KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Eachanari (po), Coimbatore-21

Semester – II

19ITU201 PROGRAMMING IN JAVA 4H – 4C

Instruction Hours / week: L: 4 T: 0 P: 0 **Marks:** Internal : **40** External : **60** Total: **100**

End Semester Exam : 3 Hours

Course Objectives

- To understand the fundamentals of programming such as variables, conditional and iterative execution, methods, etc.
- To understand fundamentals of object-oriented programming in Java, including defining classes, invoking methods, using class libraries, etc.
- To use the Java SDK environment to create, debug and run simple Java programs.
- To use Java in various technologies in different platforms.
- To understand the fundamental of Packages and access modifiers and interface in java.
- To understand the fundamental of Exception Handling and AWT component and AWT classes.

Course Outcomes (COs)

1. Student will obtain knowledge of the structure and model of the Java programming language.
2. How to use the Java programming language for various programming technologies (understanding)
3. Develop software in the Java programming language (application)
4. Evaluate user requirements for software functionality required to decide whether the Java programming language can meet user requirements (analysis)
5. propose the use of certain technologies by implementing them in the Java programming language to solve the given problem (synthesis)
6. choose an engineering approach to solving problems, starting from the acquired knowledge of programming and knowledge of operating systems. (evaluation)

Unit I

Introduction to Java: Object Oriented Paradigm and Concepts-Structured versus Object Oriented Approach. Java Language: Features of Java -Environment-Java Architecture-Java Development Kit-Types of Java Program. Variable Declaration and Arrays: Data Types-Java Tokens –Variable Declaration – Type Casting and Conversion – Arrays, Operators, And Control Statements: Selection Constructs – Iteration Constructs –Jump Statements.

Unit II

Classes and Objects

Introduction to classes: Instance variables, Class variables, Instance Methods, Constructors, Class methods, Declaring Objects, Garbage Collection, Method Overloading - Constructor Overloading - This Reference. Inheritance: Super class variables- Method Overriding - final Keyword, Abstract Classes and Interfaces.

Unit III

Exception Handling: Fundamentals – Hierarchy of Classes – Types of Exceptions-Exception Class – Uncaught Exceptions – Handling Exceptions – User Defined Exceptions.

Multithreaded Programming: The Java Thread Model – Runnable Interface - Thread Class – Thread Creation – Thread's Life Cycle – Thread Scheduling -Synchronization and Deadlock. Packages and Access Modifiers: Package Declaration – The CLASSPATH variable - import statement – The Java Language Packages - Access Protection.

Unit IV

Strings: Creation – Operation on strings - Character Extraction Methods – Comparison – Searching and Modifying –Data Conversion and valueOf() Methods – Changing case of characters - String Buffer Class and its methods. Collection and Utilities: Collection of Objects – Core Interfaces and Classes – Iterators – List, Set, Map Implementations.

Unit V

Input Output Classes: I/O Operations –Hierarchy of Classes – File class – Input Stream, Output Stream, FilterInputStream, FilterOutputStream, Reader and Writer classes – Random Access File class –Stream Tokenizer. Applets: Basics – Life Cycle –Methods –Graphics Class- Color, Font, and Font Metrics Class – Using the Status window – Passing parameters to Applets – getDocumentBase() and getCodeBase(). AWT Components: AWT Classes – Basic Component and Container Classes – Frame Window in an Applet.

Suggested Readings

1. Herbert Schildt, 2014, Java Complete Reference, 9th Edition, Tata McGraw Hill, New Delhi.
2. ISRD Group, 2007, Introduction to Object Oriented Programming through Java, 1st Edition, Tata McGraw Hill, New Delhi.[Unit -I (3-104), Unit -II (105-127), Unit -III (129-164), Unit -IV (219-236, 253-280), Unit -V (165-199, 283-307)]
3. Deitel H.M. and P.J.Deitel, 2005, Java-How to Program, 6th Edition, Pearson Education, New Delhi.
4. Dr.S Somasundaram, 2004, Java Programming**,** 1st Edition, Techmedia. New Delhi.
5. E.Balagurusamy, 2010, Programming with Java – A Primer, 4th Edition, Tata McGraw Hill, New Delhi.

Web Sites

1. www.java.sun.com
2. www.knking.com
3. www.webdeveloper.com
4. www.forums.sun.com
5. www.netbeans.com
6. java.sun.com/docs/books/tutorial/
7. www.java.net/

ESE Pattern	
Part – A (Online)	20 * 1 = 20
Part – B	5 * 2 = 10
Part – C (Either or)	5 * 6 = 30
Total	60 marks

CIA Pattern	
Part – A	20 * 1 = 20
Part – B	3 * 2 = 6
Part – C (Either or)	3 * 8 = 24
Total	50 marks

Faculty **HOD**

sKARPAGAM UNIVERSITY
Karpagam Academy of Higher Education
(Deemed University Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

LECTURE PLAN

SUBJECT NAME: PROGRAMMING IN JAVA

SUBJECT CODE: 19ITU201 **SEMESTER: II**

STAFF: Dr.D.SHANMUGA PRIYAA **CLASS: I B.Sc. IT**

S. No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
Unit I			
1.	1	Introduction to Object Oriented Programming - Object Oriented Paradigm and Concepts, Structured vs Object oriented approach	S1: 1-9
2.	1	The JAVA Language - Features of Java - Java Architecture, JDK - Types of Java Program	S1: 10-19
3.	1	Variable Declaration and Arrays - Data types in Java	S1: 20-30
4.	1	Java Tokens - Variable declaration, Type casting and conversion - Arrays	S5: 45 -57 W1
5.	1	Operators in Java – Operators, Operators - Operator precedence	S1: 31 - 40 S5: 60 - 76 W1
6.	1	Control Statements – Introduction - Selection constructs, Iteration constructs - Jump statements	S1: 41 - 53
7.	1	Recapitulation and discussion of important questions	
	Total No. of Hours planned for Unit-I		**7**
Unit II			
1.	1	Introduction to Classes - Class-An Introduction - Instance variables, Constructors - Class methods	S1: 54 -65
2.	1	Declaring objects - Garbage collection, Classes and Methods - Method overloading	S1: 65-78 W1
3.	1	Constructor overloading , this reference	S1: 79 - 88
4.	1	Inheritance - Basics of inheritance - Super class variables	S1: 89 - 97 W1
5.	1	Method overriding - The *final* keyword	S1: 98-106
6.	1	Abstract classes and Interfaces - The abstract classes and methods - Defining interfaces	S1: 107 - 111 W1

