array.

The following fragment uses a traditional for loop to compute the sum of the values in an array:

The following fragment uses a traditional for loop to compute the sum of the values in an array:

```
int nums [ ] =  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
int sum=0;
for (int i=0;  i<10;  i++)  sum+= nums[i];
```

To compute the sum, each element in nums is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the nums array by i, the loop control variable.

5.7 Iterating Over Multidimensional Arrays

The enhanced for also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of arrays of arrays. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array because each iteration obtains the next array, not an individual element. Furthermore, the iteration variable in the for loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each for to iterate over an array of N dimensions, the objects obtained will be arrays of N–1 dimensions.

```
// Use for-each style for on a two-dimensional array.
 class ForEach  {
      public static void main (String args[ ]) {
            int sum=0;
            int nums[ ]  [ ]  = new int [3] [5];
           // give nums some values
          for (int i=0; i<3; i++)
          for (int j=0; j<5; j++)
                nums[i][j] = (i + i) * (j + 1);
         // Use for each for loop to display and sum the values.
         for( int x[ ] : nums) {
            for (int y: x) {
               System.out.println( “Value is : “ + y);
               Sum += y;
            }
         }
         System.out.println( “Summation : “ + sum);
     }
}
```

The output from this program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

Notice how x is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the for obtains the next array in nums, beginning with the array specified by nums[0]. The inner for loop then cycles through each of these arrays, displaying the values of each element.

5.8 Applying the Enhanced for

Since the for-each style for can only cycle through an array sequentially, from start to finish. One of the most common is searching. For example, the following program uses a for loop to search an unsorted array for a value. It stops if the value is found.

```
//Search an array using for-each style for.
class Search {
    public static void main (String args[ ]) {
        int nums [ ] = { 6, 8, 3, 7, 5, 6, 1, 4} ;
        int val = 5;
        boolean found = flase;
        // Use for-each style for to search nums for val.
        for (int x : nums ) {
            if (x = = val)  {
                found = true;
                break;
            }
          }
        if (found)
            System.out.println("Value found!" );
     }
}
```

The for-each style for is an excellent choice in this application because searching an unsorted array involves examining each element in sequence. (of course, if the array were sorted, a binary search could be used, which would require a different style loop.) Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

5.9 Strings

String manipulation is the most common part of many Java programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in Java is by using a character array. Example:

```
char charArray [  ]  = new char [4];
charArray [ 0 ]  = 'J' ;
charArray [ 1 ]  = 'a' ;
charArray [ 2 ]  = 'v' ;
charArray [ 3 ]  = 'a' ;
```

Although character arrays have the advantage of being able to query, their length, they themselves are not good enough to support the range of operations we may like to perform on strings. For example, copying one character array into another might require a lot of book

keeping effort. Java is equipped to handle these situations more efficiently. In Java, strings are class objects and implemented using two classes, namely, **String** and **StringBuffer.** A Java string is an instantiated object of the **String** class. Java strings, as compared to C strings, are more reliable and predictable. This is basically due to C's lack of bounds-checking. A Java string is not a character array and is not NULL terminated. Strings may be declared and created as follows:

```
String      stringName;
stringName  = new String (“string”);
```

Example:

```
String firstName;
firstName = new String("Anil");
```

These two statements may be combined as follows:

```
String firstName = new String("Anil");
```

Like arrays, it is possible to get the length of string using the **length** method of the **String** class.

```
int m = firstName.length( ) ;
```

Note the use of parentheses here. Java strings can be concatenated using the + operator. Examples:

```
String fullname   =  name1 +  name2;
String city1      =  “New” + “Delhi” ;
```

where namel and name2 are Java strings containing string constants.

String Arrays: We can also create and use arrays that contain strings. The statement

```
String itemArray[ ] = new String[3];
```

Will create an itemArray of size 3 to hold three string constants. We can assign the strings to the itemArray element by element using three different statements or more efficiently using a for loop

String Methods

The String class defines a number of methods that allow us to accomplish a variety of string manipulation tasks.

Some Most Commonly Used String Methods

Method Call	**Task performed**
s2 = sl.toLowerCase;	Converts the string sl to all lowercase
s2 = sl.toUpperCase;	Converts the string sl to all Uppercase
s2 = sl. replace ('x' , 'y') ;	Replace all appearances of x with y
s2 = s1. trim ( ) ;	Remove white spaces at the beginning and end of the string s1
s1.equa1s (s2)	Returns 'true' if sl is equal to s2
sl.equa1sIgnoreCase(s2)	Returns 'true' if sl = s2, ignoring the case of characters
sl.length ( )	Gives the length of sl
sl.ChartAt(n)	Gives nth character of sl
sl.compareTo(s2)	Returns negative if sl < s2, positive if sl > s2, and zero if sl is equal s2

sl.concat(s2)	Concatenates sl and s2
sl.substring(n)	Gives substring starting from nth character
sl.substring(n, m)	Gives substring starting from nth character up to mth (not including m th)
String.Va1ueOf(p)	Creates a string object of the parameter p (simple type or object)
p. toString( )	Creates a string representation of the object p
sl. indexOf ('x')	Gives the position of the first occurrence of 'x' in the string s1
sl.indexOf('x', n)	Gives the position of 'x' that occurs after nth position in the string s1
String.ValueOf(Variable)	Converts the parameter value to string representation

Table 5.1 Some most commonly used String Method

StringBuffer Class

SringBuffer is a peer class of String. While String creates strings of fixed_length, StringBuffer creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string, or append another string to the end.

Method	**Task**
s1.setCharAt(n, 'x')	Modifies the nth character to x
s1.append (s2)	Append the string s2 to s1 at the end
s1.insert (n, s2)	Insert the string s2 at the position n of the string s1
s1.setLength (n)	Sets the length of the string s1 to n. if n<s1.length ( ) s1 is truncated. If n>s1.length ( ) zeros are added to s1

Table 5.2 Commonly Used StringBuffer Methods

Alphabetical ordering of strings

```
class StringOrdering
{
        static String name[] = {"Madras", "Delhi", "Ahmedabad",
                                "Calcutta", "Bombay"};
        public static void main (String args[ ])
         {
              int size = name. length;
             String temp = null;
            for (int i=0; i < size; i++)
            {
                for (int j = i+l; j < size; j++)
                 {
                    if (name[j] .compareTo(name[i]) < 0)
                    {
                            // swap the strings
                            temp = name[i];
                            name[i] = name[j];
                            name[j] = temp;
```

```
                        }
                    }
                }

            for (int i = 0; i < size; i++)
            {
                System.out.println(name[i]);
            }
        }
}
```

Program produces the following sorted list:

```
Ahmedabad
Bombay
Calcutta
Delhi
Madras
```

Manipulation of strings

```
class StringManipulation
{
    public static void main (String args[ ])
    {
        StringBuffer str = new StringBuffer("Object language");
        System.out.println("Original String :" + str);
        // obtaining string length
        System.out.println("Length of string :" + str.length());
        // Accessing characters in a string
        for (int i = 0; i < str.length(); i++)
        {
            int p = i + 1;
            System.out.println("Character at position:"  is" " + p +
                                " is " + str.charAt(i));
        }
        // Inserting a string in the middle
        String aString = new String(str.toString( ));
        int pos = aString.indexOf(" language");
        str.insert(pos," Oriented ");
        System.out.println("Modified string: " + str);
        //Modifying characters
        str. setCharAt (6, '-' ) ;
        System.out.println("String now: " + str);
        //Appending a string at the end
        str.append(" improves security."};
        System.out.println("Appended string: " + str);
    }
}
```

Output

```
Original String; Object language
```

```
Length of string: 15
Character at position: 1 is 0
Character at position: 2 is b
Character at position: 3 is j
Character at position: 4 is e
Character at position: 5 is c
Character at position: 6 is t
Character at position: 7 is
Character at position: 8 is l
Character at position: 9 is a
Character at position: 10 is n
Character at position: 11 is g
Character at position: 12 is u
Character at position: 13 is a
Character at position: 14 is g
Character at position: 15 is e
Modified string: Object Oriented language
String now: Object-Oriented language
Appended string: Object-Oriented language improves security.
```

5.10 Using Command-Line Arguments

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution. This is achieved in Java programs by using what are known as command line arguments. Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution.

We can write Java programs that can receive and use the arguments provided in the command line. Consider a main function statement as public static void main (String **args[ ]).** args is declared as an array of strings (known as String objects). Any arguments provided in the command line (at the time of execution) are passed to the array args as its elements. The individual elements of an array are accessed by using an index or subscript like arts [i]. The value of i denotes the position of the elements inside the array. For example, args[2] denotes the third element.

Use of command line arguments

```
/* This program uses command line
 * arguments as input.
*/
Class ComLineTest
{
        public static void main(String args[ ])
          {
               int count, i=O;
               String string;
               count = args.length;
               System.out.println("Number of arguments " + count);
               while(i < count)
```

```
            {
                string = args[i];
                i = i + 1;
                System.out.println(i+ " : " + "Java is " +string+ "!");
            }
        }
}
```

Compile and run the program with the command line as follows:

java ComLineTest Simple Object_Oriented Distributed Robust Secure
Portable Multithreaded Dynamic
Upon execution, the command line arguments Simple, Object_Oriented, etc. are passed to the program through the array args as discussed earlier. That is the element args[0] contains Simple, args[1] contains Object_Oriented, and so on. These elements are accessed using the loop variable i as an index like name = args[i].The index i is incremented using a while loop until all the arguments are accessed.
The number of arguments is obtained by statement
count = args.length;

The output of the program would be as follows:

Number of arguments = 8
1 : Java is Simple!
2 : Java is Object_Oriented!
3 : Java is Distributed!
4 : Java is Robust!
5 : Java is Secure!
6 : Java is Portable!
7 : Java is Multithreaded!
8 : Java is Dynamic!

Note how the output statement concatenates the strings while printing.

5.11 Assignment-2

Short Answer questions (2 marks each)

1. What are command line arguments?
2. Define Array. Write the purpose of length member.
3. How to create and instantiate a one dimensional array reference variable? Give example.
4. How to initialize an array at the time of creation? Give example.
5. What is the use of length member of arrays?
6. What are different ways of constructing String? Give example.
7. List any four methods associated with Strings.
8. What is an irregular array?
9. Mention any four Stringbuffer Class methods?

10. Write a statement to declare and instantiate an array to hold marks obtained by students in different subjects in a class. Assume that there are up to 60 students in a class and there are 8 subjects (M.U. Oct/Nov. 2012)
11. Identify the type of errors in the program segment given below: (M.U. Oct/Nov. 2012)

```
class Exam
{
  Public static void main(String args[ ])
{
    int S, n, X[ ]= {10, 20, 30, 40};
    n=X.length;    System.out.println("Given List:");
   for(i=0;i<=n;i++)
   System.out.println(X[i]);
}
```

12. What are the two forms of declaring array? (M.U. Oct./Nov. 2013)
13. Write the method to find array length.

Long Answer Questions

1. What are command Line arguments? How are they useful? (5 Marks-M.U. Oct/Nov.2012, 2014)
2. What are arrays? Explain how to declare instantiate initialize and use a one-dimensional array with suitable code example. (5 marks-M.U.Oct/Nov.2014)
3. Explain how to create irregular arrays? Explain with an example (4 Marks- M.U. Oct/Nov.2010, 2011, 2013)
4. Define StringBuffer class. Explain the different StringBuffer methods. (7 Marks- M.U. Oct/Nov. 2012, 2013)
5. How does the string class differ from string buffer class? Give the syntax and use of following method I) indexOf() II) subString() III)insert() IV)setCharAt()
6. Write a java code to input 'N' strings and display the string in alphabetical order. (4 marks- M.U. Oct/Nov.2012)
7. Explain the use of length member with suitable code example.(5 marks)
8. Explain different methods to initialize an array with suitable example. (5 marks)
9. How do you declare and initialize a two dimensional array? Give an example? (6 Marks-M.U. Oct/Nov.2012)
10. What is StringBuffer Class? Explain any two methods of String Buffer class. (4 Marks-M.U. Oct/Nov.2010)
11. Differentiate between the following methods. (6 Marks- M.U. Oct/Nov.2009)
    i) s1. CharAt (n) and s1.setCharAt(n, 'x')
    ii) s1.setLength(n) and s1.length( )
    iii) s1.append(s2) and s1.insert(n, s2)
12. Write a short note on assigning array references. (5 Marks)
13. Explain the For..Each Style for loop with an example (5 Marks)
14. Explain iterating over multidimensional array with an example (5 Marks)
15. Write a short note on Applying the Enhanced for. (5 Marks)

UNIT-II

Chapter 6

Classes, Objects and Methods

6.1 Class Fundamentals

Java is a true object-oriented language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a class that defines the *state* and *behaviour* of the basic program components known as *objects.*
Classes create objects and objects use methods to communicate between them. That is all about object-oriented programming.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In Java, the data items are called fields and the functions are called *methods.* Calling a specific method in an object is described as sending the object a message.

A class is essentially a description of how to make an object that contains fields and methods.
It provides a sort of *template* for an object and behaves like a basic data type such as into it is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OOP concepts such as *encapsulation, inheritance* and *polymorphism.*

General Form of a class: A class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create "variables" of that type. In Java, these variables are termed as *instances* of classes, which are the actual *objects.* The basic form of a class definition is:

```
class     classname [extends superclassname]
  {
        [ variable declaration; ]
        [ methods declaration; ]
  }
```

Everything inside the square brackets is optional. Even body of the class can be empty. Because of the body is empty, this class does not contain any properties and therefore cannot do anything. We can, however, compile it and even create objects using it. Classname and superclassname are any valid Java identifiers. The keyword extends indicates that the properties of the superclassname class are extended to the classname class. This concept is known as inheritance.

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables. Example:

```
        class Rectangle
          {
                int width;
                int length;
           }
```

The class **Rectangle** contains two integer type instance variables. It is allowed to declare them in one line as

```
int length, width;
```

These variables are only declared and therefore no storage space has been created in the memory. Instance variables are also known as member variables.

6.2 Creating objects

An object in Java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as instantiating an object.

Objects in Java are created using the new operator. The new operator creates an object of the specified class and returns a reference to that object. Here is an example of creating an object of type Rectangle.

```
Rectangle rect1 / / declare
rect1 = new Rectangle ( ) / / instantiate
```

The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable rect1 is now an object of the Rectangle class. Both statements can be combined into one as shown below:

```
Rectangle rect1 = new Rectangle ( );
```

The method Rectangle ( ) is the default constructor of the class. We can create any number of objects of Rectangle.

```
Rectangle rect1 = new Rectangle( ) ;
Rectangle rect2 = new Rectangle( ) ;
```

It is important to understand that each object has its own copy of the instance variables of its class. This means that any changes to the variables of one object have no effect on the variables of another.

6.3 Reference Variables and Assignment

In an assignment operation, object reference variables act differently than do variables of a primitive type, such as int. When you assign one primitive-type variable to another, the situation is straightforward. The variable on the left receives a copy of the value of the variable on the right. When you assign one object reference variable to another, the situation is a bit more complicated because you are changing the object that the reference variable refers to. The effect of this difference can cause some counterintuitive results. For example, consider the following fragment

```
Vehicle car1= new Vehicle ( );
Vehicle car2= car1
```

car1 and car2 will both refer to the same object. The assignment of car1 to car2 simply makes car2 refer to the same object as does car1. Thus, the object can be acted upon by either car1 or car2. Although car1 and car2 both refer to the same object, they are not linked in any other way.

6.4 Adding Methods

A class with only data fields (and without methods that operate on that data) has no life. The objects created by such a class cannot respond to, any, messages. We must therefore add methods that are necessary for manipulating the data contained in the class. The general form of a method declaration is

```
type methodname (parameter-list)
  {
```

```
        method-body;
    }
```

Method declarations have four basic parts:

- The name of the method (method name)
- The type of the value the method returns (type)
- A list of parameters (parameter-list)
- The body of the method

The type specifies the type of value the method would return. This could be a simple data type such as int as well as any class type. It could even be void type, if the method does not return any value. The method name is a valid identifier. The parameter list is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input. The variables in the list are separated by commas. In the case where no input data are required, the declaration must retain the empty parentheses.

Examples:

```
(int m, float x, float y)    / / Three parameters
 ( )                         / / Empty list
```

The body actually describes the operations to be performed on the data. Let us consider the Rectangle class again and add a method getData ( ) to it.

```
class Rectangle
{
        int length;
        int width;
        void getData(int x , int y )
         {
            length= x ;
            width= y ;
         }
}
```

Note that the method has a return type of void because it does not return any value. We pass two integer values to the methods which are then assigned to the instance variables length and width. The getData() method is basically added to provide values to the instance variables. Let us add some more properties to the class. Assume that we want to compute the area of the rectangle defined by the class. This can be done as follows:

```
  class Rectangle
   {
          int length, width; / / Com bind declaration
          void getData(int x ,   int y)
          {
               length = x ;
               width = Y ;
          }
          int rectArea( )
```

```
        {
            int area = length * width;
            return (area)
        }
    }
```

The new method rectArea() computes area of the rectangle and returns the result. Since the result would be an integer, the return type of the method has been specified as into Also note that the parameter list is empty. Declaration of instance variables (and also local variables) can be combined as

```
int length, width;
```

The parameter list used in the method header should always be declared independently separated by commas. That is,

```
void getData (int x, y) // Incorrect is illegal.
```

Now, our class Rectangle contains two instance variables and two methods. We can add more variables and methods, if necessary. Instance variables and methods in classes are accessible by all the methods in the class but a method cannot access the variables declared in other methods. Example:

```
class Access
{
    int x ;
    void method1( )
     {
        int y;
        x = 10 ;   // legal
        y = x      // legal
     }
    void method2( )
     {
       int z ;
       x = 5;   //legal
       z = 10; //legal
       y = 1;   //illegal
     }
 }
```

6.5 Returning From a Method

In general, there are two conditions that cause a method to return, when the method's closing curly brace is encountered. The second is when a return statement is executed. There are two forms of return, one for use in void methods (those that do not return a value) and one for returning values. In a void method, you can cause the immediate termination of a method by using this form of return:

```
return;
```

When this statement executes, program control returns to the caller, skipping any remaining code in the method. For example, consider this method:

```
void myMeth ( ) {
    int i;
```

```
        for (i=0;   i<10;   i++ ) {
            if ( i = = 5) return;    // stop at 5
            System.out.println( );
        }
    }
```

Here, the for loop will only run from 0 to 5, because once i equals 5, the method returns. It is permissible to have multiple return statements in a method, especially when there are two or more routes out of it.

6.6 Returning a Value

Although methods with a return type of void are not rare, most methods will return a value. In fact, the ability to return a value is one of the most useful features of a method. Return values are used for a variety of purposes in programming. In some cases, such as with sqrt( ), the return value contains the outcome of some calculation. In other cases, the return value may simply indicate success or failure. In still others, it may contain a status code. Methods return a value to the calling routine using this form of return:

return value;

Here, value is the value returned. This form of return can be used only with methods that have a non-void return type.

Example:

```
class Rectangle
  {
       int length, width;                  // Declaration of variables
      void getData(int x, int y)          // Definition of method
       {
          length = x;
          width = y;
          int rectArea ( )     // Definition of another method
           {
              int area = length * width;
              return (area);
           }
      }
  }
class RectArea                                  // Class with main method
  {
    public static void main (String args[ ])
      {
          int areal,area2;
          Rectangle rect1 = new Rectangle();    //Creating objects
          Rectangle rect2 = new Rectangle();
          rect1.length = 15;                    // Accessing variables
          rect1. width = 10;
          areal = rect1.length * rect1.width;  I I Accessing methods
          rect2.getData(20,12);
          area2 = rect2.rectArea();
          System.out.println("Area1 =  "+ areal);
          System.out.println("Area2 =  "+ area2);
```

```
    }
  }
```

Output:

```
 Area1= 150
 Area1=240
```

6.7 Using Parameters

It is possible to pass one or more values to a method when the method is called. Inside the method, the variable that receives the argument is called a parameter. Parameters are declared inside the parentheses that follow the method's name. The parameter declaration syntax is the same as that used for variables. A parameter is within the scope of its method, and aside from its special task of receiving an argument, it acts like any other local variable.

Example:

```
class Rectangle
  {
      int length, width;                  // Declaration of variables
     void getData(int x, int y)          // Definition of method
      {
         length = x;
         width = y;
     }
  }
class RectArea                                  // Class with main method
  {
    public static void main (String args[ ])
     {
         Rectangle rect1 = new Rectangle();    //Creating objects
         rect1.length = 15;                    // Accessing variables
         rect1. width = 10;
         System.out.println("Length =  "+ length);
         System.out.println("Area2 =  "+ width);
     }
  }
```

6.8 Constructors

All objects that are created must be given initial values. We can do these using two approaches. The first approach uses the dot operator to access the instance variables and then assigns values to them individually. It can be a tedious approach to initialize all the variables of all the objects.

The second approach takes the help of a method like getData to initialize each object individually using statements like,

```
rect1.getData (15, l0);
```

It would be simpler and more concise to initialize an object when it is first created. Java supports a special type of method, called a constructor that enables an object to initialize itself when it is created. Constructors have the same name as the class itself. They do not return any value, not even void. This is because they return the instance of the class itself.

Let us consider our Rectangle class again. We can now replace the getData method by a constructor method as shown below:

```
class Rectangle
{
    int length;
    int width;
    Rectangle ( ) / / Constructor method
      {
        length = 10 ;
        width = 20 ;
      }
    int rectArea ( )
    {
      return(length * width);
    }
}
class RectangleArea
{
     public static void main (string args[ ])
     Rectangle rect1 = new Rectangle ( );        // Calling constructor
     int areal = rect1.rectArea( ) ;
     System.out.println("Area1 = "+ areal) ;
}
```

Output:
Area1=200;

6.9 Parameterized Constructors

In the preceding example, a parameter-less constructor was used. Although this is fine for some situations, most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parentheses after the constructor's name.

Example:

```
class Rectangle
{
    int length;
    int width;
    Rectangle (int x, int y) / / Constructor method
      {
        length = x ;
        width = y ;
      }
    int rectArea ( )
    {
      return(length * width);
    }
}
```

```
class RectangleArea
   {
        public static void main (string args[ ])
        Rectangle rect1 = new Rectangle (15,10);        // Calling constructor
        int areal = rect1.rectArea( ) ;
        System.out.println("Area1 = "+ areal) ;
   }
```

Output:
Area1=150;

6.10 Adding a Constructor

We can improve the class by adding a constructor that automatically initializes the fields or data when an object is constructed.

Example:

```
 class Rectangle
   {
        int length;
        int width;
        Rectangle (int x, int y) / / Constructor method
          {
             length = x ;
             width = y ;
          }
   }
class RectangleArea
   {
        public static void main (string args[ ])
        Rectangle rect1 = new Rectangle (15,10);        // Calling constructor
        System.out.println("Length = "+  length) ;
        System.out.println("Width = "+  length) ;
   }
```

Output:
Length=15
Width=10

6.11 The new operator

In the context of an assignment, the new operator has this general form:

class-var = new class-name (arg-list);

Here, class-var is a variable of the class type being created. The class-name is the name of the class that is being instantiated. The class name followed by a parenthesized argument list (which can be empty) specifies the constructor for the class. If a class does not define its own constructor, new will use the default constructor supplied by Java. Thus, new can be used to create an object of any class type. The new operator returns a reference to the newly created object, which (in this case) is assigned to class-var. Some time since memory is finite, it is possible that new will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur.

6.12 Garbage Collection and Finalizers

objects are dynamically allocated from a pool of free memory by using the new operator. As explained, memory is not infinite, and the free memory can be exhausted. Thus, it is possible for new to fail because there is insufficient free memory to create the desired object. For thisreason, a key component of any dynamic allocation scheme is the recovery of free memory from unused objects, making that memory available for subsequent reallocation. In many programming languages, the release of previously allocated memory is handled manually. For example, in C++, you use the delete operator to free memory that was allocated. However, Java uses a different, more trouble-free approach: garbage collection. Java's garbage collection system reclaims objects automatically-occurring transparently, behind the scenes, without any programmer intervention. It works like this: When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object is released. This recycled memory can then be used for a subsequent allocation. Garbage collection occurs only periodically during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. For efficiency, the garbage collector will usually run only when two conditions are met: there are objects to recycle, and there is a need to recycle them. Remember, garbage collection takes time, so the Java run-time system does it only when it is appropriate. Thus, you can't know precisely when garbage collection will take place.

6.13 The finalize Method ( )

Java supports a concept called finalization, which is just opposite to initialization. Java run-time is an automatic garbage collecting system. It automatically frees up the memory resources used by the objects. But objects may hold other non-object resources such as file descriptors or window system fonts. The garbage collector cannot free these resources. In order to free these resources we must use a finalizer method. This is similar to destructors in c++. The finalizer method is simply finalize ( ) and can be added to any class. Java calls that method whenever it is about to reclaim the space for that object. The finalize method should explicitly define the tasks to be performed.

The finalize ( ) method has this general form

```
protected void finalize ( )
 {
     //finalization code here
 }
```

Here, the keyword protected is a specifier that prevents access to finalize ( ) by code defined outside its class. It is important to understand that finalize ( ) is called just before garbage collection. It is not called when an object goes out of scope, for example. This means that you cannot know when or even if finalize ( ) will be executed. For example, if your program ends before garbage collection occurs, finalize ( ) will not execute. Therefore, it should be used as a "backup" procedure to ensure the proper handling of some resource, or for special-use applications, not as the means that your program uses in its normal operation. In short, finalize ( ) is a specialized method that is seldom needed by most programs.

6.14 The this keyword

When a method is called, it is automatically passed an implicit argument that is a reference to the invoking object (that is, the object on which the method is called). This reference is called this. To understand this, consider following example

```
class Rectangle
{
     int length, width;                  // Declaration of variables
    void getData(int x, int y)          // Definition of method
     {
        this.length = x;
        this.width = y;
        int rectArea ( )     // Definition of another method
         {
            int area = this. length * this.width;
            return (area);
         }
    }
}
class RectArea                                  // Class with main method
{
   public static void main (String args[ ])
    {
        int areal,area2;
        Rectangle rect1 = new Rectangle();    //Creating objects
        Rectangle rect2 = new Rectangle();
        rect1.length = 15;                      // Accessing variables
        rect1. width = 10;
        areal = rect1.length * rect1.width;  I I Accessing methods
        rect2.getData(20,12);
        area2 = rect2.rectArea();
        System.out.println("Area1 =  "+ areal);
        System.out.println("Area2 =  "+ area2);
    }
}
```

Output:

Area1= 150
Area1=240

6.15 Controlling Access to Class Members

In its support for encapsulation, the class provides two major benefits. First, it links data with the code that manipulates it. Second, it provides the means by which access to members can be controlled. Although Java's approach is a bit more sophisticated, in essence, there are two basic types of class members: public and private. A public member can be freely accessed by code defined outside of its class. A private member can be accessed only by other methods defined by its class. It is through the use of private members that access is controlled.

Restricting access to a class' members is a fundamental part of object-oriented programming because it helps prevent the misuse of an object. By allowing access to private data only through a well-defined set of methods, you can prevent improper values from being assigned to that data by performing a range check, for example. It is not possible for code outside the class to set the value of a private member directly. You can also control precisely how and

when the data within an object is used. Thus, when correctly implemented, a class creates a "black box" that can be used, but the inner workings of which are not open to tampering.

