Chapter 1

2M Questions

1) Enlist the properties of object oriented programming.

Ans:

a. Classes and Objects
b. Data hiding
c. Data Encapsulation
d. Polymorphism
e. Inheritance
f. Abstraction

2) Enlist the features of Java language.

Ans:

The Java programming language is a high-level language which has following features:

- Simple
- Object oriented
- Distributed
- Interpreted
- Robust
- Secure
- Architecture neutral
- Portable
- High performance
- Multithreaded
- Dynamic

3) State the primitive data types of Java language.

Ans:

Primitive Data Types

Keyword	**Description**	**Size/Format**
	(integers)	
`byte`	Byte-length integer	8-bit two's complement
`short`	Short integer	16-bit two's complement
`int`	Integer	32-bit two's complement
`long`	Long integer	64-bit two's complement
	(real numbers)	

`float`	Single-precision floating point	32-bit IEEE 754
`double`	Double-precision floating point	64-bit IEEE 754
	(other types)	
`char`	A single character	16-bit Unicode character
`boolean`	A boolean value (`true` or `false`)	true or false

4) Enlist the types of operators.
Ans:

a. Arithmetic Operator
b. Relational & Conditional Operator
c. Shift & Logical Operator
d. Assignment Operator

5) Enlist the types of literals.
Ans:

a. Integer literal
b. Floating Point literal
c. Boolean literal
d. Character literal

6) What are the types of Java programming language?
Ans:

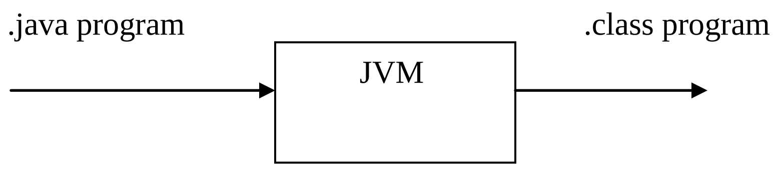
There are two basic types of Java Programming language:
a. Application
b. Applet

7) What is the “main” thread?
Ans:

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:
• It is the thread from which other "child" threads will be spawned.
• It must be the last thread to finish execution. When the main thread stops, your program terminates.
Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown here:
static Thread currentThread( )
This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, it can be controlled like any other thread.

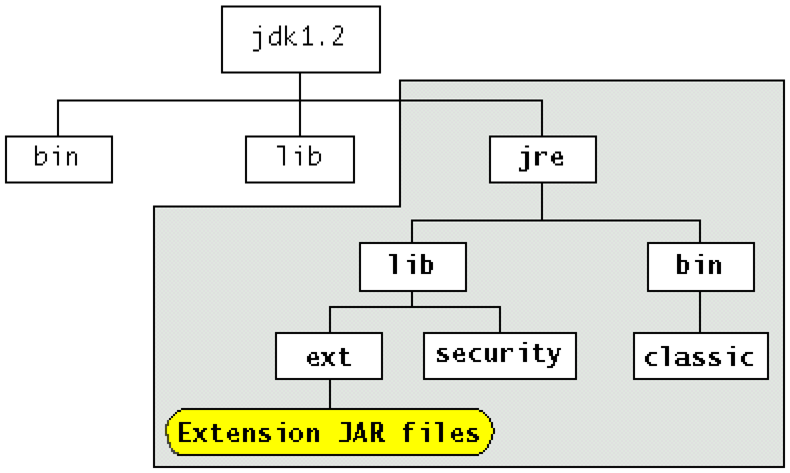
8) What is JVM?
Ans:

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*).
JIT (Just In Time) compiler is part of the JVM, it compiles bytecode into executable
Many browsers include a Java Virtual Machine that is used to execute applets. Unfortunately, browser JVMs typically do not include the latest Java features. The Java Plug-In solves this problem. It directs a browser to use a Java Runtime Environment (JRE) rather than the browser's JVM. The JRE is a subset of the JDK. It does not include the tools and classes that are used in a development environment. Various tools such as **javac**, **java**, and **javadoc** have been enhanced. Debugger and profiler interfaces for the JVM are available.



9) What is JRE?
Ans:
Installed extensions are JAR files in the `lib/ext` directory of the Java Runtime Environment (JRE) software. As its name implies, the JRE is the runtime portion of the Java Development Kit containing the platform's core API but without development tools such as compilers and debuggers. The JRE is available either by itself or as part of the Java Development Kit.

In the 1.2 platform, the JRE is a strict subset of the JDK™ software. The JDK 1.2 software directory tree looks like this:



JRE 1.2 consists of those directories within the highlighted box in the diagram. Whether your JRE is stand-alone or part of the JDK software, any JAR file in the JRE's `lib/ext` directory is automatically treated by the runtime environment as an extension.

10) What is JDK? State the version.

Ans:
JDK is JAVA Development Kit.
JDK 1.1.x is upwards binary-compatible with 1.0.x except for these incompatibilities. This means that class files built with a 1.0.x compiler (such as 1.0.2) will run correctly in 1.1.x.

Downward binary compatibility is generally supported, though not guaranteed. That is, class files built with a 1.1.x compiler, but relying only on APIs defined in 1.0.x, will generally run on 1.0.x versions of the Java Virtual Machine, but this "downwards" compatibility has not been extensively tested and cannot be guaranteed. Of course, if the class files depend on any new 1.1.x APIs, those files will not work on 1.0.x systems.

In general, the JavaSoft policy is that:

- Bug-fix releases (e.g. 1.1.1, 1.1.2) within a family (1.1.x) will maintain both upward and downward binary-compatibility with each other.
- Functionality releases (e.g. 1.1, 1.2) within a family (1.x) will maintain upward but not necessarily downward binary-compatibility with each other.
- Major releases (e.g. 2.0, 3.0) will not necessarily maintain any binary compatibility.

Latest version of JDK is:
jdk1.6.0_03

11) State the use of documentation of Java.
Ans:
The programmer must aware about the documentation of JAVA. That is called as JAVA API (Application Programming Interface). The documentation includes the syntax of all classes, their methods, constructors, etc. With the help of these information, the programmer is coming to know the usage of classes in the program.

12) What is J2SE?
Ans:
Java has three editions. One of them is called as J2SE. i.e. Java 2 Standard Edition. This edition includes core java technology.

13) State the utilities provided by JDK.
Ans:
Once JDK is included in the PC. The number of directories are being created. They are:
a. bin: which includes the java compiler, interpreter, documentation & more utilities.
b. lib: which includes the .jar files that internally includes the predefined classes.

14) Enlist different types of statements used in Java programming language.

Ans:

a. Selection statement
b. Transfer statement
c. Iteration statement
d. Guarding statement

4M Questions

1) Write a program in Java to print “Hello World”.

Ans:

Algorithm:

```
a. Define a class “HelloWorld”
        1. Define a main method
                1.1 Print the message “Hello World!”
b. End of Main method & then class
```

Code:

```
class HelloWorld
{
        public static void main(String args[])
        {
                System.out.println("Hello World!");
        }
}
```

Compilation:

```
c:\>javac Helloworld.java
```

Execution:

```
c:\>java HelloWorld
```

Output:

```
Hello World!
```

2) Describe the meaning of following statement.

public static void main(String args[])

Ans:

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

This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**. (This is just like C/C++.) The exact meaning of each part of this line cannot be given now, since it involves a detailed understanding of Java's approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let's take a brief look at each part now.
The **public** keyword is an *access specifier,* which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made. The keyword **void** simply tells the compiler that **main( )** does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don't worry. All of these concepts will be discussed in detail in subsequent chapters.
As stated, **main( )** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But the Java interpreter has no way to run these classes. So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the **main( )** method.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters.* If there are no parameters required for a given method, you still need to include the empty parentheses. In **main( )**, there is only one parameter, albeit a complicated one. **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.

3) Describe the concept of data abstraction.
Ans:

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
certain methods be overridden by subclasses by specifying the
abstract type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
To declare an abstract method, use this general form:
abstract *type name(parameter-list)*;
Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

4) Describe any four Java language features.

Ans: The following features are listed below:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple:

Java was designed to be easy for the professional programmer to learn and use effectively. Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have *surprising* features. In Java, there are a small number of clearly defined ways to accomplish a given task.

Object Oriented:

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as highperformance nonobjects.

Robust:

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development.

Multithreaded:

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

Architecture-Neutral:

A central issue for the Java designers was that of code longevity and portability. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation.

Interpreted and High Performance:
Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at crossplatform solutions have done so at the expense of performance. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed:
Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. The original version of Java (Oak) included features for intra-addressspace messaging. This allowed objects on two different computers to execute procedures remotely. Java has recently revived these interfaces in a package called *Remote Method Invocation* (*RMI*). This feature brings an unparalleled level of abstraction to client/server programming.

Dynamic:
Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

5) Justify whether “Java language is 100 % object oriented programming language.”

Ans:

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as highperformance nonobjects.
Actually Java language is not 100% object oriented programming language. There are reasons for the same. One of them is: Java includes native code that is nothing but the C/C++ language can be included in Java. Second thing is that java has one JDBC driver that is Type-1 which is not purely written in Java.

6) Justify whether “Java language is made for internet language.”

Ans:

The Java programming language is not just for writing cute, entertaining applets for the Web. The general-purpose, high-level Java programming language is also a powerful software platform. Using the generous API, you can write many types of programs.

An application is a standalone program that runs directly on the Java platform. A special kind of application known as a *server* serves and supports clients on a network. Examples of servers are Web servers, proxy servers, mail servers, and print servers. Another specialized program is a *servlet*. A servlet can almost be thought of as an applet that runs on the server side. Java Servlets are a popular choice for building interactive web applications, replacing the use of CGI scripts. Servlets are similar to applets in that they are runtime extensions of applications. Instead of working in browsers, though, servlets run within Java Web servers, configuring or tailoring the server.

How does the API support all these kinds of programs? It does so with packages of software components that provide a wide range of functionality. Every full implementation of the Java platform gives you the following features:

- **The essentials**: Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, and so on.
- **Applets**: The set of conventions used by applets.
- **Networking**: URLs, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) sockets, and IP (Internet Protocol) addresses.
- **Internationalization**: Help for writing programs that can be localized for users worldwide. Programs can automatically adapt to specific locales and be displayed in the appropriate language.
- **Security**: Both low level and high level, including electronic signatures, public and private key management, access control, and certificates.
- **Software components**: Known as JavaBeans$^{\text{TM}}$, can plug into existing component architectures.
- **Object serialization**: Allows lightweight persistence and communication via Remote Method Invocation (RMI).
- **Java Database Connectivity (JDBC$^{\text{TM}}$)**: Provides uniform access to a wide range of relational databases.

7) Describe the primitive data types of Java language.
Ans:
Every variable must have a data type. A variable's data type determines the values that the variable can contain and the operations that can be performed on it. Integers can contain only integral values (both positive and negative). You can perform arithmetic operations, such as addition, on integer variables.

The Java programming language has two categories of data types: *primitive* and reference. A variable of primitive type contains a single value of the appropriate size and format for its type: a number, a character, or a boolean value. For example, an integer value is 32 bits of data in a format known as two's complement, the value of a `char` is 16 bits of data formatted as a Unicode character, and so on.

variableName
value

The following table lists, by keyword, all of the primitive data types supported by Java, their sizes and formats, and a brief description of each. The `MaxVariablesDemo` program declares one variable of each primitive type.

Primitive Data Types

Keyword	**Description**	**Size/Format**
	(integers)	
`byte`	Byte-length integer	8-bit two's complement
`short`	Short integer	16-bit two's complement
`int`	Integer	32-bit two's complement
`long`	Long integer	64-bit two's complement
	(real numbers)	
`float`	Single-precision floating point	32-bit IEEE 754
`double`	Double-precision floating point	64-bit IEEE 754
	(other types)	
`char`	A single character	16-bit Unicode character
`boolean`	A boolean value (`true` or `false`)	true or false

A literal primitive value can be put directly in your code. For example, if you need to assign the value 4 to an integer variable you can write this:

```
int anInt = 4;
```

The digit 4 is a literal integer value. Here are some examples of literal values of various primitive types:

Examples of Literal Values and Their Data Types

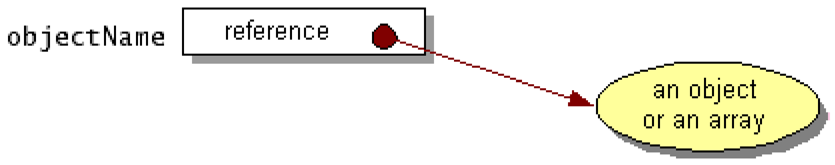
Literal	**Data Type**
178	`int`
8864L	`long`
`37.266`	`double`
37.266D	`double`
`87.363F`	`float`
26.77e3	`double`
`'c'`	`char`
`true`	`boolean`

`false`	`boolean`

Generally speaking, a series of digits with no decimal point is typed as an integer. You can specify a long integer by putting an `'L'` or `'l'` after the number. `'L'` is preferred as it cannot be confused with the digit `'1'`. A series of digits with a decimal point is of type double. You can specify a float by putting an `'f'` or `'F'` after the number. A literal character value is any single Unicode character between single quote marks. The two boolean literals are simply `true` and `false`.

Arrays, classes, and interfaces are reference types. The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable.

A reference is called a pointer, or a memory address in other languages. The Java programming language does not support the explicit use of addresses like other languages do. The variable's name can be used instead.



8) Describe any four elements of Java language.
Ans:

Like any other programming language, the Java programming language is defined by grammar rules that specify how syntactically legal constructs can be formed using the language elements, and by a semantic definition that specifies the meaning of syntactically legal constructs.

Lexical Tokens

The low-level language elements are called lexical tokens (or just tokens for short) and are the building blocks for more complex constructs. Identifiers, numbers, operators, and special characters are all examples of tokens that can be used to build high-level constructs like expressions, statements, methods, and classes.

Identifiers

A name in a program is called an identifier. Identifiers can be used to denote classes, methods, variables, and labels.

In Java an identifier is composed of a sequence of characters, where each character can be either a letter, a digit, a connecting punctuation (such as underscore _), or any currency symbol (such as `$`, `¢`, `¥`, or `£`). However, the first character in an identifier cannot be a digit. Since Java programs are written in the Unicode character set (see p. 23), the definitions of letter and digit are interpreted according to this character set.

Keywords

Keywords are reserved identifiers that are predefined in the language and cannot be used to denote other entities. All the keywords are in lowercase, and incorrect usage results in compilation errors.

Literals

A literal denotes a constant value, that is, the value a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, boolean or string values.

9) Describe the escape sequences.
Ans:

Escape Sequences

Certain escape sequences define special character values as shown in below table. These escape sequences can be single-quoted to define character literals. For example, the character literals `'\t'` and `'\u0009'` are equivalent. However, the character literals `'\u000a'` and `'\u000d'` should not be used to represent newline and carriage return in the source code. These values are interpreted as line-terminator characters by the compiler, and will cause compile time errors. One should use the escape sequences `'\n'` and `'\r'`, respectively, for correct interpretation of these characters in the source code.

Table. Escape Sequences

Escape Sequence	Unicode Value	Character
`\b`	`\u0008`	Backspace (BS)
`\t`	`\u0009`	Horizontal tab (HT or TAB)
`\n`	`\u000a`	Linefeed (LF) a.k.a., Newline (NL)
`\f`	`\u000c`	Form feed (FF)
`\r`	`\u000d`	Carriage return (CR)
`\'`	`\u0027`	Apostrophe-quote
`\"`	`\u0022`	Quotation mark

Table. Escape Sequences

Escape Sequence	Unicode Value	Character
`\\`	`\u005c`	Backslash

We can also use the escape sequence `\ddd` to specify a character literal by octal value, where each digit `d` can be any octal digit (`0`–`7`), as shown in below table. The number of digits must be three or fewer, and the octal value cannot exceed `\377`, that is, only the first 256 characters can be specified with this notation.

Table. Examples of Escape Sequence `\ddd`

Escape Sequence `\ddd`	Character Literal
`'\141'`	`'a'`
`'\46'`	`'&'`
`'\60'`	`'0'`

10) Explain different types of comments used in Java language along with example.

Ans:

A program can be documented by inserting comments at relevant places. These comments are for documentation purposes and are ignored by the compiler.

Java provides three types of comments to document a program:

- A single-line comment: `// ... to the end of the line`
- A multiple-line comment: `/* ... */`
- A documentation (Javadoc) comment: `/** ... */`

Single-line Comment

All characters after the comment-start sequence `//` through to the end of the line constitute a single-line comment.

```
// This comment ends at the end of this line.
int age;        // From comment-start sequence to the end of the line
is a comment.
```

Multiple-line Comment

A multiple-line comment, as the name suggests, can span several lines. Such a comment starts with `/*` and ends with `*/`.

```
/*  A comment
    on several
    lines.
*/
```

The comment-start sequences (`//`, `/*`, `/**`) are not treated differently from other characters when occurring within comments, and are thus ignored. This means trying to nest multiple-line comments will result in compile time error:

```
/*  Formula for alchemy.
    gold = wizard.makeGold(stone);
    /* But it only works on Sundays. */
*/
```

The second occurrence of the comment-start sequence `/*` is ignored. The last occurrence of the sequence `*/` in the code is now unmatched, resulting in a syntax error.

Documentation Comment

A documentation comment is a special-purpose comment that when placed before class or class member declarations can be extracted and used by the `javadoc` tool to generate HTML documentation for the program. Documentation comments are usually placed in front of classes, interfaces, methods and field definitions. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with `/**` and ends with `*/`:

```
/**
 *  This class implements a gizmo.
 *  @author K.A.M.
 *  @version 2.0
 */
```

11) How to declare the variable in Java. Give Example.
Ans:

A variable stores a value of a particular type. A variable has a name, a type, and a value associated with it. In Java, variables can only store values of primitive data types and references to objects. Variables that store references to objects are called reference variables.

Declaring and Initializing Variables

Variable declarations are used to specify the type and the name of variables. This implicitly determines their memory allocation and the values that can be stored in them. We show some examples of declaring variables that can store primitive values:

```
char a, b, c;            // a, b and c are character variables.
double area;             // area is a floating-point variable.
boolean flag;            // flag is a boolean variable.
```

The first declaration above is equivalent to the following three declarations:

```
char a;
char b;
char c;
```

A declaration can also include initialization code to specify an appropriate initial value for the variable:

```
int i = 10,              // i is an int variable with initial value 10.
    j = 101;             // j is an int variable with initial value
101.
long big = 2147483648L;  // big is a long variable with specified
initial value.
```

Object Reference Variables

An object reference is a value that denotes an object in Java. Such reference values can be stored in variables and used to manipulate the object denoted by the reference value.

A variable declaration that specifies a reference type (i.e., a class, an array, or an interface name) declares an object reference variable. Analogous to the declaration of variables of primitive data types, the simplest form of reference variable declaration only specifies the name and the reference type. The declaration determines what objects a reference variable can denote. Before we can use a reference variable to manipulate an object, it must be declared and initialized with the reference value of the object.

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of
class Pizza.
Hamburger bigOne,   // Variable bigOne can reference objects of class
Hamburger,
          smallOne; // and so can variable smallOne.
```

12) Describe binary string concatenation operator.
Ans:

The binary operator + is overloaded in the sense that the operation performed is determined by the type of the operands. When one of the operands is a `String` object, the other operand is implicitly converted to its string representation and string concatenation is performed. Non-`String` operands are converted as follows:

- For a operand of a primitive data type, its value is converted to a `String` object with the string representation of the value.

- Values like `true`, `false`, and `null` are represented by string representations of these literals. A reference variable with the value `null` also has the string representation `"null"` in this context.
- For all reference value operands, a string representation is constructed by calling the `toString()` method on the referred object. Most classes override this method from the `Object` class in order to provide a more meaningful string representation of their objects.

```
String theName = " Uranium";
theName = " Pure" + theName;                  // " Pure Uranium"
String trademark1 = 100 + "%" + theName;      // "100% Pure Uranium"
(1)
```

The integer literal `100` is implicitly converted to the string `"100"` before concatenation. This conversion is corresponds to first creating an object of the wrapper class `Integer`, which represents the integer `100`, and then creating a string from this object by using the `toString()` method supplied by this class:

```
new Integer(100).toString();
```

Note that using the character literal `'%'`, instead of the string literal `"%"` in line (1) above, does not give the same result:

```
String trademark2 = 100 + '%' + theName;      // "137 Pure Uranium"
```

Integer addition is performed by the first + operator: `100 + '%'`, that is, `(100 + 37)`. Caution should be exercised as the + operator might not be applied as intended, as shown by the following example:

```
System.out.println("We put two and two together and get " + 2 + 2);
```

The above statement prints `"We put two and two together and get 22"` and not `"We put two and two together and get 4"`.

13) Describe Boolean logical operators.
Ans:

Boolean logical operators include the unary operator `!` (logical complement) and the binary operators `&` (logical AND), `|` (logical inclusive OR), and `^` (logical exclusive OR, a.k.a. logical XOR). Boolean logical operators can be applied to `boolean` operands, returning a `boolean` value. The operators `&`, `|`, and `^` can also be applied to integral operands to perform bitwise logical operations.

Given that x and y represent boolean expressions, the boolean logical operators are defined in following table. In the table, the operators are ranked, with the operator having the highest precedence first.

Table. Boolean Logical Operators

Logical complement	`!x`	Returns the complement of the truth-value of x.
Logical AND	`x & y`	`true` if both operands are true; otherwise, `false`.
Logical OR	`x | y`	`true` if either or both operands are true; otherwise, `false`.
Logical XOR	`x ^ y`	`true` if and only if one operand is true; otherwise, `false`.

In evaluation of boolean expressions involving boolean logical AND, XOR, and OR operators, both the operands are evaluated. The order of operand evaluation is always from left to right.