7.	1	Implementing interfaces - Extending interface - Interface reference	S1: 112 - 119
8.	1	Recapitulation and discussion of important questions	
	Total No. of Hours planned for Unit-II		**8**
Unit - III			
1.	1	Exception Handling – Fundamentals, Hierarchy of exception class	S1: 120 - 125 S5: 220 - 233 W2
2.	1	Types of exceptions - Exception class, Uncaught exceptions - Handling exceptions	S1: 120 - 137
3.	1	Multithreaded Programming - Java thread model, Runnable interface - Thread class	S1: 138 -150 S5: 198 - 219
4.	1	Synchronization and Deadlock	S1: 151 - 160
5.	1	Packages and Access Modifiers – Introduction, Package declaration - Classpath variable, Import statement - Access protection	S1: 161 -176 W1
6.	1	Recapitulation and discussion of important questions	
	Total No. of Hours planned for Unit-III		**6**
Unit – IV			
1.	1	Handling Strings - Creating strings, Operations on strings	S1: 177 -183
2.	1	Character extraction methods - Comparison –Searching and Modifying	S1: 184 -187
3.	1	Data Conversion and valueOf() Methods – Changing case of characters	S1: 188 -191
4.	1	Searching and modifying strings - StringBuffer class	S1: 192-194
5.	1	Collection and Utilities - Collections of objects, Core interfaces and classes- Iterators, List implementations	S1: 218 - 243
6.	1	Set implementations	S1: 244-249
7.	1	Map implementations	S1: 250 - 252
8.	1	Recapitulation and discussion of important questions	
	Total No. of Hours planned for Unit-IV		**8**
Unit – V			
1.	1	Input Output Classes -I/O operations - Hierarchy of classes	S1: 253 -255 W2
2.	1	File class - InputStream and OutputStream	S1: 256 -260

3.	1	FilterInputStream - FilterOutputStream	S1: 261 -265
4.	1	Reader and Writer classes, RandomAccessFile class - Stream tokenizer	S1: 266 - 275
5.	1	Applets - Applet Basics - Applet Life cycle, Running applets - Methods of applet class	S1: 292 - 295 S5: 234 - 260 W3
6.	1	Graphics class, Color class, Font class - FontMetrics class - Limitations of applet	S1: 296 -310
7.	1	AWT Components - AWT classes - Basic component class	S1: 311 -320 W4
8.	1	Container classes - Frame window in an applet	S1: 321 -330
9.	1	Recapitulation and discussion of important questions	
10.	1	Previous ESE Question Paper Discussion	
11.	1	Previous ESE Question Paper Discussion	
	Total No. of Hours planned for Unit-V		**11**

Total No. of Hours planned: 40

Text Books

S1. Herbert Schildt, 2014, Java Complete Reference, 9th Edition, Tata McGraw Hill, New Delhi.

S2. ISRD Group, 2007, Introduction to Object Oriented Programming through Java, 1st Edition, Tata McGraw Hill, New Delhi.

S3. Deitel H.M. and P.J.Deitel, 2005, Java-How to Program, 6th Edition, Pearson Education, New Delhi.

S4. Dr.S Somasundaram, 2004, Java Programming**,** 1st Edition, Techmedia. New Delhi.

S5. E.Balagurusamy, 2010, Programming with Java – A Primer, 4th Edition, Tata McGraw Hill, New Delhi.

Web Sites

W1. www.java.sun.com
W2. www.knking.com
W3. www.webdeveloper.com
W4. www.forums.sun.com
W5. www.netbeans.com

Faculty **HOD**

KARPAGAM ACADEMY OF HIGHER EDUCATION
CLASS: I B.Sc IT COURSE NAME: Programming in Java
COURSE CODE: 19ITU201 UNIT: I (Introduction, Datatypes) BATCH-2019-2022
SYLLABUS

Introduction to Object Oriented Programming: Object Oriented Paradigm and Concepts-Structured versus Object Oriented Approach. Java Language: Features of Java -Environment-Java Architecture-Java Development Kit-Types of Java Program. Variable Declaration and Arrays: Data Types-Java Tokens –Variable Declaration – Type Casting and Conversion – Arrays, Operators,
And Control Statements: Selection Constructs – Iteration Constructs –Jump Statements.

Introduction to Object Oriented Programming

Is car an Object, how will you decide that?

If you want to term something as an object then sure it must have properties and behaviors. What are the properties and behaviors of a car? Let's list out

Properties of a car

Properties
Name
Color
Gas_Accepted
Passenger Capacity
Top Speed
Current Speed

Behavior of a car

Behavior
Accelerate (speed up)
Brake (slow down)
Turn Left (turn the wheels)
Turn right (turn the wheels)
Beep (horn)
Monitor_Tank

Yes of course, car is an object simply because it has its own properties and behavior. In other words object is a collection of properties and behavior. Properties can be handled by the data and the behaviors can be handled by the methods.

Finally, Object is a collection of data and methods.

Definition

Object oriented programming approach organizes data about real world entities (objects) in problem domain and a set of well defined interfaces to the data.

Object Oriented Paradigm and Concepts

1) **Object**

In object oriented programming, the object is the basic unit; the focus is mainly on data and behaviors. The purpose of object oriented programming is to combine data and behavior into a package, just as objects in the real world do.

2) **Class**

Classes are the base-structures or blueprints or templates from which objects are created. These structures define all the properties and behavior an object will possess.

3) **Data and Behavior**

In OOP, the properties used to describe an object are known as data. Data generally defines how an object looks like.
The behaviors are implemented as functions called methods.
For example, Mobile Phone
Data defines size, color, screen size of the mobile phone whereas the behavior describes making calls, sending messages and taking pictures etc.
These data and methods combined together into single, self contained unit called object.