6.16 Java's Access Modifiers

In java we can restrict the access to certain variables and methods from outside the class. For example, we may not like the objects of a class directly alter the value of a variable or access a method. We can achieve this in Java by applying *visibility modifiers* to the instance variables and methods. The visibility modifiers are also known as *access modifiers*. Java provides three types of visibility modifiers: public, private and protected. They provide different levels of protection as described below

public Access

public Visibility control is visible to all the classes even outside the class where it is defined. Example:

```
public int number;
public void sum( )   { ……….}
```

A variable or method declared as public has the widest possible visibility and accessible everywhere.

friendly Access

When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access. The difference between the "public" access and the "friendly" access is that the public modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages.

protected Access

The visibility level of a "protected" field lies in between the public access and friendly access. That is, the protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. Note that non-subclasses in other packages cannot access the "protected" members.

private Access

private fields enjoy the highest degree of protection. They are accessible only within their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. A method declared as private behaves like a method declared as final. It prevents the method from being subclassed. Also note that we cannot override a non-private method in a subclass and then make it private.

private protected Access

A field can be declared with two keywords private and protected together like:

```
Private protected int codeNumber;
```

This gives a visibility level in between the "protected" access and "private" access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package. Table summarises the visibility provided by various access modifiers.

Access modifier → / Access Location ↓	Public	Protected	friendly (default)	private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in Same package	Yes	Yes	Yes	No	No
Subclass in same packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

Table 6.1 Visibility Control

6.17 Pass Objects to Methods

In Java it is both correct and common to pass objects to methods. Consider following example

```
class Rectangle
{
    int length;
    int width;
    Rectangle (int x, int y) / / Constructor method
      {
        length = x ;
        width = y ;
      }
    void area (Rectangle r1 )
    {
       int arearect =  r1.length * r1.width;
      System.out.println ("Areaa of Rectangle : " + arearect);
    }
}
class RectangleArea
{
     public static void main (string args[ ])
     Rectangle rect1 = new Rectangle (15,10);        // Calling constructor
     Rectangle rect2 = new Rectangle (20,10);
     rect2.rectArea(rect1 ) ;
  }
```

Output:

Area1=150;

When you pass an object to a method, objects are implicitly passed by reference. Keep in mind that when you create a variable of a class type, you are creating a reference to an object. It is the reference, not the object itself, which is actually passed to the method. As a result, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference.

6.18 Returning Objects

A method can return any type of data, including class types. The class ErrorMsg could be used to report errors. Its method, getErrorMsg( ), returns a String object that contains a description of an error based upon the error code that it is passed.

```
// Return a String object.
    class ErrorMsg  {
        string msgs[ ]=  {“Output Error”, “Input Error”, Disk Full”, “Index Out-Of-Bounds”};
// Return the error message.
  String getErrorMsg ( int i) {
        if (i > 0 & i < msgs.length)
              return msgs[ i];
        else
                  return “Invalid  Error Code:’
    }
}
 class ErrMsg {
    public static void main (String args[ ])
        ErrorMsg err = new ErrorMsg()’
        System.out.println (err.getErrorMsg(2));
        System.out.println (err.getErrorMsg(4));
    }
```

Output:
Disk Full
Invalid Error Code

When an object is returned by a method, it remains in existence until there are no more references to it. At that point it is subject to garbage collection. Thus, an object won’t be destroyed just because the method that created it terminates.

6.19 Method Overloading

In Java it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of Parameters to decide which one of the definitions to execute. This process is known as polymorphism

To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists. The difference may either be in the number or type of arguments. That is, each parameter list should be unique.

Method's return type does not play any role in this. Here is an example of creating an overloaded method.

```
class Room
{
      float length;
      float breadth;
      Room(float x, float y)          // constructor1
        {
            length= x;
            breadth = y;
        }
       Room(float x)              // constructor2
        {
             length = breadth = x ;
        }
       int area ( )
        {
             return (length * breadth) ;
        }
 }
```

Here, we are overloading the constructor method Room ( ). An object representing a rectangular room will be created as

```
Room room1 = new Room (25.0, 15.0);              //using constructor
```

On the other hand, if the room is square, then we may create the corresponding object as

```
Room room2 = new Room (20.0);                    // using constructor2
```

6. 20 Overloading Constructers

Like methods, constructors can also be overloaded. Doing so allows you to construct objects in a variety of ways. For example, consider the following program:

```
class Rectangle
   {
        int length;
        int width;
        Rectangle (  )
          {
               length= 10;
               width = 20;
          }
         Rectangle ( int k)
           {
               length = k;
               width = k;
           }
        Rectangle (int x, int y) / / Constructor method
          {
```

```
            length = x ;
            width = y ;
         }
       int rectArea ( )
        {
           return(length * width);
        }
   }
class RectangleArea
   {
        public static void main (string args[ ])
        Rectangle rect1= new Rectangle ( );
        Rectangle rect2 = new Rectangle (10);
        Rectangle rect3 = new Rectangle (15,10);
        int areal = rect1.rectArea( ) ;
        System.out.println("Area1 = "+ areal) ;
        int area2 = rect2.rectArea( ) ;
        System.out.println("Area2 = "+ area2) ;
        int area3 = rect3.rectArea( ) ;
        System.out.println("Area3 = "+ area3) ;
   }
```

Output:
Area1=200;
Area2= 100;
Area3=150;

The proper constructor is called based upon the parameters specified when new is executed. By overloading a class' constructor, you give the user of your class flexibility in the way objects are constructed. One of the most common reasons that constructors are overloaded is to allow one object to initialize another.

6.21 Recursion

In Java, a method can call itself. This process is called recursion, and a method that calls itself is said to be recursive. In general, recursion is the process of defining something in terms of itself and is somewhat similar to a circular definition. The key component of a recursive method is a statement that executes a call to itself. Recursion is a powerful control mechanism.
The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1 × 2 × 3, or 6. The following program shows a recursive way to compute the factorial of a number. For comparison purposes, a non recursive equivalent is also included.

```
// A simple example of recursion.
  class Factorial  {
      //This is a recursive function.
      int factR (int n)
          int result;
      if (n= = 1 ) return 1;
```

```
        result = factR (n-1) * n;
        return result;
    }

    //This is an iterative equivalent.
   int factI ( int n) {
         int t, result;
         result = 1;
         for (t=1; t<=n; t++)     result * = t;
         return result;
   }

class Recursion {
     public static void main (String args [ ] )  {
          Factorial f = new Factorial ();
          System.out.println("Factorial using recursive method. ");
          System.out.println("Factorial of 3 is " + f.factR (3) );
          System.out.println("Factorial of 4 is " + f.factR (4) );
          System.out.println("Factorial of 5 is " + f.factR (5) );
          System.out.println( );
          System.out.println("Factorial using iterative method. ");
          System.out.println("Factorial of 3 is " + f.factI (3) );
          System.out.println("Factorial of 4 is " + f.factI (4) );
          System.out.println("Factorial of 5 is " + f.factI (5) );
      }
}
```

The operation of the non recursive method factI( ) should be clear. It uses a loop starting at 1 and progressively multiplies each number by the moving product.

The operation of the recursive factR( ) is a bit more complex. When factR( ) is called with anargument of 1, the method returns 1; otherwise, it returns the product of factR(n–1)*n. To evaluate this expression, factR( ) is called with n–1. This process repeats until n equals 1 and the calls to the method begin returning. For example, when the factorial of 2 is calculated, the first call to factR( ) will cause a second call to be made with an argument of 1. This call will return 1, which is then multiplied by 2 (the original value of n). The answer is then 2.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. A recursive call does not make a new copy of the method. Only the arguments are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.

6.22 Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are

created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called by the JVM when your program begins. Outside the class, to use a static member, you need only specify the name of its class followed by the dot operator. No object needs to be created. For example, if you want to assign the value 10 to a static variable called count that is part of the Timer class, use this line:

Timer.count = 10;

This format is similar to that used to access normal instance variables through an object, except that the class name is used. A static method can be called in the same way by use of the dot operator on the name of the class.

Variables declared as static are, essentially, global variables. When an object is declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable. Here is an example that shows the differences between a static variable and an instance variable:

```
//Use a static variable.
class StaticDecmo
{
    int x;  // a normal instance variable
    static int y;  // a static variable
    //Return the sum of the instance variable x
   // and the static variable y.
   int sum ( ) {
            return x + y;
    }
}
class Sdemo  {
      public static void main (String args[ ] ) {
           StaticDemo ob1 = new StaticDemo ( );
            StaticDemo ob2 = new StaticDemo ( );
           // Each object has its own copy of an instance variable.
           ob1.x =  10;
           ob2. x= 20;
           System.out.println( “Of course, ob1.x  and ob2.x “ +  “are independent. “ );
           System.out.prinln( “ob1.x: “ + ob1.x + “\nob2.x: “ + ob2.x);
           System.out.printl( );
          // each object share one copy of a static variable.
          System.out.println (“The static variable y is shared. “);
         StaticDemo.y = 19;
         System.out.println (“Set StaticDemo.y to 19. “ );
         System.out.println(“ob1.sum( ) : “ + ob1.sum( ));
         System.out.println(“ob2.sum( ) : “ + ob2.sum( ));
         System.out.println( );
         StaticDemo. y = 100;
         System.out.println (“Change StaticDemo.y to 100 “ );
         System.out.println(“ob1.sum( ) : “ + ob1.sum( ));
         System.out.println(“ob2.sum( ) : “ + ob2.sum( ));
         System.out.println( );
```

```
    }
  }
```

The output of the program is shown here
Of course, ob1.x and ob2.x " + "are independent.
ob1.x =10;
ob2.x = 20;
The static variable y is shared.
Set staticDemo.y to 19.
ob1.sum( ) : 29
ob2.sum ( ) : 39
change StaticDemo.y to 100
ob1.sum( ) : 110
ob2.sum ( ) : 120

As you can see, the static variable y is shared by both ob1 and ob2. Changing it affects the entire class, not just an instance. The difference between a static method and a normal method is that the static method is called through its class name, without any object of that class being created. Program given bellow Defining and using static members class Math operation

```
class Mathoperation
{
    static float mul(float x, float y)
       {
          return x*y;
    static float divide(float x, float y)
      {
          return x/y ;
      }
}
class MathApplication
   {
        public void static main(string args[ ])
          {
              float a = MathOperation.mul(4.0,5.0);
              float b = MathOperation.divide(a,2.0);
              System.out.println("b = "+ b) ;
          }
    }
```

Output of Above Program is:
b = 10.0

6.22.1 Static Block

Sometimes a class will require some type of initialization before it is ready to create objects. For example, it might need to establish a connection to a remote site. It also might need to initialize certain static variables before any of the class static methods are used. To handle these types of situations Java allows you to declare a static block. A static block is

executed when the class is first loaded. Thus, it is executed before the class can be used for any other purpose. Here is an example of a static block:

```
// use static block
class StaticBlock {
   static double rootOf2;
   static double rootOf3;
   static {
        System.out.println (“Inside static block.”);
        rootOf2 = Math.sqrt (2.0);
        rootOf3 =  Math.sqrt (3.0);
     }
  StaticBlock(String msg) {
          System.out.println (msg);
   }
}
  class Sdemo3 {
      public static void main (String args[ ]) {
           StaticBlock ob = new StaticBlock (“Inside Constructor”);
           System.out.println (“Square root  of 2 is” + StaticBlock.rootOf2);
           System.out.println (“Square root  of 3 is” + StaticBlock.rootOf3);
      }
}
```

The output is shown here

```
Inside static block.
Inside constructor
Square root of 2 is 1.4142135623730951
Square root of 3 is 1.7320508075688772
```

The static block is executed before any objects are constructed.

6.23 Introducing Nested and Inner Classes

In Java you can define a nested class. This is a class that is declared within another class. A nested class does not exist independently of its enclosing class. Thus, the scope of a nested class is bounded by its outer class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block. There are two general types of nested classes: those that are preceded by the static modifier and those that are not. The only type that we are concerned about is the non-static variety. This type of nested class is also called an inner class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Sometimes an inner class is used to provide a set of services that is used only by its enclosing class. Here is an example that uses an inner class to compute various values for its enclosing class:

```
/ / Use an inner class.
class Outer {
   int nums[ ];
```

```
Outer (int n [ ] ) {
    nums = n;
}
void analyze ( ) {
   Inner inOb = new Inner ( );
   System.out.println ( “Minimum: “ +inOb.min ( ) );
   System.out.println ( “Maximum: “ +inOb.max ( ) );
   System.out.println ( “Average: “ +inOb.avg ( ) );
}
/ / This is an inner class.
 class Inner  {
       int min ( ) {
           int m = nums [0];
           for (int i=1; i<nums.length; i++)
               if (nums [i] < m) m= nums[i];
           return m;
      }
        int max ( ) {
           int m = nums [0];
           for (int i=1; i<nums.length; i++)
               if (nums [i] > m) m= nums[i];
           return m;
       }
       int avg ( ) {
           int a = 0;
           for (int i=1; i<nums.length; i++)
               a += nums [i];
           return a/ nums.length;
       }
   }
}

Class NestedClassDemo {
     public static void main (String args [ ] ) {
          int x[ ] = { 3, 2, 1, 5, 6, 9, 7, 8 };
          Outer outOb = new Outer (x);
          outOb. Analyze ( );
      }
}
```

The output of the program is shown here:

```
Minimum: 1
Maximum: 9
Average: 5
```

In this example, the inner class Inner computes various values from the array nums, which is a member of Outer. As explained, an inner class has access to the members of its enclosing class, so it is perfectly acceptable for Inner to access the nums array directly. Of course, the opposite is

not true. For example, it would not be possible for analyze( ) to invoke the min( ) method directly, without creating an Inner object.

6.24 Variable Length Arguments

Sometimes you will want to create a method that takes a variable number of arguments, based on its precise usage. For example, a method that opens an Internet connection might take a user name, password, file name, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. To create such a method implies that there must be some way to create a list of arguments that is variable in length, rather than fixed.

A variable-length argument is specified by three periods (...). For example, here is how to write a method called vaTest( ) that takes a variable number of arguments:

```
class VariableTest
{
/ / vaTest ( ) uses a vararg.
 static void vaTest (int . . . v) {
     System.out.println(Number of args: " + v.length);
     System.out.println("Contents: ");
     for (int i=0; i < v.length; i++)
         System.out.println ("arg " + i + ": "  + v[i]);
     System.out.println();
        }
    public static void main main (String args [ ] ) {
        / / Notice how vaTest ( ) can be called with a
       / /  variable number of arguments.
       vaTest (10 ) ;          / / 1 arg
       vaTest (1, 2, 3);      / /  3 args
       vaTest ( );             / /   no args
   }
}
```

The output of the program is shown here:

```
   Number of args: 1
   Contents :
      arg 0: 10
   Number of args : 3
   Contents :
      args 0: 1
    args 1: 2
    args 2: 3
Number of args: 0
   Contents :
```

Notice that v is declared as shown here:

```
int . . . v
```

This syntax tells the compiler that vaTest( ) can be called with zero or more arguments. Furthermore, it causes v to be implicitly declared as an array of type int [ ]. Thus, inside vaTest( ) , v is accessed using the normal array syntax.

There are two important things to notice about this program. First, as explained, inside vaTest( ) , v is operated on as an array. This is because v is an array. The ... syntax simply tells the compiler that a variable number of arguments will be used, and that these arguments will be stored in the array referred to by v. Second, in main( ), vaTest( ) is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to v. In the case of no arguments, the length of the array is zero. A method can have "normal" parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method.

6.25 Assignment-6

Short Answer Questions (2 marks each)

1. What is meant by method overloading? What is its use? (M.U. Oct/Nov. 2008, 2011)
2. What are static members?
3. What are classes and objects?
4. What is a class? How does it accomplish data hiding? (M.U. Oct/Nov. 2009)
5. What is the purpose of new operator?
6. What is constructor?
7. Write the syntax to create a class in Java.
8. What is the purpose of a constructor?
9. What is garbage collection?
10. What is the use of finalize() method in Java?
11. When do we declare a member of a class static? (M.U. Oct/Nov. 2008)
12. Write a statement that creates and instantiates an object in Java.
13. Write the necessary statements to instantiate an object called **rect1** which is of type **Rectangle.**
14. What do you mean by parameterized constructor?
15. What is the purpose of this keyword? (M.U. Oct/Nov. 2014)
16. List out different java's access modifiers?
17. Differentiate public and private access modifiers.
18. What is recursion?
19. What are static blocks?
20. What is the purpose of variable length arguments.
21. Mention one advantage and one drawback of recursion.
22. What is the purpose of static keyword?
23. List any two restrictions of static methods. (Hw to access a static method in Java)
24. What is the specialty of static variable? (How to access static variable in Java)

Long Answer Questions

1. What is a class? How does it accomplish data hiding? Explain with example? (5 Marks-M.U. Oct/nov-2010)
2. Explain with syntax and example, how a method is defined in a class? (4 or 5 Marks-M.U. Oct/Nov. 2012, 2009)
3. With an example explain the different types of constructors supported by Java (5 Marks-M.U. Oct/nov-2014)
4. What are objects? How is they created? Explain with an example. (5 Marks-M.U. Oct/Nov. 2012, 2010)
5. Design a class to represent a bank account, include the following members: Name of the depositor, Account number, Balance. Write suitable methods to
    1. Assign initial values
    2. Display the name and balance (5 Marks-M.U. Oct/Nov. 2012)
6. What is the need of using static members? How do you access them? Give example (5 Marks-M.U. Oct/Nov. 2011, 2009)
7. What do you mean by method overloading? Explain. (5 Marks-M.U. Oct/Nov. 2012, 2010)
8. Write a note on nesting of class. (4 Marks)
9. Explain with an example, how constructor can be used to initialize the object of a class? (4/6 Marks-M.U. Oct/Nov. 2012, 2010)
10. How do you define a method? Explain with an example (3 Marks-M.U. Oct/Nov 2008)
11. Explain with example the general format of declaring instance method in java. (5 Marks-M.U. Oct/Nov 2008)
12. What do you mean by instanting an object? With an example declare a class and instantiate the object. (3 Marks-M.U. Oct/Nov 2008)
13. Explain recursion with an example. (5 Marks)
14. Write a short note on this keyword? (4 Marks)
15. What is static block? Explain with an example (5 Marks)
16. Explain variable length arguments with an example (5 Marks-M.U.Oct/Nov.2014)
17. Explain how garbage collection is achieved in Java. (5 marks)
18. Explain the use of finalize() method in Java with suitable example (5 marks)
19. Explain the use of access modifiers with suitable example. (5 marks)
20. Explain how to pass objects to methods with suitable example. (4 marks)
21. Explain how to return an object from a method with suitable example. (5 marks)
22. Explain method overloading with suitable example. (5 marks-M.U. Oct/Nov.2014)
23. Explain constructor overloading with suitable example.(5 marks)
24. Create a class Student with members Register number, name and marks in three subjects. Use constructors to initialize objects and write a method that displays the information of a student.
25. Explain nesting of classes with suitable example.
26. Explain how to pass variable-length arguments with suitable example.

UNIT-II

Chapter 7

Inheritance

7.1 Inheritance Basics

Inheritance is one of the three foundation principles of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the language of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the variables and methods defined by the superclass and add its own, unique elements.

Java supports inheritance by allowing one class to incorporate another class into its declaration. This is done by using the extends keyword. Thus, the subclass adds to (extends) the superclass

A subclass is defined as follows:

```
class subclassname extends superclassname
{
    variables declaration;
    methods declaration;
}
```

The keyword extends signifies that the properties of the superclassname are extended to the subclass name. The subclass will now contain its own variables and methods as well those of the superclass. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

```
 / / A simple class hierarchy.
 / / A class for two-dimensional objects.
 class TwoDShape  {
    double width;
    double height;
 void showDim ( ) {
    System.out.println( “”Width and height are “ + width + “ and “ +height);
 }
}

/ / A subclass of TwoDshape for triangles.
 class Triangle extends TwoDshape {
    String style;
    double area ( ) {
        return width * height / 2;
    }
    void showStyle ( ) {
         system.out.println(“Triangle is “ + style);
   }
 }

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

```
        Triangle t1 = new Triangle ( );
        Traingle t2 = new Traingle ( );
         t1.width = 4.0;
         t1.height = 4.0;
         t1. Style = "filled";
         t2.width = 8.0;
         t2.height = 12.0;
         t2. Style = "outlined";
        System.out.println("Info for t1: ");
        t1.showStyle ( );
        t1.showDim ( );
        System.out.println("Area is " + t1.area ( ));
        System.out.println ( );
        System.out.println("Info for t2: ");
        t2.showStyle ( );
        t2.showDim ( );
        System.out.println("Area is " + t2.area ( ));
    }
}
```

The output of the program shown here

```
Info for t1
Triangle is filled
Width and Height are 4.0 and 4.0
Area is 8.0

Info for t2
Triangle is outlined
Width and Height are 8.0 and 12.0
Area is 48.0
```

Here, TwoDShape defines the attributes of a "generic" two-dimensional shape, such as a square, rectangle, triangle, and so on. The Triangle class creates a specific type of TwoDShape, in this case, a triangle. The Triangle class includes all of TwoDObject and adds the field style, the method area( ), and the method showStyle( ). The triangle's style is stored in style. This can be any string that describes the triangle, such as "filled", "outlined", "transparent", or even something like "warning symbol", "isosceles", or "rounded". The area( ) method computes and returns the area of the triangle, and showStyle( ) displays the triangle style. Because Triangle includes all of the members of its superclass, TwoDShape, it can access width and height inside area ( ). Also, inside main( ), objects t1 and t2 can refer to width and height directly, as if they were part of Triangle.

Even though TwoDShape is a superclass for Triangle, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. For example, the following is perfectly valid:

```
        TwoDShape shape = new TwoDShape ( );
        shape.width = 10;
        shape.showDim ( );
```

A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.

7.2 Member Access and Inheritance

Often an instance variable of a class will be declared private to prevent its unauthorized use or tampering. Inheriting a class does not overrule the private access restriction. Thus, even though a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared private. For example, if, as shown here, width and height are made private in TwoDShape, then Triangle will not be able to access them:

```
 / / Private members are not inherited.
 / /  This example will not compile.
 / / A class for two-dimensional objects.
 class TwoDShape  {
     private double width;
     private double height;
 void showDim ( ) {
     System.out.println( “”Width and height are “ + width + “ and “ +height);
  }
}

/ / A subclass of TwoDshape for triangles.
 class Triangle extends TwoDshape {
     String style;
     double area ( ) {
         return width * height / 2;   / / Error! Can’t access
     }
     void showStyle ( ) {
          system.out.println(“Triangle is “ + style);
    }
 }
```

The Triangle class will not compile because the reference to width and height inside the area( ) method causes an access violation. Since width and height are declared private, they are accessible only by other members of their own class. Subclasses have no access to them. A class member that has been declared private will remain private to its class. It is not accessible by any code outside its class, including subclasses. Java programmers typically use accessor methods to provide access to the private members of a class. Here is a rewrite of the TwoDShape and Triangle classes that uses methods to access the private instance variables width and height:

```
 / / Use accessor methods to set and get private members
 / / A class for two-dimensional objects.
 class TwoDShape  {
     private double width;
     private double height;
 / / Accessor methods for width and height
   double getWidth( ) { return width; }
   double getHeight( ) { return height; }
```

```
   void setWidth(double w ) { width=w; }
   void setHeight(double h ) { height=h; }
  void showDim ( ) {
     System.out.println( “”Width and height are “ + width + “ and “ +height);
  }
}

/ / A subclass of TwoDshape for triangles.
 class Triangle extends TwoDshape {
     String style;
     double area ( ) {
         return getWidth ( ) * getHeight ( ) / 2;
     }
     void showStyle ( ) {
          system.out.println(“Triangle is “ + style);
    }
 }

class Shapes2 {
    public static void main (String args [ ]) {
          Triangle t1 = new Triangle ( );
          Traingle t2 = new Traingle ( );
           t1.setWidth (4.0);
           t1.setHeight (4.0);
           t1. style = “filled”;
           t2.setWidth (8.0);
           t2.setHeight (12.0);
           t2. Style = “outlined”;
          System.out.println(“Info for t1: “);
          t1.showStyle ( );
          t1.showDim ( );
          System.out.println(“Area is “ + t1.area ( ));
          System.out.println ( );
          System.out.println(“Info for t2: “);
          t2.showStyle ( );
          t2.showDim ( );
          System.out.println(“Area is “ + t2.area ( ));
      }
}
```

Generally if an instance variable is to be used only by methods defined within its class, then it should be made private. If an instance variable must be within certain bounds, then it should be private and made available only through accessor methods. This way, you can prevent invalid values from being assigned.

7.3 Constructors and Inheritance

In a hierarchy, it is possible for both superclasses and subclasses to have their own constructors. The constructor for the superclass constructs the superclass portion of the object, and the

constructor for the subclass constructs the subclass part. This makes sense because the superclass has no knowledge of or access to any element in a subclass. Thus, their construction must be separate. However, in practice, most classes will have explicit constructors. When only the subclass defines a constructor, the process is straightforward: simply construct the subclass object. The superclass portion of the object is constructed automatically using its default constructor. For example, here is a reworked version of Triangle that defines a constructor. It also makes style private, since it is now set by the constructor.

```
  / / add a constructor to Triangle.
 / / A class for two-dimensional objects.
 class TwoDShape  {
     private double width;
     private double height;
 / / Accessor methods for width and height
   double getWidth( ) { return width; }
   double getHeight( ) { return height; }
   void setWidth(double w ) { width=w; }
   void setHeight(double h ) { height=h; }
 void showDim ( ) {
     System.out.println( “”Width and height are “ + width + “ and “ +height);
  }
}

/ / A subclass of TwoDshape for triangle.
 class Triangle extends TwoDshape {
     private String style;
    / / Constructor
    Triangle (String s, double w, double h) {
         setWidth(w);
         setHeight(h);
         style (s);
    }
     double area ( ) {
         return getWidth ( ) * getHeight ( ) / 2;
     }
     void showStyle ( ) {
          system.out.println(“Triangle is “ + style);
    }
 }

class Shapes2 {
    public static void main (String args [ ]) {
          Triangle t1 = new Triangle ( “filled”, 4.0, 4.0);
          Traingle t2 = new Traingle ( “outlined”, 8.0, 12.0);
          System.out.println(“Info for t1: “);
          t1.showStyle ( );
          t1.showDim ( );
          System.out.println(“Area is “ + t1.area ( ));
```

```
        System.out.println ( );
        System.out.println("Info for t2: ");
        t2.showStyle ( );
        t2.showDim ( );
        System.out.println("Area is " + t2.area ( ));
    }
}
```

Here, Triangle's constructor initializes the members of TwoDClass that it inherits along with its own style field. When both the superclass and the subclass define constructors, the process is a bit more complicated because both the superclass and subclass constructors must be executed. In this case you must use another of Java's keywords, super, which has two general forms. The first calls a superclass constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

7.4 Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of super:

```
super(parameter-list);
```

Here, parameter-list specifies any parameters needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass constructor. To see how super( ) is used, consider the version of TwoDShape in the following program. It defines a constructor that initializes width and height.