```
if (i > 0 & i++ < 10) {/*...*/} // i will be incremented, regardless of
value in i.
```

The binary boolean logical operators have precedence lower than arithmetic and relational operators, but higher than assignment, conditional AND, and OR operators. This is illustrated in the following examples:

```
boolean b1, b2, b3 = false, b4 = false;
b1 = 4 == 2 & 1 < 4;           // false, evaluated as (b1 = ((4 == 2) &
(1 < 4)))
b2 = b1 | !(2.5 >= 8);         // true
b3 = b3 ^ b2;                  // true
b4 = b4 | b1 & b2;             // false
```

Order of evaluation is illustrated for the last example:

```
  (b4 = (b4 | (b1 & b2)))
-> (b4 = (false | (b1 & b2)))
-> (b4 = (false | (false & b2)))
-> (b4 = (false | (false & true)))
-> (b4 = (false | false))
-> (b4 = false)
```

14) Describe conditional operators by giving examples.
Ans:

Conditional operators `&&` and `||` are similar to their counterpart logical operators & and `|`, except that their evaluation is short-circuited. Given that `x` and `y` represent values of `boolean` expressions, the conditional operators are defined in below table. In the table, the operators are listed in decreasing precedence order.

Table. Conditional Operators

Conditional AND	`x && y`	`true` if both operands are true; otherwise, `false`.
Conditional OR	`x || y`	`true` if either or both operands are true; otherwise, `false`.

Unlike their logical counterparts `&` and `|`, which can also be applied to integral operands for bitwise operations, the conditional operators `&&` and `||` can only be applied to `boolean` operands. Their evaluation results in a `boolean` value.

The binary conditional operators have precedence lower than either arithmetic, relational, or logical operators, but higher than assignment operators. The following examples illustrate usage of conditional operators:

```
boolean b1 = 4 == 2 && 1 < 4;    // false, short-circuit evaluated as
                                 // (b1 = ((4 == 2) && (1 < 4)))
boolean b2 = !b1 || 2.5 > 8;     // true, short-circuit evaluated as
                                 // (b2 = ((!b1) || (2.5 > 8)))
boolean b3 = !(b1 && b2);        // true
boolean b4 = b1 || !b3 && b2;    // false, short-circuit evaluated as
                                 // (b4 = (b1 || ((!b3) && b2)))
```

Order of evaluation for computing the value of boolean variable `b4` proceeds as follows:

```
   (b4 = (b1 || ((!b3) && b2)))
-> (b4 = (false || ((!b3) && b2)))
-> (b4 = (false || ((!true) && b2)))
-> (b4 = (false || ((false) && b2)))
-> (b4 = (false || false))
-> (b4 = false)
```

15) Describe shift operators by giving examples.
Ans:

The binary shift operators form a new value by shifting bits either left or right a specified number of times in a given integral value. The number of shifts (also called the shift distance) is given by the right-hand operand, and the value that is to be shifted is given by the left-hand operand. Note that unary numeric promotion is applied to each operand individually. The value returned has the promoted type of the left-hand operand. Also, the value of the left-hand operand is not affected by applying the shift operator.

The shift distance is calculated by AND-ing the value of the right-hand operand with a mask value of `0x1f` (31) if the left-hand has the promoted type `int`, or using a mask value of `0x3f` (63) if the left-hand has the promoted type `long`. This effectively means masking the five lower bits of the right-hand operand in the case of an `int` left-hand operand, and masking the six lower bits of the right-hand operand in the case of a `long` left-hand operand. Thus, the shift distance is always in the range 0 to 31 when the promoted type of left-hand operand is `int`, and in the range 0 to 63 when the promoted type of left-hand operand is `long`.

Given that `a` contains the value whose bits are to be shifted and `n` specifies the number of bits to shift, the bitwise operators are defined in below table. It is implied that the value `n` in table is subject to the shift distance calculation outlined above, and that the shift operations are always performed on the value of the left-hand operand represented in 2's complement.

Table. Shift Operators

Shift left	`a << n`	Shift all bits in `a` left `n` times, filling with `0` from the right.
Shift right with sign bit	`a >> n`	Shift all bits in `a` right `n` times, filling with the sign bit from the left.
Shift right with zero fill	`a >>> n`	Shift all bits in `a` right `n` times, filling with `0` from the left.

As the bits are shifted left, zeros are always filled in from the right.

```
int i = 12;
int result = i << 4;    // 192
```

The bits in the `int` value for `i` are shifted left four places as follows:

```
i << 4
= 0000 0000 0000 0000 0000 0000 0000 1100 << 4
= 0000 0000 0000 0000 0000 0000 1100 0000
= 0x000000c0
= 192
```

As the bits are shifted right, the sign bit (the most significant bit) is used to fill in from the left. So, if the left-hand operand is a positive value, zeros are filled in from the left, but if the operand is a negative value, ones are filled in from the left.

```
int i = 12;
int result = i >> 2;    // 3
```

The value for `i` is shifted right with sign-fill two places.

```
i >> 2
= 0000 0000 0000 0000 0000 0000 0000 1100 >> 2
= 0000 0000 0000 0000 0000 0000 0000 0011
= 0x00000003
= 3
```

16) Describe conditional operator.
Ans:

The ternary conditional operator allows conditional expressions to be defined. The operator has the following syntax:

<condition> `?` <expression$_1$> `:` <expression$_2$>

If the boolean expression <condition> is `true` then <expression$_1$> is evaluated; otherwise, <expression$_2$> is evaluated. Of course, <expression$_1$> and <expression$_2$> must evaluate to values of compatible types. The value of the expression evaluated is returned by the conditional expression.

```
boolean leapYear = false;
int daysInFebruary = leapYear ? 29 : 28;   // 28
```

The conditional operator is the expression equivalent of the `if-else` statement. The conditional expression can be nested and the conditional operator associates from right to left:

`(a?b?c?d:e:f:g)` evaluates as `(a?(b?(c?d:e):f):g)`

17) Describe new, [ ], instanceof operators.
Ans:

The `new` operator is used to create objects, that is, instances of classes and arrays. It is used with a constructor call to instantiate classes, and with the `[]` notation to create arrays. It is also used to instantiate anonymous arrays, and anonymous classes.

```
Pizza onePizza = new Pizza();       // Create an instance of Pizza
class.
```

The `[]` notation is used to declare and construct arrays and also to access array elements.

```
int[] anArray = new int[5];// Declare and construct an int array of 5
elements.
anArray[4] = anArray[3];   // Element at index 4 gets value of element
at index 3.
```

The boolean, binary, and infix operator `instanceof` is used to test an object's type.

```
Pizza myPizza = new Pizza();
boolean test1 = myPizza instanceof Pizza;  // True.
boolean test2 = "Pizza" instanceof Pizza;  // Compile error. String is
not Pizza.
boolean test3 = null instanceof Pizza;     // Always false. null not an
instance.
```

18) Describe various selection statements by giving examples.
Ans:

Java provides selection statements that allow the program to choose between alternative actions during execution. The choice is based on criteria specified in the selection statement. These selection statements are

- simple `if` Statement
- `if-else` Statement
- `switch` Statement

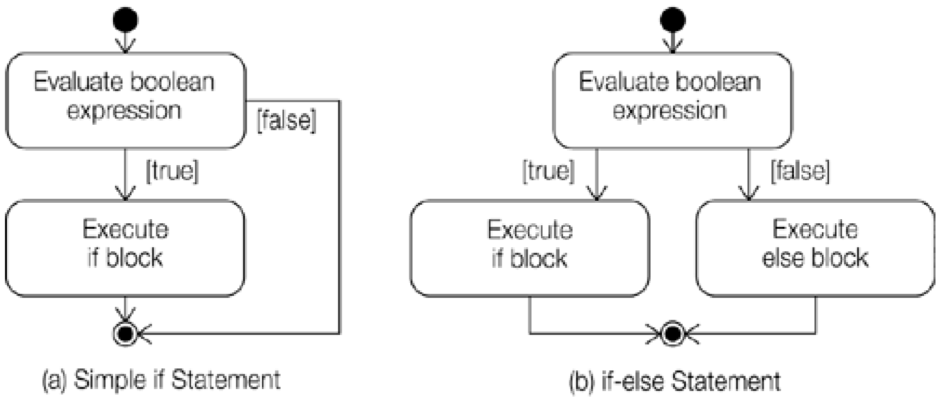
Simple `if` Statement

The simple `if` statement has the following syntax:

```
if (<conditional expression>)
    <statement>
```

It is used to decide whether an action is to be performed or not, based on a condition. The condition is specified by <conditional expression> and the action to be performed is specified by <statement>.

The following figure shows the activity diagram of if statement.



`if-else` Statement

The `if-else` statement has the following syntax:

```
if (<conditional expression>)
      <statement1>
  else
      <statement2>
```

It is used to decide between two actions, based on a condition.

`switch` Statement

Conceptually the `switch` statement can be used to choose one among many alternative actions, based on the value of an expression. Its general form is as follows:

```
switch (<non-long integral expression>) {
   case label1: <statement1>
   case label2: <statement2>
   ...
   case labeln: <statementn>
   default:  <statement>
} // end switch
```

The syntax of the `switch` statement comprises a `switch` expression followed by the `switch` body, which is a block of statements. The type of the `switch` expression is non-`long` integral (i.e., `char`, `byte`, `short`, or `int`). The statements in the `switch` body can be labeled, defining entry points in the `switch` body where control can be transferred depending on the value of the `switch` expression. The semantics of the `switch` statement are as follows:

- The `switch` expression is evaluated first.
- The value of the `switch` expression is compared with the `case` labels. Control is transferred to the <statement$_i$> associated with the `case` label that is equal to the value of the `switch` expression. After execution of the associated statement, control falls through to the next statement unless appropriate action is taken.
- If no `case` label is equal to the value of the `switch` expression, the statement associated with the `default` label is executed.

19) Describe iteration statements by giving examples.
Ans:

Loops allow a block of statements to be executed repeatedly (i.e., iterated). A boolean condition (called the loop condition) is commonly used to determine when to terminate the loop. The statements executed in the loop constitute the loop body. The loop body can be a single statement or a block.

Java provides three language constructs for constructing loops:

- `while` statement
- `do-while` statement
- `for` statement

These loops differ in the order in which they execute the loop body and test the loop condition. The `while` and the `for` loops test the loop condition before executing the loop body, while the `do-while` loop tests the loop condition after execution of the loop body.

`while` Statement

The syntax of the `while` loop is

```
while (<loop condition>)
    <loop body>
```

The loop condition is evaluated before executing the loop body. The `while` statement executes the loop body as long as the loop condition is `true`. When the loop condition becomes `false`, the loop is terminated and execution continues with the statement immediately following the loop. If the loop condition is `false` to begin with, the loop body is not executed at all. In other words, a `while` loop can execute zero or more times. The loop condition must be a `boolean` expression.

`do-while` Statement

The syntax of the `do-while` loop is

```
do
    <loop body>
  while (<loop condition>);
```

The loop condition is evaluated after executing the loop body. The `do-while` statement executes the loop body until the loop condition becomes `false`. When the loop condition becomes `false`, the loop is terminated and execution continues with the statement immediately following the loop. Note that the loop body is executed at least once.

`for` Statement

The `for` loop is the most general of all the loops. It is mostly used for counter-controlled loops, that is, when the number of iterations is known beforehand.

The syntax of the loop is as follows:

```
for (<initialization>; <loop condition>; <increment expression>)
    <loop body>
```

The <initialization> usually declares and initializes a loop variable that controls the execution of the <loop body>. The <loop condition> is a `boolean` expression, usually involving the loop variable, such that if the loop condition is `true`, the loop body is executed; otherwise, execution continues with the statement following the `for` loop. After each iteration (i.e., execution of the loop body), the <increment expression> is executed. This usually modifies the value of the loop variable to ensure eventual loop termination. The loop condition is then tested to determine if the loop body should be executed again. Note that the <initialization> is only executed once on entry to the loop.

20) Describe different transfer statements by giving examples.
Ans:

Java provides six language constructs for transferring control in a program:

- `break`
- `continue`
- `return`
- `try-catch-finally`
- `throw`
- `assert`

This section discusses the first three statements, and the remaining statements are discussed in subsequent sections.

Note that Java does not have a `goto` statement, although `goto` is a reserved word.

Labeled Statements

A statement may have a label.

<label> : <statement>

A label is any valid identifier and it always immediately proceeds the statement. Label names exist in their own name space, so that they do not conflict with names of packages, classes, interfaces, methods, fields, and local variables. The scope of a label is the statement prefixed by the label, meaning that it cannot be redeclared as a label inside the labeled statement—analogous to the scope of local variables.

```
L1: if (i > 0) {
    L1: System.out.println(i);  // (1) Not OK. Label redeclared.
}
```

```
L1: while (i < 0) {              // (2) OK.
    L2: System.out.println(i);
}
```

`break` Statement

The `break` statement comes in two forms: the unlabeled and the labeled form.

```
break;             // the unlabeled form

break <label>;     // the labeled form
```

The unlabeled `break` statement terminates loops (`for`, `while`, `do-while)` and `switch` statements which contain the `break` statement, and transfers control out of the current context (i.e., the closest enclosing block). The rest of the statement body is skipped, terminating the enclosing statement, with execution continuing after this statement.

`continue` Statement

Like the `break` statement, the `continue` statement also comes in two forms: the unlabeled and the labeled form.

```
continue;             // the unlabeled form
continue <label>;     // the labeled form
```

The `continue` statement can only be used in a `for`, `while`, or `do-while` loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible. In the case of the `while` and `do-while` loops, the rest of the loop body is skipped, that is, stopping the current iteration, with execution continuing with the <loop condition>. In the case of the `for` loop, the rest of the loop body is skipped, with execution continuing with the <increment expression>.

`return` Statement

The `return` statement is used to stop execution of a method and transfer control back to the calling code (a.k.a. the caller). The usage of the two forms of the `return` statement is dictated by whether it is used in a `void` or a non-`void` method. The first form does not return any value to the calling code, but the second form does. Note that the keyword `void` does not represent any type.

21) Write a program to describe the use of command line arguments.

Ans:

```
class CmdDemo
{
    public static void main(String args[])
    {
            System.out.println("java is "+args[0]);
            System.out.println("java is "+args[1]);
    }
}
```

22) Write a program to accept the value from a keyboard.

Ans:

```
import java.io.DataInputStream;

class ReadDemo
{
    public static void main(String args[])
    {
            String str = "";
            int a = 0;
            DataInputStream in = new DataInputStream(System.in);
            System.out.println("please enter the number:");
    try
    {
            str = in.readLine();
    a = Integer.parseInt(str);
    }
    catch(Exception e)
    {
            System.out.println("The Error is: "+e);
    }

    System.out.println("The inputed data str is: "+str);

    System.out.println("The inputed data a is: "+a);
    }
}
```

23) Write a program to find out the grade of a student by entering the value from keyboard using switch case statement.

Ans:

```
class CommandLine
{
    public static void main(String[] args)
    {
      int marks;
            float c;
            marks = Integer.parseInt(args[0]);
            marks = marks/10;
            c = (float) marks /10;
```

```
                if (c >= 8.5f)
                {
                        marks=9;
                }
                switch(marks)
                {
                        case 10:
            case 9:
                                    System.out.println("Distiction");
                 break;
                        case 8:
                        case 7:
            case 6:
                                   System.out.println("First Class");
                 break;
            case 5:
                        case 4:
                                   System.out.println("Second Class");
                 break;
                        case 3:
            case 2:
            case 1:
                                   System.out.println("Fail");
                 break;
            default:
                                    System.out.println("Marks must be less than or equal to
    100");
                }
        }
    }
```

24) Write a program to describe the concept of a correct implementation of a producer and consumer.

Ans:

```
// A Correct Implmentation of a producer and consumer

class Q
{
        int n;
        boolean valueSet = false;

        synchronized int get()
        {
                if(!valueSet)
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println("Interrupted Exception");
                        }

                        System.out.println("Got: "+n);
                        valueSet = false;
                        notify();
                        return n;
        }

        synchronized void put(int n)
        {
                if(valueSet)
                {
                        try
                        {
                                wait();
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println("Interrupted Exception");
                        }


                        this.n = n;
                        valueSet = true;
                        System.out.println("Put: "+n);
                        notify();
                }
        }
}


class Producer implements Runnable
{
        Q q;

        Producer(Q q)
        {
                this.q = q;
                new Thread(this, "Producer").start();
        }
```

```
	public void run()
	{
		int i = 0;

		while(true)
		{
			q.put(i++);
		}
	}
}


class Consumer implements Runnable
{
	Q q;

	Consumer(Q q)
	{
		this.q = q;
		new Thread(this, "Consumer").start();
	}


	public void run()
	{
		while(true)
		{
			q.get();
		}
	}
}


class InterProcess
{
	public static void main(String args[])
	{
		Q q = new Q();
		new Producer(q);
		new Consumer(q);

		System.out.println("Press Control-C to stop.");
	}
}
```

25) Write a program to describe the concept of major model of an object.

Ans:

```
// Abstraction - Major element of object model.
// Define an abstract class which includes abstract methods
abstract class Figure
{
        double dim1;
        double dim2;

        Figure(double a, double b)
        {
                dim1 = a;
                dim2 = b;
        }

        abstract double area();
}

// Define a class "Rectangle" which inherits from the above abstract class
class Rectangle extends Figure
{
        Rectangle(double a, double b)
        {
                super(a, b);
        }

        double area()
        {
                System.out.println("Inside Area of Rectangle");
                return dim1 * dim2;
        }
}

// Define a class "Triangle" which inherits from the above abstract class
class Triangle extends Figure
{
        Triangle(double a, double b)
        {
                super(a, b);
        }

        double area()
        {
                System.out.println("Inside area of Triangle");
                return dim1 * dim2 / 2;
        }
}


// Define a main class
class AbsEle
{
        public static void main(String args[])
        {
                Rectangle r = new Rectangle(9, 5);
                Triangle t = new Triangle(10, 8);
                System.out.println("Area is: "+r.area());

                System.out.println("Area is: "+t.area());
        }
}
```

26) Write a program to describe the concept of encapsulation.

Ans:

```
// Encapsulation - Major element of object model.

// The following program shows a simple inheritance where encapsulation
// takes place. Encapsulation is the mechanism that binds together code and
// the data it manipulates, and keeps both safe from outside interference and
// misuse.

// Create a superclass.
class A
{
        int i,j;

        void showij()
        {
                System.out.println("i and j: " + i + " " +j);
        }
}

// Create a subclass by extending class A
class B extends A
{
        int k;

        void showk()
        {
                System.out.println("k: "+k);
        }

        void sum()
        {
                System.out.println("i+j+k: "+(i+j+k));
        }
}

class Encapsulate
{
        public static void main(String args[])
        {
                A superob = new A();
                B subob = new B();

                // The superclass may be used by itself.
                superob.i = 10;
                superob.j = 20;
                System.out.println("Contents of superob: ");
                superob.showij();
                System.out.println();
```

```
                    // The superclass has access to all public members of
                    // its superclass.

                    subob.i = 7;
                    subob.j = 8;
                    subob.k = 9;
                    System.out.println("Contents of subob: ");
                    subob.showij();
                    subob.showk();
                    System.out.println();

                    System.out.println("Sum of i, j and k in subob: ");
                    subob.sum();
          }
}
```

27) Write a program to describe the concept of abstraction.

Ans:

```
abstract class Animal
{
          abstract void voice(String j);

          void walk(String g)
          {
                    System.out.println("Each animal has four legs: "+g);
          }
}

class cat extends Animal
{
          void voice(String j)
          {
                    System.out.println("Cat Says: "+j);
          }
}

class dog extends Animal
{
          void voice(String i)
          {
                    System.out.println("Dog says: "+i);
          }
}

class ElAbstract
{
          public static void main(String args[])
          {
                    cat c = new cat();
                    c.voice("MAW MAW");
                    c.walk("CAT WALK");

                    dog d = new dog();
                    d.voice("BOW BOW");
```

```
                d.walk("DOG RUN");
        }
}
```

28) Write a program to describe the concept of multiple inheritance.

Ans:

```
// Multiple inheritance can be achieved by Interface

// Define an integer stack interface
interface intStack
{
        void push(int item);        // store an item
        int pop();                                  // retrieve an item
}

// Define a class for a "growable" class
class DynStack implements intStack
{
        private int stck[];
        private int tos;

        // allocate and initialize stack
        DynStack(int size)
        {
                stck = new int[size];
                tos = -1;
        }

        // Push an item onto the stack
        public void push(int item)        // if stack is full, allocate a larger stack
        {
                if(tos == stck.length-1)
                {
                        int temp[] = new int[stck.length * 2];        // double size
                        for(int i = 0; i < stck.length; i++)
                                temp[i] = stck[i];
                        stck = temp;
                        stck[++tos] = item;
                }
                else
                        stck[++tos] = item;
        }

        // Pop an item from the stack
        public int pop()
        {
                if(tos < 0)
                {
                        System.out.println("Stack underflow.");
                        return 0;
                }
                else
                        return stck[tos--];
        }
}
```

```
// Define a main class
class Hierarchy
{
	public static void main(String args[])
	{
		DynStack mystack1 = new DynStack(5);
		DynStack mystack2 = new DynStack(8);

		// these loops cause each stack to grow
		for(int i = 0; i < 12; i++)
			mystack1.push(i);
		for(int i = 0; i < 20; i++)
			mystack2.push(i);

		System.out.println("Stack in mystack1: ");
		for(int i = 0; i < 12; i++)
			System.out.println(mystack1.pop());

		System.out.println("Stack in mystack2: ");
		for(int i = 0; i < 20; i++)
			System.out.println(mystack2.pop());
	}
}
```

29) Write a program to describe the concept of polymorphism.

Ans:

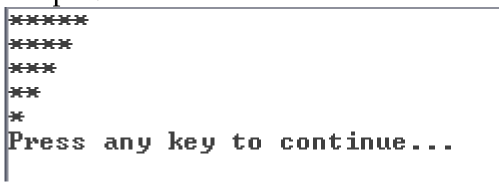
```
// The following program shows Dynamic Method Dispatch i.e. Dynamic Binding

class A
{
	void callme()
	{
		System.out.println("Inside A's callme method");
	}
}

class B extends A
{
	// Override callme()

	void callme()
	{
		System.out.println("Inside B's callme method");
	}
}

class C extends A
{
	// Override callme()

	void callme()
	{
		System.out.println("Inside C's callme method");
	}
}

class Typing
{
        public static void main(String args[])
        {
                A a = new A();   // object of type A
                B b = new B();  // object of type B
                C c = new C();  // object of type C