4) **Abstraction**

Abstraction enables us to focus only on essential and ignore the non-essential. I other words exposing only the necessary details and ignore the unnecessary.
For example,

1) To drive a car it is not mandatory that one has to be aware of internal workings of a car engine
2) Coimbatore to Salem, what's the route map.
Coimbatore→ Avinashi→ Perundurai→Salem. Only the major towns are focused and the small villages, houses, trees in between them are ignored.

5) **Encapsulation**

Capsules may be used when more mixes of sensitive drugs needs to be taken, but those drugs can't be viewed from outside world.Similarly encapsulation or information hiding permits objects to operate as complete independent, self contained package of data and methods. It hides the data and method implementation from the outside world.

6) Inheritance

Inheritance allows the new class to automatically inherit the data and methods of another class. It also allows adding new data and methods to the inherited ones. This dynamically increases the proficiency.

7) Message Passing

Communication among the objects can be made through message passing, any object can send message to any other object.

8) Polymorphism

Polymorphism is a feature that allows one interface to be used for a general class of actions. For example, a single button of a mobile phone is used to call, take pictures, send messages etc.

Polymorphism achieves extensibility.

Structured versus Object Oriented Approach

In conventional programming methodology the focus is only on algorithm whereas in object oriented programming the focus is on data rather than the algorithm.

In the traditional approach the problem is divided into functions whereas in OOP the problem is divided into objects. Complex real world objects can be modeled/represented only on object oriented programming which is a tedious task in traditional method.

In structured approach, the data are mostly defined as global so that any function can access which leads to lack in data security and data integrity. In OOP this is avoided with the help of encapsulation concept.

The Java Language: Features of Java

Java changes the passive nature of the Internet and World Wide Web by enabling architecturally neutral code to be dynamically loaded and run on a heterogeneous network of machines. It is also a leading programming language for wireless technology and real-time systems. ·

Sun Microsystems officially describes Java as a programming language with the following attributes:

- Compiled and Interpreted
- Platform independent and Portable
- Object oriented
- Robust and Secure
- Distributed
- Multithreaded
- Dynamic

Compiled and Interpreted

Java is both a compiled and an interpreted language. Java translates source code into bytecode instructions. Java interpreter generates machine code that can directly be executed by the particular machine that is running the Java program.

Platform Independent and Portable

Java programs once written can be run anywhere anytime .Java's portability is one of the major reasons for its popularity. A program written in Java can easily be moved from one computer system to another.

The Java programmer need not make any alterations in the code for using it on a computer having a different operating system, processor and system resources.

This feature has made Java a popular language for the Internet.

Object Oriented

Java is clean, usable, pragmatic approach to object orientation. The object model in java is simple and easy to extend, while simple types, such as integers are kept as high performance non objects.

Robust and Secure

The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably on a variety of systems. Accordingly, 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 life cycle.

Further, it also checks your code at runtime. ln fact, many hard-to-track-down bugs that often tum up in hard-to-reproduce runtime situations arc simply impossible to occur in Java.

For a language that is widely used for programming on the Internet, security becomes a crucial issue. Java systems safeguard the memory by ensuring that no viruses are communicated with an applet. As there are no pointers in Java, the programs are not allowed to gain access to memory locations without proper authorization.

Distributed

Java is a distributed language; it can be used for creating applications that can be run on networks. It can share both data and programs and Java applications can easily access remote objects on Internet.

Multithreaded

The word Multithreaded implies handling multiple tasks simultaneously. Java supports multithreaded programs. i.e. the user need not wait for the application to execute one task completely before starting the other. For example. one can Listen 10 sound clip while browsing a page and at the same time download an applet from a remote computer.

A multithreaded application can have several threads of execution running independently and simultaneously. These threads may communicate and cooperate and will appear to be a single stream to the user.

Dynamic

Java was designed to adapt in a constantly evolving environment It is capable of incorporating new functionality whether it comes from local system, local network or the Internet. Java dynamically links new class libraries and methods at runtime. This gives Java programs a high level of flexibility during execution.

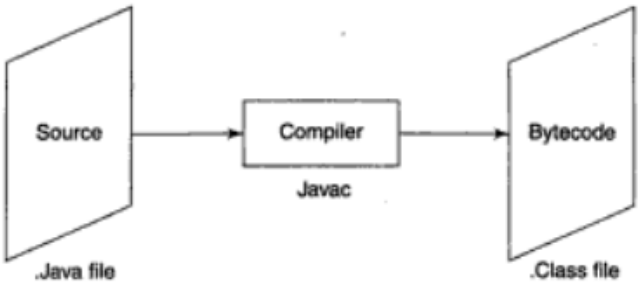
Environment: Java Architecture

The Java technology is actually a group of technologies. It not only provides the language part for developing applications, but also supports architecture for running these applications. It concurrently provides necessary tools to develop compile and run the Java applications. The Java architecture provides a portable, high-performance, robust runtime environment within which the Java language can be used.

The Bytecode

The first step in the Java application life cycle is the compilation of code. The Java compiler acts just as any other compiler. lt creates the machine code for execution from a higher level language Java compiler compiles the code for a machine that physically does not exist. i.e. for a virtual machine.

This compiled code is known as bytecode, and hypothetical machine is called Java Virtual Machine (JVM). Bytecode is a highly optimized set or instructions designed to be executed by JVM. Converting a Java program into bytecode makes it easier to run a program in a wide variety of environments.



Java Virtual Machine

Java language is platform independence. usually referred to as "write once run anywhere (WORA) ... This is accomplished by tbc JVM that runs on the local machine, interprets the Java bytecode, and converts it into platform-specific machine code.

The JVM is invoked differently depending on the type of Java program.

JVM performs the following functions:

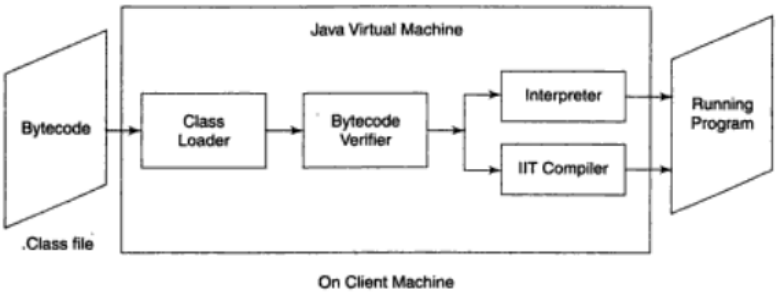
I. When a class file is executed. JVM loads all required classes automatically from the local disk from across the network. This is the function of "Class loader"' utility of JVM.