```
 / / add a constructor to TwoDShape
 class TwoDShape  {
    private double width;
    private double height;
 / / parameterized constructor
   TwoDShape (double w, double h )  {
        width=w;
        height=h;
    }
 / / Accessor methods for width and height
  double getWidth( ) { return width; }
  double getHeight( ) { return height; }
  void setWidth(double w ) { width=w; }
  void setHeight(double h ) { height=h; }
 void showDim ( ) {
    System.out.println( ""Width and height are " + width + " and " +height);
 }
}

/ / A subclass of TwoDshape for triangle.
class Triangle extends TwoDshape {
    private String style;
   / / Constructor
   Triangle (String s, double w, double h) {
        super(w, h)  / / call superclass constructor
        style (s);
```

```
    }
     double area ( ) {
         return getWidth ( ) * getHeight ( ) / 2;
     }
     void showStyle ( ) {
          system.out.println("Triangle is " + style);
    }
 }

class Shapes4 {
    public static void main (String args [ ]) {
          Triangle t1 = new Triangle ( "filled", 4.0, 4.0);
          Traingle t2 = new Traingle ( "outlined", 8.0, 12.0);
          System.out.println("Info for t1: ");
          t1.showStyle ( );
          t1.showDim ( );
          System.out.println("Area is " + t1.area ( ));
          System.out.println ( );
          System.out.println("Info for t2: ");
          t2.showStyle ( );
          t2.showDim ( );
          System.out.println("Area is " + t2.area ( ));
      }
}
```

Here, Triangle( ) calls super( ) with the parameters w and h. This causes the Two Shape( ) constructor to be called, which initializes width and height using these values. Triangle no longer initializes these values itself. It need only initialize the value unique to it: style. This leaves TwoDShape free to construct its sub object in any manner that it so chooses. Furthermore, TwoDShape can add functionality about which existing subclasses have no knowledge, thus preventing existing code from breaking. Any form of constructor defined by the superclass can be called by super( ). The constructor executed will be the one that matches the arguments. For example, here are expanded versions of both TwoDShape and Triangle that include default constructors and constructors that take one argument:

```
  / / add more constructor to TwoDShape
  class TwoDShape  {
     private double width;
     private double height;
 / / A default constructor
    TwoDShape( )  {
         Width = height = 0.0;
    }
 / / parameterized constructor
    TwoDShape (double w, double h )  {
         width=w;
          height=h;
      }
/ / constructor object with equal width and height
```

```
    TwoDShape (double x )  {
         width=x;
          height=x;
      }

  / / Accessor methods for width and height
    double getWidth( ) { return width; }
    double getHeight( ) { return height; }
    void setWidth(double w ) { width=w; }
    void setHeight(double h ) { height=h; }
  void showDim ( ) {
     System.out.println( “”Width and height are “ + width + “ and “ +height);
  }
}

/ / A subclass of TwoDshape for triangle.
 class Triangle extends TwoDshape {
     private String style;
  / / A default constructor
    Triangle( )  {
         super( );
         style = “none”;
    }

    / / Constructor
    Triangle (String s, double w, double h) {
         super(w, h)  / / call superclass constructor
         style (s);
    }

  / / one argument constructor
    Triangle (double x )  {
         super (x); // call superclass constructor
         style = “filled”;
      }

     double area ( ) {
         return getWidth ( ) * getHeight ( ) / 2;
     }
     void showStyle ( ) {
          system.out.println(“Triangle is “ + style);
    }
 }

class Shapes5 {
    public static void main (String args [ ]) {
         Triangle t1 = new Triangle ( );
```

```
        Traingle t2 = new Traingle ( "outlined", 8.0, 12.0);
        Triangle t3 = new Traingle (4.0);
        t1 = t2;

        System.out.println("Info for t1: ");
        t1.showStyle ( );
        t1.showDim ( );
        System.out.println("Area is " + t1.area ( ));
        System.out.println ( );
        System.out.println("Info for t2: ");
        t2.showStyle ( );
        t2.showDim ( );
        System.out.println("Area is " + t2.area ( ));
        System.out.println ( );
        t3.showStyle ( );
        t3.showDim ( );
        System.out.println("Area is " + t3.area ( ));

    }
}
```

The output of the program shown here

```
Info for t1:
Triangle is outlined
Width and Height are 8.0 and 12.0
Area is 48.0

Info for t2:
Triangle is outlined
Width and Height are 8.0 and 12.0
Area is 48.0

Info for t3:
Triangle is outlined
Width and Height are 8.0 and 4.0
Area is 8.0
```

When a subclass calls super( ), it is calling the constructor of its immediate superclass. Thus, super( ) always refers to the superclass immediately above the calling class. This is true even in a multilevel hierarchy. Also, super( ) must always be the first statement executed inside a subclass constructor.

7.5 Using super to Access Superclass Member

There is a second form of super that acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

```
super.member
```

Here, member can be either a method or an instance variable. This form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
/ / using super to overcome name hiding
class A {
    int i;
}
/ / creates a subclass by extending class A.
class B extends A {
    int i;    / / this i hides the i in A
    B (int a, int b)  {
        super.i =a;    / / i in A
        i = b;    / / i in B
    }
    void show ( ) {
       System.out.println( " i in superclass: " + super.i);
       System.out.println ( " I in subclass: " + i);
     }
  }

class UseSuper {
     public static void main (String args [ ]) {
          B subOb = new B(1, 2);
          subOb.show ( );
    }
}
```

This program displays the following;

```
 i in superclass: 1
 i in subclass: 2
```

Although the instance variable i in B hides the i in A, super allows access to the i defined in the superclass. Super can also be used to call methods that are hidden by a subclass.

7.6 Creating a Multilevel Hierarchy

We can build hierarchies that contain as many layers of inheritance as you like. It is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A.

To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass Triangle is used as a superclass to create the subclass called ColorTriangle. ColorTriangle inherits all of the traits of Triangle and TwoDShape and adds a field called color, which holds the color of the triangle.

```
/ / A multilevel hierarchy.
class TwoDShape  {
   private double width;
```

```
    private double height;
 / / A default constructor
   TwoDShape( )  {
        Width = height = 0.0;
   }
 / / parameterized constructor
   TwoDShape (double w, double h )  {
        width=w;
         height=h;
     }
/ / constructor object with equal width and height
   TwoDShape (double x )  {
        width=x;
         height=x;
     }

 / / Accessor methods for width and height
  double getWidth( ) { return width; }
  double getHeight( ) { return height; }
  void setWidth(double w ) { width=w; }
  void setHeight(double h ) { height=h; }
 void showDim ( ) {
    System.out.println( “”Width and height are “ + width + “ and “ +height);
 }
}
/ / Extend TwoDshape.
class Triangle extends TwoDshape {
    private String style;
 / / A default constructor
   Triangle( )  {
        super( );
        style = “none”;
   }

   / / Constructor
   Triangle (String s, double w, double h) {
        super(w, h)  / / call superclass constructor
        style (s);
   }

 / / one argument constructor
   Triangle (double x )  {
        super (x); // call superclass constructor
        style = “filled”;
     }

    double area ( ) {
```

```
        return getWidth ( ) * getHeight ( ) / 2;
     }
     void showStyle ( ) {
          system.out.println("Triangle is " + style);
    }
 }

/ / Extend triangle
class ColorTriangle extends Triangle {
   private String color;
    ColorTriangle (String c, String s, double w, double h) {
         super (s,w,h);
       color = c;
  }
   String getColor ( ) { return color; }
     void showColor ( ) {
          System.out.println ( "Color is" + color);
     }
  }

class Shapes6 {
    public static void main (String args [ ]) {
         ColorTriangle t1 = new Triangle ("Blue", "outlined", 8.0, 12.0);
         ColorTraingle t2 = new Traingle (("Red", "filled",2.0, 2.0);
         System.out.println("Info for t1: ");
         t1.showStyle ( );
         t1.showDim ( );
         t1.showColor( );
         System.out.println("Area is " + t1.area ( ));
         System.out.println ( );
         System.out.println("Info for t2: ");
         t2.showStyle ( );
         t2.showDim ( );
         t1.showColor( );
         System.out.println("Area is " + t2.area ( ));
     }
}
```

The output of the program shown here
Info for t1:
Triangle is outlined
Width and Height are 8.0 and 12.0
Color is Blue
Area is 48.0

Info for t2:
Triangle is outlined

Width and Height are 2.0 and 2.0
Color is Red
Area is 4.0

Because of inheritance, ColorTriangle can make use of the previously defined classes of Triangle and TwoDShape, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code. This example illustrates one other important point: super( ) always refers to the constructor in the closest superclass. The super( ) in ColorTriangle calls the constructor in Triangle. The super( ) in Triangle calls the constructor in TwoDShape. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters "up the line." This is true whether or not a subclass needs parameters of its own.

In a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default (parameter less) constructor of each superclass will be executed. The following program illustrates when constructors are executed.

7.7 Superclass References and Subclass Objects

Java is a strongly typed language. Aside from the standard conversions and automatic promotions that apply to its primitive types, type compatibility is strictly enforced. Therefore, a reference variable for one class type cannot normally refer to an object of another class type. For example, consider the following program

```
 / / this will not compile.
class X {
     int a;
    X(int i)  { a  = i; }
}
class Y {
    int a;
   Y (int i) { a =  i;}
}
classIncompatibleRef {
      public static void main(String args [ ] ) {
        X x= new X (10);
        X x2;
        Y y = new Y(5);
         x2 = x;    / / Ok, both of same type
         x2= y ;   / / Error, not of same type
   }
}
```

Here, even though class X and class Y are physically the same, it is not possible to assign an X reference to a Y object because they have different types. In general, an object reference variable can refer only to objects of its type.

There is, however, an important exception to Java's strict type enforcement. A reference variable of a superclass can be assigned a reference to an object of any subclass derived from that superclass. In other words, a superclass reference can refer to a subclass object. Here is an example:

```
/ / A super class reference can refer to a subclass object.
class X {
     int a;
    X(int i)  { a  = i; }
}
class Y extends X{
    int b;
   Y (int i, int j) {
         super (j);
         b =  i;
   }
}
classSupSubRef {
      public static void main(String args [ ] ) {
        X x= new X (10);
        X x2;
        Y y = new Y(5, 6);
         x2 = x;    / / Ok, both of same type
         System.out.println( " x2.a: " + x2.a);
         x2= y ;   / / Still ok, because Y is derived from X
        System.out.println( " x2.a: " + x2.a);
       / / X references Know only about X members
       x2.a =19;
       x2.b=27;  // Error, X doesn't have a b member
     }
}
```

Here, Y is now derived from X; thus, it is permissible for x2 to be assigned a reference to a Y object. It is important to understand that it is the type of the reference variable not the type of the object that it refers to that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why x2 can't access b even when it refers to a Y object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it.

An important place where subclass references are assigned to superclass variables is when constructors are called in a class hierarchy. As you know, it is common for a class to define a constructor that takes an object of the class as a parameter. This allows the class to construct a copy of an object. Subclasses of such a class can take advantage of this feature.

7.8 Method Overriding

Method inheritance enables us to define and use methods repeatedly in subclasses without having to define the methods again in subclass. However, there may be occasions when we want

an object to respond to the same method but have different behaviour when that method is called. That means, we should override the method defined in the superclass. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding. Program illustrates the concept of overriding. The method display( ) is overridden.

```
class Super
{
   int x ;
   Super(int x)
    {
        this.x = x ;
  }
void display( )          // method defined
  {
       System.out.println("Super x-" + x);
 }

class Sub extends Super
 {
     int y ;
     Sub(int x, int y)
       {
           super(x) ;
           this.y = y ;
       }
     void display( )                    // method defined again
       {
            System.out.println("Super x =" + x) ;
            System.out.println("Sub y = " + y);
       }
}

class  OverrideTest
    {
         public static void main (String args[ ])
           {
                Sub s1= new Sub(100,200);
                S1. display ( );
           }
    }
```

Output of above Program is:

```
Super x = 100
Sub y = 200
```

7.9 Overridden Methods Support Polymorphism

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
/ / demonstrate dynamic method dispatch.
class Sup  {
   void who ( ) {
        System.out.println ( "who( ) in Sup");
    }
 }
class Sub1 extends Sup {
  {
        void who ( ) {
        System.out.println ( "who( ) in Sub1");
     }
  }
class Sub2 extends Sup {
  {
        void who ( ) {
        System.out.println ( "who( ) in Sub2");
     }
  }
class DynDispDemo  {
     public static void main (String args [ ] ) {
          Sup superOb = new Sup ( );
          Sub1 subOb1= new Sub1( );
          Sub2 subOb2 = new Sub2 ( );
          Sup supRef;
          supRef = superOb;
          supRef.who ( );
          supRef = subOb1;
          supRef.who ( );
          supRef = subOb2;
          supRef.who ( );
  }
```

```
}
```

The output of the program is shown here:

```
who( ) in Sup
who( ) in Sub1
who( ) in Sub2
```

This program creates a superclass called Sup and two subclasses of it, called Sub1 and Sub2. Sup declares a method called who ( ), and the subclasses override it. Inside the main( ) method, objects of type Sup, Sub1, and Sub2 are declared. Also, a reference of type Sup, called supRef, is declared. The program then assigns a reference to each type of object to supRef and uses that reference to call who ( ). As the output shows, the version of who( ) executed is determined by the type of object being referred to at the time of the call, not by the class type of supRef.

7.10 Use of Overridden Methods

Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism. Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy that moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

To better understand the power of method overriding, we will apply it to the TwoDShape class. In the preceding examples, each class derived from TwoDShape defines a method called area( ). This suggests that it might be better to make area( ) part of the TwoDShape class, allowing each subclass to override it, defining how the area is calculated for the type of shape that the class encapsulates.

```
 / / Use dynamic method dispatch.
 class TwoDShape  {
    private double width;
    private double height;
    private String name;
 / / A default constructor
   TwoDShape( )  {
        Width = height = 0.0;
         Name = "none";
   }
 / / parameterized constructor
   TwoDShape (double w, double h, String n )  {
        width=w;
         height=h;
```

```
        name=n;
      }
 / / constructor object with equal width and height
    TwoDShape (double x, String n )  {
        width= height=x;
        name = n;
      }
  / / constructor an object from an object
      TwoDShape ( TwoDShape ob)  {
         width =  ob.width;
         height =  ob.height;
         name =  ob.name;
     }
  / / Accessor methods for width and height
   double getWidth( ) { return width; }
   double getHeight( ) { return height; }
   void setWidth(double w ) { width=w; }
   void setHeight(double h ) { height=h; }
   String getName( ) { return name; }
  void showDim ( ) {
     System.out.println( “”Width and height are “ + width + “ and “ +height);
  }
   double area ( )   {
        system.out.println (“ area ( ) must be overridden );
        return 0.0;
   }
}
/ / A sub class of  TwoDshape for triangles.
 class Triangle extends TwoDshape {
     private String style;
  / / A default constructor
    Triangle( )  {
        super( );
        style = “none”;
    }

    / / Constructor for triangle
    Triangle (String s, double w, double h) {
        super(w, h, “triangle”)  / / call superclass constructor
        style =s;
    }

  / / one argument constructor
    Triangle (double x )  {
        super (x, “triangle”); // call superclass constructor
        style = “filled”;
      }
```

```
   / / constructor an object from an object
     Triangle (Triangle ob) {
          super (ob);
          style=ob.style;
      }
   / / override area ( ) for Triangle
     double area ( ) {
         return getWidth ( ) * getHeight ( ) / 2;
     }
     void showStyle ( ) {
          system.out.println("Triangle is " + style);
    }
 }

/ / A subclass of TwoDShape for rectangle
class Rectangle  extends TwoDShape {
    / / A default constructor
   Rectangle ( ) {
       super ( );
      }
   / / Constructor for Rectangle.
  Rectangle (double w, double h)  {
       super (w, h, "rectangle" );    / /call superclass constructor
   }
   / / construct a square
  Rectangle ( double x) {
    super (x, "rectangle");       / /call superclass constructor
  }
  / / construct an object from an object
  Rectangle (Rectangle ob)  {
     super (ob)  / / pass object to TwoDShape constructor
   }
  double area ( ) {
         return getWidth ( ) * getHeight ( );
     }
}
class DynShapes {
    public static void main (String args [ ]) {
          TwoDShape shapes [ ] = new TwoDShape [5];
          Shapes [0] = new Triangle ("outlined", 8.0, 12.0);
          Shapes [1] = new Rectangle (10);
          Shapes [2] = new Rectangle (10, 4);
          Shapes [3] = new Triangle (7.0);
          Shapes [4] = new TwoDShape(10,20, "generic");
          for (int i=0; i< shapes.length; i++)   {
              System.out.println ("object is " + shapes[i].getName( ));
              System.out.println ("Area is  " + shapes [i]. area( ));
```

```
            System.out.println ( );
    }
}
```

The output of the program shown here

```
Object is triangle
Area is 48.0
Object is rectangle
Area is 100.0
Object is rectangle
Area is 40.0
Object is triangle
Area is 24.5
area ( ) must be overridden
Area is 0.0
```

Let's examine this program closely. First, as explained, area( ) is now part of the TwoDShape class and is overridden by Triangle and Rectangle. Inside TwoDShape, area( ) is given a placeholder implementation that simply informs the user that this method must be overridden by a subclass. Each override of area( ) supplies an implementation that is suitable for the type of object encapsulated by the subclass. Thus, if you were to implement an ellipse class, for example, then area( ) would need to compute the area( ) of an ellipse.

In main( ) that shapes is declared as an array of TwoDShape objects. However, the elements of this array are assigned Triangle, Rectangle, and TwoDShape references. This is valid because, as explained, a superclass reference can refer to a subclass object. The program then cycles through the array, displaying information about each object. Although quite simple, this illustrates the power of both inheritance and method overriding. The type of object referred to by a superclass reference variable is determined at run time and acted on accordingly. If an object is derived from TwoDShape, then its area can be obtained by calling area( ). The interface to this operation is the same no matter what type of shape is being used.

7.11 Using Abstract classes

Sometimes you will want to create a superclass that defines only a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement but does not, itself, provide an implementation of one or more of these methods. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. An abstract method is created by specifying the abstract type modifier. An abstract method contains no body and is, therefore, not implemented by the superclass. Thus, a subclass must override it, it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present. The abstract modifier can be used only on normal methods. It cannot be applied to static methods or to constructors. A class that contains one or more abstract methods must also be declared as abstract by preceding its class declaration with the abstract modifier. Since an abstract class does not define a complete implementation, there can be no objects of an abstract class. Thus, attempting to create an object of an abstract class by

using new will result in a compile-time error. When a subclass inherits an abstract class, it must implement all of the abstract methods in the superclass. If it doesn't, then the subclass must also be specified as abstract. Thus, the abstract attribute is inherited until such time as a complete implementation is achieved.

Example:

```
abstract class Shape
{
    ….. ……
    .. ……
    abstract void draw( };
    ... …. .....
}
```

7.12 Using Final

As powerful and useful as method overriding and inheritance are, sometimes you will want to prevent them. For example, you might have a class that encapsulates control of some hardware device. Further, this class might offer the user the ability to initialize the device, making use of private, proprietary information. In this case, you don't want users of your class to be able to override the initialization method. Whatever the reason, in Java it is easy to prevent a method from being overridden or a class from being inherited by using the keyword final.

Example:

```
final int SIZE = 100;
final void showstatus ( ) {……}

final class Aclass {……..}
final class Bclass extends Someclass {… ……..}
```

Making a method final ensures that the functionality defined in this, method will never be altered in any way. Similarly, the value of a final variable can never be changed. Final variables, behave like class variables and they do not take any space on individual objects of the class. Any attempt to inherit these classes will cause an error and the compiler will not allow it. Declaring a class final prevents any unwanted extensions to the class. It also allows the compiler to perform some optimizations when a method of a final class is invoked.

7.13 The object class

Java defines one special class called Object that is an implicit superclass of all other classes. In other words, all other classes are subclasses of Object. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array. Object defines the following methods, which means that they are available in every object

Table 7.1 Object Methods

Method	Purpose
Object clone( )	Creates a new object that is same as the object being cloned.
boolean equals(Object object)	Determines whether one object is

	equal to another
void finalize( )	Called before an unused object is recycled
Class <? > getClass( )	Obtains the class of an object at run time
int hashCode( )	Returns the hash code associated with the invoking object
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait ( ) void wait (long milliseconds) void wait (long millseconds, int nanoseconds)	Waits on another thread of execution.

The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final. You can override the others. equals( ) and toString( ). The equals( ) method compares two objects. It returns true if the objects are equivalent and false otherwise. The toString( ) method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using println( ). Many classes override this method. The getClass( ) method relates to Java's generics feature. Generics allow the type of data used by a class or method to be specified as a parameter.

7.14 Assignmenmt-7

Short Answer Questions (2 Marks each)

1. What is the purpose of super keyword?
2. What is inheritance? Why it is used. (M.U. Oct/Nov. 2014)
3. What is the purpose of keyword extends?
4. What is the purpose of super()?
5. How to prevent methods from overriding?
6. What is method overriding?
7. What do you mean by finalize( ) method? (M.U. Oct/Nov.2012, 2010)
8. What do you mean by overriding methods? (M.U. Oct/Nov.2012)
9. What is the purpose of final class?
10. Write general syntax of abstract method?
11. What are abstract classes?
12. What are abstract methods?
13. Write syntax for defining a subclass?
14. What do you mean by superclass references?
15. List out the use of overridden methods.
16. How to prevent methods from overriding?
17. What are final methods?

Long Answer Questions

1. Write a note on method overriding. (5 Marks-M.U. Oct/Nov. 2011)
2. Explain the purpose of final variables and methods, final classes and finalize( ) method (6 Marks-M.U. Oct/Nov. 2011)
3. What is inheritance? Explain single inheritance with suitable example code. Also use subconstructors. (6 Marks-M.U. Oct/Nov. 2011)
4. Explain overriding method with an example. (5 Marks-M.U. Oct/Nov. 2010)
5. Explain the single level inheritance with a program example. (5 Marks-M.U. Oct/Nov. 2010, 2014)
6. Create a base class 'Shape' to include shapename and define a constant PI=22/7. From this derive two classes 'Rectangle' and 'Circle'. Calculate the area of both the Shapes. For Rectangle area is Length X breadth and for circle area is PI X $Radius^2$ (4 Marks-M.U. Oct/Nov. 2009)
7. What is inheritance? How does it help us to create a new class? Explain. (5 Marks-M.U. Oct/Nov. 2009)
8. What is an Abstract class? What are its features? Give an example. (5 Marks-M.U. Oct/Nov. 2009, 2014)
9. Explain final classes and methods? (5 Marks, 2014)
10. Create a base class Room with length and breadth as data members. Add a method to calculate area. Write all possible constructors. Create a derived class with additional data members height and method volume. Output area and volume. Make usage of the keyword super. (8 Marks-M.U. Oct/Nov. 2008)
11. Explain with an example using super to call superclass constructor. (5Marks)
12. Explain multilevel inheritance with an example (6 Marks)
13. Explain how to pass variable-length arguments with suitable example.
14. With an example explain superclass reference and subclass objects. (5Marks)
15. Explain the use of overridden methods with an example. (5 Marks)
16. Write a short note on the object class. (5 Marks)
17. Explain how a superclass reference can refer to a subclass object with suitable example.

UNIT-III

Chapter 8

Packages and Interfaces

8.1 Packages

One of the main features of OOP is its ability to reuse the code already created. One way of achieving this is by extending the classes and implementing the interfaces. This is limited to reusing the classes within a program. If we need to use classes from other programs we have to use packages. Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes into packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

For most applications, we will need to use two different sets of classes, one for the internal representation of our program's data, and the other for external presentation purposes. We may have to build our own classes for handling our data and use existing class libraries for designing user interfaces.

Defining packages: All classes in Java belong to some package. When no package statement is specified, the default (or global) package is used. Furthermore, the default package has no name, which makes the default package transparent. This is why you haven't had to worry about packages before now. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define one or more packages for your code.

To create a package, put a package command at the top of a Java source file. The classes declared within that file will then belong to the specified package. Since a package defines a namespace, the names of the classes that you put into the file become part of that package's namespace.

This is the general form of the package statement:

```
package pkg;
```

Here, pkg is the name of the package. For example, the following statement creates a package called Mypack

```
package mypack;
```

Java uses the file system to manage packages, with each package stored in its own directory. For example, the .class files for any classes you declare to be part of mypack must be stored in a directory called mypack. Like the rest of Java, package names are case sensitive. This means that the directory in which a package is stored must be precisely the same as the package

name. Lowercase is often used for package names. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here

```
package pack1.pack2.pack3...packN;
```

Of course, you must create directories that support the package hierarchy that you create.

Finding packages and CLASSPATH: Java run-time system knows where to look for packages has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable. Third, you can use the class path option with java and javac to specify the path to your classes. For example, assuming the following package specification:

```
package mypack;
```

In order for a program to find mypack, one of three things must be true: The program can be executed from a directory immediately above mypack, or CLASSPATH must be set to include the path to mypack, or the classpath option must specify the path to mypack when the program is run via java. To avoid problems, it is best to keep all .java and .class files associated with a package in that package's directory. Also, compile each file from the directory above the package directory.

The listing below shows a package named package1 containing a single class ClassA.

```
package package1;
public class ClassA
  {
        public void displayA( )
        {
           System.out.println("Class A");
        }
  }
```

This source file should be named ClassA.java and stored in the subdirectory packagel as stated earlier.
Now compile this java file. The resultant ClassA.class will be stored in the same subdirectory.
Now consider the listing shown below:

```
import package1.ClassA;
class PackageTest1
{
        public static void main (String args[ ] )
        {
                ClassA objectA = new ClassA( ) ;
                objectA.displayA( )
        }
}
```

This listing shows a simple program that imports the class ClassA from the package packagel. The source file should be saved as PackageTestl.java and then compiled. The source file and the compiled file would be saved in the directory of which package 1 was a subdirectory. Now we can run the program and obtain the results.

During the compilation of PackageTestl.java the compiler checks for the file ClassA.class in the packagel directory for information it needs, but it does not actually include the code from ClassA.class in the file PackageTestl.class. When the PackageTestl program is run, Java looks for the file PackageTestl.class and loads it using something called *class loader*. Now the interpreter knows that it also needs the code in the file ClassA.class and loads it as well.

8.2 Packages and Member Access

The visibility of an element is determined by its access specification, private, public, protected, or default and the package in which it resides. Thus, the visibility of an element is determined by its visibility within a class and its visibility within a package. This multilayered approach to access control supports a rich assortment of access privileges. Table 8.1 summarizes the various access levels.

Access modifier → / Access Location ↓	Public	Protected	default (friendly)	private
Same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Other classes in Same package	Yes	Yes	Yes	No
Subclass in same packages	Yes	Yes	No	No
Non-subclasses in other packages	Yes	No	No	No

Table 8.1 Class Member Access

If a member of a class has no explicit access modifier, then it is visible within its package but not outside its package. Therefore, you will use the default access specification for elements that you want to keep private to a package but public within that package. Members explicitly declared public are visible everywhere, including different classes and different packages. There is no restriction on their use or access. A private member is accessible only to the other members of its class. A private member is unaffected by its membership in a package. A member specified as protected is accessible within its package and to all subclasses, including subclasses in other packages.

Table 8-1 applies only to members of classes. A top-level class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other

code. If a class has default access, it can be accessed only by other code within its same package. Also, a class that is declared public must reside in a file by the same name.

When two classes were in the same package, so there one class can use another class because the default access privilege grants all members of the same package access. For example if Book were inone package and BookDemo were in another, the situation would be different. In this case, access to Book would be denied. To make Book available to other packages, you must declare class name, constructor and methods as public.

```
package package2;
public class ClassB
{
        public int m = 10
        public void displayB( )
        {
                System.out.println("Class B");
                System.out.println("m = " + m);
        }
}
```

The source file and the compiled file of this package are located in the subdirectory package2.

8.3 Understanding Protected Members

The protected modifier creates a member that is accessible within its package and to subclasses in other packages. Thus, a protected member is available for all subclasses to use but is still protected from arbitrary access by code outside its package. To better understand the effects of protected, consider an example.