                A r;       // Obtain a reference of type A

                r = a;                // r refers to an A object
                r.callme(); // calls A's version of callme

                r = b;                // r refers to an B object
                r.callme(); // calls B's version of callme

                r = c;                // r refers to an C object
                r.callme(); // calls C's version of callme
        }}
```

30) Write a program to display various Pascal trees.
Ans:

First program:

```
class Pascal2
{
  public static void main(String args[])
    {
     for(int i=5;i>=1;i--)
       {
          for(int j=1;j<=i;j++)
             {
              System.out.print("*");
             }
          System.out.println();
        }
     }
}
```

Output:

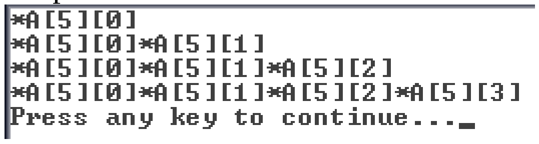


```
*****
****
***
**
*
Press any key to continue...
```

Second Program:

```
class Pascalloc
{
   public static void main(String args[])
     {
              int i,j,k;
            for( i=0;i<4;i++)
              {
                   for( j=i;j<=4;j++)
                      {
                       System.out.print("");
                      }
                     for(k=0;k<=i;k++)
                      {
                       System.out.print("*");
                          System.out.print("A[");
                       System.out.print(+ j);
                          System.out.print("]");
                          System.out.print("[");
                          System.out.print(+k );
                          System.out.print("]");
                      }
                  System.out.println("");
                }
            }
}
```

Output:



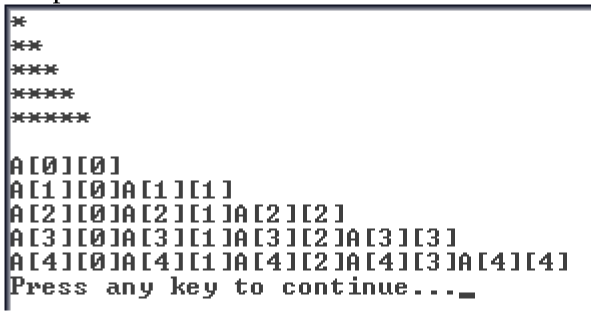
```
*A[5][0]
*A[5][0]*A[5][1]
*A[5][0]*A[5][1]*A[5][2]
*A[5][0]*A[5][1]*A[5][2]*A[5][3]
Press any key to continue...
```

Third Program:

```
class Pascalarr
{
  public static void main(String args[])
  {
    int i, j;
    int A[][]={  {0,0,0,0,0},
                  {0,0,0,0,0},
                  {0,0,0,0,0},
                  {0,0,0,0,0},
                  {0,0,0,0,0} };

             for(i=0;i<5;i++)
            {
              for(j=0;j<=i;j++)
              {
```

```
              System.out.print("*");
               A[i][j]=1;
             }
            System.out.println(" ");
           }
        System.out.println("");
         for(i=0;i<5;i++)
         {
           for(j=0;j<5;j++)
           {
             if(A[i][j]==1)
              {
                 System.out.print("A[");
                 System.out.print(+ i);
                 System.out.print("][");
                 System.out.print(+ j);
                 System.out.print("]");
              }
            }
          System.out.println(" ");
        }
     }
   }
```

Output:



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

A[0][0]
A[1][0]A[1][1]
A[2][0]A[2][1]A[2][2]
A[3][0]A[3][1]A[3][2]A[3][3]
A[4][0]A[4][1]A[4][2]A[4][3]A[4][4]
Press any key to continue..._
```

8M Questions

1) Describe the precedence and associativity rules for operators.
Ans:

Precedence and associativity rules are necessary for deterministic evaluation of expressions. The operators are summarized in below table. They are discussed in subsequent sections in this chapter.

The following remarks apply to below table:

- The operators are shown with decreasing precedence from the top of the table.

- Operators within the same row have the same precedence.
- Parentheses, `( )`, can be used to override precedence and associativity.
- The unary operators, which require one operand, include the postfix increment (`++`) and decrement (`--`) operators from the first row, all the prefix operators (`+`, `-`, `++`, `--`, `~`, `!`) in the second row, and the prefix operators (object creation operator `new`, cast operator `(type)`) in the third row.
- The conditional operator (`? :`) is ternary, that is, requires three operands.
- All operators not listed above as unary or ternary, are binary, that is, require two operands.
- All binary operators, except for the relational and assignment operators, associate from left to right. The relational operators are nonassociative.
- Except for unary postfix increment and decrement operators, all unary operators, all assignment operators, and the ternary conditional operator associate from right to left.

Table. Operator Summary

Postfix operators	`[] . (parameters) expression++ expression--`
Unary prefix operators	`++expression --expression +expression -expression ~ !`
Unary prefix creation and cast	`new (type)`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >> >>>`
Relational	`< <= > >= instanceof`
Equality	`== !=`
Bitwise/logical AND	`&`
Bitwise/logical XOR	`^`
Bitwise/logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Conditional	`?:`
Assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`

Precedence rules are used to determine which operator should be applied first if there are two operators with different precedence, and these follow each other in the expression. In such a case, the operator with the highest precedence is applied first.

`2 + 3 * 4` is evaluated as `2 + (3 * 4)` (with the result `14`) since `*` has higher precedence than `+`.

Associativity rules are used to determine which operator should be applied first if there are two operators with the same precedence, and these follow each other in the expression.

Left associativity implies grouping from left to right:

`1 + 2 - 3` is interpreted as `((1 + 2) - 3)`, since the binary operators `+` and `-` both have same precedence and left associativity.

Right associativity implies grouping from right to left:

`- - 4` is interpreted as `(- (- 4))` (with the result `4`), since the unary operator `-` has right associativity.

The precedence and associativity rules together determine the evaluation order of the operators.

2) Describe different kinds of type conversions that occur in program.
Ans:

Some type conversions must be explicitly stated in the program, while others are done implicitly. Some type conversions can be checked at compile time to guarantee their validity at runtime, while others will require an extra check at runtime.

Unary Cast Operator: (type)

Java, being a strongly typed language, checks for type compatibility (i.e., checks if a type can substitute for another type in a given context) at compile time. However, some checks are only possible at runtime (for example, which type of object a reference actually denotes during execution). In cases where an operator would have incompatible operands (for example, assigning a `double` to an `int`), Java demands that a cast be used to explicitly indicate the type conversion. The cast construct has the following syntax:

(<type>) <expression>

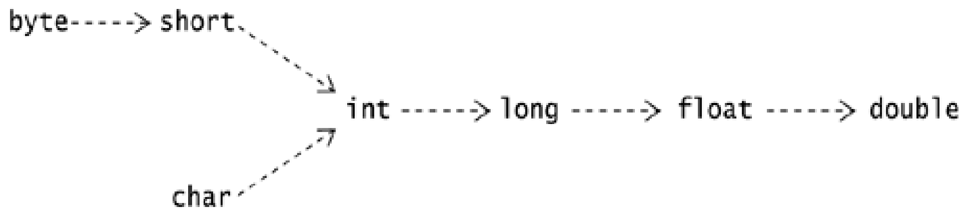
The cast (<type>) is applied to the value of the <expression>. At runtime, a cast results in a new value of <type>, which best represents the value of the <expression> in the old type. We use the term casting to mean applying the cast operator for explicit type conversion.

Casting can be applied to primitive values as well as references. Casting between primitive data types and reference types is not permitted. Boolean values cannot be cast to other data values, and vice versa. The reference literal `null` can be cast to any reference type.

Examples of casting between primitive data types are provided in this chapter.

Narrowing and Widening Conversions

For the primitive data types, the value of a narrower data type can be converted to a value of a broader data type without loss of information. This is called a widening primitive conversion. Widening conversions to the next broader type for primitive data types are summarized in following figure. The conversions shown are transitive. For example, an `int` can be directly converted to a `double` without first having to convert it to a `long` and a `float`.



Converting from a broader data type to a narrower data type is called a narrowing primitive conversion, which can result in loss of magnitude information. Any conversion which is not a widening conversion according to above figure is a narrowing conversion. Note that all conversions between `char` and the two integer types `byte` and `short` are considered narrowing conversions: the reason being that the conversions between the unsigned type `char` and the signed types `byte` or `short` can result in loss of information.

Widening and narrowing conversions are also defined for reference types. Conversions up the inheritance hierarchy are called widening reference conversions (a.k.a. upcasting). Conversions down the inheritance hierarchy are called narrowing reference conversions (a.k.a. downcasting).

Both narrowing and widening conversions can be either explicit (requiring a cast) or implicit. Widening conversions are usually done implicitly, whereas narrowing conversions typically require a cast. It is not illegal to use a cast for a widening conversion. However, the compiler will flag any conversions that require a cast.

3) Describe arithmetic operators along with examples.
Ans:

The arithmetic operators are used to construct mathematical expressions as in algebra. Their operands are of numeric type (which includes the `char` type).

Arithmetic Operator Precedence and Associativity

In below table the precedence of the operators is in decreasing order, starting from the top row, which has the highest precedence. Unary subtraction has higher precedence than multiplication. The operators in the same row have the same precedence. Binary multiplication, division, and remainder operators have the same precedence. The unary operators have right associativity, and the binary operators have left associativity.

Table . Arithmetic Operators

Unary	`+`	Addition	`-`	Subtraction		
Binary	`*`	Multiplication	`/`	Division	`%`	Remainder
	`+`	Addition	`-`	Subtraction		

Evaluation Order in Arithmetic Expressions

Java guarantees that the operands are fully evaluated from left to right before an arithmetic binary operator is applied. Of course, if evaluation of an operand causes an exception, the subsequent operands will not be evaluated.

In the expression `a + b * c`, the operand `a` will always be fully evaluated before the operand `b,` which will always be fully evaluated before the operand `c.` However, the multiplication operator `*` will be applied before the addition operator `+`, respecting the precedence rules. Note that `a`, `b`, and `c` can be arbitrary arithmetic expressions that have been determined to be the operands of the operators.

Integer Arithmetic

Integer arithmetic always returns a value that is in range, except in the case of integer division by zero and remainder by zero, which causes an `ArithmeticException` (see the division operator `/` and the remainder operator `%` below). A valid value does not necessarily mean that the result is correct, as demonstrated by the following examples:

```
int tooBig   = Integer.MAX_VALUE + 1;    // -2147483648 which is
Integer.MIN_VALUE.
int tooSmall = Integer.MIN_VALUE - 1;    //  2147483647 which is
Integer.MAX_VALUE.
```

The results above should be values that are out-of-range. However, integer arithmetic wraps if the result is out-of-range, that is, the result is reduced modulo in the range. In order to avoid wrapping of out-of-range values, programs should either use explicit checks or a wider type. If the type `long` is used in the examples above, the results would be correct in the `long` range:

```
long notTooBig   = Integer.MAX_VALUE + 1L;   //  2147483648L in range.
long notTooSmall = Integer.MIN_VALUE - 1L;   // -2147483649L in range.
```

Floating-point Arithmetic

Certain floating-point operations result in values that are out-of-range. Typically, adding or multiplying two very large floating-point numbers can result in an out-of-range value which is represented by Infinity. Attempting floating-point division by zero also returns infinity. The examples below show how this value is printed as signed infinity.

```
System.out.println( 4.0 / 0.0);          // Prints:  Infinity
System.out.println(-4.0 / 0.0);          // Prints: -Infinity
```

Multiplication Operator: *

Multiplication operator * multiplies two numbers.

```
int    sameSigns     = -4    * -8;         // result:  32
double oppositeSigns =  4.0  * -8.0;       // result: -32.0
int    zero          =  0    * -0;         // result:   0
```

Division Operator: /

The division operator `/` is overloaded. If its operands are integral, the operation results in integer division.

```
int    i1 = 4  / 5;   // result: 0
int    i2 = 8  / 8;   // result: 1
double d1 = 12 / 8;   // result: 1 by integer division. d1 gets the
value 1.0.
```

Integer division always returns the quotient as an integer value, i.e. the result is truncated toward zero. Note that the division performed is integer division if the operands have integral values, even if the result will be stored in a floating-point type.

4) Describe increment and decrement operator along with example.
Ans:

Variable increment (`++`) and decrement (`--`) operators come in two flavors: prefix and postfix. These unary operators have the side effect of changing the value of the arithmetic operand, which must evaluate to a variable. Depending on the operator used, the variable is either incremented or decremented by 1.

These operators are very useful for updating variables in loops where only the side effect of the operator is of interest.

Increment Operator ++

Prefix increment operator has the following semantics:

`++i` adds `1` to `i` first, then uses the new value of `i` as the value of the expression. It is equivalent to the following statements.

```
i += 1;
result = i;
return result;
```

Postfix increment operator has the following semantics:

`j++` uses the current value of `j` as the value of the expression first, then adds `1` to `j`. It is equivalent to the following statements:

```
result = j;
j += 1;
return result;
```

Decrement Operator --

Prefix decrement operator has the following semantics:

`--i` subtracts `1` from `i` first, then uses the new value of `i` as the value of the expression.

Postfix decrement operator has the following semantics:

`j--` uses the current value of `j` as the value of the expression first, then subtracts `1` from `j`.

Examples of Increment and Decrement Operators

```
// (1) Prefix order: increment operand before use.
int i = 10;
int k = ++i + --i;  // ((++i) + (--i)). k gets the value 21 and i
becomes 10.
--i;                // Only side effect utilized. i is 9. (expression
statement)

// (2) Postfix order: increment operand after use.
long i = 10;
long k = i++ + i--;  // ((i++) + (i--)). k gets the value 21L and i
becomes 10L.
```

```
i++;                // Only side effect utilized. i is 11L.
(expression statement)
```

An increment or decrement operator, together with its operand can be used as an expression statement.

Execution of the assignment in the second declaration in (1) proceeds as follows :

```
k = ((++i) + (--i))          Operands are determined.
k = ( 11   + (--i))          i now has the value 11.
k = ( 11   +  10)            i now has the value 10.
k = 21
```

Expressions where variables are modified multiple times during the evaluation should be avoided, because the order of evaluation is not always immediately apparent.

We cannot associate increment and decrement operators. Given that `a` is a variable, we cannot write `(++(++a))`. The reason is that any operand to `++` must evaluate to a variable, but the evaluation of `(++a)` results in a value.

In the case where the operand is of type `char`, `byte`, or `short`, both binary numeric promotion and an implicit narrowing conversion are performed to achieve the side effect of modifying the value of the operand. In the example below, the `int` value of `(++b)` (i.e., `11`), is assigned to `int` variable `i`. The side effect of incrementing the value of `byte` variable `b` requires binary numeric promotion to perform `int` addition, followed by an implicit narrowing conversion of the `int` value to `byte`.

```
byte b = 10;
int  i = ++b;         // i is 11, and so is b.
```

The increment and decrement operators can also be applied to floating-point operands. In the example below, the side effect of the ++ operator is overwritten by the assignment.

```
double x = 4.5;
x = x + ++x;          // x gets the value 10.0.
```

5) By giving examples describe relational operators.
Ans:

Given that `a` and `b` represent numeric expressions, the relational (also called comparison) operators are defined as shown in below table.

Table. Relational Operators

`a < b`	`a` less than `b`?
`a <= b`	`a` less than or equal to `b`?

Table. Relational Operators

`a < b`	`a` less than `b`?
`a > b`	`a` greater than `b`?
`a >= b`	`a` greater than or equal to `b`?

All relational operators are binary operators, and their operands are numeric expressions. Binary numeric promotion is applied to the operands of these operators. The evaluation results in a `boolean` value. Relational operators have precedence lower than arithmetic operators, but higher than that of the assignment operators.

```
double  hours = 45.5;
boolean overtime = hours >= 35.0;     // true.
boolean order = 'A' < 'a';            // true. Binary numeric promotion
applied.
```

Relational operators are nonassociative. Mathematical expressions like a   b   c must be written using relational and boolean logical/conditional operators.

```
int a = 1, b = 7, c = 10;
boolean valid1 = a <= b <= c;             // (1) Illegal.
boolean valid2 = a <= b && b <= c;        // (2) OK.
```

Since relational operators have left associativity, the evaluation of the expression `a <= b <= c` at (1) in the examples above would proceed as follows: `((a <= b) <= c)`. Evaluation of `(a <= b)` would yield a `boolean` value that is not permitted as an operand of a relational operator, that is, `(`<boolean value> `<= c)` would be illegal.

6) Describe integer bitwise operators by giving examples.
Ans:

Integer bitwise operators include the unary operator `~` (bitwise complement) and the binary operators `&` (bitwise AND), `|` (bitwise inclusive OR), and `^` (bitwise exclusive OR, a.k.a. bitwise XOR).

The binary bitwise operators perform bitwise operations between corresponding individual bit values in the operands. Unary numeric promotion is applied to the operand of the unary bitwise complement operator `~`, and binary numeric promotion is applied to the operands of the binary bitwise operators. The result is a new integer value of the promoted type, which can only be either `int` or `long`.

Given that `A` and `B` are corresponding bit values (either `0` or `1`) in the left-hand and right-hand operands, respectively, these bitwise operators are defined as shown in below table. The operators are listed in decreasing precedence order.

The operators `&`, `|`, and `^` can also be applied to boolean operands to perform boolean logical operations.

Table. Integer Bitwise Operators

Operator Name	Notation	Effect on Each Bit of the Binary Representation
Bitwise complement	`~A`	Invert the bit value: `1` to `0`, `0` to `1`.
Bitwise AND	`A & B`	`1` if both bits are `1`; otherwise, `0`.
Bitwise OR	`A | B`	`1` if either or both bits are `1`; otherwise, `0`.
Bitwise XOR	`A ^ B`	`1` if and only if one of the bits is `1`; otherwise, `0`.

The result of applying bitwise operators between two corresponding bits in the operands is shown in below table, where `A` and `B` are corresponding bit values in left-hand and right-hand operands, respectively.

Table. Result Table for Bitwise Operators

A	B	~A	A & B	A | B	A ^ B
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Examples of Bitwise Operator Application

```
char v1 = ')';          // Unicode value 41
byte v2 = 13;

int result1 = ~v1;      // -42
int result2 = v1 & v2;  // 9
int result3 = v1 | v2;  // 45
int result4 = v1 ^ v2;  // 36
```

Chapter 2

2M Questions

1) What is class in Java?
Ans:

One of the fundamental ways in which we handle complexity is abstractions. An abstraction denotes the essential properties and behaviors of an object that differentiate it from other objects. The essence of OOP is modelling abstractions, using classes and objects. The hard part in this endeavour is finding the right abstractions.

A class denotes a category of objects, and acts as a blueprint for creating such objects. A class models an abstraction by defining the properties and behaviors for the objects representing the abstraction. An object exhibits the properties and behaviors defined by its class. The properties of an object of a class are also called attributes, and are defined by fields in Java. A field in a class definition is a variable which can store a value that represents a particular property. The behaviors of an object of a class are also known as operations, and are defined using methods in Java. Fields and methods in a class definition are collectively called members.

An important distinction is made between the contract and the implementation that a class provides for its objects. The contract defines what services, and the implementation defines how these services are provided by the class. Clients (i.e., other objects) only need to know the contract of an object, and not its implementation, in order to avail themselves of the object's services.

2) State the general form of a method.
Ans:

The general syntax of a method declaration is

```
<method modifiers> <return type> <method name> (<formal parameter list>)
      <throws clause> // Method prototype
{ // Method body
   <local variable declarations>
   <nested local class declarations>
   <statements>
}
```

In addition to the name of the method, the method prototype can specify the following information:

- scope or accessibility modifier

- additional method modifiers
- the type of the return value, or `void` if the method does not return any value a formal parameter list
- checked exceptions thrown by the method in a `throws` clause

The formal parameter list is a comma-separated list of parameters for passing information to the method when the method is invoked by a method call. An empty parameter list must be specified by `( )`. Each parameter is a simple variable declaration consisting of its type and name:

<parameter modifier> <type> <parameter name>

The parameter names are local to the method. The signature of a method comprises the method name and the formal parameter list only.

The method body is a block containing the local declarations and the statements of the method.

3) Enlist different types of classes available in Java.

Ans:

The following are the different types of classes:

a. Local class
b. Anonymous class
c. final class
d. abstract class
e. Wrapper class

4) Which operator is used to instantiate an object? Give example.

Ans:

"new" operator is used to instantiate an object.

e.g.

Student s1 = new Student();

5) What is the use of constructor? Give syntax to define a constructor.

Ans:

The main purpose of constructors is to set the initial state of an object when the object is created by using the `new` operator.

A constructor has the following general syntax:

<accessibility modifier> <class name> (<formal parameter list>)

```
        <throws clause> // Constructor header
 { // Constructor body
   <local variable declarations>
   <nested local class declarations>
   <statements>
 }
```

Constructor declarations are very much like method declarations. However, the following restrictions on constructors should be noted:

- Modifiers other than an accessibility modifier are not permitted in the constructor header.
- Constructors cannot return a value and, hence, cannot specify a return type, not even `void`, in the constructor header, but they can contain the simple form of the `return` statement in the constructor body.
- Constructor name must be the same as the class name.

4M Questions

1) Explain the following statement:

   HelloWorld h1 = new HelloWorld();

Ans:
In the above statement, the first execution is to call constructor.

a. So before creating an object, java interpreter calls constructor HelloWorld().
b. Next new operator is called to create an object in heap memory.
c. Next s1 reference has been created in the stack frame.
d. Next using "=" operator the reference s1 has been linked with the object which is present in heap memory.