2. After loading the required classes. JVM verifies to make sure that the classes do not violate any of the basic rules of the Java language, This is the function of the "Bytecode verifier"

3. Theo JVM keeps track of all memory usage. It takes care of memory allocation and also performs the release of memory after the object is no longer needed. This process which manages dereferenced objects is called Garbage Collection.

Just In Time (JIT) Compilers

Within JVM, Just-In-Time (JIT) compilers are used to improve performance. JIT compiler translates bytecodes only the first time. If repeated execution of the code is required. it is automatically mapped to the corresponding native machine code. This is especially effective in repetitive code such as loops or recursive functions.



Java Development Kit (JDK)

The package that provides the basic functionality of Java language as a series of classes and methods, and the tools that are used to develop and execute Java programs is known as Java Development Kit (JDK).

The major part of it comprises of Software Development Kit (SDK). Java 2 SDK 1.4 (j2sdk) includes the following: sets of tools:

Javac	The compiler for the java language
Java	The launcher for java applications
Javadoc	API document generator
Appletviewer	Run and debug applets without a web browser
Jar	Manage Java archive files
Jdb	The Java debugger
Javah	C header generator
Javap	Class file disassemble

bin It contains the executable files for the development toots contained in the JDK. like javac, java , appletviewer etc. The PATH environment variable shou1d contain an entry for this directory.

lib This directory contains files used by the development tools. It includes rools.jar, which contains non-core classes for support of the tools and utilities in the SDK.

Jre This is the root directory of the Java runtime environment used by the SDK development tools. This is the directory represented by the Java ".home" system property.

Types of Java Program

Though Java was created by James Gosling for developing small, platform-independent and robust programs that were used in consumer electronics it can be used to develop more dynamic programs. It is the leading programming language for wireless technology, web services and real-time embedded programming for cell phones. Broadly we can categorize Java programs into the following two main groups:

• Applets
• Applications

Applets

A Java applet is a small program embedded in a web page and is run when that page is browsed using a web browser. Applets arc downloaded over the network and can make network

connections only to the host they are issued from. Applets arc inherently graphical in nature and lend to contain controls such as buttons, labels, text fields, etc..

For execution of an applet, JVM is built into the browser. Applets can connect into a database on the Web server communicate with the web server and can play audio clips, animations and images. But they are restricted from accessing a local machine.

Applications

Applications are stand-alone program written in Java. They are invoked by using a JVM which resides within a local operating system. Unlike applets, Java applications can access the local file- system or establish connections with other machines on the network.

An application must contain a static method 'main()' from where its execution begins. .

Java applications can also execute on the server machine. The multitier model of Internet computing uses these types of server-side Java applications.

Simple Java Program

```
class Condition {
  public static void main(String[] args) {
    boolean learning = true;

    if (learning) {
      System.out.println("Java programmer");
    }
    else {
      System.out.println("What are you doing here?");
    }
  }
}
```

Variable declaration and arrays

Primitive Data Types:

There are totally eight primitive data types in Java. They can be categorized as given below:

Integer types (Does not allow decimal places)

- byte
- short
- int
- long

Rational Numbers(Numbers with decimal places)

- float
- double

- characters
  - char
- conditional
  - boolean

Please notice that all the data type keywords are in small letters. These are part of the java keywords and every keyword in java is in small letters.

In the **integer data types**, we have four different data types. But, why do we need four different types when one can do the job. Yes, it is extremely important to understand that each and every data type has limitations to the amount of numbers it can represent. There is a memory constraint defined for every data type.
Understanding the memory limitations is extremely important in deciding which data type should be used. For example, when you are representing the age of a person, for sure it will not cross 120, so, using short data type is enough instead of long which has very big memory foot print.
The following should be understood for every data type:

1. Memory size allocated.
2. Default value
3. Range of values it can represent.

Data types in Java details

Data Type	Memory Size	Default value	Declaration
Byte	8 bits	0	byte a=9;
Short	16 bits	0	short b=89;
Int	32 bits	0	int c=8789;
Long	64 bits	0	long=9878688;
Float	32 bits	0.0f	float b=89.8f;
Double	64 bits	0.0	double c =87.098
Char	16 bits	'u0000'	char a ='e';
Boolean	JVM Dependent	false	boolean a =true;

Java Tokens

A token is the smallest element in a program that is meaningful to the compiler. These tokens define the structure of the language. The Java token set can be divided into five categories: Identifiers, Keywords, Literals, Operators, and Separators.

1. Identifiers

Identifiers are names provided by you. These can be assigned to variables, methods, functions, classes etc. to uniquely identify them to the compiler.

2. Keywords

Keywords are reserved words that have a specific meaning for the compiler. They cannot be used as identifiers. Java has a rich set of keywords. Some examples are: boolean, char, if, protected, new, this, try, catch, null, threadsafe etc.

3. Literals

Literals are variables whose values remain constant throughout the program. They are also called Constants. Literals can be of four types. They are:

a. String Literals

String Literals are always enclosed in double quotes and are implemented using the java.lang.String class. Enclosing a character string within double quotes will automatically create a new String object. For example,String s = "this is a string";. String objects are immutable, which means that once created, their values cannot be changed.

b. Character Literals

These are enclosed in single quotes and contain only one character.

c. Boolean Literals

They can only have the values true or false. These values do not correspond to 1 or 0 as in C or C++.

d. Numeric Literals

Numeric Literals can contain integer or floating point values.

4. Operators

An operator is a symbol that operates on one or more operands to produce a result.

5. Separators

Separators are symbols that indicate the division and arrangement of groups of code. The structure and function of code is generally defined by the separators. The separators used in Java are as follows:

parentheses ( )

Used to define precedence in expressions, to enclose parameters in method definitions, and enclosing cast types.

braces { }

Used to define a block of code and to hold the values of arrays.

brackets [ ]

Used to declare array types.

semicolon ;

Used to separate statements.

comma ,

Used to separate identifiers in a variable declaration and in the for statement.

period .