```
/ / Make the instance variable in book protected
 package bookpack;
public class Book  {
    / / these are now protected
     protected String title;
     protected String author;
     protected int pubDate;
     public Book (String t, String a, int d) {
          title=t;
          author=a;
          pubDate=d;
      }
   public void show ( ) {
         System.out.println(title);
         System.out.println(author);
         System.out.println(pubDate);
         System.out.println();
     }
 }
```

Next, create a subclass of Book, called ExtBook, and a class called ProtectDemo that uses

ExtBook. ExtBook adds a field that stores the name of the publisher and several accessor methods. Both of these classes will be in their own package called bookpackext. They are shown here:

```
/ / Make the instance variables in book protected.
  package bookpack;
  public class Book {
       / / these are now protected
      protected String tiltle
      protected String author;
      protected int pubDate;
      public Book (String t, String a, int d)  {
            title=t;
            author=a;
            pubDate = d;
     }
     public void show() {
          System.out.println(title);
          System.out.println(author);
          System.out.println(pubDate);
          System.out.println( );
      }
 }
```

Next, create a subclass of Book, called ExtBook, and a class called ProtectDemo that uses ExtBook. ExtBook adds a field that stores the name of the publisher and several accessor methods. Both of these classes will be in their own package called bookpackext. They are shown here:

```
 / / demonstrate protected.
 package bookpackext;
 class ExtBook extends bookpack.Book {
       private String publisher;
       public ExtBook(String t, String a, int d, String p) {
            super (t, a, d);
            publisher = p;
      }
      public string getPublisher ( ) {return publisher; }
      public void setPublisher (String p) { publisher = p;}
      / / These are OK because subclass can access a protected member.
     public String getTitle( ) {return title; }
     public void setTitle(String t ) {title = t; }
     public String getAuthor( ) {return author; }
     public void  setAuthor( String a) {author=a; }
     public int getPubDate( ) {return pubDate; }
     public void  setPubDate( int d)   {pubDate=d; }
}

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

```
        ExtBook books[ ] = new  ExtBook [5];
        book [0 ]=  new ExtBook ( “Java: A Beginner’s Guide”, “Schildt”, 2011, “McGraw-
                                    Hill”);
        book [1]=  new ExtBook ( “Java: The Complete Reference”, “Schildt”, 2011,
                                   “McGraw-Hill”);
        book [2]=  new ExtBook ( “TheArt of java”, “Schildt and Holmes”, 2003,
                                   “Osborne/McGraw-Hill”);
        book [3]=  new ExtBook ( “Red Storm Rising”, “Clancy”, 1986, “Putnam” );
        book [4]=  new ExtBook ( “On the Road”, “kerouac”, 1955, “Viking” );

        / / Find book by author
       System.out.println(“showing all book by Schildt. “ );
       for(int i=0; i<books.length; i++)
           if(book[i].getAuthor ( ) = = “Schildt” )
               System.out.println(books[i]. getTitle ( ) );
        / / books [0]. Title = “test title “;          / / Error-not accessible
    }
}
```

In above example ExtBook extends Book, it has access to the protected members of Book, even though ExtBook is in a different package. Thus, it can access title, author, and pubDate directly; as it does in the accessor methods it creates for those variables. However, in ProtectDemo, access to these variables is denied because ProtectDemo is not a subclass of Book

8.4 Importing packages

Using import you can bring one or more members of a package into view. This allows you to use those members directly, without explicit package qualification.

```
import package name. class name;
or
import packagename.*;
```

These are known as ***import statements*** and must appear at the top of the file, before any class declarations. The first statement allows the specified class in the specified package to be imported.

```
/ / demonstrate import.
 package Bookpackext;
 import bookpack.*;
 / / Use the Book class from bookpack.
 class UseBook {
     public static void main(String args [ ])  {
        ExtBook books[ ] = new  ExtBook [5];
        book [0 ]=  new ExtBook ( “Java: A Beginner’s Guide”, “Schildt”, 2011, “McGraw-
                                    Hill”);
        book [1]=  new ExtBook ( “Java: The Complete Reference”, “Schildt”, 2011,
                                   “McGraw-Hill”);
        book [2]=  new ExtBook ( “TheArt of java”, “Schildt and Holmes”, 2003,
                                   “Osborne/McGraw-Hill”);
        book [3]=  new ExtBook ( “Red Storm Rising”, “Clancy”, 1986, “Putnam” );
```

```
        book [4]=  new ExtBook ( “On the Road”, “kerouac”, 1955, “Viking” );
        for(int i=0; i<books.length; i++) book[i]. show( );
    }
}
```

8.5 Java’s Standard Packages

Java defines a large number of standard classes that are available to all programs. This class library is often referred to as the Java API (Application Programming Interface). The Java API is stored in packages. At the top of the package hierarchy is java. Descending from java are several subpackages, including these;

Package Name	Contents
java.lang	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, date, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets.

Table 8.2 Java System Packages and Their Classes

8.6 Interfaces

Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass. For instance, a definition like

```
class A extends B extends C
{

}
```

A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes. Java provides an alternate approach known as *interfaces* to support the concept of multiple inheritance. Although a Java class cannot be a subclass of more than one superclass, it can *implement* more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

Defining interfaces: An interface is basically a kind of class. Like classes, interfaces contains methods and variables but with a major difference. The difference is that interfaces define only

abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constant
Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods. The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is

```
interface InterfaceName
{
    variables declaration;
    methods declaration;
}
```

Here, interface is the key word and *InterfaceName* is any valid Java variable Gust like class names). Variables are declared as follows

static final *type VariableName = Value;*
Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example

return-type *methodNamel* (parameter_list);

Here is an example of an interface definition that contains two variables and one method

```
interface Item
{
        static final int code = 1001;
        static final String name = “Fan";
        void display ( ) ;
}
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon.
The class that implements this interface must define the code for the method. Another example of an interface is:

```
interface Area
{
        final static float pi = 3.142F;
        float compute (float X, float y);
        void show ( );
}
```

8.7 Implementing interfaces
Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows

```
class classname implements Interfacename
{
body of classname
}
```

Here the class *classname* "implements" the interface *interfacename. A* more general form of implementation may look like this:

class classname extends superclass implements interfacel,interface2, ...
{
body of classname
}

This shows that a class can extend another class while implementing interfaces. When a class implements more than one interface, they are separated by a comma.
In this program, first we create an interface Area and implement the same in tw6 different classes, Rectangle and Circle. We create an instance of each class using the new operator. Then we declare an object of type Area, the interface class. Now, we assign the reference to the Rectangle object rect to area. When we call the compute method of area, the compute method of Rectangle class is invoked. We repeat the same thing with the Circle object.

```
//InterfaceTest.java
interface Area                  //Interface defined
{
        final static float pi = 3.14F;
        float compute (float x, float y);
}
class Rectangle implements Area
{
        // Interface implemented
        public float compute (float x, float y)
        {
              return (x*y);
        }
}

class Circle implements Area
{
        // Another implementation
        public float compute (float x, float y)
        {
                return (pi*x*x);
        }
}
class InterfaceTest
{
        public static void main (String args[ ])
        {
                Rectangle rect = new Rectangle( );
                Circle cir = new Circle( );
                Area area;      //Interface object
                area = rect;
```

```
                System.out.println("Area of Rectangle = "+ area.compute(10,20));
                area = cir;
                System.out.println("Area of Circle = " + area.compute(10,10));
        }
}
```

8.8 Using Interface References

You can create an interface reference variable. Such a variable can refer to any object that implements its interface. When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed. This process is similar to using a superclass reference to access a subclass object. The following example illustrates this process. It uses the same interface reference variable to call methods on objects of both ByTwos and ByThrees.

```
 / / demonstrate interface references.
  class ByTwos implements Series {
        int start;
        int val;
        ByTwos ( )  {
            start=0;
            val=0;
         }
       public int getnext ( ) {
            val +=2;
            return val;
        }
      public void reset ( ) {
            val=start;
      }
     public void setStart ( int x) {
          start = x;
          val = x;
     }
  }

  class ByThrees implements Series {
        int start;
        int val;
        ByThrees ( )  {
            start=0;
            val=0;
         }
       public int getnext ( ) {
            val +=3;
            return val;
        }
      public void reset ( ) {
```

```
            val=start;
       }
      public void setStart ( int x) {
          start = x;
          val = x;
      }
  }
class SeriesDemo  {
    public static void main (String args[ ] ) {
        ByTwos twoOb = new ByTwos ( );
        ByThrees threeOb = new ByThrees ( );
        Series ob;
        for (int i=0; i < 5; i++) {
             ob= twoOb;
             System.out.println("Next ByTwos value is " + ob. getNext ( ) );
             Ob = threeOb;
             System.out.println("Next ByTwos value is " + ob. getNext ( ) );
       }
    }
}
```

In main ( ), ob is declared to be a reference to a Series interface. This means that it can be used to store references to any object that implements Series. In this case, it is used to refer to twoOb and threeOb, which are objects of type ByTwos and ByThrees, respectively, which both implement Series. An interface reference variable has knowledge only of the methods declared by its interface declaration. Thus, ob could not be used to access any other variables or methods that might be supported by the object.

8.9 Variables in Interfaces

To define a set of shared constants, create an interface that contains only these constants, without any methods. Each file that needs access to the constants simply "implements" the interface. This brings the constants into view. Variables are declared as follows

static final *type VariableName = Value;*

Note that all variables are declared as constants.

Here is an example of an interface definition that contains two variables and one method

```
interface Item
{
        static final int code = 1001;
        static final String name = "Fan";
        void display ( ) ;
}
```

By default all the variables declared inside interface are static final. So we can re-write above interface as

```
interface Item
{
        int code = 1001;
        String name = "Fan";
```

```
        void display ( ) ;
}
```

8.10 Extending interfaces

Like classes, interfaces can also be extended. That is, an interface can be sub interfaced from other interfaces. The new sub interface will inherit all the members of the super interface in the manner similar to subclasses. This is achieved using the keyword extends as shown below:

```
interface name2 extends name1
{
        body of name2
}
```

For example, we can put all the constants in one interface and the methods in the other. This will enable us to use the constants in classes where the methods are not required.

```
interface ItemConstants
{
        int code = 1001;
        string name = "Fan";
}
interface Item extends ItemConstants
{
        void display ( );
}

interface ItemConstants
{
        int code = 1001;
        String name = "Fan";
}
interface ItemMethods
{
        void display( );
}
interface Item extends ItemConstants, ItemMethods
{
…….
…….
}
```

While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in the superinterfaces. After all, subinterfaces are still interfaces, not classes. Instead, it is the responsibility of any class that implements the derived interface to define all the methods. Note that when an interface extends two or more interfaces, they are separated by commas. It is important to remember that an interface cannot extend classes. This would violate the rule that an interface can have only abstract methods and constants.

8.11 Assignemt-8

Short Answer Questions (2 Marks Each)

1. What is a package? Name any two advantages of using package. (M.U. Oct/Nov. 2014)
2. How to define a package?
3. What is namespace?
4. How to import a package?
5. How to import a specific class of a package?
6. List any four API packages of Java.
7. Explain any two java standard packages. (M.U. Oct/Nov. 2009)
8. How to hide packages?
9. What is an interface?
10. Write the syntax for defining interface. (M.U. Oct/Nov. 2008, 2010)
11. How do you extend one interface with other? Give an example. (M.U. Oct/Nov. 2009)
12. Write the general syntax of implementing interface
13. What is interface? Why is it needed? (M.U. Oct/Nov.. 2012)
14. What is the purpose of protected members?

Long Answer Questions

1. What are packages? What are the benefits of designing packages? (5 Marks-M.U. Oct/Nov. 2008, 2009, 2011)
2. How a package is created? Explain with an example. (5 Marks-M.U. Oct/Nov. 2010, 2011)
3. With an example explain how to create and importing the user defined package. (6 marks-M.U. Oct/Nov. 2014)
4. Design a package program to contain the class 'Add' to add two integers. Write a java program to include and use this package. (5 Marks- M.U. Oct/Nov. 2009)
5. What is an interface? How does it help the programmer to implement multiple inheritance. Explain with a stand alone program (10 Marks- M.U. Oct/Nov. 2008)
6. Give the general syntax for defining and implementing an interface. Explain with suitable example. (7 Marks-M.U. Oct/Nov. 2011)
7. How do you define and access member of an interface? Explain with an example (5 Marks- M.U. Oct/Nov. 2009)
8. Explain using interface reference with an example. (5 Marks)
9. Write a short note on variables in interfaces (4 Marks)
10. Explain protected members with an example (5 Marks)

UNIT-III

Chapter 9

Exception Handling

9.1 The Exception Hierarchy

In Java, all exceptions are represented by classes. All exception classes are derived from a class called Throwable. Thus, when an exception occurs in a program, an object of some type of exception class is generated. There are two direct subclasses of throwable: Exception and Error. Exceptions of type Error are related to errors that occur in the Java virtual machine itself, and not in your program. These types of exceptions are beyond your control, and your program will not usually deal with them. Thus, these types of exceptions are not described here. Errors that result from program activity are represented by subclasses of Exception. For example, divide-by-zero, array boundary, and file errors fall into this category. In general, your program should handle exceptions of these types. An important subclass of Exception is RuntimeException, which is used to represent various common types of run-time errors.

9.2 Exception Handling Fundamentals

An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it(i.e., informs us that an error has occurred). If the exception object is not caught and handled properly, the interpreter will display an error message and will terminate the program.

If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling*

The purpose of exception handling mechanism is to provide a means to detect and report an exceptional circumstance so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

1. Find the problem (Hit the exception).
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions. When writing programs, we must always be on the lookout for places in the program where an exception could be generated.

9.3 try and catch

Syntax of exception handling code

The basic concepts of exception handling are *throwing* an exception and catching it.

Java uses a keyword try to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword catch "catches" the exception "thrown" by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple try and catch statements:

……
……
try
{

```
        statement;    //generates an exception
}
catch (Exception-type e)
{
        statement;    //processes the exception
}
……
……
```

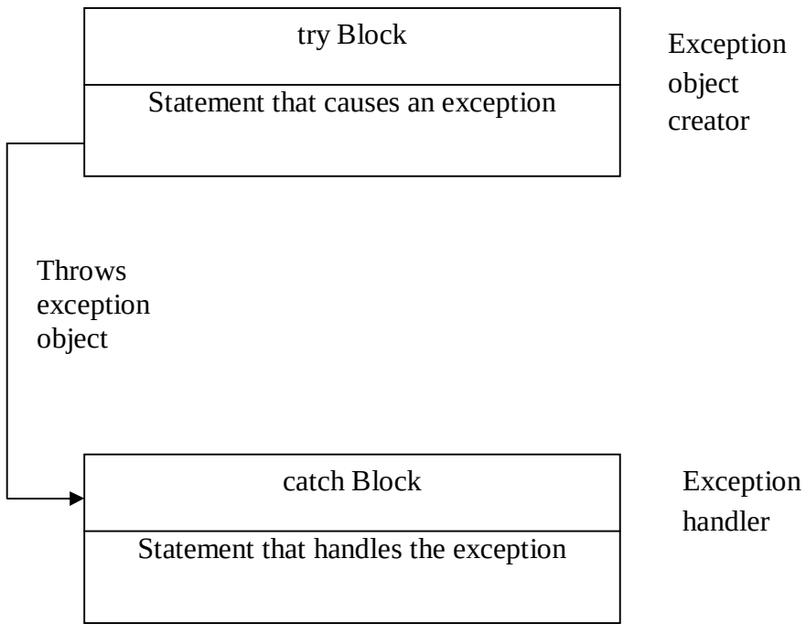

Figure 9.1 Exception handling

The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block. The catch block too can have one or more statements that are necessary to process the exception. Remember that (every try statement should be followed by *at least one* catch statement; otherwise compilation error will occur.). Catch statement works like a method definition. The catch statement is passed a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

Using* try *and* catch *for* exception *handling

```
class Error3
{
        public static void main (String args[ ])
        {
                int a - 10;
                int b = 5;
```

```
                int c = 5;
                int x, y ;
                try
                {
                        x= a / (b-c);     //Exception here
                }
                catch (Arithmetic Exception e)
                {
                        System.out.println("Division by zero");
                }
                y = a / (b+c);
                System.out.println("y   =" + y);
        }
}
```

Program displays the following output:

Division by zero
y=1

The program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and then continues the execution, as if nothing has happened.

9.4 The Consequences of an Uncaught Exception

Catching one of Java's standard exceptions, as the preceding program does, has a side benefit: It prevents abnormal program termination. When an exception is thrown, it must be caught by some piece of code, somewhere. In general, if your program does not catch an exception, then it will be caught by the JVM. The trouble is that the JVM's default exception handler terminates execution and displays a stack trace and error message. For example, the index out-of-bounds exception is not caught by the program.

```
 / / Let JVM handle the error.
 class Nothandled {
      public static void main (String args[ ] ) {
      int nums[ ] = new int [4];
      System.out.println("Before exception is generated.");
      / / generate an index out-of-bounds exception
      nums[7] =10;
  }
}
```

When the array index error occurs, execution is halted, and the following error message is displayed.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
                At NotHandled.main (NotHandled.java: 9)
```

While such a message is useful for you while debugging, this is why it is important for your program to handle exceptions itself, rather than rely upon the JVM. As mentioned earlier, the

type of the exception must match the type specified in a catch statement.If it doesn't, the exception won't be caught.

9.5 Using multiple catch statements

It is possible to have more than one catch statement in the catch block as illustrated below:

```
……..
……..
try
{
        statement;      //generates an exception
}
catch (Exgeption-Type-1 e)
{
        statement;      //processes exception type 1
}
catch (Exgeption-Type-2 e)
{
        statement;      //processes exception type 2
}
.
.
.
catch (Exgeption-Type-N e)
{
        statement;      //processes exception type N
}
……..
……..
}
```

When an exception in a try block is generated, the Java treats the multiple catch statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will skipped. Note that Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

Example: catch (Exception e) ;

The catch statement simply ends with a semicolon, which does nothing. This statement will catch an exception and then ignore it.

Using *multiple catch blocks*

```
class Error4
{
        public static void main (String args[ ])
        {
                int a[ ] = {5, 10};
                int b = 5;
        try
```

```
        {
                int x = a[2] / b – a[1];
        }
        catch (ArithmeticException  e)
        {
                Systern.out.println("Division by zero");
        }
        catch (ArraylndexOutOfBoundsException e)
        {
                Systern.out.println("Array index error");
        }
        catch(ArrayStoreException e)
        {
                Systern.out.println("Wrong data type");
        }
        int y = a[l] / a[0];
        Systern.out.println("y = " + y);
        }
}
```

uses a chain of catch blocks and, when run, produces the following output:

```
Array index error
y = 2
```

Note that the array element a[2] does not exist because array a is defined to have only two elements, a[0] and a[l]. Therefore, the index 2 is outside the array boundary thus causing the block. Catch(ArraylndexOutOfBoundsException e) to catch and handle the error. Remaining catch blocks are skipped.

9.6 Catching Subclass Exceptions

There is one important point about multiple catch statements that relates to subclasses. A catch clause for a superclass will also match any of its subclasses. For example, since the superclass of all exceptions is Throwable, to catch all possible exceptions, catch Throwable. If you want to catch exceptions of both a superclass type and a subclass type, put the subclass first in the catch sequence. If you don't, then the superclass catch will also catch all derived classes. This rule is self-enforcing because putting the superclass first causes unreachable code to be created, since the subclass catch clause can never execute. In Java, unreachable code is an error. For example, consider the following program:

```
/ / Subclasses must precede superclasses in each statements.
class ExcDemo {
    public static void main (String args [ ] ) {
        / / Here, number is longer than denom.
        int numer [ ] = {4, 8, 16, 32, 64, 128, 256, 512} ;
        int denom [ ] = {2, 0, 4, 4, 0, 8};
        for (int i=0; i<number.lengthh; i++) {
          try {
            System.out.println (numer [i] + " / " + denom [i] + "is " + numer[i]/denom[i]);
```

```
            }
          catch (ArrayIndexOutOfBoundsException exc) {
            / / catch the exception
            System.out.println ("No matching element found. ");
          }
          catch (Throwable exc) {
             system.out.println("Some exception occurred. " );
          }
        }
    }
 }
```

The output of the program is shown here:

```
4 / 2  is  2
Some exception occurred.
16 / 4  is  4
32 / 4  is  8
Some exception occurred.
128 / 8 is 16
No matching element found.
No matching element found.
```

9.7 Try Blocks Can Be Nested

One try block can be nested within another. An exception generated within the inner try block that is not caught by a catch associated with that try is propagated to the outer try block. For example, here the ArrayIndexOutOfBoundsException is not caught by the inner catch, but by the outer catch.

```
/ / Use a nested try block.
 class NestTrys {
      public static void main (String args [ ] ) {
           / / Here, number is longer than denom.
          int numer [ ] = {4. 8, 16, 32, 64, 128, 256, 512} ;
          int denom [ ] = {2, 0, 4, 4, 0, 8};
          for (int i=0; i<numer.lengthh; i++) {
            try {   // outer try
               for (int i=0; i<numer.length; i++) {
                  try {
                     System.out.println (numer [i] + " / " + denom [i] + "is " +
                                          numer[i]/denom[i]);
                  }
             catch (ArithmeticException exc) {
              / / catch the exception
              System.out.println ("Can't divide by Zero!");
            }
          }
      }
          catch (ArrayIndexOutOfBoundsException exc) {
```

```
            / / catch the exception
            System.out.println ("No matching element found. ");
            System.out.println ("Fatal error-program terminated.");
          }
        }
    }
```

The output of the program is shown here:

```
4 / 2  is  2
Some exception occurred.
16 / 4  is  4
32 / 4  is  8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error-program terminated.
```

In this example, an exception that can be handled by the inner try in this case, a divide-by-zero error allows the program to continue. However, an array boundary error is caught by the outer try, which causes the program to terminate. Although certainly not the only reason for nested try statements, the preceding program makes an important point that can be generalized. Often nested try blocks are used to allow different categories of errors to be handled in different ways. Some types of errors are catastrophic and cannot be fixed. Some are minor and can be handled immediately. Many programmers use an outer try block to catch the most severe errors, allowing inner try blocks to handle less serious ones.

9.8 Throwing an Exception

It is possible to manually throw an exception by using the throw statement. Its general form is shown here:

```
throw exceptOb;
```

Here, exceptOb must be an object of an exception class derived from Throwable. Here is an example that illustrates the throw statement by manually throwing an Exception.

Arithmetic Exception:

```
/ / manually throw an exception.
class ThrowDemo {
    public static void main (String args [ ]) {
        try {
            System.out.println ("Before throw. " );
            Throw new ArithmeticException ( );
            }
            catch (ArithmeticException exc) {
                / /catch the exception
               System.out.println ("Exception caught. ");
             }
            System.out.println ("After try/catch block.");
          }
        }
```

The output from the program is shown here:

```
Before throw.
Exception caught.
After try/catch block.
```

Notice how the ArithmeticException was created using new in the throw statement. Remember, throw throws an object. Thus, you must create an object for it to throw. That is, you can't just throw a type.

9.9 Rethrowing an Exception

An exception caught by one catch statement can be rethrown so that it can be caught by an outer catch. The most likely reason for rethrowing this way is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect. Remember, when you rethrow an exception, it will not be recaught by the same catch statement. It will propagate to the next catch statement. The following program illustrates rethrowing an exception.

```
/ / rethrow an exception
class Rethrow {
    public static void main (String args [ ] ) {
        / / Here, number is longer than denom.
        int number [ ] = {4. 8, 16, 32, 64, 128, 256, 512} ;
        int denom [ ] = {2, 0, 4, 4, 0, 8};
        for (int i=0; i<numer.lengthh; i++) {
          try {
             System.out.println (numer [i] + " / " + denom [i] + "is " + numer[i]/denom[i]);
          }
         catch (ArithmeticException exc) {
               / /catch the exception
              System.out.println ("Can't divide by zero! ");
            }
          catch (ArrayIndexOutOfBoundsException exc) {
            / / catch the exception
            System.out.println ("No matching element found. ");
            throw exc; / / rethrow the exception
          }
        }
     }
  }
    class RethrowDemo {
        public static void main (String args [ ]) {
            try {
                Rethrow. genException ( );
            }
           catch(ArrayIndexOutOfBoundsException exc) {
               / / recatch exception
              System.out.println("Fatal error – " + "program terminated.");
           }
         }
```

```
    }
```

In this program, divide-by-zero errors are handled locally, by genException( ), but an array boundary error is rethrown. In this case, it is caught by main( ).

9.10 Using finally

Java supports another statement known as finally statement that can be used to handle an Exception that is not caught by any of the previous catch statements. finally block can be used to handle any exception generated with in a try block. It may be added immediately after the try block or after the last catch block shown as follows:

```
try                                     try
{                                       {
        …….                                      …….
        …….                                      …….

}                                       }

finally                                 catch( …….)
{                                       {
        …….                                      …….
        …….                                      …….

}                                       }
                                        .
                                        .
                                        .
                                        catch( …….)
                                        {
                                                …….
                                                …….
                                        }
```

When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

9.11 Using Throws

In some cases, if a method generates an exception that it does not handle, it must declare that exception in a throws clause. Here is the general form of a method that includes a throws clause:

```
ret-type methName (param-list) throws except-list {
        // body
 }
```

Here, except-list is a comma-separated list of exceptions that the method might throw outside of itself. Exceptions that are subclasses of Error or RuntimeException don't need to be specified in a throws list. Java simply assumes that a method may throw one. All other types of exceptions do need to be declared. Failure to do so causes a compile-time error.

```
throws   java.io.IOException
```

to main( ). Thus, such an exception would be thrown out of main ( ) and needed to be specified as such.

Example:- It creates a method called prompt( ), which displays a prompting message and then reads a character from the keyboard. Since input is being performed, an IOException might occur. However, the prompt( ) method does not handle IOException itself. Instead, it uses a throws clause, which means that the calling method must handle it. In this example, the calling method is main( ), and it deals with the error.

```
/ / Use throws.
 class ThrowDemo {
     public static char prompt (String str)
        throws java.io.IOException {
        System.out.print(str + ";");
        return (char) System.in.read ( );
     }
  public static void main(String args [ ])  {
      char ch;
      try {
          ch = prompt ("Enter a letter");
      }
     catch(Java.io.IOException exc) {
       System.out.println("I/O exception occurred. ");
       ch = 'X';
      }
       System.out.println("You pressed " +ch);
   }
}
```

Note that IOException is fully qualified by its package name java.io. Java's I/O system is contained in the java.io package. Thus, the IOException is also contained there. It would also have been possible to import java.io and then refer to IOException directly.

9.12 Java's Built-in Exceptions

Inside the standard package java.lang, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they need not be included in any method's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang. Where as some exceptions defined by java.lang that must be included in a methods throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. Java defines several other types of exceptions that relate to its various class libraries.

Exception Type	Cause of Exception
ArithrneticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes

ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
ClassCastException	Invalid cast
EnumConstNotFoundException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Caused by referencing a null object
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state
NumberFormatException	Caused when a conversion between strings and number fails
IndexOutOfBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with negative size
NullPointerException	Invalid use of a null reference
NumberFormatException	Caused when a conversion between strings and number fails
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
TypeNotPresentException	Type not found.
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

Table 9.1 the unchecked exception defined in Java.lang

Exception Type	Cause of Exception
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread
NoSuchFiledException	A requested filed does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exception

Table 9.2 The checked exception defined in Java.lang

9.13 Creating Exception Subclasses

There may be times when we would like to throw our own exceptions. We can do this by using the keyword throw as follows:

```
        throw new Throwable_subclass;
```

Examples:
throw new ArithmeticException( );
throw new NumberFormatException( );
Exception is a subclass of Throwable and therefore MyException is a subclass of Throwable class. An object of a class that extends Throwable can be thrown and caught.

Throwing our own exception

```
import java.lang.Exception;
class MyException extends Exception
{
    MyException(String message)
    {
        super (message) ;
    }
}
class TestMyException
{
    public static void main (String args[ ])
    {
        int x = 5, y = 1000;
        try
        {
            float z = (float) x / (float) y ;
        }
        if(z < 0.01)
        {
            throw new MyException("Number is too small");
        }
        catch (MyException e)
        {
            System.out.println("Caught my exception");
            System.out.println(e.getMessage( ) );
        }
        finally
        {
            System.out.println("I am always here");
        }
    }
}
```

9.14 Assignment-9

Short Answer Questions

1. What is an exception?
2. What is the purpose of exception handler?
3. Differentiate the keywords throw and throws.

4. Give an example code which creates runtime error. (M.U. Oct/Nov. 2011)
5. What is an exception? What are the advantages of exception handler? Give an example. (M.U. Oct/Nov. 2011, 2010, 2014)
6. How do we create try block and catch block? (M.U. Oct/Nov. 2008)
7. What is the purpose of try and catch block?
8. What is finally statement in exception?
9. What is meant by rethrowing an exception?
10. List the sublcasses of Throwable class and mention their purpose.
11. List any four types of exceptions in Java.
12. List any 2 methods defined by Throwable class along with their purpose.