2) Describe the general form of a class.

Ans:

class declaration introduces a new reference type. It has the following general syntax:

```
<class modifiers> class <class name>
        <extends clause> <implements clause> // Class header
{ // Class body
    <field declarations>
    <method declarations>
    <nested class declarations>
    <nested interface declarations>
    <constructor declarations>
    <initializer blocks>
}
```

In the class header, the name of the class is preceded by the keyword `class`. In addition, the class header can specify the following information:

- scope or accessibility modifier
- additional class modifiers
- any class it extends
- any interfaces it implements

The class body can contain member declarations which comprise

- field declarations
- method declarations
- nested class and interface declarations

Members declared `static` belong to the class and are called static members, and non-static members belong to the objects of the class and are called instance members. In addition, the following can be declared in a class body:

- constructor declarations
- static and instance initializer blocks

The member declarations, constructor declarations and initializer blocks can appear in any order in the class body.

In order to understand what code is legal to declare in a class, we distinguish between static context and non-static context. A static context is defined by static methods, static field initializers, and static initializer blocks. A non-static context is defined by instance methods, constructors, non-static field initializers, and instance initializer blocks. By static code we mean expressions and statements in a static context, and similarly by non-static code we mean expressions and statements in a non-static context. One crucial difference between the two contexts is that static code cannot refer to non-static members.

3) Write a program to describe the use of class.
Ans:
The following sample program shows the scope of class members.

```
class SuperName {
    int instanceVarInSuper;
    static int staticVarInSuper;

    void instanceMethodInSuper()      { /* ... */ }
    static void staticMethodInSuper() { /* ... */ }
    // ...
}

class ClassName extends SuperName {
    int instanceVar;
    static int staticVar;

    void instanceMethod()       { /* ... */ }
    static void staticMethod() { /* ... */ }
    // ...
}
```

4) Write a program to define the method in a class and also include the parameter passing and returning a value in class.

Ans:

Source code:

```
import java.util.*;
interface Stack
{
   void push(int x);
   int pop();
}

class Fixedstack implements Stack
{

    int s[] = new int[5];
    int top=-1;

 public void push(int x)
  {
    if(top==s.length-1)
     {
        System.out.println("Fixed Stack overflow");
     }
    else
     {
       top++;
       s[top]=x;
     }
  }

 public int  pop()
  {
    int t;
   if(top<0)
```

```
         {
           System.out.println("Fixed Stack underflow:");
           t = -1;
         }
       else
        {
          t=s[top];
          top--;
        }
       return t;
       }
     }

     class Dynamicstack implements Stack
     {

          Vector S;
          int top;

      Dynamicstack()
           {
            S = new Vector();
            top = -1;
          }
      public void push(int x)
       {

          top++;
          S.addElement(new Integer(x));
       }


      public int  pop()
       {
          int t;
            Integer n;
         if(top == -1)
          {
            System.out.println("Dynamic Stack underflow:");
            t = -1;
          }
       else
        {
          n=(Integer)S.elementAt(top);
          top--;
            t = n.intValue();
        }
       return t;
       }
     }

      class Q1
     {
        public static void main(String args[])
         {
           Fixedstack f=new Fixedstack();
           Dynamicstack d = new Dynamicstack();
            int a[]={10,20,30,40,50,60,70};
            int i,j;

            for(i=0;i<a.length;i++)
             {
                f.push(a[i]);
                              d.push(a[i]);
             }
            System.out.println("Popped from FIXED STACK :");
            for(i=0;i<a.length;i++)
             {
                j = f.pop();
                              if (j != -1)
                                              System.out.println(i+" : " +j);
             }
                  System.out.println("Popped from DYNAMIC STACK :");
            for(i=0;i<a.length+2;i++)
             {
                j = d.pop();
                              if (j != -1)
                                              System.out.println(i+" : " +j);
             }

         }

     }
```

5) Describe access modifiers.
Ans:

The following chart shows the access level permitted by each specifier.

Specifier	**class**	**subclass**	**package**	**world**
`private`	X			
`protected`	X	X*	X	
`public`	X	X	X	X
`package`	X		X	

The first column indicates whether the class itself has access to the member defined by the access specifier. As you can see, a class always has access to its own members. The second column indicates whether subclasses of the class (regardless of which package they are in) have access to the member. The third column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The fourth column indicates whether all classes have access to the member.

6) Write java program to define the access modifiers.
Ans:

```
    package p1;

public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
    }
```

7) Write java program to pass object reference values.
Ans:

```
public class CustomerTwo {
    public static void main (String[] args) {
        Pizza favoritePizza = new Pizza();          // (1)
        System.out.println("Meat on pizza before baking: " +
favoritePizza.meat);
        bake(favoritePizza);                        // (2)
        System.out.println("Meat on pizza after baking: " +
favoritePizza.meat);
    }
    public static void bake(Pizza pizzaToBeBaked) { // (3)
        pizzaToBeBaked.meat = "chicken";  // Change the meat on the
pizza.
```

```
        pizzaToBeBaked = null;                          // (4)
    }
}

class Pizza {                                           // (5)
    String meat = "beef";
}
```

Output from the program:

```
Meat on pizza before baking: beef
Meat on pizza after baking: chicken
```

8) What is an array? Give example.

Ans:

Array is a collection of similar type of elements.

```
// This program demonstrates the length array member.
class Length {
public static void main(String args[]) {
- 128 -
int a1[] = new int[10];
int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
}
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use.

9) Write a java program to demonstrate string array.

Ans:

```
// Demonstrate String arrays.
class StringDemo3 {
public static void main(String args[]) {
String str[] = { "one", "two", "three" };
for(int i=0; i<str.length; i++)
System.out.println("str[" + i + "]: " +
str[i]);
}
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
    str[2]: three
```

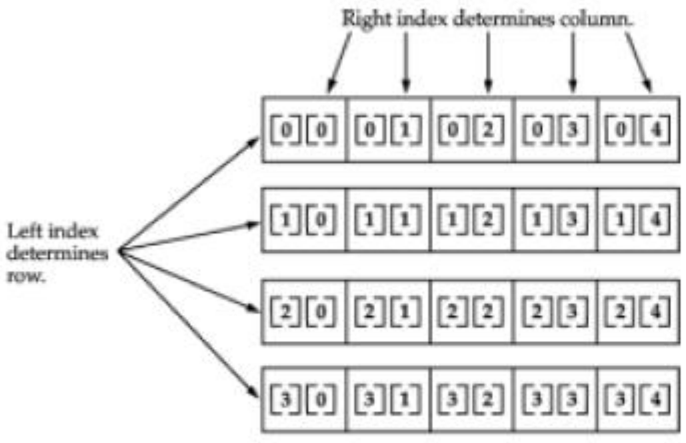
10) Describe multidimensional array.
Ans:

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in following Figure

Figure:



Given: int twoD [ ] [ ] = new int [4] [5] ;

11) Write a java program to demonstrate two dimensional array.
Ans:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
```

```
10 11 12 13 14
15 16 17 18 19
```

8M

1. Describe a “String” class.
Ans:

```
The syntax of String class is as follows:

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.
Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
          String str = "abc";
```

is equivalent to:

```
          char data[] = {'a', 'b', 'c'};
          String str = new String(data);
```

Handling character strings is supported through two `final` classes: `String` and `StringBuffer`. The `String` class implements immutable character strings, which are read-only once the string has been created and initialized, whereas the `StringBuffer` class implements dynamic character strings.

Character strings implemented using these classes are genuine objects, and the characters in such a string are represented using 16-bit characters.

This section discusses the class `String` that provides facilities for creating, initializing, and manipulating character strings. The next section discusses the `StringBuffer` class.

Creating and Initializing Strings

String Literals Revisited

The easiest way of creating a `String` object is using a string literal:

```
String str1 = "You cannot change me!";
```

A string literal is a reference to a `String` object. The value in the `String` object is the character sequence enclosed in the double quotes of the string literal. Since a string literal is a reference, it can be manipulated like any other `String` reference. The reference value of a string literal can be assigned to another `String` reference: the reference `str1` will denote the `String` object with the value `"You cannot change me!"` after the assignment above. A string literal can be used to invoke methods on its `String` object:

```
int len = "You cannot change me!".length(); // 21
```

The compiler optimizes handling of string literals (and compile-time constant expressions that evaluate to strings): only one `String` object is shared by all string-valued constant expressions with the same character sequence. Such strings are said to be interned, meaning that they share a unique `String` object if they have the same content. The `String` class maintains a private pool where such strings are interned.

```
String str2 = "You cannot change me!";
```

Both `String` references `str1` and `str2` denote the same `String` object, initialized with the character string: `"You cannot change me!"`. So does the reference `str3` in the following code. The compile-time evaluation of the constant expression involving the two string literals, results in a string that is already interned:

```
String str3 = "You cannot" + " change me!"; // Compile-time constant
expression
```

In the following code, both the references `can1` and `can2` denote the same `String` object that contains the string `"7Up"`:

```
String can1 = 7 + "Up";  // Value of compile-time constant expression:
"7Up"
String can2 = "7Up";     // "7Up"
```

However, in the code below, the reference `can4` will denote a new `String` object that will have the value `"7Up"` at runtime:

```
String word = "Up";
String can4 = 7 + word;  // Not a compile-time constant expression.
```

The sharing of `String` objects between string-valued constant expressions poses no problem, since the `String` objects are immutable. Any operation performed on one `String` reference will never have any effect on the usage of other references denoting the same object. The `String` class is also declared `final`, so that no subclass can override this behavior.

String Constructors

The `String` class has numerous constructors to create and initialize `String` objects based on various types of arguments. The following shows two of them:

```
String(String s)
```

This constructor creates a new `String` object, whose contents are the same as those of the `String` object passed as argument.

```
String()
```

This constructor creates a new `String` object, whose content is the empty string, `""`.

In the following code, the `String` object denoted by `str4` is different from the `String` object passed as argument:

```
String str4 = new String("You cannot change me!");
```

Constructing `String` objects can also be done from arrays of bytes, arrays of characters, or string buffers:

```
byte[] bytes = {97, 98, 98, 97};
char[] characters = {'a', 'b', 'b', 'a'};
StringBuffer strBuf = new StringBuffer("abba");
//...
String byteStr = new String(bytes);      // Using array of bytes:
"abba"
String charStr = new String(character);  // Using array of chars:
"abba"
String buffStr = new String(strBuf);     // Using string buffer: "abba"
```

2. Write a java program to construct “String”.
Ans:

```
public class StringConstruction {

    static String str1 = "You cannot change me!";            //
Interned

    public static void main(String[] args) {
        String emptyStr = new String();                      // ""
        System.out.println("emptyStr: " + emptyStr);

        String str2 = "You cannot change me!";               //
Interned
        String str3 = "You cannot" + " change me!";          //
Interned
        String str4 = new String("You cannot change me!");   // New
String object

        String words = " change me!";
        String str5 = "You cannot" + words;                  // New
String object

        System.out.println("str1 == str2:                 " +
                           (str1 == str2));                  // (1)
true
```

```
        System.out.println("str1.equals(str2):            " +
                           str1.equals(str2));                // (2)
true

        System.out.println("str1 == str3:                 " +
                           (str1 == str3));                   // (3)
true
        System.out.println("str1.equals(str3):            " +
                           str1.equals(str3));                // (4)
true

        System.out.println("str1 == str4:                 " +
                           (str1 == str4));                   // (5)
false
        System.out.println("str1.equals(str4):            " +
                           str1.equals(str4));                // (6)
true

        System.out.println("str1 == str5:                 " +
                           (str1 == str5));                   // (7)
false
        System.out.println("str1.equals(str5):            " +
                           str1.equals(str5));                // (8)
true
        System.out.println("str1 == Auxiliary.str1:       " +
                           (str1 == Auxiliary.str1));         // (9)
true
        System.out.println("str1.equals(Auxiliary.str1): " +
                           str1.equals(Auxiliary.str1));      // (10)
true

        System.out.println("\"You cannot change me!\".length(): " +
                           "You cannot change me!".length());// (11) 21
    }
}

class Auxiliary {
    static String str1 = "You cannot change me!";            //
Interned
}
```

Output from the program:

```
emptyStr:
str1 == str2:                 true
str1.equals(str2):            true
str1 == str3:                 true
str1.equals(str3):            true
str1 == str4:                 false
str1.equals(str4):            true
str1 == str5:                 false
str1.equals(str5):            true
str1 == Auxiliary.str1:       true
str1.equals(Auxiliary.str1): true
"You cannot change me!".length(): 21
```

3. Write a java program to read characters from a string.
Ans:
```
public class ReadingCharsFromString {
    public static void main(String[] args) {
        int[] frequencyData = new int [Character.MAX_VALUE];// (1)
        String str = "You cannot change me!";               // (2)

        // Count the frequency of each character in the string.
        for (int i = 0; i < str.length(); i++)              // (3)
            try {
                frequencyData[str.charAt(i)]++;             // (4)
            } catch(StringIndexOutOfBoundsException e) {
                System.out.println("Index error detected: "+ i +" not 
in range.");
            }
        // Print the character frequency.
        System.out.println("Character frequency for string: \"" + str + 
"\"");
        for (int i = 0; i < frequencyData.length; i++)
            if (frequencyData[i] != 0)
                System.out.println((char)i + " (code "+ i +"): " +
                                   frequencyData[i]);

        System.out.println("Copying into a char array:");
        char[] destination = new char [str.length()];
        str.getChars( 0,            7, destination, 0);     // (5) "You 
can"
        str.getChars(10, str.length(), destination, 7);     // (6) " 
change me!"
        // Print the character array.
        for (int i = 0; i < 7 + (str.length() - 10); i++)
            System.out.print(destination[i]);
        System.out.println();
    }
}
```

Output from the program:

```
Character Frequency for string: "You cannot change me!"
  (code 32): 3
! (code 33): 1
Y (code 89): 1
a (code 97): 2
c (code 99): 2
e (code 101): 2
g (code 103): 1
h (code 104): 1
m (code 109): 1
n (code 110): 3
o (code 111): 2
t (code 116): 1
u (code 117): 1
Copying into a char array:
You can change me!
```

4. Write short note on “Comparing Strings”.
Ans:

Characters are compared based on their integer values.

```
boolean test = 'a' < 'b';    // true since 0x61 < 0x62
```

Two strings are compared lexicographically, as in a dictionary or telephone directory, by successively comparing their corresponding characters at each position in the two strings, starting with the characters in the first position. The string `"abba"` is less than `"aha"`, since the second character `'b'` in the string `"abba"` is less than the second character `'h'` in the string `"aha"`. The characters in the first position in each of these strings are equal.

The following public methods can be used for comparing strings:

```
boolean equals(Object obj)
boolean equalsIgnoreCase(String str2)
```

The `String` class overrides the `equals()` method from the `Object` class. The `String` class `equals()` method implements `String` object value equality as two `String` objects having the same sequence of characters. The `equalsIgnoreCase()` method does the same, but ignores the case of the characters.

```
int compareTo(String str2)
int compareTo(Object obj)
```

The first `compareTo()` method compares the two strings and returns a value based on the outcome of the comparison:

- the value `0`, if this string is equal to the string argument
- a value less than `0`, if this string is lexicographically less than the string argument
- a value greater than `0`, if this string is lexicographically greater than the string argument

The second `compareTo()` method (required by the `Comparable` interface) behaves like the first method if the argument `obj` is actually a `String` object; otherwise, it throws a `ClassCastException`.

Here are some examples of string comparisons:

```
String strA = new String("The Case was thrown out of Court");
String strB = new String("the case was thrown out of court");

boolean b1 = strA.equals(strB);             // false
boolean b2 = strA.equalsIgnoreCase(strB);   // true
```

```
String str1 = new String("abba");
String str2 = new String("aha");

int compVal1 = str1.compareTo(str2);         // negative value => str1
< str2
```

5. Describe short note on Searching for characters and substrings.
Ans:

The following overloaded methods can be used to find the index of a character, or the start index of a substring in a string. These methods search forward toward the end of the string. In other words, the index of the first occurrence of the character or substring is found. If the search is unsuccessful, the value –1 is returned.

```
int indexOf(int ch)
```

Finds the index of the first occurrence of the argument character in a string.

```
int indexOf(int ch, int fromIndex)
```

Finds the index of the first occurrence of the argument character in a string, starting at the index specified in the second argument. If the index argument is negative, the index is assumed to be 0. If the index argument is greater than the length of the string, it is effectively considered to be equal to the length of the string—returning the value -1.

```
int indexOf(String str)
```

Finds the start index of the first occurrence of the substring argument in a string.

```
int indexOf(String str, int fromIndex)
```

Finds the start index of the first occurrence of the substring argument in a string, starting at the index specified in the second argument.

The `String` class also defines a set of methods that search for a character or a substring, but the search is backward toward the start of the string. In other words, the index of the last occurrence of the character or substring is found.

```
int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

The following method can be used to create a string in which all occurrences of a character in a string have been replaced with another character:

```
String replace(char oldChar, char newChar)
```

Examples of search methods:

```
String funStr = "Java Jives";
//               0123456789

String newStr = funStr.replace('J', 'W');      // "Wava Wives"

int jInd1a = funStr.indexOf('J');              // 0
int jInd1b = funStr.indexOf('J', 1);           // 5
int jInd2a = funStr.lastIndexOf('J');          // 5
int jInd2b = funStr.lastIndexOf('J', 4);       // 0

String banner = "One man, One vote";
//               01234567890123456

int subInd1a = banner.indexOf("One");          // 0
int subInd1b = banner.indexOf("One", 3);       // 9
int subInd2a = banner.lastIndexOf("One");      // 9
int subInd2b = banner.lastIndexOf("One", 10); // 9
int subInd2c = banner.lastIndexOf("One", 8);  // 0
int subInd2d = banner.lastIndexOf("One", 2);  // 0
```

Extracting Substrings

```
String trim()
```

This method can be used to create a string where white space (in fact all characters with values less than or equal to the space character `'\u0020'`) from the front (leading) and the end (trailing) of a string has been removed.

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

The `String` class provides these overloaded methods to extract substrings from a string. A new `String` object containing the substring is created and returned. The first method extracts the string that starts at the given index `startIndex` and extends to the end of the string. The end of the substring can be specified by using a second argument `endIndex` that is the index of the first character after the substring, that is, the last character in the substring is at index `endIndex-1`. If the index value is not valid, a `StringIndexOutOfBoundsException` is thrown.

Examples of extracting substrings:

```
String utopia = "\t\n  Java Nation \n\t  ";
utopia = utopia.trim();                          // "Java Nation"
utopia = utopia.substring(5);                    // "Nation"
String radioactive = utopia.substring(3,6);      // "ion"
```

5. Describe wrapper classes along with example.
Ans:

Wrapper classes were introduced with the discussion of the primitive data types. Primitive values in Java are not objects. In order to manipulate these values as objects, the `java.lang` package provides a wrapper class for each of the primitive data types. All wrapper classes are `final`. The objects of all wrapper classes that can be instantiated are immutable, that is, their state cannot be changed.

Although the `Void` class is considered a wrapper class, it does not wrap any primitive value and is not instantiable (i.e., has no `public` constructors). It just denotes the `Class` object representing the keyword `void`. The `Void` class will not be discussed further in this section.

In addition to the methods defined for constructing and manipulating objects of primitive values, the wrapper classes also define useful constants, fields, and conversion methods.

Common Wrapper Class Constructors

The `Character` class has only one `public` constructor, taking a `char` value as parameter. The other wrapper classes all have two `public` one-argument constructors: one takes a primitive value and the other takes a string.

```
WrapperType( type v )
WrapperType( String str )
```

Converting Primitive Values to Wrapper Objects

A constructor that takes a primitive value can be used to create wrapper objects.