Used to separate package names from classes and subclasses and to separate a variable or a method from a reference variable.

Variables

There are different types of variables in Java. They are as follows:

1. Instance Variables (Non-Static Fields)

Objects store their individual states in "non-static fields", that is, fields declared without the static keyword.

Non-static fields are also known as instance variables because their values are unique to each instance of a class. For example, the currentSpeed of one bicycle is independent from the currentSpeed of another.

2. Class Variables (Static Fields)

A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as static since, conceptually, the same number of gears will apply to all instances. The code static int numGears = 6; would create such a static field.

3. Local Variables

A method stores its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (for example, int count = 0;). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared—between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

4. Parameters

They are the variables that are passed to the methods of a class.

Variable Declaration

Identifiers are the names of variables. They must be composed of only letters, numbers, the underscore, and the dollar sign ($). They cannot contain white spaces. Identifiers may only begin with a letter, the underscore, or the dollar sign. A variable cannot begin with a number. All variable names are case sensitive.

Syntax for variable declaration

datatype1 variable1, datatype2 variable2, … datatypen variablen;

For example:

int a, char ch;

Initialisation

Variables can be assigned values in the following way: Variablename = value;
For example;

```
ch='a';
a=0;
```

Type Casting and Conversions

Java data type casting comes with 3 flavors.

1. Implicit casting
2. Explicit casting
3. Boolean casting.

1. Implicit casting

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion.

Examples:

```
int x = 10;                 // occupies 4 bytes
double y = x;               // occupies 8 bytes
System.out.println(y);      // prints 10.0
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

2. Explicit casting

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires explicit casting; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

```
double x = 10.5;            // 8 bytes
int y = x;                  // 4 bytes ;  raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

```
double x = 10.5;
int y = (int) x;
```

The double x is explicitly converted to int y. The thumb rule is, on both sides, the same data type should exist.

3. Boolean casting

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is incompatible for conversion. Maximum we can assign a boolean value to another boolean.

Following raises error.

```
boolean x = true;
int y = x;              // error
boolean x = true;
int y = (int) x;        // error
```

byte –> short –> int –> long –> float –> double

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, byte can be assigned to short implicitly but short to byte requires explicit casting.

Arrays in Java

Introduction to Arrays

A Java array is an ordered collection of primitives, object references, or other arrays.
Java arrays are homogeneous: except as allowed by polymorphism, all elements of an array must be of the same type.
Each variable is referenced by array name and its index.
Arrays may have one or more dimensions.

One-Dimensional Arrays

A one-dimensional array is a list of similar-typed variables. The general form of a one-dimensional array declaration is:
type var-name[ ];

type declares the array type.
type also determines the data type of each array element.
The following declares an array named days with the type "array of int":

int days[];

days is an array variable.
The value of days is set to null.
Allocate memory for array

You allocate memory using new and assign it to array variables. new is a special operator that allocates memory. The general form is:

arrayVar = new type[size];

type specifies the type of data being allocated.
size specifies the number of elements.
arrayVar is the array variable.
The following two statements first create an int type array variable and then allocate memory for it to store 12 int type values.

int days[];
days = new int[12];
days refers to an array of 12 integers.

All elements in the array is initialized to zero.
Array creation is a two-step process.

declare a variable of the desired array type.
allocate the memory using new.
In Java all arrays are dynamically allocated.
You can access a specific element in the array with [index].
All array indexes start at zero.
For example, the following code assigns the value 28 to the second element of days.

```
public class Main {
  public static void main(String[] argv) {
    int days[];
    days = new int[12];

    days[1] = 28;

    System.out.println(days[1]);
  }
}
```

It is possible to combine the declaration of the array variable with the allocation of the array itself.

int month_days[] = new int[12];

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. For example, the following declares a two-dimensional array variable called twoD.

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

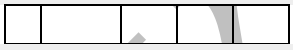
This allocates a 4-by-5 array and assigns it to twoD. This array will look like the one shown in the following:

```
  [leftIndex][rightIndex]


  [0][0] [0][1] [0][2] [0][3] [0][4]
  [1][0] [1][1] [1][2] [1][3] [1][4]
  [2][0] [2][1] [2][2] [2][3] [2][4]
  [3][0] [3][1] [3][2] [3][3] [3][4]
```

The wrong way to think about multi-dimension arrays

```
+----+----+----+
|  1|  2|  3|
+----+----+----+
|  4|  5|  6|
+----+----+----+
|  7|  8|  9|
+----+----+----+
```

right way to think about multi-dimension arrays



```
| |--------|  1|  2|  3|
+--+      +----+----+----+    +----+----+----+
| |----------------------------|  4|  5|  6|
+--+  +----+----+----+        +----+----+----+
| |---|  7|  8|  9|
+--+  +----+----+----+
```

An irregular multi-dimension array

```
+--+      +----+----+
| |--------|  1|  2|
+--+      +----+----+        +----+----+----+
| |----------------------------|  4|  5|  6|
+--+  +----+----+----+----+    +----+----+----+
| |---|  7|  8|  9| 10|
+--+  +----+----+----+----+
```

The following code use nested for loop to assign values to a two-dimensional array.

```
public class Main {
  public static void main(String args[]) {
    int twoD[][] = new int[4][5];
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 5; j++) {
        twoD[i][j] = i*j;
      }
    }
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 5; j++) {
        System.out.print(twoD[i][j] + " ");
      }
      System.out.println();
    }
  }
}
```

Java Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2

%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

The Relational Operators:

There are following relational operators supported by Java language
Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.
Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100
b = 0000 1101
---
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right	A >>>2 will give 15 which is 0000 1111

	operand and shifted values are filled up with zeros.	

The Logical Operators:
The following table lists the logical operators:
Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators:
There are following assignment operators supported by Java language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the	C *= A is equivalent to C = C * A

	result to left operand	
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {

   public static void main(String args[]){
      int a , b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
```

```
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

Control Statements in Java

Java Control statements in a programming language are very useful as they allow a programmer to change the flow of program execution i.e. altering the normal program flow to jump directly on some statement(s) or to skip a statement(s). In Java control statements are divided into following 3 categories:

Selection/Decision making Statements

Using these statements, a piece of code would be executed only if a certain condition(s) is true. These are of 3 types:

1. if

Statement(s) between the set of curly braces '{ }' will be executed only if the condition(s), between the set of brackets '( )' after 'if' keyword, is/are true.