Long answer questions

1. List out any 5 common unchecked java exceptions. Also mention the cause for throwing those exceptions?
2. Illustrate the use multiple catch statement with an example. (5 Marks- M.U. Oct/Nov. 2008, 2014)
3. Define an exception called 'InvalidEntry' that is thrown when an accepted numeric value is less than zero and greater than 100. Write a program that uses this exception. (5 Marks- M.U. Oct/Nov. 2009)
4. What is exception? Explain with examples any two exceptions which may arise while working with numeric values. (4 Marks- M.U. Oct/Nov. 2009)
5. Illustrate the use of finally with suitable example. (4 Marks- M.U. Oct/Nov. 2009, 2012, 2014)
6. Explain the exception handling mechanism in java with an example (4 Marks- M.U. Oct/Nov. 2010, 2012)
7. Explain the purpose of try and catch statements in java. (4 Marks- M.U. Oct/Nov. 2010)
8. With suitable example code, explain how to manipulate multiple catch blocks to handle several exceptions. (6 Marks- M.U. Oct/Nov. 2011)
9. List some common type of exception that might occur in java. Give examples. (5Marks- M.U. Oct/Nov. 2012)
10. How we can throw our own exceptions in Java? Define an exception called "NoMatchEx" that is thrown when a string is not equal to "India". Write a java program that uses this exception. (5 Marks)
11. Write a short note on Rethrowing exception (4 Marks)
12. List and explain different methods of Throwable class.
13. Write a short note on "using throws" (5 Marks)
14. List and explain any 5 built-in exceptions of Java (5 marks)
15. Explain how to create user defined exceptions in Java with an ecample (6 marks- M.U. Oct/Nov. 2014)

UNIT-III
Chapter 10
Multithreaded Programming

10.1 Multithreading fundamentals

There are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, it is process-based multitasking that allows you to run the Java compiler at the same time you are using a text editor or browsing the Internet. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks at once. For instance, a text editor can be formatting text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Although Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. Multithreaded multitasking is.

A principal advantage of multithreading is that it enables you to write very efficient programs because it lets you utilize the idle time that is present in most programs. As you probably know, most I/O devices, whether they be network ports, disk drives, or the keyboard, are much slower than the CPU. Thus, a program will often spend a majority of its execution time waiting to send or receive information to or from a device. By using multithreading, your program can execute another task during this idle time. For example, while one part of your program is sending a file over the Internet, another part can be reading keyboard input, and still another can be buffering the next block of data to send.

It is important to understand that Java's multithreading features work in both types of systems. In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multiprocessor/multicore systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations. A thread can be in one of several states. It can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which is a temporary halt to its execution. It can later b e resumed.

A thread can be blocked when waiting for a resource. A thread can be terminated, in which case its execution ends and cannot be resumed. Along with thread-based multitasking comes the need for a special type of feature called synchronization, which allows the execution of threads to be coordinated in certain well-defined ways. However, the fact that Java manages threads through language elements makes multithreading especially convenient.

10.2 The Thread Class and Runnable Interface

Java's multithreading system is built upon the Thread class and its companion interface, Runnable. Both are packaged in java.lang. Thread encapsulates a thread of execution. To create a new thread, your program will either extend Thread or implement the Runnable interface. The Thread class defines several methods that help manage threads. Here are some of the more commonly used ones.

Method	Meaning
final int getName( )	Obtains a thread's name
final int getPriority( )	Obtains a thread's priority
final Boolean isAlive( )	Determines whether a thread is still running.
final void join( )	Waits for a thread to terminate
void run( )	Entry point for the thread
static void sleep(long milliseconds)	Suspends a thread for a specified period of milliseconds
void start( )	Starts a thread by calling its run ( ) method

Table 10.1 Thread Methods

All processes have at least one thread of execution, which is usually called the main thread, because it is the one that is executed when your program begins.. From the main thread, you can create other threads.

10.3 Creating Thread

We stated earlier that we can create threads in two ways: one by using the extended Thread class and another by implementing the Runnable interface, The Runnable interface declares the run( ) method that is required for implementing threads in our programs. To do this, we must perform the steps listed below:

1. Declare the class as implementing the Runnable interface.
2. Implement the run( ) method.
3. Create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.
4. Call the thread's start( ) method to run the thread.

Using Runnable interface

```
class X implements Runnable                    // Step 1
{
        public void run( )                     // Step 2
        {
                for(int i = 1; i<=10; i++)
                {
                        System.out.println("\tThreadX : " +i);
                }
        System.out.println("End of ThreadX");
        }
}

class RunnableTest
{
        public static void main (String args[ ])
        {
                X runnable = new X( ) ;
```

```
                Thread threadX = new Thread(runnable);          //Step 3
        }
}
```

Extending the thread: We can create a thread, by extending the Thread class that is defined under package java.lang.Thread. This gives us access to all the thread methods directly. It includes the following steps:
1. Declare the class as extending the Thread class.
2. Implement the run( ) method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the start() method to initiate the thread execution.

Declaring the Class
The Thread class can be extended as follows:

```
class MyThread extends Thread
{
        ……..
        ……..
        ……..
}
```

Implementing the run ( ) Method
The run() method has been inherited by the class MyThread. We have to override this method in order to implement the code to be executed by our thread. The basic implementation of run ( ) is:

```
public void run( )
{
        ……
        ……...          // Thread code here
}
```

When we start the new thread, Java calls the thread's run () method, so it is the run () where all the action takes place.

Starting New Thread
To actually create and run an instance of our thread class, we must write the following:

```
MyThread aThread = new MyThread( );
aThread.start ( );      // invokes run( ) method
```

The first line instantiates a new object of class MyThread. Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a *newborn* state. The second line calls the start ( ) method pausing the thread to move into the *runnable* state. Then, the Java runtime will schedule the thread to run by invoking its run ( ) method. Now, the thread is said to be in the *running* state

10.4 Creating Multiple Threads

The below program illustrates the use of Thread class for creating and running threads in an application. The program creates three threads A, B, and C for undertaking three different tasks. The main method in the ThreadTest class also constitutes another thread which we may call the "main thread".
The main thread dies at the end of its main method. However, before it dies, it creates and starts all the three threads A, B, and C. Note the statements like
new A ( ).start ( );
in the main thread. This is just a compact way of starting a thread. This is equivalent to:
A threadA = new A( );
threadA.start( );
Immediately after the thread A is started, there will be two threads running in the program i.e. the main thread and the thread A. The start( ) method returns back to the main thread immediately after invoking the rune) method, thus allowing the main thread to start the thread B.

Creating threads using the thread class

```
class A extends Thread
{
        public void run( )
        {
                for(int i=l; i<=5;i++)
                {
                        System.out.println("\tFrom Thread A : i = " + i);
                }
                System.out.println("Exit form A");
        }
}

class B extends Thread
{
        public void run( )
        {
                for(int j=l; j<=5; j++)
                {
                        System.out.println("\tFrom Thread B : j = " + j);
                }
                System.out.println("Exit from B ");
        }
}

class C extends Thread
{
        public void run( )
        {
                for(int k=l; k<=5; k++)
                {
                        System.out.println("\tFrom Thread C : k = " + k);
                }
```

```
                System.out.println("Exit from C ");
        }
}

class ThreadTest
{
        public static void main (String args[ ])
        {
                new A( ). start ( );
                new B( ). start ( );
                new C( ) .start( );
        }
}
```

Output of Program

```
First run
From Thread A : i = 1
From Thread A : i = 2
From Thread B : j = 1
From Thread B : j = 2
From Thread C : k = 1
From Thread C : k = 2
From Thread A : i = 3
From Thread A : i = 4
From Thread B : j = 3
From Thread B : j = 4
From Thread C : k = 3
From Thread C : k = 4
From Thread A : i = 5
Exit from A
From Thread B : j = 5
Exit from B
From Thread C : k = 5
Exit from C

Second run
From Thread A : i = 1
From Thread A : i = 2
From Thread C : k = 1
From Thread C : k = 2
From Thread A : i = 3
From Thread A : i = 4
From Thread B : j = 1
From Thread B : j = 2
From Thread C : k = 3
From Thread C : k = 4
From Thread A : i = 5
Exit from A
```

```
From Thread B : k = 4
From Thread B : j = 5
From Thread C : k = 5
Exit from C
From Thread B : j = 5
Exit from B
```

All the four threads including main( ) are running concurrently on their own. Note that the output from the threads are not specially sequential. They do not follow any specific order. They are running independently of one another and each executes whenever it has a chance. Remember, once the threads are started, we cannot decide with certainty the order in which they may execute statements. Second run has a different output sequence.

10.5 Determining When a Thread Ends

It is often useful to know when a thread has ended. For example, in the preceding examples, for the sake of illustration it was helpful to keep the main thread alive until the other threads ended. In those examples, this was accomplished by having the main thread sleep longer than the child threads that it spawned. This is, of course, hardly a satisfactory or generalizable solution. Fortunately, Thread provides two means by which you can determine if a thread has ended. First, you can call isAlive( ) on the thread. Its general form is shown here:

```
final boolean isAlive( )
```

The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

```
/ / Use isAlive( ).
class MoreThreads {
   public static void main (String args[ ]) {
      System.out.println("Main thread starting.");
      MyThread mt1 = new MyThread ("Child #1");
      MyThread mt2 = new MyThread ("Child #2");
      MyThread mt3 = new MyThread ("Child #3");
       do {
            System.out.print ( " . ");
            try {
               Thread.sleep (100);
            }
           catch(InterruptedException exc) {
              System.out.println("Main thread interrupted.");
           }
        } while (mt1.thrd.isAlive ( ) | | mt2.thrd.isAlive ( ) | | mt3.thrd.isALive( ) );
         System.out.println("Main thread ending.");
      }
}
```

This version produces output that is similar to the previous version, except that main( ) ends as soon as the other threads finish. The difference is that it uses isAlive( ) to wait for the child threads to terminate. Another way to wait for a thread to finish is to call join( ), shown here:

```
final void join( ) throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate. Here is a program that uses join( ) to ensure that the main thread is the last to stop:

```
/ / Use Join ( ).
   class Mythread implements Runnable {
          Thread thrd;
      // Construct a new thread.
      Mythread (String name) {
           thrd = new Thread (this, name);
           thrd.start( );  / /start the thread
      }
    / / Begin execution of new thread.
     public void run ( )   {
        System.out.println (thrd.getName ( ) + "starting. ");
        try {
            for (int count=0; count<10; count++) {
                Thread.sleep(400);
                System.out.println("In" + thrd.getName( ) + ", count is " + count);
             }
        }
       catch(InterruptedException exc) {
           System.out.println(thrd.getName ( ) + "interrupted.");
       }
       System.out.println(thrd.getName ( ) + "terminating.");
   }
}

class JoinThreads {
      public static void main(String args [ ] ) {
           System.out.println ("Main thread starting.");
           MyThread mt1 = new MyThread ("Child #1");
           MyThread mt2 = new MyThread ("Child #2");
           MyThread mt3 = new MyThread ("Child #3");
            try{
               mt1.thrd.join();
               System.out.println ("Child #1 joined.");
               mt2.thrd.join();
               System.out.println ("Child #2 joined.");
               Mt3.thrd.join();
               System.out.println ("Child #2 joined.");
             }
            catch(InterruptedException exc) {
               System.out.println("Main thread interrupted.");
            }
         }
          System.out.println("Main thread ending.");
```

```
    }
  }
```

10.6 Thread Priorities

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The threads that we have discussed so far are of the same priority. The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a first-come, first-serve basis. Java permits us to set the priority of a thread using the setPriority( ) method as follows:

```
ThreadName.setPriority(intNumber);
```

The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:

MIN_PRIORITY= 1
NORM_PRIORITY= 5
MAX_PRIORITY= 10

The intNumber may assume one of these constants or any value between 1 and 10. Note that the default setting is NORM_PRIORITY.

Most user-level processes should use NORM_PRIORITY, plus or minus 1. Back-ground tasks such as network I/O and screen repainting should use a value very near to the lower limit. We should be very cautious when trying to use very high priority values. This may defeat the very purpose of using multithreads. By assigning priorities to threads, we can ensure that they are given the attention (or lack of it) they deserve. For example, we may need to answer an input as quickly as possible. Whenever multiple threads are ready for execution, the Java system chooses the highest priority thread and executes it. For a thread of lower priority to gain control, one of the following things should happen:

1. It stops running at the end of run( ).
2. It is made to sleep using sleep( ).
3. It is told to wait using wait( ).

However, if another thread of a higher priority comes along, the currently running thread will be *preempted* by the incoming thread thus forcing the current thread to move to the runnable state. Remember that the highest priority thread always preempts any lower priority threads.

Use of priority in thread

```
class A extends Thread
{
        public void run( )
        {
                System.out.println("threadA started");
                for(int i=l; i<=4; i++)
                {
```

```
                        System.out.println(~\tFrom Thread A : i = ~ +i);
                }
                System.out.println(“Exit from A”);
        }
}

class B extends Thread
{
        public void run( )
        {
                System.out.println(“threadB started");
                for(int j=l; j<=4; j++)
                {
                        System.out.println("\tFrom Thread B : j = " +j);
                }
                System.out.println("Exit from B ");
        }
}

class C extends Thread
{
        public void run( )
        {
                System.out.println("threadC started");
                for(int k=l; k<=4; k++)
                {
                        System.out.println("\tFrom Thread C : k = " +k);
                }
                System.out.println("Exit from C ");
        }
}

class ThreadPriority
{
        public static void main (String args[ ])
        {
                A threadA = new A( );
                B threadB = new B( );
                C threadC = new C( );
                threadC.setPriority(Thread.MAX_PRIORITY);
                threadB.setpriority(threadA.getPriority( )+1);
                threadA.setPriority(Thread.MIN_PRIORITY);
                System.out.println("Start thread A");
                threadA.start( );
                System.out.println("Start thread B");
                threadB.start( );
                System.out.println("Start thread C");
```

```
                thread.start( );
                System.out.println("End of main Thread");
        }
}
```

Output:

```
Start thread A
Start thread B
Start thread C
threadB started
From Thread B :  j=1
From Thread B :  j=2
threadC started
From Thread C :  k=1
From Thread C :  k=2
From Thread C :  k=3
From Thread C :  k=4
Exit from C
End of main thread
From Thread B :  j = 3
From Thread B :  j = 4
Exit from B
threadA started
From Thread A : i = 1
From Thread A : i = 2
From Thread A : i = 3
From Thread A : i = 4
Exit from A
```

10.7 Synchronization

If a thread try to use data and methods outside their run( ) method then they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as *synchronization.* In case of Java, the keyword synchronised helps to solve such problems by keeping a watch on such locations. For example, the method that will read information from a file and the method that will update the same file may be declared as synchronized. Example:

```
synchronized void update( )
{
        ……..
        …….
        …….          // code here is synchronized
}
```

When we declare a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the

key can only open the lock. It is also possible to mark a block of code as synchronized as shown below:

```
synchronized (lock-object)
{
        ……..          //code here is synchronized
        …….
}
```

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

An interesting situation may occur when two or more threads are waiting to gain control of a resource. Due to some reason, the condition on which the waiting threads rely on to gain control does not happen. This results in what is known as *deadlock*. For example, assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets hold of Method2. Because these are mutually exclusive conditions, a deadlock occurs.

```
Thread A
        synchronized method2( )
        {
                synchronized methodl( )
                {
                        …….
                        …….
                }
        }
ThreadB
        synchronized method1( )
        {
                synchronized method2( )
                {
                        …….
                        …….
                }
        }
```

10.8 Using Synchronized Methods

You can synchronize access to a method by modifying it with the synchronized keyword. When that method is called, the calling thread enters the object's monitor, which then locks the object. While locked, no other thread can enter the method, or enter any other synchronized method defined by the object's class. When the thread returns from the method, the monitor unlocks the object, allowing it to be used by the next thread. Thus, synchronization is achieved with virtually no programming effort on your part. The following program demonstrates synchronization by controlling access to a method called sumArray( ), which sums the elements of an integer array.

```
 / / Use Synchronize to control access.
  class SumArray {
      private int sum;
```

```
        Synchronized int sumArray (int nums [ ] ) {
            sum= 0;  / / resetsum
        for ( int i=0; i<nums. Length; i++) {
            sum+= nums[i];
            System.out.println("Running total for" + Thread.getName ( ) + "is " +sum);
            try {
              thread.sleep (10);  / /allow task-switch
              }
          catch (InterruptedException exc) {
                System.out.println( "Thread Interrupted");
         }
      }
       return sum;
   }
}

class Mythread implements Runnable {
     Thread thrd;
     static SumArray sa= new SumArray ( );
      int a[ ];
      int answer;
      / / Construct a new thread.
     Mythread (String name, int nums[ ]) {
        thrd = new Thread (this, name);
         a = nums;
        thrd.start ( ); / /start the thread
    }
 / / begin execution of new thread.
   public void run ( ) {
       int sum;
       System.out.println(thrd.getName ( ) + "starting.");
       answer = sa.sumArray (a);
       System.out.println("Sum for " + thrd.getName ( ) + "is " +answer);
       Systeem.out.println(thrd.getname( ) +  "terminating.");
   }
}

    class Sync {
       public static void main(String args[ ]) {
          int a [ ] = {1, 2, 3, 4, 5};
          Mythread mt1  = new MyThread ("Child #1", a);
          Mythread mt2 = new Mythread("Child #2",a);
          try {
             mt1. Thrd.join ( );
             mt2.thrd.join ( );
            }
          catch (InterruptedException exc)
```

```
            {
              System.out.println ("Main thread interrupted.");
            }
        }
    }
```

The program creates three classes. The first is SumArray. It contains the method sumArray( ), which sums an integer array. The second class is MyThread, which uses a static object of type SumArray to obtain the sum of an integer array. This object is called sa and because it is static, there is only one copy of it that is shared by all instances of MyThread. Finally, the class Sync creates two threads and has each compute the sum of an integer array.

Inside sumArray( ), sleep( ) is called to purposely allow a task switch to occur. Because sumArray( ) is synchronized, it can be used by only one thread at a time. Thus, when the second child thread begins execution, it does not enter sumArray( ) until after the first child thread is done with it. This ensures that the correct result is produced.

10.9 The synchronized Statement

Although creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. For example, you might want to synchronize access to some method that is not modified by synchronized. This can occur because you want to use a class that was not created by you but by a third party and you do not have access to the source code. Thus, it is not possible for you to add synchronized to the appropriate methods within the class. You can simply put calls to the methods defined by this class inside a synchronized block. This is the general form of a synchronized block:

```
synchronized (objref) {
    / /statements to be synchronized
}
```

Here, objref is a reference to the object being synchronized. Once a synchronized block has been entered, no other thread can call a synchronized method on the object referred to by objref until the block has been exited.

```
/ / Use Synchronized block to control access to SumArray.
  class SumArray {
      private int sum;
      int sumArray (int nums [ ] ) {
          sum= 0;  / / reset sum
       for ( int i=0; i<nums. Length; i++) {
          sum+= nums[i];
          System.out.println("Running total for" + Thread.getName ( ) + "is " +sum);
          try {
            thread.sleep (10);  / /allow task-switch
            }
        catch (InterruptedException exc) {
              System.out.println( "Thread Interrupted");
      }
    }
```

```
      return sum;
   }
}

class Mythread implements Runnable {
     Thread thrd;
     static SumArray sa= new SumArray ( );
      int a[ ];
      int answer;
      / / Construct a new thread.
     Mythread (String name, int nums[ ]) {
        thrd = new Thread (this, name);
         a = nums;
        thrd.start ( ); / /start the thread
    }
 / / begin execution of new thread.
   public void run ( ) {
      int sum;
      System.out.println(thrd.getName ( ) + "starting.");
     / /synchronize call to sumArray ( )
       synchronized (sa);
      answer = sa.sumArray (a);
      System.out.println("Sum for " + thrd.getName ( ) + "is " +answer);
      Systeem.out.println(thrd.getname( ) +  "terminating.");
   }
}

    class Sync {
      public static void main(String args[ ]) {
         int a [ ] = {1, 2, 3, 4, 5};
         Mythread mt1  = new MyThread ("Child #1", a);
         Mythread mt2 = new Mythread("Child #2",a);
         try {
            mt1. Thrd.join ( );
            mt2.thrd.join ( );
           }
         catch (InterruptedException exc)
           {
             System.out.println ("Main thread interrupted.");
           }
        }
    }
```

10.10 Thread Communication Using notify( ), wait( ), and notifyAll( )

A thread called T is executing inside a synchronized method and needs access to a resource called R that is temporarily unavailable. If T enters some form of polling loop that waits for R, T ties up the object, preventing other threads' access to it. This is a less than optimal solution

because it partially defeats the advantages of programming for a multithreaded environment. A better solution is to have T temporarily relinquish control of the object, allowing another thread to run. When R becomes available, T can be notified and resume execution. Such an approach relies upon some form of interthread communication in which one thread can notify another that it is blocked and be notified that it can resume execution. Java supports interthread communication with the wait( ), notify( ), and notifyAll( ) methods.

The wait( ), notify( ), and notifyAll( ) methods are part of all objects because they are implemented by the Object class. These methods can be called only from within a synchronized context. Here is how they are used. When a thread is temporarily blocked from running, it calls wait( ). This causes the thread to go to sleep and the monitor for that object to be released, allowing another thread to use the object. At a later point, the sleeping thread is awakened when some other thread enters the same monitor and calls notify( ), or notifyAll( ).
Following are the various forms of wait( ) defined by Object:

final void wait( ) throws InterruptedException
final void wait(long millis) throws InterruptedException
final void wait(long millis, int nanos) throws InterruptedException

The first form waits until notified. The second form waits until notified or until the specified period of milliseconds has expired. The third form allows you to specify the wait period in terms of nanoseconds.
Here are the general forms for notify( ) and notifyAll( ):

final void notify( )
final void notifyAll( )

A call to notify( ) resumes one waiting thread. A call to notifyAll( ) notifies all threads, with the highest priority thread gaining access to the object. Although wait( ) normally waits until notify( ) or notifyAll( ) is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a spurious wakeup.

An Example That Uses wait( ) and notify( ) :To understand the need for and the application of wait( ) and notify( ), we will create a program that simulates the ticking of a clock by displaying the words Tick and Tock on the screen. To accomplish this, we will create a class called TickTock that contains two methods: tick( ) and tock( ). The tick( ) method displays the word “Tick”, and tock( ) displays “Tock”. To run the clock, two threads are created, one that calls tick( ) and one that calls tock( ). The goal is to make the two threads execute in a way that the output from the program displays a consistent “Tick Tock” that is, a repeated pattern of one tick followed by one tock.

```
/ / Use wait ( ) and notify ( ) to create a ticking clock.
    class TickTock {
        String state; / /contains the state of the clock
     synchronized void tick (boolean running) {
       if(!running) { / /stop the clock
           state = “ticked”;
           notify ( ); / /notify any waiting threads
           return;
         }
         System.out.print (“Tick”);
         State= “ticked”; / /set the current state to ticked
```

```
            notify ( );   / / let tock ( ) run
            try {
              while( !state.equals("tocked"))
                  wait( )  / / wait for tock ( ) to complete
              }
            catch (InterruptedException exc) {
                System.out.println ("Thread interrupted.");
            }
        }
        synchronized void tock (boolean running) {
          if(!running) { / /stop the clock
              state = "tocked";
              notify ( ); / /notify any waiting threads
              return;
            }
            System.out.print ("Tock");
            State= "tocked"; / /set the current state to tocked
            notify ( );   / / let tock ( ) run
            try {
              while( !state.equals("ticked"))
                  wait( )  / / wait for tick ( ) to complete
              }
            catch (InterruptedException exc) {
                System.out.println ("Thread interrupted.");
            }
        }
    }

class Mythread implements Runnable {
      Thread thrd;
      TickTock ttOb;
      / / Construct a new thread.
      Mythread (String name, TickTock tt) {
            thrd = new Thread (this, name);
            ttob = tt;
            thrd.start ( );  / /start the thread
      }
     / / Begin execution of new thread.
     public void run ( )  {
          if (thrd.getName ( ).compareTo ("Tick") = = 0) {
              for(int i=0; i<5; i++) ttOb.tick (true);
              ttob.tick(false);
          }
          else  {
              for (int i=0; i<5; i++) ttOb.tock (true);
              ttob.tock(false);
          }
```

```
    }
  }

  class ThreadCom {
     public static void main (String args [ ]) {
        TickTock tt = new TickTock ( );
         Mythread mt1= new Mythread ("Tick", tt);
         Mythread mt2= new MyThread ("Tock", tt);
         try {
           mt1.thrd.join ( );
           mt2.thrd.join ( );
         }
            catch (InterruptedException exc) {
             System.out.println("Main thread interrupted.");
         }
      }
    }
```

Here is the output of the program

```
Tick Tock
Tick Tock
Tick Tock
Tick Tock
Tick Tock
```

10. 11 Suspending, Resuming, and Stopping Threads

It is sometimes useful to suspend execution of a thread. For example, a separate thread can be used to display the time of day. If the user does not desire a clock, then its thread can be suspended. Whatever the case, it is a simple matter to suspend a thread. Once suspended, it is also a simple matter to restart the thread. The mechanisms to suspend, stop, and resume threads differ between early versions of Java and more modern versions, beginning with Java 2. Prior to Java 2, a program used suspend( ), resume( ), and stop( ), which are methods defined by Thread, to pause, restart, and stop the execution of a thread. They have the following forms:

```
final void resume( )
final void suspend( )
final void stop( )
```

Since you cannot now use the suspend( ), resume( ), or stop( ) methods to control a thread. A thread must be designed so that the run( ) method periodically checks to determine if that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing two flag variables: one for suspend and resume, and one for stop. For suspend and resume, as long as the flag is set to "running," the run( ) method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. For the stop flag, if it is set to "stop," the thread must terminate. The following example shows one way to implement your own versions of suspend( ), resume( ), and stop( )

```
  class MyThread implements Runnable {
     thread thrd;
     boolean suspend;
```

```
    boolean stopped;
    Mythread (String name) {
        thrd = new Thread(this, name);
        suspend = false;
        stopped = false;
        thrd.start ( );
    }

/ / this is the entry point for thread.
public void run ( ) {
    System.out.println (thrd.getName ( ) + "starting. ";
    try {
      for(int i=1; i<1000; i++) {
        system.out.print(I + " ");
        if ((i % 10) = =0) {
            System.out.println ( );
            Thread.sleep(250);
        }
       / / Use synchronized block to check suspend and stopped.
       synchronized (this) {
         while (suspended) {
           wait ( );
         }
        if (stopped) break;
       }
    }
  } catch (InterruptedException exc) {
    System.out.println (thrd.getName ( ) + "interrupted. ";
  }
   System.out,println (thrd.getName ( ) + "exiting. " );
}

 / / Stop the thread.
 Sychronized void mystop ( ) {
    stopped = true;
        / / the following ensures that a suspended thread can be stopped.
        suspend =false;
        notify ( );
    }
     / / suspend the thread.
     synchronized void mysuspend ( ) {
        suspended = true;
     }
    / / resumes the thread.
          synchronized void myresume ( ) {
        suspended = false;
         notify ( );
```

```
        }
    }

class suspend {
   public static void main (String args [ ] ) {
     Mythread ob1 =new Mythread (“My Thread “);
     try {
        Thread.sleep (1000);  / / let ob1 thread start executing
        ob1.mysuspend ( );
        System.out.println (“Suspending thread.”);
        Thread.sleep (1000);
        ob1.myresume ( );
        System.out.println (“Resuming thread.”);
        Thread.sleep (1000);
        ob1.mysuspend ( );
        System.out.println (“Suspending thread.”);
        Thread.sleep (1000);
        ob1.myresume ( );
        System.out.println (“Resuming thread.”);
        Thread.sleep (1000);
        ob1.mysuspend ( );
        System.out.println (“Stoping thread.”);
        ob1.mystop( );
      }catch (InterruptedException e) {
        System.out.println(“Main thread Interrupted” );
     }

    / / wait for thread to finish
    try {
       ob1.thrd.join ( );
      } catch (InterruptedException e ) {
        System.out.println (“Main thread Interrupted “ );
      }
      System.out.println (“main thread exiting.”);
    }
   }
```

Sample output from this program is shown here (your output may differ slightly)