```
Character charObj1   = new Character('\n');
Boolean   boolObj1   = new Boolean(true);
Integer   intObj1    = new Integer(2003);
Double    doubleObj1 = new Double(3.14);
```

6. Write a java program to represent the string of integers.
Ans:

```
public class IntegerRepresentation {
    public static void main(String[] args) {
        int positiveInt = +41;    // 051, 0x29
        int negativeInt = -41;    // 037777777727, -051, 0xffffffd7, -
0x29
        System.out.println("String representation for decimal value: "
                           + positiveInt);
        integerStringRepresentation(positiveInt);
        System.out.println("String representation for decimal value: "
                           + negativeInt);
        integerStringRepresentation(negativeInt);
    }

    public static void integerStringRepresentation(int i) {
        System.out.println("    Binary:\t\t" +
Integer.toBinaryString(i));
        System.out.println("    Hex:\t\t"    + Integer.toHexString(i));
        System.out.println("    Octal:\t\t"  +
Integer.toOctalString(i));
        System.out.println("    Decimal:\t"  + Integer.toString(i));

        System.out.println("    Using toString(int i, int base)
method:");
        System.out.println("    Base 2:\t\t" + Integer.toString(i, 2));
        System.out.println("    Base 16:\t"  + Integer.toString(i,
16));
        System.out.println("    Base 8:\t\t" + Integer.toString(i, 8));
        System.out.println("    Base 10:\t"  + Integer.toString(i,
10));
    }
}
```

Output from the program:

```
String representation for decimal value: 41
    Binary:     101001
    Hex:        29
    Octal:      51
    Decimal:    41
    Using toString(int i, int base) method:
    Base 2:     101001
    Base 16:    29
    Base 8:     51
    Base 10:    41
String representation for decimal value: -41
    Binary:     11111111111111111111111111010111
    Hex:        ffffffd7
    Octal:      37777777727
    Decimal:    -41
    Using toString(int i, int base) method:
    Base 2:     -101001
    Base 16:    -29
    Base 8:     -51
    Base 10:    -41
```

7. Describe the meaning and usage of “Vector”.
Ans:

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the collections framework. With the release of Java 2, **Vector** was reengineered to extend **AbstractList** and implement the **List** interface, so it now is fully compatible with collections.
Here are the **Vector** constructors:
Vector( )
Vector(int *size*)
Vector(int *size*, int *incr*)
Vector(Collection *c*)
The first form creates a default vector, which has an initial size of 10. The second form

creates a vector whose initial capacity is specified by *size.* The third form creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr.* The increment specifies the number of elements to allocate each time that a vector is resized upward. The fourth form creates a vector that contains the elements of collection *c*. This constructor was added by Java 2.
All vectors start with an initial capacity. After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place. This reduction is important, because allocations are costly in terms of time. The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If you don't specify an increment, the vector's size is doubled by each allocation cycle.
Vector defines these protected data members:

```
int capacityIncrement;
int elementCount;
Object elementData[ ];
```

The increment value is stored in **capacityIncrement**. The number of elements currently in the vector is stored in **elementCount**. The array that holds the vector is stored in **elementData**.
In addition to the collections methods defined by **List**, **Vector** defines several legacy methods.
Because **Vector** implements **List**, you can use a vector just like you use an **ArrayList** instance. You can also manipulate one using its legacy methods. For example, after you instantiate a **Vector**, you can add an element to it by calling **addElement( )**. To obtain the element at a specific location, call **elementAt( )**. To obtain the first element in the vector, call **firstElement( )**. To retrieve the last element, call **lastElement( )**. You can obtain the index of an element by using **indexOf( )** and **lastIndexOf( )**. To remove an element, call **removeElement( )** or **removeElementAt( )**.

8. Write a java program to demonstrate the use of Vector.
Ans:

```
import java.util.*;
import java.io.*;
class Vec
{
        public static void main(String[] args)
        {
                Vector v1=new Vector();
      Vector v2=new Vector();
                Vector v3=new Vector();
                Vector v4=new Vector();

                int item=0,qty=0,t1,t2,tot=0;
                char ch = 'Y';
                DataInputStream x = new DataInputStream(System.in);

                v1.insertElementAt(new Integer(1),0);
                v1.insertElementAt(new String("Shirt"),1);
                v1.insertElementAt(new Integer(50),2);
                v1.insertElementAt(new Integer(150),3);

                v2.insertElementAt(new Integer(2),0);
                v2.insertElementAt(new String("Pants"),1);
                v2.insertElementAt(new Integer(75),2);
                v2.insertElementAt(new Integer(350),3);

                v3.insertElementAt(new Integer(1),0);
                v3.insertElementAt(new String("Shirt"),1);
                v3.insertElementAt(new Integer(0),2);
                v3.insertElementAt(new Integer(0),3);

                v4.insertElementAt(new Integer(2),0);
                v4.insertElementAt(new String("Pants"),1);
                v4.insertElementAt(new Integer(0),2);
                v4.insertElementAt(new Integer(0),3);

                System.out.println("ITEM\tNAME \tQUANTITY\tCOST(PER ITEM)");
     System.out.println(v1.elementAt(0)+"
\t"+v1.elementAt(1)+"\t"+v1.elementAt(2)+"      \t"+v1.elementAt(3));
                System.out.println(v2.elementAt(0)+"
\t"+v2.elementAt(1)+"\t"+v2.elementAt(2)+"      \t"+v2.elementAt(3));

                try
          {
                     while(ch == 'Y' || ch == 'y')
                     {

                System.out.print("Enter Item No. to Purchase - ");
                               item = Integer.parseInt(x.readLine());
           System.out.print("Enter Quantity to Purchase - ");
                               qty = Integer.parseInt(x.readLine());
                               System.out.print("Purchase More?(Y/N) - ");
                               ch = (x.readLine()).charAt(0);
                               switch(item)
                                {
               case 1 :
                                                t1 = ((Integer)v3.elementAt(2)).intValue();
                                                v3.add(2,new Integer(t1+qty));
                                          break;
               case 2 :
                                                t1 = ((Integer)v4.elementAt(2)).intValue();
                 v4.add(2,new Integer(t1+qty));
                                          break;
               default:
```

```
                                                System.out.println("Wrong Item
Entered.");
                              }
        }

        t1 = ((Integer)v3.elementAt(2)).intValue();
                  v3.add(3,new Integer(t1*150));
                  t2 = ((Integer)v4.elementAt(2)).intValue();
                  v4.add(3,new Integer(t2*350));
        tot = (t1*150)+(t2*350);

       }catch (Exception e)   { }

     System.out.println("\n\nYOUR SHOPPING SUMMARY\n\n");
     System.out.println("ITEM\tNAME \tQUANTITY\tCOST");
     System.out.println(v3.elementAt(0)+"
\t"+v3.elementAt(1)+"\t"+v3.elementAt(2)+"      \t"+v3.elementAt(3));
               System.out.println(v4.elementAt(0)+"
\t"+v4.elementAt(1)+"\t"+v4.elementAt(2)+"      \t"+v4.elementAt(3));
               System.out.println("          \t\t        -------");
               System.out.println("          \t\tTOTAL   "+tot);

        }
}
```

Chapter 3 Interfaces and Packages

2M

1. Define Interface.
Ans:

Extending classes using single implementation inheritance creates new class types. A superclass reference can denote objects of its own type and its subclasses strictly according to the inheritance hierarchy. Because this relationship is linear, it rules out multiple implementation inheritance, that is, a subclass inheriting from more than one superclass. Instead Java provides interfaces, which not only allow new named reference types to be introduced, but also permit multiple interface inheritance.

2. Define package.
Ans:

A package in Java is an encapsulation mechanism that can be used to group related classes, interfaces, and subpackages.

4M

1. How to define the interface?
Ans:

Defining Interfaces

A top-level interface has the following general syntax:

```
<accessibility modifier> interface <interface name>
                        <extends interface clause> // Interface header

{ // Interface body
    <constant declarations>
    <method prototype declarations>
    <nested class declarations>
    <nested interface declarations>
  }
```

In the interface header, the name of the interface is preceded by the keyword `interface`. In addition, the interface header can specify the following information:

- scope or accessibility modifier

- any interfaces it extends

The interface body can contain member declarations which comprise

- constant declarations
- method prototype declarations
- nested class and interface declarations

An interface does not provide any implementation and is, therefore, `abstract` by definition. This means that it cannot be instantiated, but classes can implement it by providing implementations for its method prototypes. Declaring an interface `abstract` is superfluous and seldom done.

2. Write a java program to demonstrate an interface.
Ans:

```
interface IStack {                                              //
(1)
    void   push(Object item);
    Object pop();
}

class StackImpl implements IStack {                             //
(2)
    protected Object[] stackArray;
    protected int      tos;  // top of stack

    public StackImpl(int capacity) {
        stackArray = new Object[capacity];
        tos        = -1;
    }

    public void push(Object item)                               //
(3)
        { stackArray[++tos] = item; }

    public Object pop() {                                       //
(4)
        Object objRef = stackArray[tos];
        stackArray[tos] = null;
        tos--;
        return objRef;
    }

    public Object peek() { return stackArray[tos]; }
}

interface ISafeStack extends IStack {                           //
(5)
    boolean isEmpty();
    boolean isFull();
}
```

```
class SafeStackImpl extends StackImpl implements ISafeStack {      //
(6)

    public SafeStackImpl(int capacity) { super(capacity); }
    public boolean isEmpty() { return tos < 0; }                   //
(7)
    public boolean isFull() { return tos >= stackArray.length-1; }//
(8)
}
public class StackUser {

    public static void main(String[] args) {                       //
(9)
        SafeStackImpl safeStackRef  = new SafeStackImpl(10);
        StackImpl     stackRef      = safeStackRef;
        ISafeStack    isafeStackRef = safeStackRef;
        IStack        istackRef     = safeStackRef;
        Object        objRef        = safeStackRef;

        safeStackRef.push("Dollars");                              //
(10)
        stackRef.push("Kroner");
        System.out.println(isafeStackRef.pop());
        System.out.println(istackRef.pop());
        System.out.println(objRef.getClass());
    }
}
```

Output from the program:

```
Kroner
Dollars
class SafeStackImpl
```

3. How to implement the interface?
Ans:

Any class can elect to implement, wholly or partially, zero or more interfaces. A class specifies the interfaces it implements as a comma-separated list of unique interface names in an `implements` clause in the class header. The interface methods must all have `public` accessibility when implemented in the class (or its subclasses). A class can neither narrow the accessibility of an interface method nor specify new exceptions in the method's `throws` clause, as attempting to do so would amount to altering the interface's contract, which is illegal. The criteria for overriding methods also apply when implementing interface methods.

A class can provide implementations of methods declared in an interface, but it does not reap the benefits of interfaces unless the interface name is explicitly specified in its `implements` clause.

class can choose to implement only some of the methods of its interfaces, (i.e., give a partial implementation of its interfaces). The class must then be declared as `abstract`. Note that interface methods cannot be declared `static`, because they comprise the contract fulfilled by the objects of the class implementing the interface. Interface methods are always implemented as instance methods.

The interfaces a class implements and the classes it extends (directly or indirectly) are called supertypes of the class. Conversely, the class is a subtype of its supertypes. Classes implementing interfaces introduce multiple interface inheritance into their linear implementation inheritance hierarchy. However, note that regardless of how many interfaces a class implements directly or indirectly, it only provides a single implementation of a member that might have multiple declarations in the interfaces.

4. Describe constants in interfaces along with example.
Ans:

An interface can also define named constants. Such constants are defined by field declarations and are considered to be `public`, `static` and `final`. These modifiers are usually omitted from the declaration. Such a constant must be initialized with an initializer expression.

An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface. However, if a client is a class that implements this interface or an interface that extends this interface, then the client can also access such constants directly without using the fully qualified name. Such a client inherits the interface constants. Typical usage of constants in interfaces is illustrated in following example, showing both direct access and use of fully qualified names at (1) and (2), respectively.

Extending an interface that has constants is analogous to extending a class having static variables. In particular, these constants can be hidden by the subinterfaces. In the case of multiple inheritance of interface constants, any name conflicts can be resolved using fully qualified names for the constants involved.

Example: Variables in Interfaces

```
interface Constants {
    double PI_APPROXIMATION = 3.14;
    String AREA_UNITS = " sq.cm.";
    String LENGTH_UNITS = " cm.";
}

public class Client implements Constants {
    public static void main(String[] args) {
        double radius = 1.5;
        System.out.println("Area of circle is " +
                           (PI_APPROXIMATION*radius*radius) +
                           AREA_UNITS);             // (1) Direct
access.
        System.out.println("Circumference of circle is " +
                           (2*Constants.PI_APPROXIMATION*radius) +
                           Constants.LENGTH_UNITS); // (2) Fully
qualified name.
    }
}
```

Output from the program:

```
Area of circle is 7.0649999999999995 sq.cm.
Circumference of circle is 9.42 cm.
```

5. How to define a package?
Ans:

A package hierarchy represents an organization of the Java classes and interfaces. It does not represent the source code organization of the classes and interfaces. The source code is of no consequence in this regard. Each Java source file (also called compilation unit) can contain zero or more definitions of classes and interfaces, but the compiler produces a separate class file containing the Java byte code for each of them. A class or interface can indicate that its Java byte code be placed in a particular package, using a `package` declaration.

The `package` statement has the following syntax:

`package` <fully qualified package name>`;`

At most one `package` declaration can appear in a source file, and it must be the first statement in the unit. The package name is saved in the Java byte code for the types contained in the package.

Note that this scheme has two consequences. First, all the classes and interfaces in a source file will be placed in the same package. Secondly, several source files can be used to specify the contents of a package.

If a `package` declaration is omitted in a compilation unit, the Java byte code for the declarations in the compilation unit will belong to an unnamed package, which is typically synonymous with the current working directory on the host system.

6. How to use package?

Ans:

The accessibility of types (classes and interfaces) in a package may deny access from outside the package. Given a reference type that is accessible from outside a package, the reference type can be accessed in two ways. The first form uses the fully qualified name of the type. However, writing long names can become tedious. The second form uses the `import` declaration to provide a shorthand notation for specifying the name of the type.

The `import` declarations must be the first statement after any `package` declaration in a source file. The simple form of the `import` declaration has the following syntax:

`import` <fully qualified type name>`;`

This is called single type import. As the name implies, such an `import` declaration provides a shorthand notation for a single class or interface. The simple name of the type (i.e., its identifier) can be used to access this particular type. Given the following `import` declaration:

```
import wizard.pandorasBox.Clown;
```

the name `Clown` can be used in the source file to refer to this class.

Alternatively, the following form of the `import` declaration can be used:

`import` <fully qualified package name>`.*;`

This is called type import on demand. It allows any type from the specified package to be accessed by its simple name.

An `import` declaration does not recursively import subpackages. The declaration does not result in inclusion of the source code of the types. The declaration only imports type names (i.e., it makes type names available to the code in a compilation unit).

All compilation units implicitly import the `java.lang` package. This is the reason why we can refer to the class `String` by its simple name, and not need to use its fully qualified name `java.lang.String` all the time.

7. How to compile and run code from package?
Ans:

The package name (for example) `wizard.pandorasBox` corresponds to the path name `wizard/pandorasBox`. The `javac` compiler can place the byte code in a directory that corresponds to the package declaration of the compilation unit. The Java byte code for all the classes (and interfaces) specified in the source files `Clown.java` and `LovePotion.java` will be placed in the directory named `wizard/pandorasBox`, as these source files have the following `package` declaration:

```
package wizard.pandorasBox;
```

The absolute path of the `wizard/pandorasBox` directory is specified by using the `-d` option (`d` for destination) when compiling with the `javac` compiler. Assuming that the current directory is called `/pgjc/work`, and all the source code files are to be found here, the command

```
>javac -d . Clown.java Ailment.java Baldness.java
```

issued in the `work` directory, will create `./wizard/pandorasBox` (and any other subdirectories required) under the current directory, and place the Java byte code for all the classes (and interfaces) in the directories corresponding to the package names. The dot (`.`) after the `-d` option denotes the current directory. After compiling the code in the example using the `javac` command above, the file hierarchy under the `/pgjc/work` directory should mirror the package hierarchy. Without the `-d` option, the default behavior of the `javac` compiler is to place all class files directly under the current directory, rather than in the appropriate subdirectories.

How do we run the program? Since the current directory is `/pgjc/work` and we want to run `Clown.class`, the fully qualified name of the `Clown` class must be specified in the `java` command

```
>java wizard.pandorasBox.Clown
```

This will load the class `Clown` from the byte code in the file `./wizard/pandorasBox/Clown.class`, and start the execution of its `main()` method.

Chapter 4 Multithreaded Programming and Exception Handling

2M

1. State the two different types of multitasking..
Ans:

Multitasking allows several activities to occur concurrently on the computer. A distinction is usually made between:

- Process-based multitasking
- Thread-based multitasking

2. What are the advantages of thread-based multitasking as compared to process-based multitasking?
Ans:

The following advantages of thread-based multitasking as compared to process-based multitasking are

- threads share the same address space
- context switching between threads is usually less expensive than between processes
- cost of communication between threads is relatively low

3. What is process based multitasking?
Ans:
At the coarse-grain level there is process-based multitasking, which allows processes (i.e., programs) to run concurrently on the computer. A familiar example is running the spreadsheet program while also working with the word-processor.

4. What is Thread based multitasking?
Ans:

At the fine-grain level there is thread-based multitasking, which allows parts of the same program to run concurrently on the computer. A familiar example is a word-processor that is printing and formatting text at the same time. This is only feasible if the two tasks are performed by two independent paths of execution at runtime. The two tasks would correspond to executing parts of the program concurrently. The sequence of code executed for each task defines a separate path of execution, and is called a thread (of execution).

5. Which type of multitasking that java supports?

Ans:

Java supports thread-based multitasking and provides high-level facilities for multithreaded programming. Thread safety is the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads.

6. What is Thread?
Ans:

A thread is an independent sequential path of execution within a program. Many threads can run concurrently within a program. At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code, that is, they are lightweight compared to processes. They also share the process running the program.

Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class. Often the thread and its associated `Thread` object are thought of as being synonymous.

Threads make the runtime environment asynchronous, allowing different tasks to be performed concurrently. Using this powerful paradigm in Java centers around understanding the following aspects of multithreaded programming:

- creating threads and providing the code that gets executed by a thread
- accessing common data and code through synchronization
- transitioning between thread states

7. What is synchronization?

Ans:

Threads share the same memory space, that is, they can share resources. However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource. For example, crediting and debiting a shared bank account concurrently amongst several users without proper discipline, will jeopardize the integrity of the account data. Java provides high-level concepts for synchronization in order to control access to shared resources.

4M

1. What is main Thread?
Ans:

The runtime environment distinguishes between user threads and daemon threads. As long as a user thread is alive, the JVM does not terminate. A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program. Daemon threads exist only to serve user threads.

When a standalone application is run, a user thread is automatically created to execute the `main()` method. This thread is called the main thread. If no other user threads are spawned, the program terminates when the `main()` method finishes executing. All other threads, called child threads, are spawned from the main thread, inheriting its user-thread status. The `main()` method can then finish, but the program will keep running until all the user threads have finished. Calling the `setDaemon(boolean)` method in the `Thread` class marks the status of the thread as either daemon or user, but this must be done before the thread is started. Any attempt to change the status after the thread has been started, throws an `IllegalThreadStateException`. Marking all spawned threads as daemon threads ensures that the application terminates when the main thread dies.

When a GUI application is started, a special thread is automatically created to monitor the user–GUI interaction. This user thread keeps the program running, allowing interaction between the user and the GUI, even though the main thread might have died after the `main()` method finished executing.

2. How to create a thread?
Ans:

A thread in Java is represented by an object of the `Thread` class. Implementing threads is achieved in one of two ways:

- implementing the `java.lang.Runnable` interface
- extending the `java.lang.Thread` class

Implementing the `Runnable` Interface

The `Runnable` interface has the following specification, comprising one method prototype declaration:

```
public interface Runnable {
    void run();
}
```

A thread, which is created based on an object that implements the `Runnable` interface, will execute the code defined in the public method `run()`. In other words, the code in the `run()` method defines an independent path of execution and thereby the entry and the exits for the thread. The thread ends when the `run()` method ends, either by normal completion or by throwing an uncaught exception.

The procedure for creating threads based on the `Runnable` interface is as follows:

1. A class implements the `Runnable` interface, providing the `run()` method that will be executed by the thread. An object of this class is a `Runnable` object.
2. An object of `Thread` class is created by passing a `Runnable` object as argument to the `Thread` constructor. The `Thread` object now has a `Runnable` object that implements the `run()` method.
3. The `start()` method is invoked on the `Thread` object created in the previous step. The `start()` method returns immediately after a thread has been spawned.

3. Write a Java Program to implement the Runnable interface.
Ans:

```
class Counter implements Runnable {

    private int currentValue;

    private Thread worker;

    public Counter(String threadName) {
        currentValue = 0;
        worker = new Thread(this, threadName);   // (1) Create a new
thread.
        System.out.println(worker);
        worker.start();                          // (2) Start the
thread.
    }

    public int getValue() { return currentValue; }

    public void run() {                          // (3) Thread entry
point
        try {
            while (currentValue < 5) {
                System.out.println(worker.getName() + ": " +
(currentValue++));
                Thread.sleep(250);               // (4) Current thread
sleeps.
            }
        } catch (InterruptedException e) {
            System.out.println(worker.getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + worker.getName());
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Counter counterA = new Counter("Counter A"); // (5) Create a
thread.

        try {
            int val;
            do {
                val = counterA.getValue();       // (6) Access the
counter value.
                System.out.println("Counter value read by main thread:
" + val);
                Thread.sleep(1000);              // (7) Current thread
sleeps.
            } while (val < 5);
        } catch (InterruptedException e) {
            System.out.println("main thread interrupted.");
        }

        System.out.println("Exit from main() method.");
    }
}
```

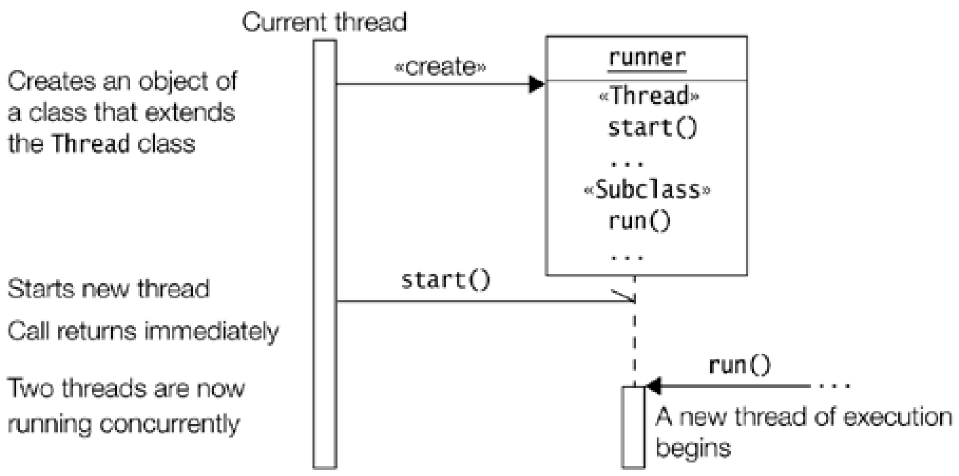
Possible output from the program:

```
Thread[Counter A,5,main]
Counter value read by main thread: 0
Counter A: 0
Counter A: 1
Counter A: 2
Counter A: 3
Counter value read by main thread: 4
Counter A: 4
Exit from thread: Counter A
Counter value read by main thread: 5
Exit from main() method.
```

4. Describe how to extends Thread class.
Ans:

A class can also extend the `Thread` class to create a thread. A typical procedure for doing this is as follows:

1. A class extending the `Thread` class overrides the `run()` method from the `Thread` class to define the code executed by the thread.
2. This subclass may call a `Thread` constructor explicitly in its constructors to initialize the thread, using the `super()` call.
3. The `start()` method inherited from the `Thread` class is invoked on the object of the class to make the thread eligible for running.


Current thread
Creates an object of a class that extends the Thread class
«create»
runner
«Thread»
start()
...
«Subclass»
run()
...
Starts new thread
Call returns immediately
start()
Two threads are now running concurrently
run()
A new thread of execution begins

5. Write a java program which extends the Thread class.
Ans:

```
class Counter extends Thread {

    private int currentValue;

    public Counter(String threadName) {
        super(threadName);                        // (1) Initialize
thread.
        currentValue = 0;
        System.out.println(this);
        start();                                  // (2) Start this
thread.
    }

    public int getValue() { return currentValue; }
    public void run() {                           // (3) Override from
superclass.
        try {
            while (currentValue < 5) {
                System.out.println(getName() + ": " +
(currentValue++));
                Thread.sleep(250);                // (4) Current thread
sleeps.
            }
        } catch (InterruptedException e) {
            System.out.println(getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + getName());
    }
}

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

        System.out.println("Method main() runs in thread " +
                Thread.currentThread().getName());   // (5) Current
thread
```

```
        Counter counterA = new Counter("Counter A"); // (6) Create a
thread.
        Counter counterB = new Counter("Counter B"); // (7) Create a
thread.