Syntax:

```
if (Condition) {
   // statements;
}
```

2. if-else

If the condition(s) between the brackets '( )' after the 'if' keyword is/are true then the statement(s) between the immediately following set of curly braces '{ }' will be executed else the statement(s) under, the set of curly braces after the 'else' keyword will be executed.

Syntax:

```
if (condition) {
   // statements;
} else {
   // statements;
}
```

3. switch

When there is a long list of cases & conditions, then if/if-else is not good choice as the code would become complicated.

Syntax:

```
switch (expression)
{
   case value1:
   //statement;
   break;
   case value2:
   //statement;
   break;
   default:
```

```
    //statement;
}
```

In the above piece of code, the user's choice (add/sub/mul) will be stored in variable 'ch'. The moment user enters his choice, it will be matched with the cases' names & program execution will jump to the matching 'Case' & the statement under that case will be executed till the keyword 'break' comes. It is very important else the other unwanted cases will also get executed. After the last case there is 'default' keyword. Statements between 'default:' and the closing bracket of switch-case region will be executed only if the user has entered any wrong value as his choice i.e. other than the cases' names.

Loop/ Iteration Statements

4. while

while statement continually executes a block of statements while a particular condition is true. Entry controlled

Syntax:

```
while(conditions)
{
//Loop body
}
```

5. do-while

It will enter the loop without checking the condition first and checks the condition after the execution of the statements. That is it will execute the statement once and then it will evaluate the result according to the condition. Exit controlled

Syntax:

```
Do
{
//Loop body
}while(condition);
```

6. for

The concept of Iteration has made our life much easier. Repetition of similar tasks is what Iteration is and that too without making any errors. Until now we have learnt how to use selection statements to perform repetition.

Syntax:

```
for(initialization;test condition;increment)
{
//Loop body
}
```

Branching/ Transfer statements

7. break

Sometimes we use Jumping Statements in Java. Using for, while and do-while loops is not always the right idea to use because they are cumbersome to read. . Break statement skips the following code lets the execution jump to point after, where the program execution has jumped from to switch-case region.

Syntax:

```
for(initialization;test condition;increment)
{
   if(condition)
   {
   //statements
   break;
   }
}
```

8. continue

Continue statement is just similar to the break statement in the way that a break statement is used to pass program control immediately after the end of a loop and the continue statement is used to force program control back to the top of a loop.

Example

```
for(initialization;test condition;increment)
{
   if(condition)
   {
   //statements
   continue;
   }}
```

Possible Questions

Part – B (2 Marks)

1. Define Abstraction
2. What is inheritance?

3. List the features of Java
4. What is bytecode?
5. List the functions performed by java virtual machine
6. Mentio nthe two types of java program
7. What is symbolic constants

Part – C (6 Marks)

1. Explain Object Oriented Paradigm Concept in detail.
2. Write note on
    a. Features of Java
    b. Java Architecture
3. Describe Java data type with neat example.
4. Explain Java Tokens.
5. Describe about Operators with example for each.
6. Explain various Control statements in Java with example.

Questions	opt1	opt2	opt3	opt4	answer
Java is a _____ language	structured programming	object oriented	procedural oriented	machine	object oriented
OOPS follows_____ approach in program design	bottom_up	top_down	middle	top	bottom_up
Objects take up _____in the memory	Space	Address	Memory	bytes	Space
_____is a collection of objects of similar type	Objects	methods	classes	messages	classes
The wrapping up of data & function into a single unit is known as	Polymorphism	encapsulation	functions	data members	encapsulation
_____refers to the act of representing essential features without	Encapsulation	inheritance	Dynamic binding	Abstraction	Abstraction
Attributes are sometimes called_____	data members	methods	messages	functions	data members
The functions operate on the datas are called_____	Methods	data members	messages	classes	Methods
_____is the process by which objects of one class acquire the properties	Polymorphism	encapsulation	data binding	Inheritance	Inheritance
_____means the ability to take more than one form	Polymorphism	encapsulation	data binding	Inheritance	Polymorphism
The process of making an operator to exhibit different behaviors in different	function overloading	operator overloading	method overloading	message overloading	operator overloading
Single function name can be used to handle different types of tasks is known	function overloading	operator overloading	polymorphism	encapsulation	function overloading
.Variables are declared in_____	only in main()	anywhere in the scope	before the main() only	only at the beginning	anywhere in the scope
._____refers to permit initialization of the variables at run time	Dynamic initialization	Dynamic binding	Data binding	Dynamic message	Dynamic initialization
Keyword _____ indicates that method do not return any value.	Static	Final	void	null	void
_____ is used to define the objects	class	functions	methods	variables	class
An _____ is a single instance of a class that retains the structure and	class member	object	instances	reference	object
A _____ is a message to take some action on an object	member	variable	method	class	method
Java does not have _____ statement	goto	if	do	do while	goto
_____ is used to separate package names from sub_packages and classes	:	,	.	!	.
The _____ is the basic unit of storage in a Java program	identifier	variable	class	object	variable
byte belongs to _____ type.	character	Boolean	floating	integer	integer
In Java an int is _____ bits	16	64	52	32	32
byte is a signed _____ type	16 bit	8 bit	32 bit	64 bit	8 bit