```
   My thread starting.
   1 2 3 4 5 6 7 8 9 10
   11 12 13 14 15 16 17 18 19 20
   21 22 23 24 25 26 27 28 29 30
   31 32 33 34 35 36 377 38 39 40
   Suspending thread.
   Resuming thread
   41 42 43 44 45 46 47 48 49 50
   51 52 53 54 55 56 57 58 59 60
   61 62 63 64 65 66 67 68 69 70
```

```
71 72 73 74 75 76 77 78 79 80
Suspending thread.
Resuming thread
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 110
Stoping thread.
My thread exiting.
Main thread exiting.
```

Here is how the program works. The thread class MyThread defines two Boolean variables, suspended and stopped, which govern the suspension and termination of a thread. Both are initialized to false by the constructor. The run( ) method contains a synchronized statement block that checks suspended. If that variable is true, the wait( ) method is invoked to suspend the execution of the thread. To suspend execution of the thread, call mysuspend( ), which sets suspended to true. To resume execution, call myresume( ), which sets suspended to false and invokes notify( ) to restart the thread. To stop the thread, call mystop( ), which sets stopped to true. In addition, mystop( ) sets suspended to false and then calls notify( ). These steps are necessary to stop a suspended thread.

10.12 Assignment-10

Short Answer Questions (2 Marks each)

1. What is a thread?
2. How do we start a thread?
3. What is multithreading?
4. List different states of a thread.
5. How to assign priority to thread?
6. What is Runnable State in thread?
7. List any four thread methods.
8. What is the purpose of join( ) method in case of threads?
9. What is the purpose of isAlive( ) method in case of threads?
10. What is the purpose of notify( ) method?
11. What is the purpose of notifyall( ) method?
12. What are the two methods by which we can stop the threads (M.U. Oct/Nov.2012)
13. What is meant by thread priority? What is its default value? (M.U. Oct/Nov.2010)
14. When does a thread move to blocked state?
15. Differentiate suspend( ) and stop( ) methods of a thread.
16. What is the purpose of resume( ) method?
17. What is the purpose of wait() and notify() methods?
18. What is synchronization? (M.U. Oct/Nov.2009)
19. What are the two methods by which we may create threads? (M.U. Oct/Nov.2008)
20. Write the two ways of identifying whether the thread has ended. (M.U. Oct/Nov.2014)

Long Answer Questions

1. Write a program using threads to generate multiplication table for 3,4 and 5 by selecting various priorities to display in the sequence of 4,5 & 3.
2. Explain creating a thread by creating a thread class with an example?

3. How do we start the thread? Also explain both the methods by which we may stop threads. (5 Marks- M.U. Oct/Nov.2008)
4. Write a note on Thread Priority. (5 Marks- M.U. Oct/Nov.2008, 2012, 2014)
5. Explain with example any four methods of thread. (6 Marks- M.U. Oct/Nov.2009)
6. List and explain different states of a thread. (5 marks)
7. With an example explain creation of thread using Runnable interface. (6 Marks- M.U. Oct/Nov.2010, 2011)
8. What is a thread? What are the different ways to create a thread? Explain any one method. (6 Marks- M.U. Oct/Nov.2012)
9. Define synchronization. With an example explain how synchronization is achieved in multithreaded environment using synchronized methods. (5 Marks-M.U. Oct/Nov.2014)
10. Explain Thread Communication using notify(), wait() and notifyAll( ). (5 Marks)
11. Explain Suspending, Resuming, and Stopping of Threads with an example. (5 Marks)
12. Explain the process of creating a thread by implementing Runnable interface with suitable example. (5 marks)

UNIT-IV

Chapter 11

Applets, Events and Miscellaneous Topics

11.1 Applet Basics

Applets are small programs that are designed for transmission over the Internet and run within a browser. Because Java's virtual machine is in charge of executing all Java programs, including applets, applets offer a secure way to dynamically download and execute programs over the Web.

There are two general varieties of applets: those based on the Abstract Window Toolkit (AWT) and those based on Swing. Both the AWT and Swing support the creation of a graphical user interface (GUI). The AWT is the original GUI toolkit and Swing is Java's lightweight alternative. It is important to understand, however, that Swing-based applets are built upon the same basic architecture as AWT-based applets. Furthermore, Swing is built on top of the AWT. Therefore, the information and techniques presented here describe the foundation of applet programming and most of it applies to both types of applets.

```
/ / A Minimal AWT-based applet.
 import java.awt.*;
 import java.applet.*;
 public class SimpleApplet extens Applet {
    public void paint (Graphics g) {
       g.drawString ("Java makes applets easy. ", 20, 20);
    }
}
```

This applet begins with two import statements. The first imports the Abstract Window Toolkit classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface. The next import statement imports the applet package. This package contains the class Applet. Every applet that you create must be a subclass (either directly or indirectly) of Applet. The next line in the program declares the class SimpleApplet.

This class must be declared as public because it will be accessed by outside code. Inside SimpleApplet, paint( ) is declared. This method is defined by the AWT Component class (which is a superclass of Applet) and is overridden by the applet. paint( ) is called each time the applet must redisplay its output. This can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or the applet window can be minimized and then restored. paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Inside paint( ), there is a call to drawString( ), which is a member of the Graphics class. This method outputs a string beginning at the specified X, Y location. It has the following general form:

```
void drawString(String message, int x, int y)
```

Here, message is the string to be output beginning at x, y. In a Java window, the upper-left corner is location 0,0. The call to drawString( ) in the applet causes the message to be displayed beginning at location 20,20.

The applet does not have a main( ) method. Unlike the standalone application program, applets do not begin execution at main( ). In fact, most applets don't even have a main( ) method. Instead, an applet begins execution when the name of its class is passed to a browser or other applet-enabled program.

Compiling applet program is similar to stand alone application program. However, running SimpleApplet involves a different process. There are two ways in which you can run an applet: inside a browser or with a special development tool that displays applets. The tool provided with the standard Java JDK is called appletviewer use it to run the applets. We can also run them in your browser, but the appletviewer is much easier to use during development.

One way to execute an applet (in either a Web browser or the appletviewer) is to write a short HTML text file that contains a tag that loads the applet. Currently, Oracle recommends using the APPLET tag for this purpose.

```
<applet code = "simple applet " width=200 height=60>
</applet>
```

The width and height statements specify the dimensions of the display area used by the applet. To execute SimpleApplet with an applet viewer, you will execute this HTML file. For example, if the preceding HTML file is called StartApp.html, then the following command line will run SimpleApplet:

```
C:\> appletviewer StartApp.html
```

11.2 Applet Organization and Essential Elements

Before building an applet program we need to know more about how applets are organized, what methods they use, and how they interact with the run-time system.

11.2.1 The Applet Architecture

An applet is a GUI-based program. Applet architecture is different from the console-based programs. First, applets are event driven, and an applet resembles a set of interrupt service routines. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the system. This is a crucial point. For the most part, your applet should not enter a "mode" of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution.

Second, it is the user who initiates interaction with an applet not the other way around. In a console-based program, when the program needs input, it will prompt the user and then call some input method. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks a mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a

key while the applet's window has input focus, a keypress event is generated. Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated. While the architecture of an applet is not as easy to understand as that of a console-based program, Java makes it as simple as possible. If you have written programs for Windows (or another GUI-based operating system), you know how intimidating that environment can be. Fortunately, Java provides a much cleaner approach that is more quickly mastered.

11.2.2 A Complete Applet Skeleton

Although SimpleApplet shown earlier is a real applet, it does not contain all of the elements required by most applets. Actually, all but the most trivial applets override a set of methods that provide the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. These lifecycle methods are init( ), start( ), stop( ), and destroy( ), and they are defined by Applet. A fifth method, paint( ), is commonly overridden by AWT-based applets even though it is not a lifecycle method. It is inherited from the AWT Component class. Since default implementations for all of these methods are provided, applets do not need to override those methods they do not use. These four lifecycle methods plus paint( ) can be assembled into the skeleton shown below

```
 / / An AWT-based Applet skeleton.
  import java.awt.*;
  import java.applet.*;
/* <applet code =”AppletSkel” width=300 height=100>
    </applet>
*/
public class AppletSkel extends Applet {
  / / called first.
  public void init ( ) {
      / / initialization
   }

/* Called second, after init ( ). Also called whenever the applet is restarted. */
  public void start ( ) {
     / /  start or resume execution
  }
  / / called when the applet is stopped.
  public void stop ( ) {
      / / suspends the execution
  }
  /* called when applet is terminated. This is the last method executed. */
  public void destroy ( ) {
      / / perform shutdown activities
   }
  / / called when an AWT-based applet's windows must be restored.
   public void paint (Graphics g) {
      / / redisplay contents of window
   }
```

}

Although this skeleton does not do anything, it can be compiled and run. Thus, it can be used as a starting point for applets that you create. Overriding paint( ) applies mostly to AWT-based applets. Swing applets use a different painting mechanism.

11.3 Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are executed. When an applet begins, the following methods are called in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

The init( ) method is the first method to be called. In init( ) applet will initialize variables and perform any other startup activities. The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped, such as when the user returns to a previously displayed Web page that contains an applet. Thus, start( ) might be called more than once during the life cycle of an applet. The paint( ) method is called each time an AWT-based applet's output must be redrawn. When the page containing your applet is left, the stop( ) method is called. We use stop( ) to suspend any child threads created by the applet and to perform any other activities required to put the applet in a safe, idle state. Remember, a call to stop( ) does not mean that the applet should be terminated because it might be restarted with a call to start( ) if the user returns to the page. The destroy( ) method is called when the applet is no longer needed. It is used to perform any shutdown operations required of the applet.

11.4 Requesting Repainting

As a general rule, an AWT-based applet writes to its window only when its paint( ) method is called by the run-time system. One of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the Java run-time system. It cannot create a loop inside paint( ) that repeatedly scrolls the banner, for example. This would prevent control from passing back to the run-time system. Whenever applet needs to update the information displayed in its window, it simply calls repaint( ). The repaint( ) method is defined by the AWT's Component class. It causes the run-time system to execute a call to your applet's paint( ) method. Thus, for another part of applet to output to its window, simply store the output and then call repaint( ). This causes a call to paint( ), which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a String variable and then call repaint( ). Inside paint( ), you will output the string using drawString( ). The simplest version of repaint( ) is shown here:

```
void repaint( )
```

This version causes the entire window to be repainted. Another version of repaint( ) specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by left and top, and the width and height of the region are passed in width and height. These dimensions are specified in pixels. You save time by specifying a region to repaint because window updates are costly in

terms of time. If you only need to update a small portion of the window, it is more efficient to repaint only that region.

11.4.1 The update ( ) Method

There is another method that relates to repainting called update( ) that your applet may want to override. This method is defined by the Component class, and it is called when your applet has requested that a portion of its window be redrawn. The default version of update( ) simply calls paint( ). However, you can override the update( ) method so that it performs more subtle repainting, but this is an advanced technique that is beyond the scope of this book. Also, overriding update( ) applies only to AWT-based applets.

To demonstrate repaint( ), a simple banner applet is presented. This applet scrolls a message, from right to left, across the applet's window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. Banners are popular Web features, and this project shows how to use a Java applet to create one.

```
import java.awt.*;
import java.applet.*;
/*
<applet code= "banner" width=300  height=50>
 </applet>
*/
public class banner extends Applet implements Runnable
{
        String msg="hello world";
        boolean stopflag;
        Thread t;
        public void init()
        {
              setBackground(Color.white);
        }
        public void start()
        {
              stopflag=false;
              t=new Thread(this);
              t.start();
        }
        public void stop()
        {
             stopflag=true;
             t=null;
        }
        public void run()
        {
             char ch;
             for(;;)
             {
```

```
                try
                {
                    int n;
                    repaint();
                    t.sleep(100);
                    ch=msg.charAt(0);
                    msg=msg.substring(1,msg.length());
                    msg=msg+ch;
                    if(stopflag)
                       break;
                }
                catch(Exception e){}
            }
        }
        public void paint(Graphics g)
        {
            Font f=new Font("ARIAL",Font.BOLD,50);
            g.setFont(f);
            g.drawString(msg,10,100);
        }
}
```

Notice that Banner extends Applet, as expected, but it also implements Runnable. This is necessary since the applet will be creating a second thread of execution that will be used to scroll the banner. The message that will be scrolled in the banner is contained in the String variable msg. A reference to the thread that runs the applet is stored in t. The Boolean variable stopFlag is used to stop the applet. Inside init( ), the thread reference variable t is set to null. The run-time system calls start( ) to start the applet running. Inside start( ), a new thread of execution is created and assigned to the Thread variable t. Then, stopFlag is set to false. Next, the thread is started by a call to t.start( ). t.start( ) calls a method defined by Thread, which causes run( ) to begin executing.

In run( ), a call to repaint( ) is made. This eventually causes the paint( ) method to be called, and the rotated contents of msg are displayed. Between each iteration, run( ) sleeps for a quarter of a second. The net effect of run( ) is that the contents of msg are scrolled right to left in a constantly moving display. The stopFlag variable is checked on each iteration. When it is true, the run( ) method terminates. If a browser is displaying the applet when a new page is viewed, the stop( ) method is called, which sets stopFlag to true, causing run( ) to terminate. It also sets t to null. Thus, there is no longer a reference to the Thread object, and it can be recycled the next time the garbage collector runs. This is the mechanism used to stop the thread when its page is no longer in view. When the applet is brought back into view, start( ) is once again called, which starts a new thread to execute the banner. Inside paint( ), the message is rotated and then displayed.

11.4.2 Using the Status Window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call showStatus( ), which is defined by Applet, with the string that you want displayed. The general form of showStatus( ) is shown here:

void showStatus(String msg)

Here, msg is the string to be displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet. The following applet demonstrates showStatus( ):

```
/ / using the Status window.
import java.awt.*;
import java.applet.*;
/*
<applet code= “StatusWindow” width=300 height=50>
</applet>
*/
 public class StatusWindow extends Applet {
   / / displaying msg in applet window.
   public void paint(Graphics g) {
     g.drawString (“This is in the applet window.”, 10, 20);
     showStatus (“This is shown in the status window.”);
   }
 }
```

11.5 Passing parameters to Applet

We can supply user-defined parameters to an applet using <PARAM ... > tags. Each <PARM… > tag has a name attribute such as color, and a value attribute such as red. Inside the applet code, the applet can refer to that parameter by name to find its value. For example, we can change the colour of the text displayed to red by an applet by using a < PARAM... > tag as follows.

```
<APPLET ..... >
<PARAM = color VALUE = "red” >
</APPLET>
```

Similarly, we can change the text to be displayed by an applet by supplying new text to the applet through a <PARAM ... >tag as shown below:

```
<PARAM NAME = text VALUE = ”I love Java “>
```

Passing parameters to an applet code using <PARAM>tag is something similar to passing parameters to the main( ) method using command line arguments. To set up and handle parameters, we need to do two things:

1. include appropriate <PARAM .... > tags in the HTML document.
2. Provide Code in the applet to parse these parameters.

Parameters are passed to an applet when it is loaded. We can define the init( ) method in the applet to get hold of the parameters defined in the <PARM> tags. This is done using the getParameter( ) method., which takes one string argument representing the name of the parameter and returns a string containing the value of that parameter.

Applet HellojavaParm

```
import java.awt.*;
import java.applet.*;
public class HelloJavaParam extends Applet
{
        String str;
        public void init( )
        {
                str = getParameter ("string“);        //Receiving parameter value
                if (str = = nu1l)
                        str=”Java”;
                str= “Hello” + str                    //Using the value
        }
        public void paint (Graphics g)
        {
                g.drawString(str, 10, 100);
        }
}
```

Now we have to create HTML file that contains this applet. below program shows a web page that passes a parameter whose name is "string" and whose VALUE is "Applet!" to the applet HelloJavaParam.

The HTML file for HelloJavaParam applet

```
<HTML>
        <! Parameterized HTML file>
        <HEAD>
        <TITLE> Welcome to Java Applets </TITLE>
        <HEAD>
        <BODY>
                <APPLET CODE = HelloJavaParam.class
                        WIDTH = 400
                        HEIGHT = 200>
                <PARAM NAME = "string"
                        VALUE = "Applet! ">
                </APPLET>
        </BODY>
</HTML>
```

This will produce output as Hello Applet. if we remove <param> tag from HTML file will produce output as Hello Java.

11.6 The Applet Class

All applets are subclasses of the Applet class. Applet inherits the following superclasses defined by the AWT: Component, Container, and Panel. Thus, an applet has access to the full functionality of the AWT. Applet contains several methods that give detailed control over the execution of your applet.

1. void destroy ( ) –Called by the browser just before an applet is terminated. Applet will override this method if it needs to perform any cleanup prior to its destruction.
2. AccessibleContext getAccessibleContext ( )- Returns the accessibility context for the invoking object.
3. AppletContext getAppletContext ( )-Returns the context associated with the applet.
4. String getAppletInfo ( )-Returns a string that associated with the applet.
5. AudioClip getAudioClip(URL url)-Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
6. AudioClip getAudioClip(URL url, String clipName) – Returns an AudioClip object that encapsulates the audio clip found at the location specified by url and having the name specified by clipName.
7. URL getCodeBase ( )- Returns the URL associated with the invoking applet.
8. URL getDocumentBase( )- Returns the URL of the HTML document that invokes the applet.
9. image getImage (URL url)- Return an image object that encapsulates the image found at location specified by url
10. image getImage (URL url, String imageName) - Return an image object that encapsulates the image found at location specified by url and having the name specified by imageName.
11. Locale getLocale ( ) –Returns a Locale object that is used by various locale-sensitive classes and methods
12. String getParameter(String paramName)-Returns the parameter associated with paramName. Null is returned if the specified parameter is not found.
13. String[ ] [ ] getParameterInfo( )- Overrides of this method should return a String table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/ or range, and an explanation of its purpose. The default implementation returns null.
14. void init ( ) – This method is called when an applet begins execution. It is the first method called for any applet.
15. boolean isActive ( )-Returns t rue if the applet has been started. It returns false if the applet has been stopped.
16. boolean isValidateRoot ( )- Returns true, which indicates that an applet is a validate root. (Added by JDk7.)
17. static final AudioClip new AudioClip (URl url)- Returns an AudioClip object that hat encapsulates the audio clip found at the location specified by url. This method is similar to getAudioClip ( ) except that it is static and can be executed without the need for an Applet object.
18. void play(URL url)- if an audio clip is found at the location specified by URl, the clip is played.
19. void play(URL url, String clipName)- if an audio clip is found at the location specified by URl with the name specifies by clipName, the clip is played.
20. void resize (Dimension dim)- Resizes the applet according to the dimension specified by dim. Dimesnsion is a class stored inside java.awt. It contains two integer fields: width and height.
21. void resize (int width, int height)-Resizes the applet according to the dimension specified by width and height.

22. Final void setStub(AppletStub stubObj)- Makes stubObj the stub for the applet. This method is used by runtime system and is not usually called by your applet. A stub is a small piece of code that provides the linkage between your applet and the browser.
23. void showStatus(String str)- Displays str in the status window of the browser or applet viewer. If the browser does not support a stuts window, then no action taken place.
24. void start ( )- Called by the browser when an applet should start (or resume) execution. It is automatically called after init ( ) when an applet first begins.
25. void stop ( ) - Called by the browser to suspend execution of an applet. Once stopped, an applet is restarted when browser calls start ( ).

11.7 Event Handling

In Java, GUI programs, such as applets, are event driven. Thus, event handling is at the core of successful GUI programming. Most events to which your program will respond are generated by the user. These events are passed to your program in a variety of ways, with the specific method depending upon the actual event. There are several types of events, including those generated by the mouse, the keyboard, and various controls, such as a push button. AWT-based events are supported by the java.awt.event package.

11.7.1 The delegation Event Model

The modern approach to handling events is based on the delegation event model. The delegation event model defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification.

Event: In the delegation model, an event is an object that describes a state change in a source. Among other reasons, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface, such as pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sources: An event source is an object that generates an event. A source must register listeners in order for the listener to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form

public void addType Listener (Type Listener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeType Listener (Type Listener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, to remove a keyboard listener, you would call removeKeyListener( ). The methods that add or

remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners: A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process AWT events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines methods that receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Classes: The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events. The class AWTEvent, defined within the java.awt package, is a subclass of EventObject. It is the superclass (either directly or indirectly) for all AWT-based events used by the delegation event model. The package java.awt.event defines several types of events that are generated by various user interface elements. Following are the most commonly used event classes.

1. ActionEvent-Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
2. AdjustmentEvent-Generated when a scroll bar is manipulated.
3. ComponentEvent-Generated when a component is hidden, moved, resized or becomes visible.
4. ContainerEvent-Generated when a component is added to or removed from a container.
5. FocusEvent-Generated when a component gains or loses keyboard focus.
6. InputEvent-Abstract superclass for all component input event classes.
7. ItemEvent-Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
8. KeyEvent-Generated when input is received from the keyboard.
9. MouseEvent-Generated when the mouse is dragged or moved, clicked, pressed, or released; also generated when the mouse enters or exists a component.
10. MouseWheelEvent-Generated when the mouse wheel is moved.
11. TextEvent-Generated when the value of a text area or text field is changed.
12. WindowEvent-Generated when the window is activated, closed, deactivated, deiconified, iconified, opened or quit.

Event Listener Interfaces: Event listeners receive event notifications. Listeners for AWT-based events are created by implementing one or more of the interfaces defined by the java.awt.event package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Following are the most commonly used listener interfaces.

1. ActionListener-Defines one method to receive action events. Action events are generated by such things as push buttons and menus.

2. AdjustmentListener-Definesone method to receive adjustment events, such as those produces by a scroll bar.
3. ComponentListener-Defines four methods to recognize when a component is hidden, moved, resized, or shown.
4. ContainerListener-Defines two methods to recognize when a component is added or removed from a container.
5. FocusListener- Defines two methods to recognize when a component gains or loses keyboard focus.
6. ItemListener-Defines one method to recognize when the state of an item changes. An item event is generated by a check box, for example.
7. KeyListener-Defines three methods to recognize when a key is pressed, released, or typed.
8. MouseListener-Defines five methods to recognize when the mouse is clicked, enters a component, exists a component, is pressed, or is released.
9. MouseMotionListener-Defines two methods to recognize when the mouse is dragged or moved.
10. MouseWheelListener-Defines one method to recognize when the mouse wheel is moved.
11. TextListener-Defines one method to recognize when a text value changes.
12. WindowListener-Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

11.7.2 Using the Delegation Event Model

Applet programming using the delegation event model is actually quite easy. Just follow these two steps: Implement the appropriate interface in the listener so that it will receive the type of event desired. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications. Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

Handling Mouse and Mouse Motion Events: To handle mouse and mouse motion events, you must implement the MouseListener and the MouseMotionListener interfaces. The MouseListener interface defines five methods. If a mouse button is clicked, mouseClicked( ) is invoked. When the mouse enters a component, the mouseEntered( ) method is called. When it leaves, mouseExited( ) is called. The mousePressed() and mouseReleased( ) methods are invoked when a mouse button is pressed and released, respectively. The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

The MouseMotionListener interface defines two methods. The mouseDragged( ) method is called multiple times as the mouse is dragged. The mouseMoved( ) method is called multiple times as the mouse is moved. Their general forms are shown here:

```
void mouseDragged(MouseEvent me)
```

void mouseMoved(MouseEvent me)

The MouseEvent object passed in me describes the event. MouseEvent defines a number of methods that you can use to get information about what happened. Possibly the most commonly used methods in MouseEvent are getX( ) and getY( ). These return the X and Y coordinates of the mouse (relative to the window) when the event occurred. Their forms are shown here:

int getX( )
int getY( )

11.8 More Java Keywords

Transient, volatile, instanceof, native, strictfp, assert, these keywords are most often used in programs more advanced purposes.

The transient and volatile Modifiers: The transient and volatile keywords are type modifiers that handle somewhat specialized situations. When an instance variable is declared as transient, then its value need not persist when an object is stored. Thus, a transient field is one that does not affect the state of an object. The volatile modifier tells the compiler that a variable can be changed unexpectedly by other parts of your program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads will share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable, possibly in a register of the CPU. The real (or master) copy of the variable is updated at various times, such as when a synchronized method is entered.

instanceof: Sometimes it is useful to know the type of an object during run time. For example, you might have one thread of execution that generates various types of objects and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object's type at run time is important involves casting. The instanceof operator has this general form:

objref instanceof type

Here, objref is a reference to an instance of a class, and type is a class or interface type. If objref is of the specified type or can be cast into the specified type, then the instanceof operator evaluates to true. Otherwise, its result is false. Thus, instanceof is the means by which your program can obtain run-time type information about an object.

strictfp: When Java-2 was released, the floating-point computation model was relaxed slightly. Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation. This prevents overflow or underflow in some cases. By modifying a class, method, or interface with strictfp, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java. When a class is modified by strictfp, all of the methods in the class are also strictfp automatically.

assert: The assert keyword is used during program development to create an assertion, which is a condition that is expected to be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an assert statement. At run time, if

the condition actually is true, no other action takes place. However, if the condition is false, then an AssertionError is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code. The assert keyword has two forms. The first is shown here:

assert condition;

Here, condition is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default AssertionError object is thrown.

The second form of assert is shown here:

assert condition: expr;

In this version, expr is a value that is passed to the AssertionError constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for expr, but any non-void expression is allowed as long as it defines a reasonable string conversion. To enable assertion checking at run time, you must specify the -ea option. Assertions are quite useful during development because they streamline the type of error checking that is common during testing. But be careful you must not rely on an assertion to perform any action actually required by the program.

Native Methods: Although rare, there may occasionally be times when you will want to call a subroutine that is written in a language other than Java. Typically, such a subroutine will exist as executable code for the CPU and environment in which you are working that is, native code. For example, you may wish to call a native code subroutine in order to achieve faster execution time. Or you may want to use a specialized, third-party library, such as a statistical package. However, since Java programs are compiled to bytecode, which is then interpreted (or compiled on the fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Java provides the native keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method. To declare a native method, precede the method with the native modifier, but do not define any body for the method. For example:

public native int meth ( );

Once you have declared a native method, you must provide the native method and follow a rathercomplex series of steps in order to link it with your Java code.

11.9 Assignment-11

Short Answer Questions (each carries 2 marks)

1. What is an applet?
2. List two types of Applets available in Java.
3. What is the purpose of appletviewer?
4. Write HTML file to run an applet program named Simple.
5. List any two features of Applets.
6. What is the purpose of paint() method?
7. List any two differences between applets and stand alone application. (M.U. Oct/Nov. 2009)
8. List any four methods associated with Applets.
9. What is the purpose of repaint ( ) method?
10. Differentiate stop() and destroy() methods.

11. What is the purpose of update( ) method.
12. Write the purpose of showStatus( ) method.
13. List the superclasses of Applet defined by AWT.
14. List any 4 methods defined by Applet.
15. What is an event? Give example.
16. What is a listener?
17. What is the purpose of event Listeners?
18. What is the purpose of MouseListener Interface?
19. List out any two commonly used Event Listener Interfaces with its purpose.
20. List any four methods of MouseListener Interface.
21. What is the purpose of transient modifier?
22. What is the purpose of volatile modifier?
23. What is the purpose of assert keyword?
24. What is the purpose of strictfp?
25. What is the purpose of native keyword?
26. What are native methods?
27. Write the syntax. of <applet> tag and also write the purpose compulsory attributes. (M.U. Oct/Nov.2014)

Long Answer Questions

1. How does Applet differ from stand-alone program? (5 Marks)
2. Explain the complete Applet skeleton. (5 Marks)
3. Explain the Applet architecture. (5 Marks)
4. Write a short note on Applet Initialization and Termination. (5 Marks)
5. Explain about requesting repaint ( ) method. (5 Marks)
6. With an exampld explain passing parameters to an applet. (5 Marks- M.U. Oct/Nov. 2014)
7. Write a short note on using status window. (5 Marks)
8. List and explain any 6 methods defined by Applet. (6 Marks)
9. List and explain the components of Delegation Event Model. (6 marks)
10. What is the purpose of MouseListener interface? With the syntax and example explain any five methods of MouseListener interface. (6 Marks- M.U. Oct/Nov. 2014)
11. List and explain different Event Listener Interfaces (6Marks)
12. Write a short note on Handling Mouse and Mouse Motion Events. (5 Marks)
13. Explain any six java keywords used for advanced purposes. (6 Marks)
14. With syntax and example explain the purpose of showStatus( ) method. (5 Marks- M.U.Oct/Nov.2014)

UNIT-IV
Chapter 12
Using AWT controls, Layout managers and menus

12.1 Control Fundamentals

Controls are components that allow a user to interact with your application in various ways for example, a commonly used control is the push button. A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them. The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text editing

These controls are subclasses of Component.

Adding and Removing Controls: To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling add( ), which is defined by Container. The add( ) method has several forms. The following form is the one that is used for the first part of this chapter:

Component add(Component compObj)

Here, compObj is an instance of the control that you want to add. A reference to compObj is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call remove( ). This method is also defined by Container. It has this general form:

void remove(Component obj)

Here, obj is a reference to the control you want to remove. We can remove all controls by calling removeAll( ).

Responding to Controls: Except for labels, which are passive, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, a program simply implements the appropriate interface and then registers an event listener for each control that we need to monitor.

The HeadlessException: Mos of the AWT controls have constructors that can throw a HeadlessException when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present).

12.1.1 Labels

The easiest control to use is a label. Alabel is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. Label defines the following constructors:

Label( ) throws HeadlessException
Label(String str) throws HeadlessException
Label(String str, int how) throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by str. This string is left-justified. The third version creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER. You can set or change the text in a label by using the setText( ) method. You can obtain the current label by calling getText( ). These methods are shown here:

void setText(String str)
String getText( )

For setText( ), str specifies the new label. For getText( ), the current label is returned. You can set the alignment of the string within the label by calling setAlignment( ). To obtain the current alignment, call getAlignment( ). The methods are as follows:

void setAlignment(int how)
int getAlignment( )

Here, *how* must be one of the alignment constants. The following example creates three labels and adds them to an applet window.

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet {
     public void init( ) {
          Label one = new Label("One");
          Label two = new Label("Two");
          Label three = new Label("Three");
          // add labels to applet window
          add(one);
          add(two);
          add(three);
       }
 }
```

The labels are organized in the window by the default layout manager.