        System.out.println("Exit from main() method.");
    }
}
```

Possible output from the program:

```
Method main() runs in thread main
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Exit from main() method.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from thread: Counter A
Exit from thread: Counter B
```

6. What is the meaning of monitor with respect to synchronization?
Ans:

A lock (a.k.a. monitor) is used to synchronize access to a shared resource. A lock can be associated with a shared resource. Threads gain access to a shared resource by first acquiring the lock associated with the resource. At any given time, at the most one thread can hold the lock (i.e., own the monitor) and thereby have access to the shared resource. A lock thus implements mutual exclusion (a.k.a. mutex).

In Java, all objects have a lock—including arrays. This means that the lock from any Java object can be used to implement mutual exclusion. By associating a shared resource with a Java object and its lock, the object can act as a guard, ensuring synchronized access to the resource. Only one thread at a time can access the shared resource guarded by the object lock.

The object lock mechanism enforces the following rules of synchronization:

- A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource. The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource. If a thread cannot immediately acquire the object lock, it is blocked, that is, it must wait for the lock to become available.

- When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is waiting for this object lock, it can proceed to acquire the lock in order to gain access to the shared resource.

Classes also have a class-specific lock that is analogous to the object lock. Such a lock is actually a lock on the `java.lang.Class` object associated with the class. Given a class `A`, the reference `A.class` denotes this unique `Class` object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

The keyword `synchronized` and the lock form the basis for implementing synchronized execution of code. There are two ways in which execution of code can be synchronized:

- synchronized methods
- synchronized blocks

7. What is synchronized method and synchronized block?
Ans:

Synchronized method

If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword `synchronized`. A thread wishing to execute a `synchronized` method must first obtain the object's lock (i.e., hold the lock) before it can enter the object to execute the method. This is simply achieved by calling the method. If the lock is already held by another thread, the calling thread waits. No particular action on the part of the program is necessary. A thread relinquishes the lock simply by returning from the `synchronized` method, allowing the next thread waiting for this lock to proceed.

Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently. A stack implementation usually defines the two operations push and pop as synchronized, so that pushing and popping of elements are mutually exclusive operations. If several threads were to share a stack, then one thread would, for example, not be able to push an element on the stack while another thread was popping the stack. The integrity of the stack is maintained in the face of several threads accessing the state of the same stack.

Synchronized block

Whereas execution of `synchronized` methods of an object is synchronized on the lock of the object, the `synchronized` block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object. The general form of the `synchronized` statement is as follows:

`synchronized (`<object reference expression>`) {` <code block> `}`

The <object reference expression> must evaluate to a non-`null` reference value, otherwise, a `NullPointerException` is thrown. The code block is usually related to the object on which the synchronization is being done. This is the case with `synchronized` methods, where the execution of the method is synchronized on the lock of the current object:

```
public Object pop() {
    synchronized (this) {           // Synchronized block on current
object
        // ...
    }
}
```

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is relinquished. This happens when the execution of the code block completes normally or an uncaught exception is thrown. In contrast to `synchronized` methods, this mechanism allows fine-grained synchronization of code on arbitrary objects.

8. Describe Thread transitions.
Ans:

Understanding the life cycle of a thread is valuable when programming with threads. Threads can exist in different states. Just because a thread's `start()` method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed.

The following figure shows the states and the transitions in the life cycle of a thread.

- Ready-to-run state

  A thread starts life in the Ready-to-run state (see p. 369).

- Running state

  If a thread is in the Running state, it means that the thread is currently executing (see p. 369).

- Dead state

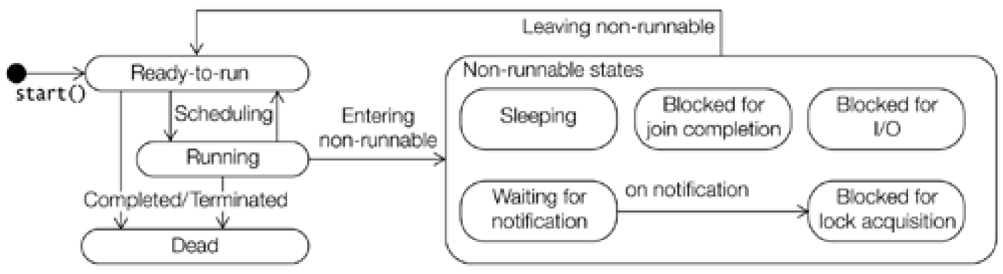
  Once in this state, the thread cannot ever run again (see p. 380).

- Non-runnable states

  A running thread can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a special transition occurs. A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state.

  The non-runnable states can be characterized as follows:

  - Sleeping: The thread sleeps for a specified amount of time (see p. 370).
  - Blocked for I/O: The thread waits for a blocking operation to complete (see p. 380).
  - Blocked for join completion: The thread awaits completion of another thread (see p. 377).
  - Waiting for notification: The thread awaits notification from another thread (see p. 370).
  - Blocked for lock acquisition: The thread waits to acquire the lock of an object (see p. 359).



Various methods from the `Thread` class are presented next. Examples of their usage are presented in subsequent sections.

```
final boolean isAlive()
```

This method can be used to find out if a thread is alive or dead. A thread is alive if it has been started but not yet terminated, that is, it is not in the Dead state.

```
final int getPriority()
final void setPriority(int newPriority)
```

The first method returns the priority of the current thread. The second method changes its priority. The priority set will be the minimum of the two values: the specified `newPriority` and the maximum priority permitted for this thread.

```
static void yield()
```

This method causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute.

```
static void sleep (long millisec) throws InterruptedException
```

The current thread sleeps for the specified time before it takes its turn at running again.

```
final void join() throws InterruptedException
final void join(long millisec) throws InterruptedException
```

A call to any of these two methods invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified time, respectively.

```
void interrupt()
```

The method interrupts the thread on which it is invoked. In the Waiting-for-notification, Sleeping, or Blocked-for-join-completion states, the thread will receive an `InterruptedException`.

9. Describe Thread priorities.
Ans:

Threads are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the Ready-to-run state. This is not necessarily the thread that has been the longest time in the Ready-to-run state. Heavy reliance on thread priorities for the behavior of a program can make the program unportable across platforms, as thread scheduling is host platform–dependent.

Priorities are integer values from 1 (lowest priority given by the constant `Thread.MIN_PRIORITY`) to 10 (highest priority given by the constant `Thread.MAX_PRIORITY`). The default priority is 5 (`Thread.NORM_PRIORITY`).

A thread inherits the priority of its parent thread. Priority of a thread can be set using the `setPriority()` method and read using the `getPriority()` method, both of which are defined in the `Thread` class. The following code sets the priority of the thread `myThread` to the minimum of two values: maximum priority and current priority incremented to the next level:

```
myThread.setPriority(Math.min(Thread.MAX_PRIORITY,
myThread.getPriority()+1));
```

10. Describe Thread Scheduler.
Ans:

Schedulers in JVM implementations usually employ one of the two following strategies:

- Preemptive scheduling.

  If a thread with a higher priority than the current running thread moves to the Ready-to-run state, then the current running thread can be preempted (moved to the Ready-to-run state) to let the higher priority thread execute.

- Time-Sliced or Round-Robin scheduling.

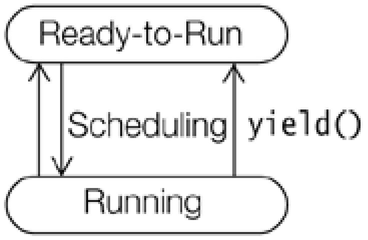
  A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready-to-run state to await its turn to run again.

It should be pointed out that thread schedulers are implementation- and platform-dependent; therefore, how threads will be scheduled is unpredictable, at least from platform to platform.

11. Describe the working of yielding and running state of a thread.
Ans:

After its `start()` method has been called, the thread starts life in the Ready-to-run state. Once in the Ready-to-run state, the thread is eligible for running, that is, it waits for its turn to get CPU time. The thread scheduler decides which thread gets to run and for how long.

The following figure illustrates the transitions between the Ready-to-Run and Running states. A call to the static method `yield()`, defined in the `Thread` class, will cause the current thread in the Running state to transit to the Ready-to-run state, thus relinquishing the CPU. The thread is then at the mercy of the thread scheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute.

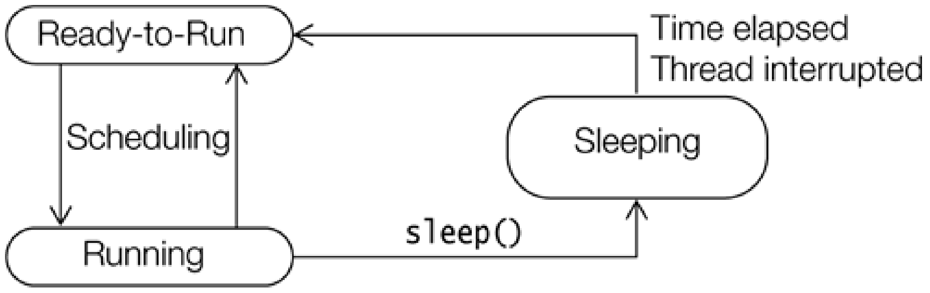


By calling the static method `yield()`, the running thread gives other threads in the Ready-to-run state a chance to run. A typical example where this can be useful is when a user has given some command to start a CPU-intensive computation, and has the option of canceling it by clicking on a Cancel button. If the computation thread hogs the CPU and the user clicks the Cancel button, chances are that it might take a while before the thread monitoring the user input gets a chance to run and take appropriate action to stop the computation. A thread running such a computation should do the computation in increments, yielding between increments to allow other threads to run. This is illustrated by the following `run()` method:

```
public void run() {
    try {
        while (!done()) {
            doLittleBitMore();
            Thread.yield();          // Current thread yields
        }
    } catch (InterruptedException e) {
        doCleaningUp();
    }
}
```

12. Describe Sleeping and Waking up states of a thread.
Ans:

The following figure shows the sleeping and waking up states of a thread.



A call to the static method `sleep()` in the `Thread` class will cause the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before transitioning to the Ready-to-run state where it takes its turn to run again. If a thread is interrupted while sleeping, it will throw an `InterruptedException` when it awakes and gets to execute.

There are serveral overloaded versions of the `sleep()` method in the `Thread` class.

13. Describe waiting and notifying of a thread.
Ans:

Waiting and notifying provide means of communication between threads that synchronize on the same object. The threads execute `wait()` and `notify()` (or `notifyAll()`) methods on the shared object for this purpose. These `final` methods are defined in the `Object` class, and therefore, inherited by all objects.

These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an `IllegalMonitorStateException`.

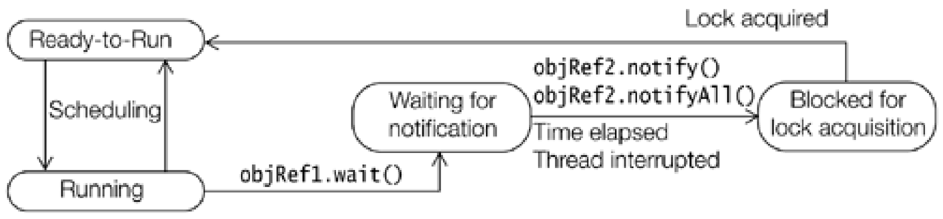
```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
```

A thread invokes the `wait()` method on the object whose lock it holds. The thread is added to the wait set of the object.

```
final void notify()
final void notifyAll()
```

A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object.

Communication between threads is facilitated by waiting and notifying, as illustrated by the following figure. A thread usually calls the `wait()` method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the Running state and transits to the Waiting-for-notification state. There it waits for this condition to occur. The thread relinquishes ownership of the object lock.



Transition to the Waiting-for-notification state and relinquishing the object lock are completed as one atomic (non-interruptable) operation. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock.

Note that the waiting thread does not relinquish any other object locks that it might hold, only that of the object on which the `wait()` method was invoked. Objects that have these other locks remain locked while the thread is waiting.

Each object has a wait set containing threads waiting for notification. Threads in the Waiting-for-notification state are grouped according to the object whose `wait()` method they invoked.

14. Describe Joining of a thread.
Ans:

A thread can invoke the overloaded method `join()` on another thread in order to wait for the other thread to complete its execution before continuing, that is, the first thread waits for the second thread to join it after completion. A running thread t_1 invokes the method `join()` on a thread t_2. The `join()` call has no effect if thread t_2 has already completed. If thread t_2 is still alive, then thread t_1 transits to the Blocked-for-join-completion state. Thread t_1 waits in this state until one of these events occur:

- Thread t_2 completes.

  In this case thread t_1 is enabled and when it gets to run, it will continue normally after the `join()` method call.

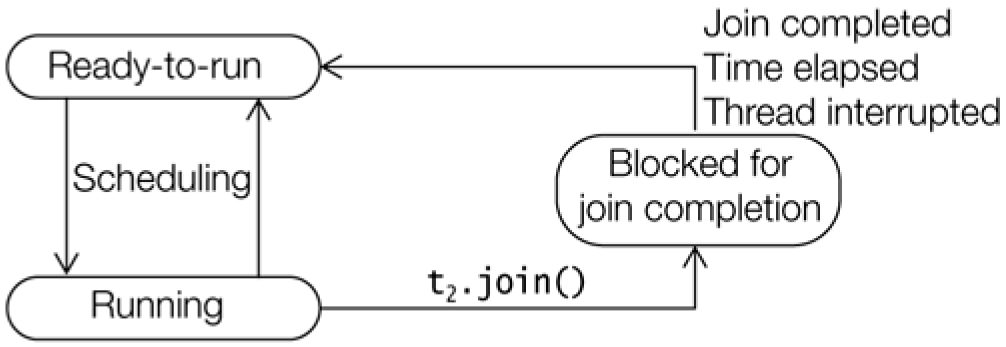
- Thread t_1 is timed out.

  The time specified in the argument in the `join()` method call has elapsed, without thread t_2 completing. In this case as well, thread t_1 is enabled. When it gets to run, it will continue normally after the `join()` method call.

- Thread t_1 is interrupted.

  Some thread interrupted thread t_1 while thread t_1 was waiting for join completion. Thread t_1 is enabled, but when it gets to execute, it will now throw an `InterruptedException`.

  The following figure shows joining of a thread.



15. Write a program to demonstrate joining of a thread.
Ans:

```
class Counter extends Thread { /* See Example 9.2. */ }

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

        Counter counterA = new Counter("Counter A");
        Counter counterB = new Counter("Counter B");

        try {
            System.out.println("Wait for the child threads to
finish.");
            counterA.join();                                 // (5)
            if (!counterA.isAlive())                         // (6)
                System.out.println("Counter A not alive.");
            counterB.join();                                 // (7)
            if (!counterB.isAlive())                         // (8)
                System.out.println("Counter B not alive.");
        } catch (InterruptedException e) {
            System.out.println("Main Thread interrupted.");
        }
        System.out.println("Exit from Main Thread.");
    }
}
```

Possible output from the program:

```
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Wait for the child threads to finish.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from Counter A.
Counter A not alive.
Exit from Counter B.
Counter B not alive.
Exit from Main Thread.
```

16. Write a program to demonstrate the thread termination.
Ans:

```
class Worker implements Runnable {                              // (1)
    private Thread theThread;                                   // (2)

    public void kickStart() {                                   // (3)
        if (theThread == null) {
            theThread = new Thread(this);
            theThread.start();
        }
    }

    public void terminate() {                                      // (4)
        theThread = null;
    }

    public void run() {                                            // (5)
        while (theThread == Thread.currentThread()) {              // (6)
            System.out.println("Going around in loops.");
        }
    }
}

public class Controller {
    public static void main(String[] args) {                       // (7)
        Worker worker = new Worker();                              // (8)
        worker.kickStart();                                        // (9)
        Thread.yield();                                            // (10)
        worker.terminate();                                        // (11)
    }
}
```

Possible output from the program:

```
Going around in loops.
Going around in loops.
Going around in loops.
Going around in loops.
Going around in loops.
```

17. Write a program to demonstrate of deadlock.
Ans:

```
public class DeadLockDanger {

    String o1 = "Lock " ;                         // (1)
    String o2 = "Step ";                          // (2)

    Thread t1 = (new Thread("Printer1") {         // (3)
        public void run() {
            while(true) {
                synchronized(o1) {                // (4)
                    synchronized(o2) {            // (5)
                        System.out.println(o1 + o2);
                    }
                }
            }
        }
    });

    Thread t2 = (new Thread("Printer2") {         // (6)
        public void run() {
            while(true) {
                synchronized(o2) {              // (7)
                    synchronized(o1) {          // (8)
                        System.out.println(o2 + o1);
                    }
                }
            }
        }
    });

    public static void main(String[] args) {
        DeadLockDanger dld = new DeadLockDanger();
        dld.t1.start();
        dld.t2.start();
    }
}
```

Possible output from the program:

```
...
Step Lock
Step Lock
Lock Step
Lock Step
Lock Step
...
```

18. Describe the fundamentals of exception handling.
Ans:

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.
Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.
This is the general form of an exception-handling block:
try {
// block of code to monitor for errors
}
catch (*ExceptionType1 exOb*) {
// exception handler for *ExceptionType1*
}

```
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

19. What are the types of exception handling?
Ans:

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

20. Describe multiple catch clauses.
Ans:

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

This program will cause a division-by-zero exception if it is started with no command-line parameters, since **a** will equal zero. It will survive the division if you provide a commandline argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

21. Describe throw statement.

Ans:

It is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:
throw *ThrowableInstance*;
Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator.
The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.
Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException,** which is caught on the next line. The exception is then rethrown.
Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. All of Java's built-in run-time exceptions have two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable.**

22. Describe throws statement.
Ans:

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.
This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try/catch** statement that catches this exception.

23.State different exception and also give the meaning.
Ans:

ArithmeticException -
Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException -
Array index is out-of-bounds.
ArrayStoreException -
Assignment to an array element of an incompatible type.
ClassCastException -
Invalid cast.
IllegalArgumentException -
Illegal argument used to invoke a method.
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.
IndexOutOfBoundsException -
Some type of index is out-of-bounds.
NegativeArraySizeException -
Array created with a negative size.
NullPointerException -
Invalid use of a null reference.
NumberFormatException -
Invalid conversion of a string to a numeric format.
SecurityException -
Attempt to violate security.
StringIndexOutOfBounds -
Attempt to index outside the bounds of a string.

24. Write a java program to demonstrate round robin scheduling.
Ans:

```
class A extends Thread
{
        public void run()
        {
                System.out.println("Thread A started");
                try
                {
                        for(int i=1;i<=5;i++)
                        {
                                System.out.println("A:"+i);
                                System.out.println("Thread A Sleeping");
                                Thread.sleep(250);
                                System.out.println("Thread A Resumed");
                        }
                }
                        catch (Exception e)
                        {
                                System.out.println("The Error Is:"+e);
                                System.out.println("Thread A Terminated");
                        }

        }
}
class B extends Thread
{
        public void run()
        {
          System.out.println("Thread B started");
          try
          {
                        for(intj=1;j<=5;j++)
                        {
                                System.out.println("B:"+j);
                                System.out.println("Thread B sleeping");
                                Thread.sleep(250);
                                System.out.println("Thread B Resumed");

                        }
                }
                        catch(Exception e)
                        {
                                System.out.println("The Error is:"+e);


                                System.out.println("Thread B terminated");
                        }
                }
}
}
class C extends Thread
{
        public void run()
        {
          System.out.println("Thread C started");
          try
          {
                        for(intk=1;k<=5;k++)
                        {
                                System.out.println("C:"+k);
                                System.out.println("Thread C sleeping");
                                Thread.sleep(250);
                                System.out.println("Thread C Resumed");

                        }
                }

                        catch(Exception e)
                        {
                                System.out.println("The Error is:"+e);


                                System.out.println("Thread C terminated");
                        }
                }
```

```
}

class RR
{
        public static void main(String args[])
        {
                A a=new A();
                B b=new B();
                C c=new C();


                a.start();
                b.start();
                c.start();

                try
                {
                        a.join();
                        b.join();
                        c.join();
                }
                catch(Exception e)
                {
                        System.out.println("The error is:"+e);
                        System.out.println("Main Thread Terminated");
                }
        }
}
```

25. Write a program to generate your own exception.
Ans:

```
import java.lang.Exception;
import java.io.*;
class myerror extends Exception
{
  myerror(String s)
   {
      super(s);
   }
}


class Q11
{
  public static void main(String args[]) throws IOException
```

```
   {
     int n=11,fact = 1;

       DataInputStream in=new DataInputStream(System.in);
       try
       {
                             System.out.print("Enter a Number : ");
           n = Integer.parseInt(in.readLine());
           if(n>=10)
                   throw new myerror("number is too big.");
           for(int i=1 ; i<= n ; i++)
               fact=fact * i ;
           System.out.print("factorial is:" + fact);
       }

     catch(myerror e)
     {
        System.out.println(e.getMessage());
     }
   }
}
```

Chapter 5 Java Applets and Graphics Programming

2M

1. Describe Applet basics.
Ans:

All applets are subclasses of **Applet**. Thus, all applets must import **java.applet**. Applets must also import **java.awt**. Recall that AWT stands for the Abstract Window Toolkit. Since all applets run in a window, it is necessary to include support for that window. Applets are not executed by the console-based Java run-time interpreter. Rather, they are executed by either a Web browser or an applet viewer. The figures shown in this chapter were created with the standard applet viewer, called **appletviewer**, provided by the JDK.