The _____ statement is often used in switch statement	break	end	do	loop	break
The keywords private and public are known as _____ labels	Static	Dynamic	Visibility	const	Visibility
The class members that have been declared as _____ can be accessed	Private	Public	Static	protected	Private
The class members that have been declared as _____ can be accessed	Private	Public	Static	protected	Public
The class variables are known as _____	Functions	members	objects	none of the above	objects
The _____ command from J2SDK compiles a Java program.	Java	Appletviewer	Javac	javad	Javac
File produced by the java compiler contains _____	ASCII	Class	Pnemonics	ByteCodes	ByteCodes
The file produced by java compiler ends with _____ file extension	Java	html	class	applet	class
Objects are instantiated from_____	Java	methods	groups		class
Which of the following lines is not a Java comment?	/** comments */	// comments	– comments	/* comments */	– comments
Which of the following statements is correct?	system.out.println('Welcome to	System.out.println("Welcome to	System.println(' Welcome to	System.out.print('Welcome to	System.out.println("Welcome to
A block is enclosed inside _____.	Parentheses	Braces	Brackets	Quotes	Braces
Wich of the following is a correct signature for the main method?	static void main(String[]	public static void main(String[]	public void main(String[]	public static void main(Strings[]	public static void main(String[]
Which of the following lines is not a Java comment?	/** comments */	// comments .	– comments	/* comments */	– comments
_____ translates the Java sourcecode to bytecode files that the	javac	java	javap	jdk	javac
In java the functions are called as _____	fields	method	variables	final	method
_____ an object is also called as instantiating an objects	deleting	creating	destroy	new	creating
Keyword _____ indicates that method do not return any value.	Static	Final	void	null	void
Java interpreter is	JVM	Javac	Compiler	JAR	JVM
The _____ method terminates the program.	System.terminate (0);	System.halt(0);	System.exit(0);	System.stop(0);	System.exit(0);
Java has no _____ function.	malloc	free	malloc and free	calloc	malloc and free
Java supports _____ inheritance	single	multiple	single and multiple	multilevel	single
Java does not have _____	sturct	header files	union	All	All
_____ is a access specifier	static	void main	public	protected	public
Java is a _____ type language.	Weak	strong	correct	incorrect	strong

Data type Short occupies _____ bytes.	1	2	4	8	2
The Properties used to describe an object are known as	Data	Attributes	Entities	Relations	Data
It enables us to ignore the non_essential	Inheritance	Encapsulation	Abstraction	DataBinding	Abstraction
It is the most powerful feature of any programming technique	top_down	bottom up	Code reusability	Security	Code reusability
Encapsulation is also known as	Abstraction	Information hiding	Polymorphism	Inheritence	Information hiding
Well defined entities that are capable of interacting with themselves	Encapsulation	Message Passing	Abstraction	Binding	Message Passing
Which of the following is a valid identifier?	area	Class	9X	8+9	area
A literal character is represented inside a pair of _____	single quotes	double quotes	brackets	paraenthesis	single quotes
short is a signed _____ type	8 bit	16 bit	32 bit	64 but	16 bit
Single precision is specified by _____keyboard	int	double	float	char	float
To add number to sum, you write (Note: Java is case-sensitive).	number += sum;	number = sum + number;	sum = Number + sum;	sum += number;	sum += number;

SYLLABUS

Introduction to classes: Instance variables, Class variables, Instance Methods, Constructors, Class methods, Declaring Objects, Garbage Collection, Method Overloading - Constructor Overloading - This Reference. Inheritance: Super class variables- Method Overriding - final Keyword, Abstract Classes and Interfaces.

Introduction to classes

A class is a template or a prototype defines a type of object. A class is to an object what a blueprint is to a house. A class is a collection of data variables and methods that define a particular entity. A class can be either user-defined or provided by one of the built in java packages.

Defining a Class

The class is defined using a keyword **class** followed by a user defined class name. The body of the class is contained in the block that is defined by curly braces{}

```
class classname
{
        [variable declarations;]
        [method declarations;]
}
```

The data or variables defined within a classes are called instance variables. The code is contained within methods, these are also called members of the class.

For example

```
class exampleclass
{
        char cc;
        int f1;
        double dd;
        void examplemethod1()
        {
        System.out.println("Hello world");
        }
        void examplemethod2()
        {
```

```
System.out.println("Hai World");
}
}
```

A class is an encapsulated collection of data, and methods to operate on data. A class definition typically includes the following

1. Access Modifier
2. The class keyword
3. Instance fields
4. Constructors
5. Instance methods
6. Class fields
7. Class method

There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/static variables

Local variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use**.**

For example

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

```
public class Test{
  public void pupAge(){
    int age = 0;
    age = age + 7;
    System.out.println("Puppy age is : " + age);
  }

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

```
    Test test = new Test();
    test.pupAge();
  }
}
```

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . *ObjectReference.VariableName*.

```
import java.io.*;

public class Employee{
  // this instance variable is visible for any child class.
  public String name;

  // salary  variable is visible in Employee class only.
  private double salary;

  // The name variable is assigned in the constructor.
  public Employee (String empName){
    name = empName;
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal){
    salary = empSal;
  }
```

```
  // This method prints the employee details.
  public void printEmp(){
    System.out.println("name  : " + name );
    System.out.println("salary :" + salary);
  }

  public static void main(String args[]){
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
  }
}
```

Class/static variables:

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name . *ClassName.VariableName.*
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

```
import java.io.*;
public class Employee{
```

```
  // salary  variable is a private static variable
  private static double salary;

  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "Development ";

  public static void main(String args[]){
    salary = 1000;
    System.out.println(DEPARTMENT+"average salary:"+salary);
  }
}
```

Instance Methods

A java method is equivalent to a function, procedure, or subroutine in other languages except that it must be defined inside a class definition. Instance methods are the foundation of encapsulation and provide a consistent interface to the class.

Adding methods to the class

Methods are declared inside the body of the class but immediately after the declaration of the instance and class variables. The general form of a method declaration is

```
returntype methodname(parameter_list)
{
        Method body;
}
```

A returntype can be a primitive type such as int, or a class type such as string or void.

A methodname begin with a lowercase letter and according to java convention, compund words in the method name should begin with uppercase letters.

The method body must be enclosed in curly braces.

An optional parameter_list/argument_list must be inside parenthesis, seperated by commas.
For example

```
String gettitle()
        {
                return title;
        }
void printdetails()
        {
```

```
        System.out.println("Title is:"+title);
    }
```

Constructors

A **java constructor** has the same name as the name of the class to which it belongs. Constructor's syntax does not include a return type, since constructors never return a value.

Constructors may include parameters of various types. When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition.
Java provides a default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided.
The only action taken by the implicit default constructor is to call the superclass constructor using the super() call. Constructor arguments provide you with a way to provide parameters for the initialization of an object.

Below is an example of a cube class containing 2 constructors. (one default and one parameterized constructor).