12.1.2 Buttons

Perhaps the most widely used control is the push button. A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors:

Button( ) throws HeadlessException
Button(String str) throws HeadlessException

The first version creates an empty button. The second creates a button that contains str as a label. After a button has been created, you can set its label by calling setLabel( ). You can retrieve its label by calling getLabel( ). These methods are as follows:

```
void setLabel(String str)
String getLabel( )
```

Here, str becomes the new label for the button.

Handling Buttons: Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the ActionListener interface. That interface defines the actionPerformed( ) method, which is called when an event occurs. An ActionEvent object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the action command string associated with the button.
By default, the action command string is the label of the button. Usually, either the button reference or the action command string can be used to identify the button.

Here is an example that creates three buttons labeled “Yes”, “No”, and “Undecided”. Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the getActionCommand( ) method on the ActionEvent object passed to actionPerformed( ).

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener {
      String msg = "";
      Button yes, no, maybe;
      public void init() {
      yes = new Button("Yes"
      no = new Button("No");
      maybe = new Button("Undecided");
      add(yes);
      add(no);
      add(maybe);
      yes.addActionListener(this);
      no.addActionListener(this);
      maybe.addActionListener(this);
   }

    public void actionPerformed(ActionEvent ae) {
        String str = ae.getActionCommand();
```

```
        if(str.equals("Yes")) {
            msg = "You pressed Yes.";
        }
        else if(str.equals("No")) {
            msg = "You pressed No.";
        }
        else {
              msg = "You pressed Undecided.";
            }
       repaint( );
     }
     public void paint(Graphics g) {
       g.drawString(msg, 6, 100);
     }
}
```

12.1.3 Check Boxes

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class. Checkbox supports these constructors:

Checkbox( ) throws HeadlessException
Checkbox(String str) throws HeadlessException
Checkbox(String str, boolean on) throws HeadlessException
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by str. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null. The value of on determines the initial state of the check box.

To retrieve the current state of a check box, call getState( ). To set its state, call setState( ). You can obtain the current label associated with a check box by calling getLabel( ). To set the label, call setLabel( ). These methods are as follows:

boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)

Here, if on is true, the box is checked. If it is false, the box is cleared. The string passed in str becomes the new label associated with the invoking check box.

Handling Checkboxes: Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item

event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection)

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener {
       String msg = "";
       Checkbox winXP, winVista, solaris, mac;
       public void init() {
       winXP = new Checkbox("Windows XP", null, true);
       winVista = new Checkbox("Windows Vista");
       solaris = new Checkbox("Solaris");
       mac = new Checkbox("Mac OS");
       add(winXP);
       add(winVista);
       add(solaris);
       add(mac);
       winXP.addItemListener(this);
       winVista.addItemListener(this);
       solaris.addItemListener(this);
       mac.addItemListener(this);
     }
 public void itemStateChanged(ItemEvent ie) {
    repaint();
}
   // Display current state of the check boxes.
   public void paint(Graphics g) {
      msg = "Current state: ";
      g.drawString(msg, 6, 80);
      msg = "  Windows XP: " + winXP.getState();
      g.drawString(msg, 6, 100);
      msg = "  Windows Vista: " + winVista.getState();
      g.drawString(msg, 6, 120);
      msg = "  Solaris: " + solaris.getState();
      g.drawString(msg, 6, 140);
      msg = "  Mac OS: " + mac.getState();
```

```
    g.drawString(msg, 6, 160);
  }
}
```

12.1.4 Checkbox Group

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type CheckboxGroup. Only the default constructor is defined, which creates an empty group. You can determine which check box in a group is currently selected by calling getSelectedCheckbox( ). You can set a check box by calling setSelectedCheckbox( ). These methods are as follows:

Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox which)

Here, which is the check box that you want to be selected. The previously selected check box will be turned off. Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/
  public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox winXP, winVista, solaris, mac;
    CheckboxGroup cbg;
    public void init() {
        cbg = new CheckboxGroup();
    winXP = new Checkbox("Windows XP", cbg, true);
    winVista = new Checkbox("Windows Vista", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);
    add(winXP);
    add(winVista);
    add(solaris);
    add(mac);
    winXP.addItemListener(this);
    winVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }
  public void itemStateChanged(ItemEvent ie) {
     repaint();
```

```
  }
 // Display current state of the check boxes.
      public void paint(Graphics g) {
      msg = "Current selection: ";
      msg += cbg.getSelectedCheckbox().getLabel();
      g.drawString(msg, 6, 100);
  }
}
```

12.1.4 Choice Controls

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. When inactive, a Choice component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines the default constructor, which creates an empty list. To add a selection to the list, call add( ). It has this general form:

```
void add(String name)
```

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur. To determine which item is currently selected, you may call either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

```
String getSelectedItem( )
int getSelectedIndex( )
```

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

```
int getItemCount( )
void select(int index)
void select(String name)
```

Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

```
String getItem(int index)
```

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method. Here is an example that creates two Choice menus. One selects the operating system.
The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
      Choice os, browser;
      String msg = " ";
      public void init() {
      os = new Choice( );
      browser = new Choice();
      // add items to os list
      os.add("Windows XP");
      os.add("Windows Vista");
      os.add("Solaris");
      os.add("Mac OS");
      // add items to browser list
      browser.add("Internet Explorer");
      browser.add("Firefox");
      browser.add("Opera");
      // add choice lists to window
      add(os);
      add(browser);
     // register to receive item events
     os.addItemListener(this);
     browser.addItemListener(this);
  }
   public void itemStateChanged(ItemEvent ie) {
      repaint();
   }
   // Display current selections.
    public void paint(Graphics g) {
         msg = "Current OS: ";
         msg += os.getSelectedItem();
         g.drawString(msg, 6, 120);
         msg = "Current Browser: ";
         msg += browser.getSelectedItem();
         g.drawString(msg, 6, 140);
    }
}
```

12.1.6 Lists

The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. List provides these constructors:

List( ) throws HeadlessException
List(int numRows) throws HeadlessException

List(int numRows, boolean multipleSelect) throws HeadlessException

The first version creates a List control that allows only one item to be selected at any one time. In the second form, the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if multipleSelect is true, then the user may select two or more items at a time. If it is false, then only one item may be selected. To add a selection to the list, call add( ). It has the following two forms:

void add(String name)
void add(String name, int index)

Here, name is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by index. Indexing begins at zero. You can specify –1 to add the item to the end of the list. For lists that allow only single selection, you can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, null is returned. getSelectedIndex( ) returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

getSelectedItems( ) returns an array containing the names of the currently selected items. getSelectedIndexes( ) returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item by using the select( ) method with a zero-based integer index. These methods are shown here:

int getItemCount( )
void select(int index)

Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

String getItem(int index)

Here, index specifies the index of the desired item.

To process list events, you will need to implement the ActionListener interface. Each time a List item is double-clicked, an ActionEvent object is generated. Its getActionCommand( ) method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an ItemEvent object is generated. Its

getStateChange( )

method can be used to determine whether a selection or deselection triggered this event. getItemSelectable( ) returns a reference to the object that triggered this event. Here is an example that converts the Choice controls in the preceding section into List components, one multiple choice and the other single choice

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
```

```
</applet>
*/
public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";
    public void init() {
    os = new List(4, true);
    browser = new List(4, false);
    // add items to os list
    os.add("Windows XP");
    os.add("Windows Vista");
    os.add("Solaris");
    os.add("Mac OS");
   // add items to browser list
   browser.add("Internet Explorer");
   browser.add("Firefox");
   browser.add("Opera");
   browser.select(1);
   // add lists to window
   add(os);
   add(browser);
   // register to receive action events
   os.addActionListener(this);
    browser.addActionListener(this);
  }
   public void actionPerformed(ActionEvent ae) {
      repaint();
  }
    // Display current selections.
  public void paint(Graphics g) {
      int idx[ ];
      msg = "Current OS: ";
      idx = os.getSelectedIndexes();
      for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + "  ";
      g.drawString(msg, 6, 120);
      msg = "Current Browser: ";
      msg += browser.getSelectedItem();
      g.drawString(msg, 6, 140);
   }
}
```

12.1.7 Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of

the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the Scrollbar class. Scrollbar defines the following constructors:

```
Scrollbar( ) throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
throws HeadlessException
```

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If style is Scrollbar.VERTICAL, a vertical scroll bar is created. If style is Scrollbar.HORIZONTAL, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in initialValue. The number of units represented by the height of the thumb is passed in thumbSize. The minimum and maximum values for the scroll bar are specified by min and max.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using setValues( ), shown here, before it can be used:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described. To obtain the current value of the scroll bar, call getValue( ). It returns the current setting. To set the current value, call setValue( ). These methods are as follows:

```
int getValue( )
void setValue(int newValue)
```

Here, newValue specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value. You can also retrieve the minimum and maximum values via getMinimum( ) and getMaximum( ), shown here:

```
int getMinimum( )
int getMaximum( )
```

They return the requested quantity. By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling setUnitIncrement( ).

By default, page-up and page-down increments are 10. You can change this value by calling setBlockIncrement( ). These methods are shown here:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

To process scroll bar events, you need to implement the AdjustmentListener interface. Each time a user interacts with a scroll bar, an AdjustmentEvent object is generated. Its getAdjustmentType( ) method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

```
BLOCK_DECREMENT- A page-down event has been generated.
BLOCK_INCREMENT -A page-up event has been generated.
TRACK -An absolute tracking event has been generated.
UNIT_DECREMENT- The line-down button in a scroll bar has been pressed.
```

UNIT_INCREMENT-The line-up button in a scroll bar has been pressed.

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
  public class SBDemo extends Applet
    implements AdjustmentListener, MouseMotionListener {
      String msg = " ";
      Scrollbar vertSB, horzSB;
      public void init() {
          int width = Integer.parseInt(getParameter("width"));
          int height = Integer.parseInt(getParameter("height"));
         vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
          horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
          add(vertSB);
          add(horzSB);
          // register to receive adjustment events
          vertSB.addAdjustmentListener(this);
          horzSB.addAdjustmentListener(this);
          addMouseMotionListener(this);
       }
      public void adjustmentValueChanged(AdjustmentEvent ae) {
         repaint( );
    }
   // Update scroll bars to reflect mouse dragging.
   public void mouseDragged(MouseEvent me) {
     int x = me.getX();
     int y = me.getY();
     vertSB.setValue(y);
     horzSB.setValue(x);
     repaint();
  }
   // Necessary for MouseMotionListener
       public void mouseMoved(MouseEvent me) {
     }
  // Display current value of scroll bars.
      public void paint(Graphics g) {
           msg = "Vertical: " + vertSB.getValue();
           msg += ",  Horizontal: " + horzSB.getValue( );
           g.drawString(msg, 6, 160);
    // show current mouse drag position
         g.drawString("*", horzSB.getValue(),
         vertSB.getValue());
```

```
  }
 }
```

12.1.8 TextField

The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. TextField is a subclass of TextComponent. TextField defines the following constructors:

TextField( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException

The first version creates a default text field. The second form creates a text field that is numChars characters wide. The third form initializes the text field with the string contained in str. The fourth form initializes a text field and sets its width.

TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call getText( ). To set the text, call setText( ). These methods are as follows:

String getText( )
void setText(String str)

Here, str is the new string. The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using select( ). Your program can obtain the currently selected text by calling getSelectedText( ). These methods are shown here:

String getSelectedText( )
void select(int startIndex, int endIndex)

getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at startIndex and ending at endIndex–1.

You can control whether the contents of a text field may be modified by the user by calling setEditable( ). You can determine editability by calling isEditable( ). These methods are shown here:

boolean isEditable( )
void setEditable(boolean canEdit)

isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( ) method. You can retrieve the echo character by calling the getEchoChar( ) method. These methods are as follows:

void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )

Here, ch specifies the character to be echoed. Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs,

an action event is generated. Here is an example that creates the classic user name and password screen:

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
    public class TextFieldDemo extends Applet
        implements ActionListener {
            TextField name, pass;
            public void init() {
                Label namep = new Label("Name: ", Label.RIGHT);
                Label passp = new Label("Password: ", Label.RIGHT);
                name = new TextField(12);
                pass = new TextField(8);
                pass.setEchoChar('?');
                add(namep);
                add(name);
                add(passp);
                add(pass);
               // register to receive action events
                    name.addActionListener(this);
                    pass.addActionListener(this);
                }
     // User pressed Enter.
     public void actionPerformed(ActionEvent ae) {
            repaint( );
     }
        public void paint(Graphics g) {
            g.drawString("Name: " + name.getText(), 6, 60);
          g.drawString("Selected text in name: " + name.getSelectedText(), 6, 80);
          g.drawString("Password: " + pass.getText(), 6, 100);
      }
}
```

12.1.9 TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea. Following are the constructors for TextArea:

```
TextArea( ) throws HeadlessException
TextArea(int numLines, int numChars) throws HeadlessException
TextArea(String str) throws HeadlessException
TextArea(String str, int numLines, int numChars) throws HeadlessException
```

TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException

Here, numLines specifies the height, in lines, of the text area, and numChars specifies its width, in characters. Initial text can be specified by str. In the fifth form, you can specify the scroll bars that you want the control to have. sBars must be one of these values: SCROLLBARS_BOTH, SCROLLBARS_NONE, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_VERTICAL_ONLY.

TextArea is a subclass of TextComponent. Therefore, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ), and setEditable( ) methods described in the preceding section. TextArea adds the following methods

void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)

The append( ) method appends the string specified by str to the end of the current text. insert( ) method inserts the string passed in str at the specified index. To replace text, call replaceRange( ).It replaces the characters fromstartIndex to endIndex–1, with the replacement text passed in str. Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Text areas only generate got-focus and lost-focus events.

Normally, your program simply obtains the current text when it is needed. The following program creates a TextArea control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
  public class TextAreaDemo extends Applet {
       public void init() {
          String val =
                "Java SE 6 is the latest version of the most\n" +
                "widely-used computer language for Internet programming.\n" +
                "Building on a rich heritage, Java has advanced both\n" +
                "the art and science of computer language design.\n\n" +
                "One of the reasons for Java's ongoing success is its\n" +
                "constant, steady rate of evolution. Java has never stood\n" +
                "still. Instead, Java has consistently adapted to the\n" +
                "rapidly changing landscape of the networked world.\n" +
                "Moreover, Java has often led the way, charting the\n" +
                "course for others to follow.";
          TextArea text = new TextArea(val, 10, 30);
          add(text);
       }
   }
```

12.2 Layout Managers

Layout manager automatically arranges controls within a window by using some type of algorithm. If you have programmed for other GUI environments, such as Windows, then you are accustomed to laying out your controls by hand. While it is possible to lay out Java controls by hand, too,you generally won't want to, for two main reasons. First, it is very tedious to manually layout a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized.

Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it. The setLayout( ) method has the following general form:

void setLayout(LayoutManager layoutObj)

Here, layoutObj is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for layoutObj. If you do this, you will need to determine the shape and position of each component manually, using the setBounds( ) method defined by Component. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container.Whenever the container needs to be resized, the layout manager is consulted via its minimumLayoutSize( ) and preferredLayoutSize( ) methods. Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ) methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise.

12.2.1 FlowLayout

FlowLayout is the default layout manager. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for FlowLayout:

FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for how are as follows:

FlowLayout.LEFT
FlowLayout.CENTER

FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. Here is a version of the CheckboxDemo applet shown earlier in this chapter, modified so that it uses left-aligned flow layout:

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=250 height=200>
</applet>
*/
  public class FlowLayoutDemo extends Applet
      implements ItemListener {
        String msg = "";
        Checkbox winXP, winVista, solaris, mac;
        public void init( ) {
        // set left-aligned flow layout
        setLayout(new FlowLayout(FlowLayout.LEFT));
        winXP = new Checkbox("Windows XP", null, true);
        winVista = new Checkbox("Windows Vista");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        add(winXP);
        add(winVista);
        add(solaris);
        add(mac);
        // register to receive item events
        winXP.addItemListener(this);
        winVista.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
        }
        // Repaint when status of a check box changes.
        public void itemStateChanged(ItemEvent ie) {
        repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = "  Windows XP: " + winXP.getState();
        g.drawString(msg, 6, 100);
```

```
        msg = "  Windows Vista: " + winVista.getState();
        g.drawString(msg, 6, 120);
        msg = "  Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = "  Mac: " + mac.getState();
        g.drawString(msg, 6, 160);
        }
        }
```

12.2.2 BorderLayout

The BorderLayout class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by BorderLayout:

```
BorderLayout( )
BorderLayout(int horz, int vert)
```

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. BorderLayout defines constants that specify the regions as BorderLayout.CENTER, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, BorderLayout.NORTH.

When adding components, you will use these constants with the following form of add( ), which is defined by Container:

```
void add(Component compObj, Object region)
```

Here, compObj is the component to be added, and region specifies where the component will be added. Here is an example of a BorderLayout with a component in each layout area:

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
        public class BorderLayoutDemo extends Applet {
             public void init( ) {
               setLayout(new BorderLayout());
               add(new Button("This is across the top."),
               BorderLayout.NORTH);
               add(new Label("The footer message might go here."),
               BorderLayout.SOUTH);
               add(new Button("Right"), BorderLayout.EAST);
               add(new Button("Left"), BorderLayout.WEST);
               String msg = "The reasonable man adapts " + "himself to the world;\n" +
               "the unreasonable one persists in " +
               "trying to adapt the world to himself.\n" +
               "Therefore all progress depends " +
```

```
            "on the unreasonable man.\n\n" + " - George Bernard Shaw\n\n";
             add(new TextArea(msg), BorderLayout.CENTER);
        }
    }
```

12.2.3 GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns. The constructors supported by GridLayout are shown here:

```
GridLayout( )
GridLayout(int numRows, int numColumns)
GridLayout(int numRows, int numColumns, int horz, int vert)
```

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns.

The third form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. Either numRows or numColumns can be zero. Specifying numRows as zero allows for unlimited-length columns. Specifying numColumns as zero allows for unlimited-length rows. Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
    public class GridLayoutDemo extends Applet {
        static final int n = 4;
          public void init( ) {
            setLayout(new GridLayout(n, n));
            setFont(new Font("SansSerif", Font.BOLD, 24));
            for(int i = 0; i < n; i++) {
              for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                  add(new Button("" + k));
                }
              }
            }
      }
```

12.2.4 Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes:MenuBar,Menu, andMenuItem. In general, a

menu bar contains one or more Menu objects. EachMenu object contains a list of MenuItem objects. Each MenuItem object represents something that can be selected by the user. SinceMenu is a subclass ofMenuItem, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type CheckboxMenuItem and will have a check mark next to them when they are selected. To create a menu bar, first create an instance ofMenuBar. This class only defines the default constructor. Next, create instances ofMenu that will define the selections displayed on the bar. Following are the constructors for Menu:

Menu( ) throws HeadlessException
Menu(String optionName) throws HeadlessException
Menu(String optionName, boolean removable) throws HeadlessException

Here, optionName specifies the name of the menu selection. If removable is true, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu. Individual menu items are of typeMenuItem. It defines these constructors:

MenuItem( ) throws HeadlessException
MenuItem(String itemName) throws HeadlessException
MenuItem(String itemName, MenuShortcut keyAccel) throws HeadlessException

Here, itemName is the name shown in the menu, and keyAccel is the menu shortcut for this item. You can disable or enable a menu item by using the setEnabled( ) method. Its form is shown here:

void setEnabled(boolean enabledFlag)

If the argument enabledFlag is true, the menu item is enabled. If false, the menu item is disabled. You can determine an item's status by calling isEnabled( ). This method is shown here:

boolean isEnabled( )

isEnabled( ) returns true if the menu item on which it is called is enabled. Otherwise, it returns false. You can change the name of a menu item by calling setLabel( ). You can retrieve the current name by using getLabel( ). These methods are as follows:

void setLabel(String newName)
String getLabel( )

Here, newName becomes the new name of the invoking menu item. getLabel( ) returns the current name. You can create a checkable menu item by using a subclass of MenuItem called CheckboxMenuItem. It has these constructors:

CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String itemName) throws HeadlessException
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException

Here, itemName is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if on is true, the checkable entry is initially checked. Otherwise, it is cleared. You can obtain the status of a checkable item by calling getState( ). You can set it to a known state by using setState( ). These methods are shown here:

boolean getState( )
void setState(boolean checked)

If the item is checked, getState( ) returns true. Otherwise, it returns false. To check an item, pass true to setState( ). To clear an item, pass false. Once you have created a menu item, you must add the item to a Menu object by using add( ), which has the following general form:

MenuItem add(MenuItem item)

Here, item is the item being added. Items are added to a menu in the order in which the calls to add( ) take place. The item is returned. Once you have added all items to a Menu object, you can add that object to the menu bar by using this version of add( ) defined by MenuBar:

Menu add(Menu menu)

Here, menu is the menu being added. The menu is returned. Menus only generate events when an item of type MenuItem or CheckboxMenuItem is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an ActionEvent object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling setActionCommand( ) on the menu item. Each time a check box menu item is checked or unchecked, an ItemEvent object is generated. Thus, you must implement the ActionListener and/or ItemListener interfaces in order to handle these menu events. The getItem( ) method of ItemEvent returns a reference to the item that generated this event. The general form of this method is shown here:

Object getItem( )

12.3 Assignment-12

Short Answer Questions (2 marks each)

1. What are controls?
2. List out any four types of AWT controls.
3. What is the purpose of add( ) and remove( ) methods of Container class?
4. What are HeadlessExceptions?
5. List any four types of adjustment events in cae of scroll bars.
6. What is the purpose of setText() and getText() methods?
7. What is the purpose of TextArea control?
8. What is the purpose of Layout Manager interface?
9. What is FlowLayout?
10. What are the two constructors of Button?
11. List out with syntax any four constructors of Checkboxes.
12. Differentiate between Checkbox and CheckboxGroup?
13. With syntax write different constructors of List control.
14. What are the purposes of Scroll Bar.?
15. List out different constructors of TextField with syntax.
16. What are the differences between TextField and TextArea.
17. What are the two constructors of BorderLayout?
18. What is gridLayout?
19. What are the different constructors of Menu?
20. Differentiate between menu and menu Bar.

Long Answer Questions

1. How to add and remove controls? Explain. (5 Marks)
2. Explain the purpose of HeadlessException. (4 Marks)
3. Illustrate the use of Label with suitable example. (5 Marks)
4. With an example demonstrate Label. (5 Marks)
5. List and explain various methods associated with Button. (5 Marks)
6. With an example explain handling Checkboxes. (5 Marks)
7. List and explain different methods associated with check boxes. (5 Marks)
8. With an example demonstrate Checkbox Group. (5 Marks)
9. Explain the purpose of Choice controls with an example. (5 Marks)
10. Explain List control with an example. (5 Marks)
11. Explain Scroll Bars. (5 Marks)
12. What is the purpose of TextField class? Explain any three methods of TextField class with syntax and example. (5 Marks-M.U.Oct/Nov.2014)
13. Explain about Layout Managers. (4 Marks)
14. With an example explain FlowLayout. (5 Marks)
15. With an example explain BoarderLayout (5 Marks)
16. What is the purpose of layout managers? With an example explain the use of Grid Layout. (5 Marks-M.U.Oct/Nov.2014)
17. Name the controls required for creating a menu. With an example explain the creation of a menu. (5 Marks-M.U.Oct/Nov.2014)

UNIT-IV
Chapter 13
Introducing Swing

13.1 The Origins and Design Philosophy of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as heavyweight.

The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque. Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Two Key Swing Features: Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel.
Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing.

Swing Components Are Lightweight: With very few exceptions, Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Because lightweight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel: Swing supports a pluggable look and feel (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes

possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style. Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time. Java SE 6 provides look-and-feels, such as metal and Motif, that are available to all Swing users. The metal look and feel is also called the Java look and feel. It is platform-independent and available in all Java execution environments. It is also the default look and feel. Windows environments also have access to the Windows and Windows Classic look and feel.

13.2 Components and Containers

A Swing GUI consists of two key items: components and containers. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a component is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a containment hierarchy, at the top of which must be a top-level container.

Components: In general, Swing components are derived from the JComponent class. JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component. All of Swing's components are represented by classes defined within the package javax.swing. The following are the class names for Swing components.

JApplet, JButton, JCheckBox, JCheckBoxMenuItem, JColorChooser, JComboBox, JComponent, JDesktopPane, JDialog, JEditorPane, JFileChooser, JFormattedTextField, JFrame, JInternalFrame, JLabel, JLayeredPane, JList JMenu, JMenuBar and JMenuItem etc.

Notice that all component classes begin with the letter J. For example, the class for a label is JLabel; the class for a push button is JButton; and the class for a scroll bar is JScrollBar.

Containers

Swing defines two types of containers. The first are top-level containers: JFrame, JApplet, JWindow, and JDialog. These containers do not inherit JComponent. They do, however, inherit the AWT classes Component and Container. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is

JFrame. The one used for applets is JApplet. The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit JComponent. An example of a lightweight container is JPanel, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as JPanel to create subgroups of related controls that are contained within an outer container.

Each top-level container defines a set of panes. At the top of the hierarchy is an instance of JRootPane. JRootPane is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar. The panes that comprise the root pane are called the glass pane, the content pane, and the layered pane. The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of JPanel. The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component, for example. In most cases, you won't need to use the glass pane directly, but it is there if you need it.

13.3 Layout Managers

layout manager automatically arranges controls within a window by using some type of algorithm. If you have programmed for other GUI environments, such as Windows, then you are accustomed to laying out your controls by hand. While it is possible to lay out Java controls by hand, too,you generally won't want to, for two main reasons. First, it is very tedious to manually layout a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized.

Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it. The setLayout( ) method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

Here, layoutObj is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for layoutObj. If you do this, you will need to determine the shape and position of each component manually, using the setBounds( ) method defined by Component. Normally, you will want to use a layout manager.

13.4 Use Jbutton

The JButton class provides the functionality of a push button. JButton allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Here, str and icon are the string and icon used for the button. When the button is pressed, an ActionEvent is generated. Using the ActionEvent object passed to the actionPerformed( ) method of the registered ActionListener, you can obtain the action command string associated with the button. By default, this is the string displayed inside the button. However, you can set the action

command by calling setActionCommand( ) on the button. You can obtain the action command by calling getActionCommand( ) on the event object. It is declared like this:

String getActionCommand( )

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

13.5 Work with JTextField

TextField is the simplest Swing text component. It is also probably its most widely used text component. JTextField allows you to edit one line of text. It is derived from JTextComponent, which provides the basic functionality common to Swing text components. JTextField uses the Document interface for its model.

Three of JTextField's constructors are shown here:

JTextField(int cols)

JTextField(String str, int cols)

JTextField(String str)

Here, str is the string to be initially presented, and cols is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string. JTextField generates events in response to user interaction. For example, an ActionEvent is fired when the user presses ENTER. ACaretEvent is fired each time the caret (i.e., the cursor) changes position. (CaretEvent is packaged in javax.swing.event.) Other events are also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call getText( ).

13.6 Create a JCheckBox

The JCheckBox class provides the functionality of a check box. Its immediate superclass is JToggleButton, which provides support for two-state buttons, as just described. JCheckBox defines several constructors. The one used here is

JCheckBox(String str)

It creates a check box that has the text specified by str as a label. Other constructors let you specify the initial selection state of the button and specify an icon. When the user selects or deselects a check box, an ItemEvent is generated. You can obtain a reference to the JCheckBox that generated the event by calling getItem( ) on the ItemEvent passed to the itemStateChanged( ) method defined by ItemListener. The easiest way to determine the selected state of a check box is to call isSelected( ) on the JCheckBox instance. In addition to supporting the normal check box operation, JCheckBox lets you specify the icons that indicate when a check box is selected, cleared, and rolled-over. We won't be using this capability here, but it is available for use in your own programs.

13.7 Work with Jlist

In Swing, the basic list class is called JList. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. JList is so widely used in Java that it is highly unlikely that you have not seen one before. JList provides several constructors. The one used here is

JList(Object[ ] items)

This creates a JList that contains the items in the array specified by items. JList is based on two models. The first is ListModel. This interface defines how access to the list data is achieved. The second model is the ListSelectionModel interface, which defines methods that determine what list item or items are selected. Although a JList will work properly by itself, most of the time you will wrap a JList inside a JScrollPane. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the JList component. AJList generates a ListSelectionEvent when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing ListSelectionListener. This listener specifies only one method, called valueChanged( ), which is shown here:

void valueChanged(ListSelectionEvent le)

Here, le is a reference to the object that generated the event. Although ListSelectionEvent does provide some methods of its own, normally you will interrogate the JList object itself to determine what has occurred. Both ListSelectionEvent and ListSelectionListener are packaged in javax.swing.event.

By default, a JList allows the user to select multiple ranges of items within the list, but you can change this behavior by calling setSelectionMode( ), which is defined by JList. It is shown here:

void setSelectionMode(int mode)

Here, mode specifies the selection mode. It must be one of these values defined by ListSelectionModel:

SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list.With single-interval selection, the user can select one range of items.With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling getSelectedIndex( ), shown here: int getSelectedIndex( ) Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned. Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling getSelectedValue( ):

Object getSelectedValue( )

It returns a reference to the first selected value. If no value has been selected, it returns null.

13. 8 Use anonymous inner classes to handle events

Event handling is one important part of programming in swing. Because JLabel does not take input from the user, it does not generate events, so no event handling was needed. However, the other Swing components do respond to user input and the events generated by those interactions need to be handled. Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any Swing-based application.

The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called the delegation event model. In many cases, Swing uses the same events as does the AWT, and these events are packaged in java.awt.event. Events specific to Swing are

stored in javax.swing.event. Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button. Event handling program imports both the java.awt and java.awt.event packages. The java.awt package is needed because it contains the FlowLayout class, which supports the standard flow layout manager used to lay out components in a frame.

The java.awt.event package is needed because it defines the ActionListener interface and the ActionEvent class. The EventDemo constructor begins by creating a JFrame called jfrm. It then sets the layout manager for the content pane of jfrmto FlowLayout. Recall that, by default, the content pane uses BorderLayout as its layout manager. However, for this example, FlowLayout is more convenient. Notice that FlowLayout is assigned using this statement