2. How to compile applet program?
Ans:

Once an applet has been compiled, it is included in an HTML file using the APPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is an example of such a comment:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called **MyApplet** in a window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

4M

1. Describe applet architecture.
Ans:

An applet is a window-based program. As such, its architecture is different from the socalled normal, console-based programs shown in the first part of this book.
Applet is basically built on Component-Container architecture.
First, applets are event driven. Although we won't examine event handling until the following chapter, it is important to understand in a general way how the event-driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs. The AWT 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 AWT. 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 AWT 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. (You will see an example later in this chapter.)
Second, the user initiates interaction with an applet—not the other way around. As you know, in a nonwindowed program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( )**. 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.

2. Describe applet skeleton.
Ans:

The five methods of applet can be assembled into the skeleton shown here:

```
// An 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 execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window 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. When run, it generates the following window when viewed with an applet viewer:

3. State the methods of applet life cycle.
Ans:

When an applet begins, the AWT calls the following methods, in this sequence:
init( )
start( )
paint( )
When an applet is terminated, the following sequence of method calls takes place:
stop( )
destroy( )

The methods are described in detail as follows.

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may 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.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

4. Describe overriding update method.
Ans:

The applet may need to override another method defined by the AWT, called **update( )**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** first fills an applet with the default background color and then calls **paint( )**. If you fill the background using a different color in **paint( )**, the user will experience a flash of the default background each time **update( )** is called—that is, whenever the window is repainted. One way to avoid this problem is to override the **update( )** method so that it performs all necessary display activities. Then have **paint( )** simply call **update( )**. Thus, for some applications, the applet skeleton will override **paint( )** and **update( )**, as shown here:

```
public void update(Graphics g) {
```

```
// redisplay your window, here.
}
public void paint(Graphics g) {
update(g);
}
```

5. Describe awt package.
Ans:

The following figure shows the class hierarchy of the AWT.

- Component
  - Button
  - Canvas
  - Checkbox
  - Choice
  - Container
    - Panel
    - Window
      - Dialog
      - Frame
  - Label
  - List
  - Scrollbar
  - TextComponent
    - TextArea
    - TextField

6. Describe class Applet.
Ans:

java.applet
Class Applet

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Panel
              └java.applet.Applet
```

```
The syntax of Applet is as follows:

public class Applet
extends Panel
```

An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.
The `Applet` class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The `Applet` class provides a standard interface between applets and their environment.

7. Describe class Component.
Ans:

The syntax of Component class is as follows:

```
public abstract class Component
extends Object
implements ImageObserver, MenuContainer, Serializable
```

A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.
The `Component` class is the abstract superclass of the nonmenu-related Abstract Window Toolkit components. Class `Component` can also be extended directly to create a lightweight component. A lightweight component is a component that is not associated with a native opaque window.

8. Describe class Container.
Ans:

The syntax of Container class is as follows:

```
public class Container
extends Component
```

A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.
Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container. If no index is specified

when adding a component to a container, it will be added to the end of the list (and hence to the bottom of the stacking order).

9. Describe class Panel and Frame.
Ans:

The syntax of Panel class is as follows:

```
public class Panel
extends Container
implements Accessible
```

`Panel` is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.
The default layout manager for a panel is the `FlowLayout` layout manager.

The hierarchy of Frame class is as follows:

java.awt
Class Frame

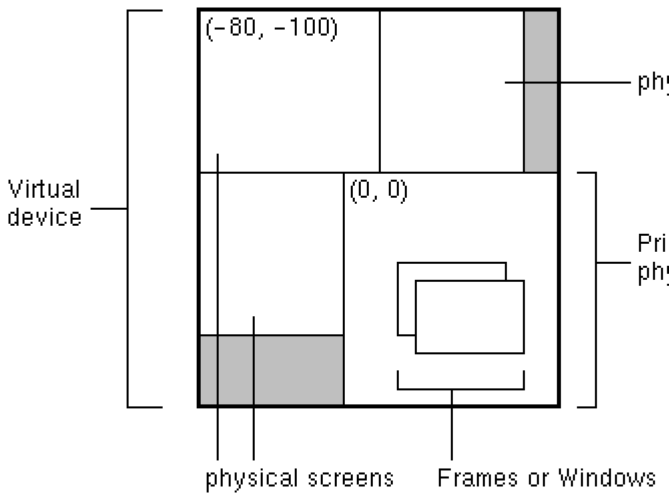
```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
```

```
The syntax of Frame class is as follows:

public class Frame
extends Window
implements MenuContainer
```

A `Frame` is a top-level window with a title and a border.
The size of the frame includes any area designated for the border. The dimensions of the border area may be obtained using the `getInsets` method, however, since these dimensions are platform-dependent, a valid insets value cannot be obtained until the frame is made displayable by either calling `pack` or `show`. Since the border area is included in the overall size of the frame, the border effectively obscures a portion of the frame, constraining the area available for rendering and/or displaying subcomponents to the rectangle which has an upper-left corner location of `(insets.left, insets.top)`, and has a size of `width - (insets.left + insets.right)` by `height - (insets.top + insets.bottom)`.
The default layout for a frame is `BorderLayout`.
A frame may have its native decorations (i.e. `Frame` and `Titlebar`) turned off with `setUndecorated`. This can only be done while the frame is not `displayable`.
In a multi-screen environment, you can create a `Frame` on a different screen device by constructing the `Frame` with `Frame(GraphicsConfiguration)` or `Frame(String title, GraphicsConfiguration)`. The `GraphicsConfiguration` object is one of the `GraphicsConfiguration` objects of the target screen device.

In a virtual device multi-screen environment in which the desktop area could span multiple physical screen devices, the bounds of all configurations are relative to the virtual-coordinate system. The origin of the virtual-coordinate system is at the upper left-hand corner of the primary physical screen. Depending on the location of the primary screen in the virtual device, negative coordinates are possible, as shown in the following figure.

are possible, as shown in the following figure.



In such an environment, when calling `setLocation`, you must pass a virtual coordinate to this method. Similarly, calling `getLocationOnScreen` on a `Frame` returns virtual device coordinates. Call the `getBounds` method of a `GraphicsConfiguration` to find its origin in the virtual coordinate system.

The following code sets the location of the `Frame` at (10, 10) relative to the origin of the physical screen of the corresponding `GraphicsConfiguration`. If the bounds of the `GraphicsConfiguration` is not taken into account, the `Frame` location would be set at (10, 10) relative to the virtual-coordinate system and would appear on the primary physical screen, which might be different from the physical screen of the specified `GraphicsConfiguration`.

```
      Frame f = new Frame(GraphicsConfiguration gc);
      Rectangle bounds = gc.getBounds();
      f.setLocation(10 + bounds.x, 10 + bounds.y);
```

Frames are capable of generating the following types of `WindowEvent`s:

- `WINDOW_OPENED`

- `WINDOW_CLOSING`:
  If the program doesn't explicitly hide or dispose the window while processing this event, the window close operation is canceled.
- `WINDOW_CLOSED`
- `WINDOW_ICONIFIED`
- `WINDOW_DEICONIFIED`
- `WINDOW_ACTIVATED`
- `WINDOW_DEACTIVATED`
- `WINDOW_GAINED_FOCUS`
- `WINDOW_LOST_FOCUS`
- `WINDOW_STATE_CHANGED`

10. Describe add, paint and update method with respect to applet.
Ans:

add

`public Component **add**(Component comp)`

Appends the specified component to the end of this container. This is a convenience method for `addImpl(java.awt.Component, java.lang.Object, int)`.
Note: If a component has been added to a container that has been displayed, `validate` must be called on that container to display the new component. If multiple components are being added, you can improve efficiency by calling `validate` only once, after all the components have been added.

paint

`public void **paint**(Graphics g)`

Paints the container. This forwards the paint to any lightweight components that are children of this container. If this method is reimplemented, super.paint(g) should be called so that lightweight components are properly rendered. If a child component is entirely clipped by the current clipping setting in g, paint() will not be forwarded to that child.
Overrides:
paint in class Component

update

`public void **update**(Graphics g)`

Updates the container. This forwards the update to any lightweight components that are children of this container. If this method is reimplemented, super.update(g) should be called so that lightweight components are properly rendered. If a child component is entirely clipped by the current clipping setting in g, update() will not be forwarded to that child.
Overrides:
update in class Component

11. Enlist listeners in java.awt.event package.
Ans:

The following are Interfaces in the java.awt.event package.
ActionListener
AdjustmentListener
AWTEventListener
ComponentListener
ContainerListener
FocusListener
HierarchyBoundsListener
HierarchyListener
InputMethodListener
ItemListener
KeyListener
MouseListener
MouseMotionListener
MouseWheelListener
TextListener
WindowFocusListener
WindowListener
WindowStateListener

8M Questions

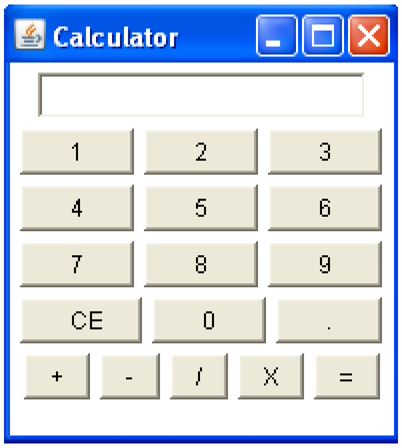
1. Write a program to demonstrate mouse events.
Ans:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<Applet code=MouseEvents.class width = 300 height = 200></applet>
*/
public class MouseEvents extends Applet implements MouseListener , MouseMotionListener
{
        String msg = "";
        int mouseX=0,mouseY=0;
        public void init()
        {
                addMouseListener(this);
                      addMouseMotionListener(this);

        }
        public void paint(Graphics g)
        {

                update(g);
        }

 public void update(Graphics g)
         {
     g.drawString(msg,mouseX,mouseY);
```

```
    }

 public void mouseClicked(MouseEvent me)
            {
        mouseX=0;
                        mouseY=15;
                        msg = "Mouse Clicked";
                        repaint();
            }

public void mouseEntered(MouseEvent me)
            {
        mouseX=0;
                        mouseY=15;
                        msg = "Mouse Entered";
                        repaint();
            }
public void mouseExited(MouseEvent me)
            {
        mouseX=0;
                        mouseY=30;
                        msg = "Mouse Exited";
                        repaint();
            }
public void mousePressed(MouseEvent me)
            {
        mouseX=me.getX();
                        mouseY=me.getY();
                        msg = "Down";
                        repaint();
            }
public void mouseReleased(MouseEvent me)
            {
        mouseX=me.getX();
                        mouseY=me.getY();
                        msg = "Up";
                        repaint();
            }
public void mouseDragged(MouseEvent me)
            {
        mouseX=me.getX();
                        mouseY=me.getY();
                        msg = "*";
                        showStatus("Dragging Mouse at " + mouseX + ", " + mouseY);
                        repaint();
            }
public void mouseMoved(MouseEvent me)
            {
        showStatus("Moving Mouse at " + me.getX() + ", " + me.getY());

            }

}
```

2. Write a program do design a calculator.
Ans:

The layout design of a calculator is as follows:



Source code:

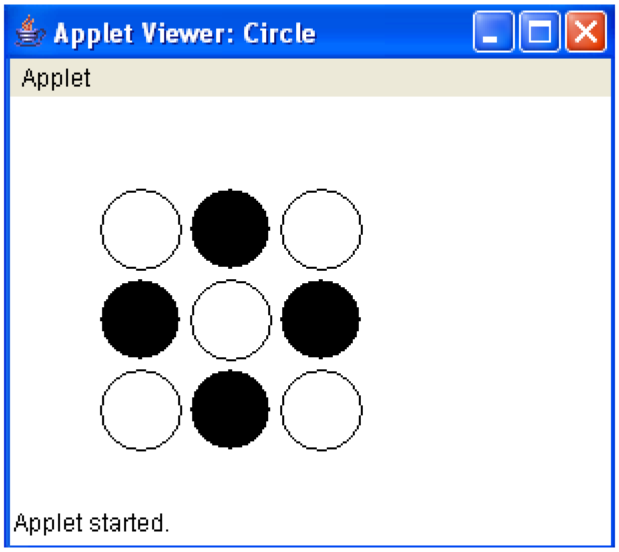
```
import java.awt.*;
import java.awt.event.*;

class Calc extends Frame implements TextListener,ActionListener
{
        Button b[] = new Button[11],plus,minus,div,mul,ce,eq;
        static TextField ans;
        static boolean newnum;
  handler h = new handler();

        public Calc(String s)
        {
   super(s);
         setLayout(new FlowLayout());
   newnum=false;
         ans = (TextField)add(new TextField(20));


         for(int i=1;i<=9;i++)
                {
                  b[i] = new Button("     "+i+"      ");
      b[i].setSize(20,10);
                  add(b[i]);
                }


         ce = new Button("      CE     ");
         b[0] =  new Button("      0      ");
   b[10] = new Button("      .      ");
         plus = new Button("  +  ");
         minus = new Button("  -  ");
         div = new Button("  /  ");
         mul = new Button("  X  ");
         eq = new Button("  =  ");
```

```
            add(ce);
            add(b[0]);
            add(b[10]);
            add(plus);
            add(minus);
            add(div);
            add(mul);
            add(eq);

            addWindowListener(new WindowAdapter() { public void windowClosing(WindowEvent
we){System.exit(0);} });
            ans.addTextListener(this);
            for(int i=0;i<11;i++)
                        b[i].addActionListener(this);
            ce.addActionListener(h);
            plus.addActionListener(h);
            minus.addActionListener(h);
            div.addActionListener(h);
            eq.addActionListener(h);
            mul.addActionListener(h);
          }

          public void textValueChanged(TextEvent te)
          {
                    String s;
                    if (newnum)
                    {
                       ans.setText("");
                    }
                    try
                              {
                                        s = ans.getText();
                                        char c = s.charAt(s.length()-1);

          if(c=='0'||c=='1'||c=='2'||c=='3'||c=='4'||c=='5'||c=='6'||c=='7'||c=='8'||c=='9'||c=='.')
                                                  { }
                                        else
                                                  ans.setText(s.substring(0,s.length()-1));
                                        ans.setCaretPosition(22);
                       }
                              catch (Exception e)
                              {

                              }
      newnum=false;
          }


          public void actionPerformed(ActionEvent ae)
          {
      String s = (ae.getActionCommand()).trim();
                    if (newnum)
                    {
                              ans.setText("");
                    }
```

```
                ans.setText(ans.getText()+s);
                newnum=false;
        }
}

class handler implements ActionListener
{
        static float op1;
        String op="";
        public void actionPerformed(ActionEvent ae)
        {
          String s = (ae.getActionCommand()).trim();
          try
                  {
                   if(s.equals("CE"))
                        {
                                op1 = 0f;
                                Calc.ans.setText("");
                                Calc.newnum = false;
                                op="";
                        }
                        else if (s.equals("+"))
                        {
                                if(op1 == 0)
                                                op1 = Float.parseFloat(Calc.ans.getText());
                                else
                                                Calculate();
                                Calc.newnum = true;
                                op=s;
                        }
                        else if(s.equals("-"))
                                {
                                        if(op1 == 0)
                                                op1 = Float.parseFloat(Calc.ans.getText());
                                        else
                                                Calculate();
                                        Calc.newnum = true;
                                        op=s;
                                }
                                else if(s.equals("/"))
                                {
                                        if(op1 == 0)
                                                op1 = Float.parseFloat(Calc.ans.getText());
                                        else
                                                Calculate();
                                        Calc.newnum = true;
                                        op=s;
                                }
                                else if(s.equals("X"))
                                {
                                        if(op1 == 0)
                                                op1 = Float.parseFloat(Calc.ans.getText());
                                        else
                                                Calculate();
                                        Calc.newnum = true;
                                        op=s;
                                }
                        else
                                {
                                        Calculate();
                                        Calc.ans.setText(op1+"");
                                        op1=0f;
                                }
                }
                catch(Exception e)  { }
        }

 public void Calculate()
         {
      try
                        {
                          if (op.equals("+"))
                                {
                                        op1 = op1 + Float.parseFloat(Calc.ans.getText());
                                }
                                else if (op.equals("-"))
                                {
                                        op1 = op1 - Float.parseFloat(Calc.ans.getText());
                                }
                                else if (op.equals("/"))
                                {
                                        op1 = op1 / Float.parseFloat(Calc.ans.getText());
                                }
                                else if (op.equals("X"))
                                {
                                        op1 = op1 * Float.parseFloat(Calc.ans.getText());
                                }
                                Calc.ans.setText(op1+"");
                        }
                        catch(Exception e) { }
         }
}

class CalcApp
{
        public static void main(String[] args)
        {
                Calc c = new Calc("Calculator");

                c.setSize(200,220);
                c.show();
                c.repaint();
        }
}
```

3. Write a program to draw the circle.
Ans:

The layout design is as follows:

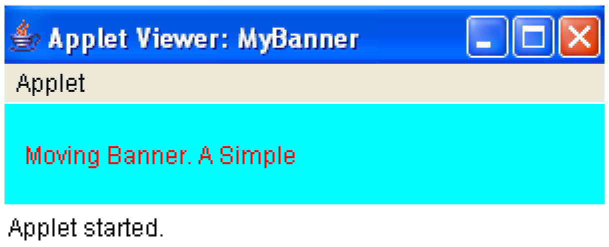


Source code:

```
import java.applet.*;
import java.awt.*;
/*
<Applet code=Circle width = 300 height = 200></applet>
*/
public class Circle extends Applet
{
        public void paint(Graphics g)
        {
                int x,y,f = 0;
                for (x = 1; x<=3 ; x++)
                {
                        for (y = 1;y <=3 ;y++ )
                        {
                                if (f == 0)
                                {
                                        g.drawOval(x*45,y*45,40,40);
            f = 1;
                                }
                                else
          {
                                  g.fillOval(x*45,y*45,40,40);
           f = 0;
                                }
                        }
                }
        }
}
```

4. Write a program to demonstrate a moving banner.
Ans:

The layout design is as follows:



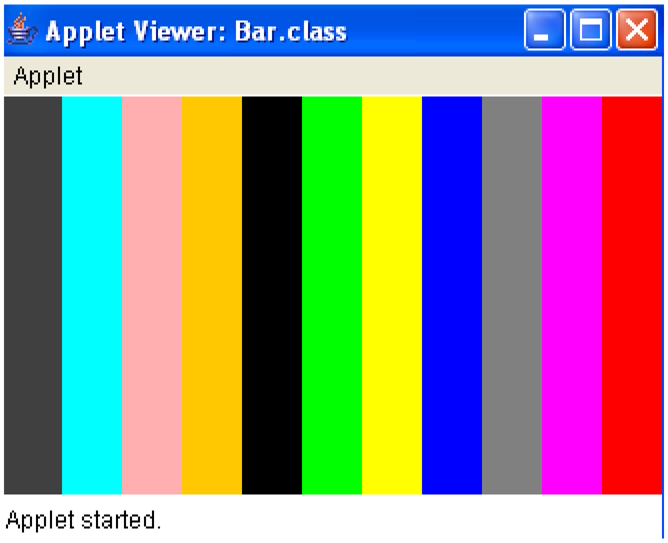
Moving Banner. A Simple

Source Code:

```
import java.awt.*;
import java.applet.*;
/*
<Applet code=MyBanner height=50 width=300></Applet>
*/
public class  MyBanner extends Applet implements Runnable
{
            String msg = "A Simple Moving Banner. ";
            boolean stopFlag;
            Thread t = null;
            int state;

            public void init()
            {
        setBackground(Color.cyan);
                        setForeground(Color.red);
            }

            public void start()
            {
              t = new Thread(this);
              stopFlag = false;
              t.start();
            }

            public void run()
            {
              char ch;
              for( ; ; )
                        {
                  try
                                {
                                  repaint();
                                  Thread.sleep(250);
                                  ch = msg.charAt(0);
                                  msg = msg.substring(1,msg.length());
                                  msg += ch;
                                  if (stopFlag)
                                      break;
                                }catch(InterruptedException e){}
                }
            }

            public void stop()
            {
                stopFlag = true;
                        t = null;
            }

            public void paint(Graphics g)
            {
              g.drawString(msg,10,30);
            }
}
```

5. Write a program to display various bars in different colors.
Ans:

Layout design is as follows:



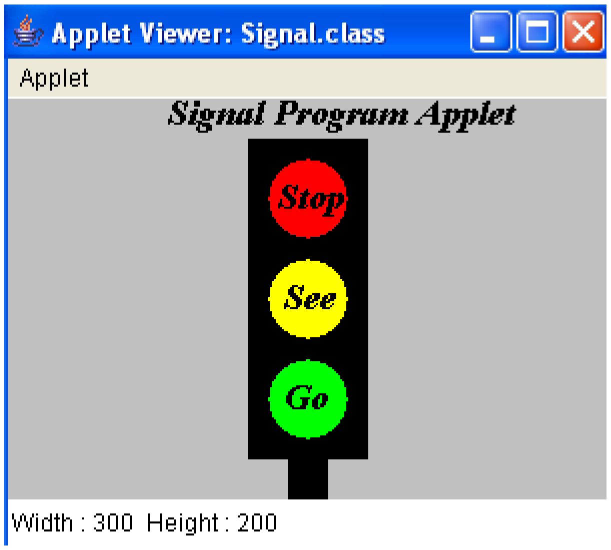
Source Code:

```
import java.applet.*;
import java.awt.*;
/*
<Applet code=Bar.class width = 330 height = 200></applet>
*/
public class Bar extends Applet
{
        int x1,x2,y1,y2;
        Dimension d;
    Color
color[]={Color.darkGray,Color.cyan,Color.pink,Color.orange,Color.black,Color.green,Color.yellow,
Color.blue,Color.gray,Color.magenta,Color.red};
```

```
            public void init()
            {
           d = getSize();
                                    x1=0;
                                    y1=0;
                                    y2 = d.height - 1;
            }
            public void paint(Graphics g)
            {
                        for (int i=0;i<=10;i++)
                        {
           g.setColor(color[i]);
                                    g.fillRect(x1,y1,30,d.height - 1);
                                    x1 = x1+30;

                        }
            }

}
```

6. Write a program to demonstrate a signal using Applet.
Ans:

Layout design is as follows:



Source Code:

```
import java.applet.*;
import java.awt.*;
/*
<Applet code=Signal.class width = 300 height = 200></applet>
*/
public class Signal extends Applet
{
            public void init()
            {
```

```
        setBackground(Color.lightGray);
                        Font f = new Font("TimesRoman",Font.BOLD|Font.ITALIC,18);
                        setFont(f);
        }
        public void paint(Graphics g)
        {
                Dimension d = getSize();
                int midx,midy,once=0;
                midx = d.width /2;
                midy = d.height/2;
                int x[] = {midx-30,midx+30,midx+30,midx+10,midx+10,midx-10,midx-10,midx-
30,midx-30};
                int y[] = {midy-80,midy-
80,midy+80,midy+80,midy+110,midy+110,midy+80,midy+80,midy-80};
                showStatus("Width : " + d.width + "  Height : " + d.height);
                g.fillPolygon(x,y,9);
                g.setColor(Color.red);
                g.fillOval(midx-20,midy-70,40,40);
                g.setColor(Color.yellow);
                g.fillOval(midx-20,midy-20,40,40);
                g.setColor(Color.green);
                g.fillOval(midx-20,midy+30,40,40);
                g.setColor(Color.black);
                g.drawString("Stop",midx-15,midy-45);
                g.drawString("See",midx-12,midy+5);
                g.drawString("Go",midx-12,midy+55);
                g.drawString("Signal Program Applet",midx-70,13);
        }
        public void update(Graphics g) { }
}
```

7. Write a program to catch the multiple keyboard events.
Ans:

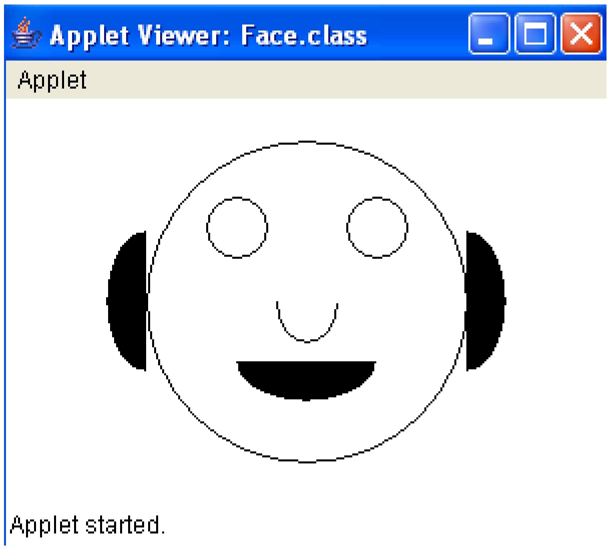
Source Code:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<Applet code=KeyBoard.class width = 300 height = 200></applet>
*/
public class KeyBoard extends Applet
{
        String msg = "";
        int mouseX=10,mouseY=15;
        public void init()
        {
                addKeyListener(new MyKeyAdapter(this));
                        requestFocus();
        }
        public void paint(Graphics g)
        {
                update(g);
        }
```

```
            public void update(Graphics g)
            {
              g.drawString(msg,mouseX,mouseY);
            }


}

class MyKeyAdapter extends KeyAdapter
{
   KeyBoard kb;
   public MyKeyAdapter(KeyBoard k)
            {
       kb = k;
    }
  public void keyTyped(KeyEvent ke)
            {
              kb.msg += ke.getKeyChar();
              kb.repaint();
            }
}
```

8. Write a program to draw the face on applet.
Ans:

Layout design is as follows:

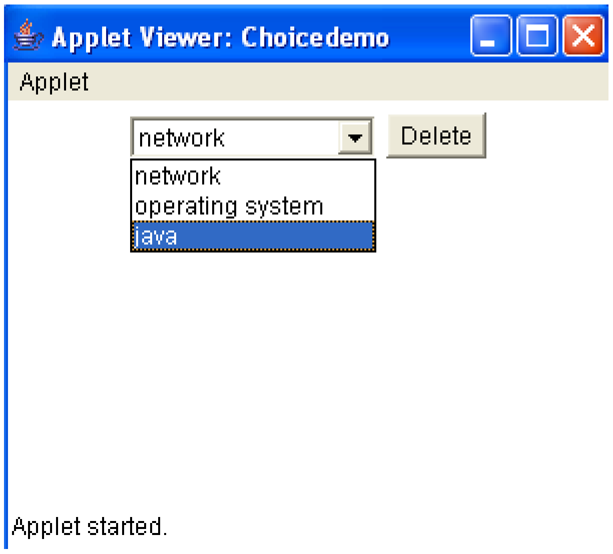


Source code:

```
import java.applet.*;
import java.awt.*;
/*
<Applet code=Face.class width = 300 height = 200></applet>
*/
public class Face extends Applet
{
        public void paint(Graphics g)
        {
                Dimension d = getSize();
                int midx,midy;
                midx = d.width /2;
                midy = d.height/2;
                g.drawOval(midx-80,midy-80,160,160);            //FACE
                g.drawOval(midx-50,midy-52,30,30);                      //LEFT EYE
                g.drawOval(midx+20,midy-52,30,30);                      //RIGHT EYE
                g.drawArc(midx-15,midy-20,30,40,0,-180);        //NOSE
                g.fillArc(midx-100,midy-35,40,70,90,180);       //LEFT EAR
                g.fillArc(midx+60,midy-35,40,70,90,-180);       //RIGHT EAR
                g.fillArc(midx-35,midy+10,70,40,0,-180); //MOUTH
        }
}
```

9. Write a program to demonstrate choice demo.
Ans:

Layout Design is as follows:



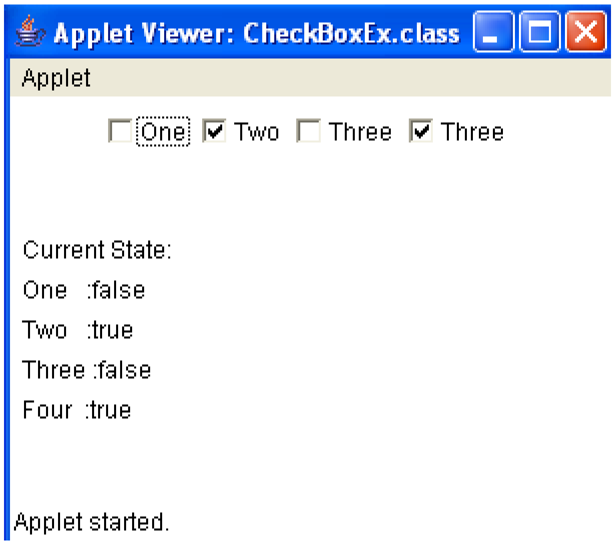
Source Code:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="Choicedemo"  width=300  height=200>
  </applet>
*/

public class Choicedemo extends Applet implements ItemListener,ActionListener
{
    Choice os;
    String s=" ";
        Button ok;
```

```
   public void init()
    {
         os=new Choice();
         ok = new Button("Delete");

        os.add("network");
                os.add("operating system");
        os.add("java");
        add(os);
                        add(ok);
        os.addItemListener(this);
                        ok.addActionListener(this);

     }

     public void paint(Graphics g)
       {

        g.drawString(s,6,120);

    }

    public void itemStateChanged(ItemEvent ie)
       {
                           s="current Choice :";
                                   s +=os.getSelectedItem();
            repaint();
       }

             public void actionPerformed(ActionEvent ae)
                {
            s = "Deleted : " + os.getSelectedItem();
                                   os.remove(os.getSelectedIndex());
                                   repaint();
                }
}
```

10. Write a program to demonstrate Checkbox demo.
Ans:

Layout design is as follows:



```
Current State:
One   :false
Two   :true
Three :false
Four  :true
```

```
Applet started.
```

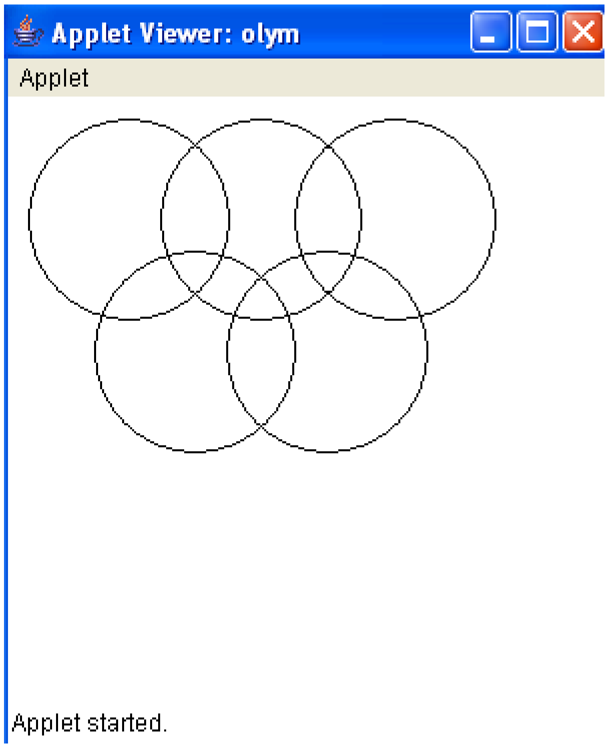
Source Code:

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

/*
<Applet code=CheckBoxEx.class width = 300 height = 200></applet>
*/

public class CheckBoxEx extends Applet implements ItemListener
{
  String msg = "";
  Checkbox one, two, three, four;


  public void init()
   {
     one = new Checkbox("One");
     two = new Checkbox("Two",true);
     three = new Checkbox("Three");
     four = new Checkbox("Three",true);


     add(one);
     add(two);
     add(three);
     add(four);


     one.addItemListener(this);
     two.addItemListener(this);
     three.addItemListener(this);
     four.addItemListener(this);
   }

  public void itemStateChanged(ItemEvent ie)
```

```
 {
  repaint();
 }
public void paint(Graphics g)
{
  msg = "Current State:";
  g.drawString(msg, 6, 80);

  msg = "One   :" + one.getState();
  g.drawString(msg, 6, 100);
  msg = "Two   :" + two.getState();
  g.drawString(msg, 6, 120);
  msg = "Three :" + three.getState();
  g.drawString(msg, 6, 140);
  msg = "Four  :" + four.getState();
  g.drawString(msg, 6, 160);
}
}
```

11. Write a program to display Olympic symbol by passing parameter to applet.
Ans:

Layout Design is as follows:



Source Code:

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

/*
```

```
<Applet code = "olym" width = 300 height = 300>
<param name = "rad" value = "50">
</Applet>
*/

public class olym extends Applet
{
        public void paint(Graphics g)
        {
          int x = Integer.parseInt(getParameter("rad"));
          g.drawOval(10,10,2*x,2*x);
          g.drawOval(10+x+(x/3),10,2*x,2*x);
          g.drawOval(10+(2*x)+((2*x)/3),10,2*x,2*x);
          g.drawOval(10+((x+x/3)/2),10+x+(x/3),2*x,2*x);
          g.drawOval(10+((x+x/3)/2)+x+x/3,10+x+(x/3),2*x,2*x);
        }
}
```

Chapter 6

2M Questions

1. Enlist two types of streams in Java I/O.
Ans:

Java 2 defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters.

2. What are the two abstract classes in Byte Stream classes?
Ans:

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses, that handle the differences between various devices, such as disk files, network connections, and even memory buffers.

3. What are the two abstract classes in Character Stream classes?
Ans:

Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.
The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively. These methods are overridden by derived stream classes.

4. Enlist Byte Stream classes.
Ans:

BufferedInputStream - Buffered input stream

BufferedOutputStream - 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
FilterOutputStream - Implements OutputStream
InputStream - Abstract class that describes stream input
OutputStream - Abstract class that describes stream output
PipedInputStream - Input pipe
PipedOutputStream - Output

5. Enlist Character Stream Classes.
Ans:

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 character array
FileReader - Input stream that reads from a file
FileWriter - Output stream that writes to a file
FilterReader - Filtered reader
FilterWriter - Filtered writer

6. Write a program to demonstrate PrintWrite class.
Ans:

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[]) {
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
int i = -7;
pw.println(i);
double d = 4.5e-7;
pw.println(d);
}
}
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

4M Questions

1. Write a program that counts the character from the specified file.
Ans:

```
import java.io.*;

class CharCount
{
        public static void main(String[] args) throws Exception
        {
                FileReader f = new FileReader("J.txt");
                int count = -1,ch = 0;


                while(ch != -1)
                {
                        count = count + 1;
                        ch = f.read();

                }

                System.out.println("Number of Characters = " + count+"(Including Spaces)");
        }
}
```

2. Write a program to demonstrate a FileReader class with the help of which the lines should be counted.
Ans:

```
// Demonstrate FileReader.
import java.io.*;

public class LineCount{
  public static void main(String args[]) throws Exception {
    FileReader fr = new FileReader("LineCount.java");
    BufferedReader br = new BufferedReader(fr);

     int count=0;
    while(br.readLine() != null) {
            count++;
    }
       System.out.println("No. of lines in this file :"+count);
    fr.close();
  }
}
```

3. Write a program to read from one file and write into another file.
Ans:

```
import java.io.*;
class ReadWrite
{
 public static void main(String args[]) throws IOException
 {
  InputStream f = new FileInputStream("/Text.txt");
  OutputStream f2 = new FileOutputStream("/Text2.txt");
  int n,ch;

  while((ch=f.read()) != -1)
            {
             f2.write(ch);
            }
```

```
   f.close();
   f2.close();

   System.out.println("File copy Complete");
 }
}
```

4. Write a program to divide the file into 3 parts and display the text.
Ans:

```
import java.io.*;
class File1by3
{
 public static void main(String args[]) throws IOException
 {
  InputStream f = new FileInputStream("/Text.txt");
  int n,ch;

  System.out.println("There are " + f.available() + " Bytes in File");
  n = f.available()/3;


  System.out.println("Displaying First 1/3 Characters");
  for(int i = 0 ; i < n ; i++)
  {
    ch = f.read();
    System.out.print((char)ch);
  }
  f.skip(n);
  System.out.println("\nDisplaying Last 1/3 Characters in File");
  for(int i = 0 ; i < n ; i++)
  {
    ch = f.read();
    System.out.print((char)ch);
  }
 }
}
```

5. What are streams?
Ans:

A *stream* is a path of communication between the source of some information and its destination. This information can come from a file, the computer's memory, or even from the Internet. In fact, the source and destination of a stream are completely arbitrary producers and consumers of bytes, respectively-you don't need to know about the source of the information when reading from a stream, and you don't need to know about the final destination when writing to one.

A *stream* is a path of communication between a source of information and its destination. For example, an input stream allows you to read data from a source, and an output stream allows you to write data to a destination.

General-purpose methods that can read from any source accept a stream argument to specify that source; general-purpose methods for writing accept a stream to specify the destination. Arbitrary *processors* of data commonly have two stream arguments. They read from the first, process the data, and write the results to the second. These processors have no idea of either the source *or* the destination of the data they are processing. Sources and destinations can vary widely: from two memory buffers on the same local computer, to the ELF (extremely low frequency) transmissions to and from a submarine at sea, to the real-time data streams of a NASA probe in deep space.

By decoupling the consuming, processing, or producing of data from the sources and destinations of that data, you can mix and match any combination of them at will as you write your program. In the future, when new, previously nonexistent forms of source or destination (or consumer, processor, or producer) appear, they can be used within the same framework, with no changes to your classes. In addition, new stream abstractions, supporting higher levels of interpretation "on top of" the bytes, can be written completely independently of the underlying transport mechanisms for the bytes themselves.

6. Describe java.io package.
Ans:

All the classes you will learn about today are part of the package `java.io`. To use any of these classes in your own programs, you will need to import each individual class or to import the entire `java.io` package, like this:

```
import java.io.InputStream;
import java.io.FilteredInputStream;
import java.io.FileOutputStream;

import java.io.*;
```

All the methods you will explore today are declared to throw `IOException`s. This new subclass of `Exception` conceptually embodies all the possible I/O errors that might occur while using streams. Several subclasses of it define a few, more specific exceptions that can be thrown as well. For now, it is enough to know that you must either `catch` an `IOException`, or be in a method that can "pass it along," to be a well-behaved user of streams.

The foundations of this stream framework in the Java class hierarchy are the two abstract classes, `InputStream` and `OutputStream`. Inheriting from these classes is a virtual cornucopia of categorized subclasses, demonstrating the wide range of streams in the system, but also demonstrating an extremely well-designed hierarchy of relationships between these streams-one well worth learning from. Let's begin with the parents, `InputStream` and `OutputStream`, and then work our way down this bushy tree.

7. Describe FileInputStream class.
Ans:

One of the most common uses of streams, and historically the earliest, is to attach them to files in the file system. Here, for example, is the creation of such an input stream on a UNIX system:

```
InputStream  s = new FileInputStream("/some/path/and/fileName");
```

You also can create the stream from a previously opened file descriptor (an instance of the `FileDescriptor` class). Usually, you get file descriptors using the `getFD()` method on `FileInputStream` or `FileOutputStream` classes, so, for example, you could use the same file descriptor to open a file for reading and then reopen it for writing:

```
FileDescriptor       fd = someFileStream.getFD();
InputStream  s  = new FileInputStream(fd);
```

In either case, because it's based on an actual (finite length) file, the input stream created can implement `available()` precisely and can skip like a champ (just as `ByteArrayInputStream` can, by the way). In addition, `FileInputStream` knows a few more tricks:

```
FileInputStream  aFIS = new FileInputStream("aFileName");

FileDescriptor  myFD = aFIS.getFD(); // get a file descriptor

 aFIS.finalize();   // will call close() when automatically called by
GC
```

The first is obvious: `getFD()` returns the file descriptor of the file on which the stream is based. The second, though, is an interesting shortcut that allows you to create `FileInputStream`s without worrying about closing them later. `FileInputStream`'s implementation of `finalize()`, a `protected` method, closes the stream. Unlike in the contrived call in comments, you almost never can nor should call a `finalize()` method directly. The garbage collector calls it after noticing that the stream is no longer in use, but before actually destroying the stream. Thus, you can go merrily along using the stream, never closing it, and all will be well. The system takes care of closing it (eventually).

8. Write a program to read a simple line by line from a file.
Ans:

```
import java.io;

public class  SimpleLineReader {
    private FilterInputStream  s;

    public  SimpleLineReader(InputStream  anIS) {
        s = new DataInputStream(anIS);
    }

    . . .    // other read() methods using stream s

    public String  readLine() throws IOException {
        char[]  buffer = new char[100];
        int     offset = 0;
        byte    thisByte;

        try {
loop:        while (offset < buffer.length) {
                switch (thisByte = (byte) s.read()) {
                    case '\n':
                        break loop;
                    case '\r':
                        byte  nextByte = (byte) s.read();

                        if (nextByte != '\n') {
                            if (!(s instanceof PushbackInputStream)) {
                                s = new PushbackInputStream(s);
                            }
                            ((PushbackInputStream) s).unread(nextByte);
                        }
                        break loop;
                    default:
                        buffer[offset++] = (char) thisByte;
                        break;
                }
            }
        } catch (EOFException e) {
            if (offset == 0)
                return null;
        }
          return String.copyValueOf(buffer, 0, offset);
     }
}
```

9. Describe object serialization.
Ans:

A topic to streams, and one that will be available in the core Java library with Java 1.1, is object serialization. *Serialization* is the ability to write a Java object to a stream such as a file or a network connection, and then read it and reconstruct that object on the other side. Object serialization is crucial for the ability to save Java objects to a file (what's called *object persistence*), or to be able to accomplish network-based applications

At the heart of object serialization are two streams classes: `ObjectInputStream`, which inherits from `DataInputStream`, and `ObjectOutputStream`, which inherits from `DataOutputStream`. Both of these classes will be part of the `java.io` package and will be used much in the same way as the standard input and output streams are. In addition, two interfaces, `ObjectOutput` and `ObjectInput`, which inherit from `DataInput` and `DataOutput`, respectively, will provide abstract behavior for reading and writing objects.

To use the `ObjectInputStream` and `ObjectOutputStream` classes, you create new instances much in the same way you do ordinary streams, and then use the `readObject()` and `writeObject()` methods to read and write objects to and from those streams.

`ObjectOutputStream`'s `writeObject()` method, which takes a single object argument, serializes that object as well as any object it has references to. Other objects written to the same stream are serialized as well, with references to already-serialized objects kept track of and circular references preserved.

`ObjectInputStream`'s `readObject()` method takes no arguments and reads an object from the stream (you'll need to cast that object to an object of the appropriate class). Objects are read from the stream in the same order in which they are written.

Here's a simple example from the object serialization specification that writes a date to a file (actually, it writes a string label, `"Today"`, and then a `Date` object):

```
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput  s  =  new  ObjectOutputStream(f);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

To deserialize the object (read it back in again), use this code:

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

One other feature of object serialization to note is the `transient` modifier. Used in instance variable declarations as other modifiers are, the `transient` modifier means that the value of that object should not be stored when the object is serialized-that its value is temporary or will need to be re-created from scratch once the object is reconstructed. Use transient variables for environment-specific information (such as file handles that may be different from one side of the serialization to the other) or for values that can be easily recalculated to save space in the final serialized object.