```
jfrm.setLayout(new FlowLayout());
```

As explained, in the past you had to explicitly call getContentPane( ) to set the layout manager for the content pane.

13.9 Create a Swing applet

The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference: ASwing applet extends JApplet rather than Applet. JApplet is derived from Applet. Thus, JApplet includes all of the functionality found in Applet and adds support for Swing. JApplet is a top-level Swing container, which means that it is not derived from JComponent. Because JApplet is a top-level container, it includes the various panes described earlier. This means that all components are added to JApplet's content pane in the same way that components are added to JFrame's content pane.

Swing applets use the same four lifecycle methods as init( ), start( ), stop( ), and destroy( ). Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the paint( ) method. All interaction with components in a Swing applet must take place on the event dispatching thread, as described in the previous section. This threading issue applies to all Swing programs.

Here is an example of a Swing applet. It provides the same functionality as the previous application, but does so in applet form.

```
// A simple Swing-based applet
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/*
This HTML can be used to launch the applet:
<object code="MySwingApplet" width=220 height=90>
</object>
*/
        public class MySwingApplet extends JApplet {
        JButton jbtnAlpha;
        JButton jbtnBeta;
        JLabel jlab;
        // Initialize the applet.
```

```
    public void init( ) {
        try {
                SwingUtilities.invokeAndWait(new Runnable ( ) {
                    public void run() {
                        makeGUI( ); // initialize the GUI
                    }
                });
            } catch(Exception exc) {
                System.out.println("Can't create because of "+ exc);
        }
    }
    // This applet does not need to override start(), stop(),
    // or destroy().
    // Set up and initialize the GUI.
    private void makeGUI( ) {
          // Set the applet to use flow layout.
          setLayout(new FlowLayout());
        // Make two buttons.
        jbtnAlpha = new JButton("Alpha");
        jbtnBeta = new JButton("Beta");
       // Add action listener for Alpha.
        jbtnAlpha.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
        jlab.setText("Alpha was pressed.");
      }
    });
    // Add action listener for Beta.
    jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
    jlab.setText("Beta was pressed.");
    }
    });
    // Add the buttons to the content pane.
    add(jbtnAlpha);
    add(jbtnBeta);
    // Create a text-based label.
    jlab = new JLabel("Press a button.");
    // Add the label to the content pane.
    add(jlab);
    }
}
```

There are two important things to notice about this applet. First, MySwingApplet extends JApplet. As explained, all Swing-based applets extend JApplet rather than Applet. Second, the init( ) method initializes the Swing components on the event dispatching thread by setting up a call to makeGUI( ). Notice that this is accomplished through the use of invokeAndWait( ) rather than invokeLater( ).

Applets must use invokeAndWait( ) because the init( ) method must not return until the entire initialization process has been completed. In essence, the start( ) method cannot be called until after initialization, which means that the GUI must be fully constructed. Inside makeGUI( ), the two buttons and label are created, and the action listeners are added to the buttons. Finally, the components are added to the content pane. Although this example is quite simple, this same general approach must be used when building any Swing GUI that will be used by an applet.

13.10 Assignment-11

Short Answer Questions (2 marks each)

1. Write the limitations of the AWT components with respect to SWING. (M.U.Oct/Nov.2014)
2. List the key items of a swing GUI .
3. What are the two key features of swing?
4. List any four layout managers supported by swing.
5. Expand PLAF and JFC.
6. What happens when the constructor JButton(String msg) is called
7. What are the two types of containers?
8. What is the purpose of Layout manager?
9. What are the three constructors of Jbutton?
10. With syntax write the three constructors of JTextfiled.
11. What re the three constant values for ListSelectionModel?
12. List out the life cycle methods of swing applet.

Long Answer Questions

1. Explain the advantages of Swing. (4 marks)
2. Explain the origin and philosophy of Swing. (6 Marks)
3. Write a short note on containers. (4 Marks)
4. Briefly explain components. (4 Marks)
5. Mention the purpose of different layout managers available in Swing.
6. Explain the purpose of JFrame and explain any four methods associated with it. (5 Marks)
7. Explain the purpose of JButton and explain any four methods associated with it. (5 Marks)
8. How to create JCheckBox explain? (5 Marks)
9. Explain the use of JTextField and any four methods associate with it. (5 Marks)
10. Explain the use of JList and any for methods associated with it. (5 Marks)
11. How to use anonymous inner classes to handle events? (5 Marks-M.U.Oct/Nov.2014)
12. Write a short note on Swing applet. (5 Marks)
13. With an example explain how to create a swing applet (5 Marks- M.U.Oct/Nov.2014)

Chapter 14
Value Added Sessions

14.1 Java Support Systems

Java and Java-enabled browsers on the Internet requires a variety of support systems. The Table 1.2 lists systems necessary to support Java for delivering information on the internet.

Support System	Description
Internet Connection	Local computer should be connected to the Internet.
Web Server	A program that accepts requests for information and sends the required documents.
Web Browser	A program that provides access to WWW and runs Java applets.
HTML	A language for creating hypertext for the Web.
APPLET Tag	For placing Java applets in HTML document.
Java Code	Java code is used for defining Java applets.
Bytecode	Compiled Java code that is referred to in the APPLET tag and transferred to the user computer.

Table 14.1: Java Support Systems

World Wide Web (WWW) is an open-ended information retrieval system designed to be used in the Internet's distributed environment. This system contains what are known as Web pages that provide both information and controls. Unlike a menu-driven system where we are guided through a particular direction using a decision tree structure, the Web system is open-ended and we can navigate to a new document in any direction. With the help of Hyper Text Markup Language (HTML) we can navigate to a new document in any direction. Web pages contain HTML tags that enable us to find, retrieve, manipulate and display documents worldwide. The incorporation of Java into Web pages has made it capable of supporting animation, graphics, games, and a wide range of special effects. With the support of Java, the Web has become more interactive and dynamic. Java communicates with a Web page through a special tag called <APPLET>.

The figure 14.1 shows following communication steps

1. The user sends a request for an HTML document to the remote computer's Web server. The Web server is a program that accepts a request, processes the request, and sends the required document.
2. The HTML document is returned to the user's browser. The document contains the APPLET tag, which identifies the applet.

3. The corresponding applet byte code is transferred to the user's computer. This bytecode had been previously created by the Java compiler using the Java source code file for that applet.
4. The Java-enabled browser on the user's computer interprets the bytecodes and provides output.
5. The user may have further interaction with the applet but with no further downloading from the provider's Web server. This is because the bytecode contains all the information necessary to interpret the applet.

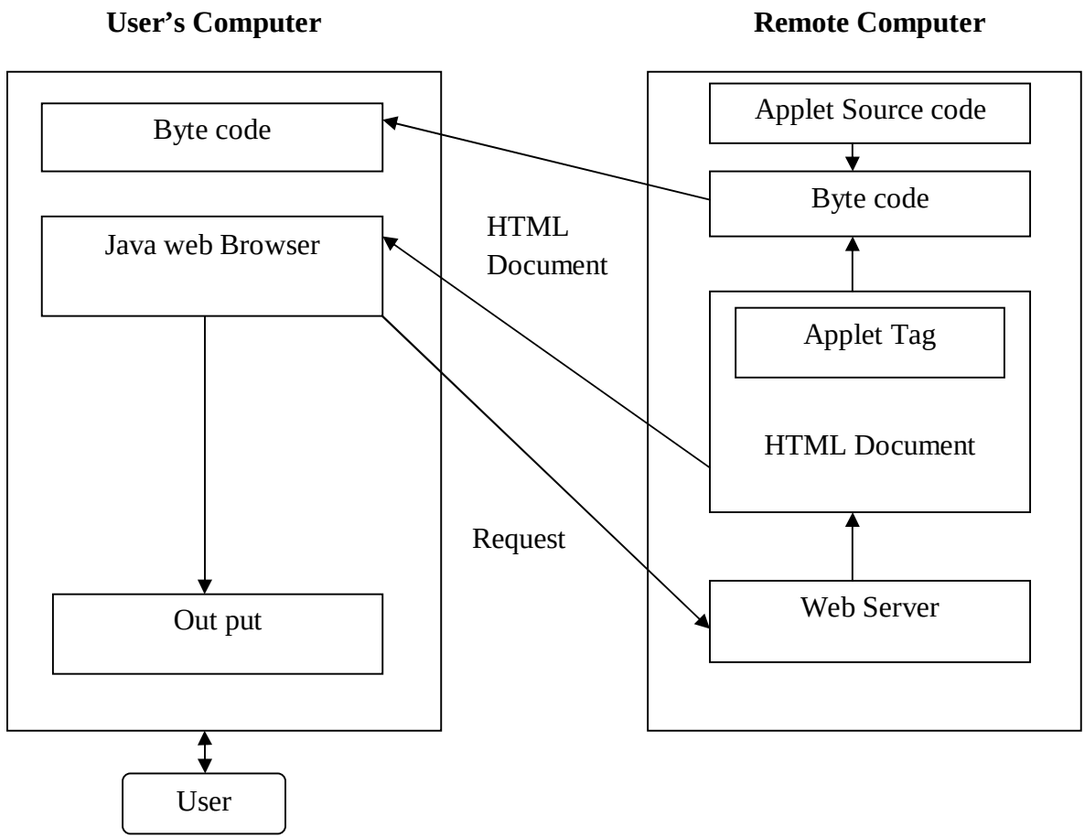

Figure 14.1: Java's interaction with web

14.2 Java Environment

Java environment includes a large number of development tools and hundreds of classes and methods he development tools are part of the system known as *Java Development Kit* (JDK) and the classes and methods are part of the *Java Standard Library* (JSL), also known as the *Application Programming Interface* (API).

Java Development Kit

The Java Development Kit comes with a collection of tools that are used for developing and running Java programs. They include:

- Ø appletviewer (for viewing Java applets)
- Ø javac (Java compiler)
- Ø java (Java interpreter)
- Ø javap (Java disassembler)
- Ø javah (for C header files)

- Ø javadoc (for creating HTML documents)
- Ø jdb (Java debugger)

appletviewer: Enables us to run Java applets (without actually using a Java-compatible browser).
java: Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
javac: The Java compiler, which translates Java source code to bytecode files that the interpreter can understand.
javadoc: Creates HTML-format documentation from Java source code files.
javah: Produces header files for use with native methods.
javap: Java disassembler, which enables us to covert bytecode files into a program description.
jdb: Java debugger, which helps us to find errors in our programs.
The way these tools are applied to build and run application programs is illustrated in Figure 14.2. To create a Java program, we need to create a source code file using a text editor. The source code is then compiled using the Java compiler javac and executed using the Java interpreter java. The Java debugger jdb is used to find errors, if any, in the source code. A compiled Java program can be converted into a source code with the help of Java disassembler javap.

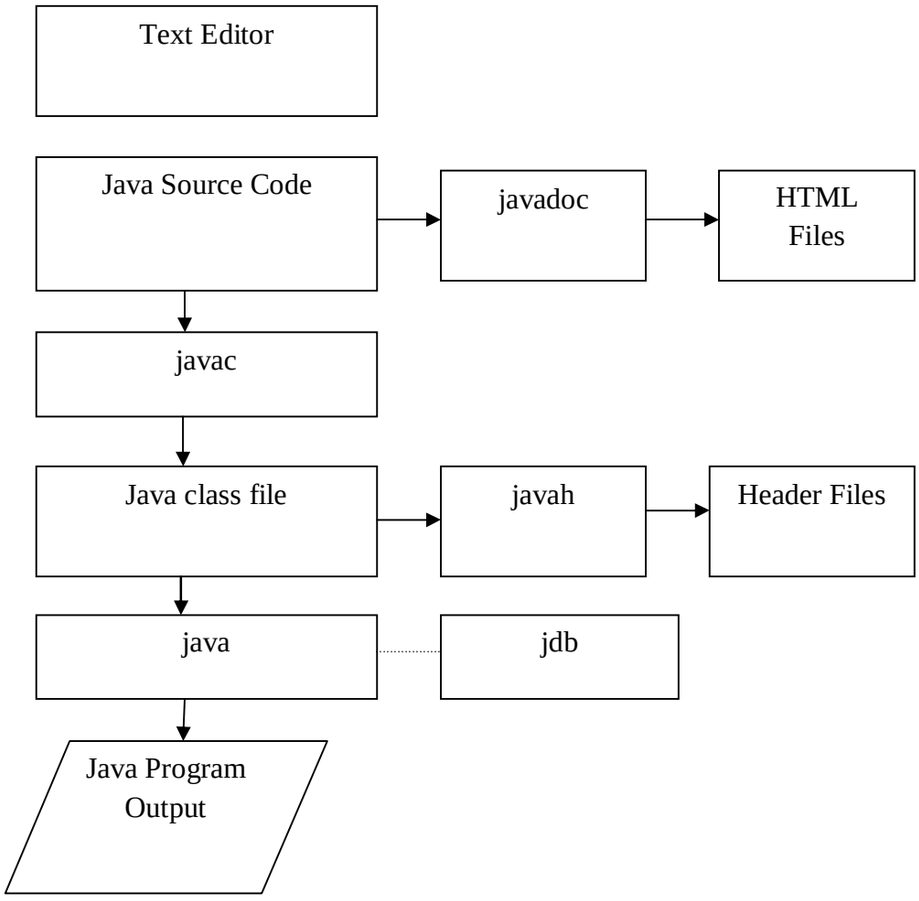

Figure 14.2 Process of building and running java application program

Application Programming Interlace

The Java Standard Library (or API) includes hundreds of classes and methods grouped into several functional packages .Most commonly used packages are:

Language Support Package: A collection of classes and methods required for implementing basic features of Java.

Utilities Package: A collection of classes to provide utility functions such as date and time functions.

Input/Output Package: A collection of classes required for input/output manipulation.

Networking Package: A collection of classes for communicating with other computers via Internet

Awt Package: The Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.

Applet Package: This includes a set of classes that allows us to create Java applets

14.3 Vectors

Java does not support the concept of variable arguments to a function. This feature can be achieved in Java through the use of the Vector class contained in the java.util package. This class can be used to create a generic dynamic array known as *vector* that can hold *objects of any type* and *any number*. The objects do not have to be homogenous. Arrays can be easily implemented as vectors. Vectors are created like arrays as follows:

```
Vector intVect = new Vector ( );      //declaring without size
Vector list = new Vector(3);          //declaring with size
```

Note that a vector can be declared without specifying any size explicitly. A vector can accommodate an unknown number of items. Even, when a size is specified, this can be overlooked and a different number of items may be put into the vector. Remember, in contrast, an array must always have its size specified. Vectors possess a number of advantages over arrays.

1. It is convenient to use vectors to store objects.
2. A vector can be used to store a list of objects that may vary in size.
3. We can add and delete objects from the list as and when required.

A major constraint in using vectors is that we cannot directly store simple data types in a vector; we can only store objects. Therefore, we need to convert simple types to objects. The vector class supports a number of methods that can be used to manipulate the vectors created.

Method Call	Task performed
list.addElement(item)	Adds the item specified to the list at the end
list.elementAt(10)	Gives the name of the 10^{th} object
list.size( )	Gives the number of objects present
list.removeElement(item)	Removes the specified item from the list
list.removeElementAt(n)	Removes the item stored in the nth position of the list
list.removeAllElements( )	Removes all the elements in the list
list.copyInto(array)	Copies all items from list to array
list.insertElementAt (item, n)	Inserts the item at nth position

Table 14.2 Important Vector Methods

Working with vectors and arrays

```
import java.util.*;                    // Importing Vector class
```

```
class LanguageVector
        {
                public static void main (String args[ ])
                {
                        Vector list = new Vector();
                        int length = args.length;
                        for (int i = 0; i < length; i++)
                        {
                                list.addElement(args[i]);
                        }
                        list.insertElementAt("COBOL",2);
                        int size = list.size();
                        String listArray[] = new String[size];
                        list.copyInto(listArray);
                        System.out.println("List of Languages");
                      for (int i = 0; i < size; i++)
                        {
                                System.out.println(listArray[i]);
                        }
                }
        }
```

Command line input and output are:

```
C:\JAVA\prog>java LanguageVector Ada BASIC C++ FORTRAN Java
List of Languages
Ada
BASIC
COBOL
C++
FORTRAN
Java
```

14.4 Wrapper classes

Vectors cannot handle primitive data types like int, float, long, char, and double. Primitive data types may be converted into object types by using the wrapper classes contained in the java.lang package. Table shows the simple data types and their corresponding wrapper class types.

Simple Type	Wrapper Class
Boolean	Boolean
Char	Character
Double	Double
Float	Float
Int	Integer
Long	Long

Table 14.3 Wrapper Classes for Converting Simple Types

The wrapper classes have a number of unique methods for handling primitive data types and objects. They are listed in the following tables.

Constructor Calling	Conversion Action
Integer IntVal = new Integer(i);	Primitive integer to Integer object
Float FloatVal = new Float(f);	Primitive float to Float object
Double DoubleVal = new Double(d);	Primitive double to Double object
Long LongVal = new Long(l);	Primitive long to Long object

Table 14.4 Converting Primitive Numbers to Object Numbers Using Constructor Methods

Note: i, f, d, and l are primitive data Values denoting int, float, double and long data types. They may be constants or Variables.

Method Calling	Conversion Action
int i = IntVal.intValue( );	Object to primitive integer
float f = FloatVal.floatValue( );	Object to primitive float
long 1 = LongVal.longValue( );	Object to primitive long
double d = DoubleVal.doubleValue( );	Object to primitive double

Table 14.5 Converting Object Numbers to Primitive Numbers Using typeValue( ) method

Method Calling	Conversion Action
str = Integer.toString(i)	Primitive integer to string
str =Float.toString(f);	Primitive float to string
str = Double. toString(d);	Primitive double to string
str = Long. toString(l);	Primitive long to string

Table 14.6 Converting Numbers to Strings Using toString( ) Method

Method Calling	Conversion Action
DoubleVal = Double.ValueOf(str);	Converts string to Double object
FloatVal = Float.ValueOf(str);	Converts string to Float object
IntVal = Integer.ValueOf(str);	Converts string to Integer object
LongVal = Long.ValueOf(str);	Converts string to Long object

Table 14.7 Converting String Objects to Numeric Objects Using the Static Method ValueOf( )

Note: These numeric objects may be converted to primitive numbers using the typeValue( )
Converting Numeric Strings to Primitive Numbers Using Parsing Methods

Method Calling	Conversion Action
int i = Integer.parseInt(str);	Converts string to primitive integer
long 1 = Long.parseLong(str);	Converts string to primitive long

Table 14.8 Converting numeric Strings to Primitive Numbers Using Parsing Method

Note: parseInt() and parseLong( ) methods throw a NumberFormatException if the value of the str does not represent an integer.

Use of *wrapper* class *methods*

```
import java.io.*;
class Invest
{
        public static void main(String args[ ])
        {
```

```
        Float principalAmount = new Float(0);
        Float interestRate = new Float(0);
        int numYears = 0;
        try
                {
                        DataInputStream in = new DataInputStream(system.in);
                        System.out.print("Enter Principal Amount: ");
                        System.out.flush( );
                        String principalString = in.readLine( );
                        principalAmount = Float.valueOf(principalString);
                        System.out.print("Enter Interest Rate: ");
                        System.out.flush( );
                        String interestString = in.readLine( );
                        interestRate = Float.valueOf(interestString);
                        System. out. print ("Enter Number of Years: ");
                        System.out.flush( );
                        String yearsString = in.readLine( );
                        numYears = Integer.parseInt(yearsString);
                }
                catch (IOException e)
                        {
                                System.out.println("I/O Error");
                                System.exit(1);
                        }
                float value = loan(principalAmount.floatValue(),
                interestRate.floatValue(),numYears);
                printline( );
                System.out.println(“Final Value = " + value);
                printline( );
}
//Method to compute Final Value
{
        static float loan(float p, float r, int n)
        int year = 1;
        float sum = p;
        while (year <= n)
        {
                sum = sum * (l+r);
                year = year + 1;
                return sum;
        }
}
// Method to draw a line
 static void printline( )
 {
        for (int i = 1; i <= 30; i++)
        {
```

```
                    System.out.print("=");
                }
                System.out.println("  ");
        }
}
```

The output of Program

```
=======================================
Enter Principal Amount: 5000
Enter Interest Rate: 0.15
Enter Number of Years: 4
=======================================
Final Value = 8745.03
```

Credit Based Fifth Semester B.C.A. Degree Examination, Oct./Nov. 2014
(2014-15 Batch) (New Syllabus)
JAVA PROGRAMMING-BCACAC314

Time: 3 Hours **Max. Marks: 100**

Note: Answer any ten questions from Part - A and answer one full question from each Unit in Part – B

PART-A **(2×10=20)**

1. a) Define Byte code. Write the benefits of Byte code.
   b) Define scope and life time of variable.
   c) Define stream. Name the two types of streams defined in Java.
   d) Define Array. Write the purpose of length member.
   e) What are constructors?
   f) What is the use of 'this' keyword in Java?
   g) Define interface. Why it is used?
   h) Define Package. Name any two advantages of using package.
   i) What is an exception? What are the advantages of exception handler?
   j) Write the two ways of identifying whether the thread has ended.
   k) Write the syntax. of <applet> tag and also write the purpose compulsory attributes.
   l) Write the limitations of the AWT components with respect to SWING.

PART-B
UNIT-I

2. a) Write a note on java Buzzwords. (5+6+4+5)
   b) Write a note on :
      i) Increment and Decrement operator
      ii) Type conversion in Expression.
   c) With syntax and example explain the process of reading a character from the keyboard.
   d) Explain nested switch statement with syntax and example.

3. a) Write a note on Java's contribution to the internet. (5+5+4+6)
   b) List and explain different primitive data types available in Java.
   c) With syntax and example explain the process of reading a string from the keyboard.
   d) Explain while loop and do while loop with syntax and example.

UNIT-II

4. a) Explain how to declare, instantiate, initialize and use a one dimensional array with example. (6+5+5+4)
   b) With an example explain command line arguments.
   c) With an example explain method overloading.
   d) Write a note on :
      i) Abstract class and Abstract method
      ii) Final class and Final method.

5. a) Explain any four string methods with suitable example. (4+6+5+5)
   b) With an example explain the different types of constructors supported by Java.
   c) Explain how to pass variable-length arguments with suitable example.
   d) With an example explain single level inheritance.

UNIT-III

6. a) With an example explain how to create and importing the user defined package. (6+5+4+5)
   b) With an example explain implementing an interface.
   c) Illustrate the use of finally with suitable example.
   d) Define synchronization. With an example explain how synchronization is achieved in multithreaded environment using synchronized methods.

7. a) Explain the Java's access control mechanism using access modifier keywords. (5+6+5+4)
   b) With an example explain how a user defined exception is defined and used for handling errors.
   c) With an example explain how a thread created by implementing runnable interface.
   d) Write a note on Thread Priority.

UNIT-IV

8. a) With an exampld explain passing parameters to an applet. (5+5+5+5)
   b) What is the purpose of TextField class? Explain any three methods of TextField class with syntax and example.
   c) What is the purpose of layout managers? With an example explain the use of Grid Layout.
   d) With an example explain how to create a swing applet.

9. a) What is the purpose of MouseListener interface? With the syntax and example explain any five methods of MouseListener interface. (6+5+5+4)
   b) Name the controls required for creating a menu. With an example explain the creation of a menu.
   c) With syntax and example explain the purpose of showStatus( ) method.
   d) With an example explain the use of anonymous inner classes to handle events

****