```
public class Cube1 {

    int length;
    int breadth;
    int height;
    public int getVolume() {
        return (length * breadth * height);
    }
    Cube1() {
        length = 10;
        breadth = 10;
        height = 10;
    }
    Cube1(int l, int b, int h) {
        length = l;
        breadth = b;
        height = h;
    }
    public static void main(String[] args) {
        Cube1 cubeObj1, cubeObj2;
        cubeObj1 = new Cube1();
        cubeObj2 = new Cube1(10, 20, 30);
        System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());
        System.out.println("Volume of Cube1 is : " + cubeObj2.getVolume());
    }
}
```

Class Methods

A class/static method is similar to a class variable in that it is assigned to a class and not an object of that class. These methods are shared by all instances of that class. A class method can only access the class variables and other class methods of its class. A static method is declared as any other method of the class except that its header is preceded by keyword static. The general form of a class method is

```
static returntype class_method_name(parameter_list)
{
        Method_body
}
```

To invoke static method from other class, the following method can be used
class_name.class_method_name(parameter_list)
(or)
Object_name. class_method_name(parameter_list)

For example

1. static void printtotalmovies()

Declaring, Instantiating and Initializing an Object

```
import java.util.Date;
class DateApp {
   public static void main (String args[]) {
      Date today = new Date();
      System.out.println(today);
   }
}
```

The main() method of the DateApp application creates a Date object named today. This single statement performs three actions: declaration, instantiation, and initialization. Date today declares to the compiler that the name today will be used to refer to an object whose type is Date, the new operator instantiates new Date object, and Date() initializes the object.

Declaring an Object

Declarations can appear as part of object creation as you saw above or can appear alone like this

Date today;

Either way, a declaration takes the form of *type name* where *type* is either a simple data type such as int, float, or boolean, or a complex data type such as a class like the Date class. *name* is the name to be used for the variable. Declarations simply notify the compiler that you will be

using *name* to refer to a variable whose type is *type*. **Declarations do not instantiate objects.** To instantiate a Date object, or any other object, use the newoperator.

Instantiating an Object

The new operator instantiates a new object by allocating memory for it. new requires a single argument: a *constructor method* for the object to be created. The constructor method is responsible for initializing the new object.

Initializing an Object

Classes provide constructor methods to initialize a new object of that type. In a class declaration, constructors can be distinguished from other methods because they have the same name as the class and have no return type. For example, the method signature for Date constructor used by the DateApp application is

Date()

A constructor such as the one shown, that takes no arguments, is known as the *default constructor*. Like Date, most classes have at least one constructor, the default constructor. However, classes can have multiple constructors, all with the same name but with a different number or type of arguments. For example, the Date class supports a constructor that requires three integers:

Date(int year, int month, int day)

that initializes the new Date to the year, month and day specified by the three parameters.

Complete Java Program

```
class Rectangle {
  double length;
  double breadth;
}

// This class declares an object of type Rectangle.
class RectangleDemo {
  public static void main(String args[]) {
    Rectangle myrect = new Rectangle();
    double area;

    // assign values to myrect's instance variables
    myrect.length = 10;
    myrect.breadth = 20;

    // Compute Area of Rectangle
    area =  myrect.length * myrect.breadth ;

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

```
  }
}
```

Garbage Collection

Garbage collection is the process that handles memory deallocation. It is incharge of cleaning the memory space allocated to the objects that are not in use.

When an object is created, memory space is allocated for the object. When there are no more references to that object , it is marked for garbage collection.

While a constructor method initializes an object, finalize() method can be created to optimize the disposing of an object.

Method Overloading

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter list

For example

```
class MyClass {
  int height;
  MyClass() {
    System.out.println("bricks");
    height = 0;
  }
  MyClass(int i) {
    System.out.println("Building new House that is "
    + i + " feet tall");
    height = i;
  }
  void info() {
    System.out.println("House is " + height
    + " feet tall");
  }
  void info(String s) {
    System.out.println(s + ": House is "
    + height + " feet tall");
  }
}
public class MainClass {
```

```
  public static void main(String[] args) {
    MyClass t = new MyClass(0);
    t.info();
    t.info("overloaded method");
    //Overloaded constructor:
    new MyClass();
  }
}
```

Constructor Overloading

Like other methods in java constructor can be overloaded i.e. we can create as many constructors in our class as desired. Number of constructors depends on the information about attributes of an object we have while creating objects

For example

```
class Language {
  String name;

  Language() {
    System.out.println("Constructor method called.");
  }

  Language(String t) {
    name = t;
  }

  public static void main(String[] args) {
    Language cpp  = new Language();
    Language java = new Language("Java");

    cpp.setName("C++");

    java.getName();
    cpp.getName();
  }

  void setName(String t) {
    name = t;
  }
```

```
  void getName() {
    System.out.println("Language name: " + name);
  }
}
```

this Reference

Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

Using this with a Field

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

For example, the Point class was written like this

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

but it could have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor x is a local copy of the constructor's first argument. To refer to the Point field x, the constructor must use this.x.

Using this with a Constructor

From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*.

```
public class Rectangle {
   private int x, y;
   private int width, height;

   public Rectangle() {
      this(0, 0, 1, 1);
   }
   public Rectangle(int width, int height) {
      this(0, 0, width, height);
   }
   public Rectangle(int x, int y, int width, int height) {
      this.x = x;
      this.y = y;
      this.width = width;
      this.height = height;
   }
   ...
}
```

Inheritance

Inheritance is one of the key features of object oriented programming. Inheritance provided a mechanism that allowed a class to inherit property of another class. When a class extends another class it inherits all non private members including fields and methods. Inheritance in java can be best understood in terms of parent and child relationship, also known as super class(parent) and sub class(child).

extends and implements keywords are used in inheritance in java.

Purpose of Inheritance

1. To promote code reuse
2. To use polymorphism

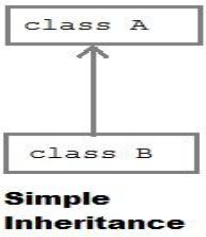
For example

```
class Box {

        double width;
        double height;
        double depth;
        Box() {
        }
```

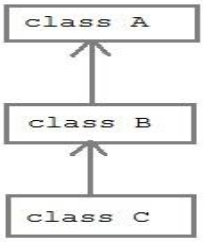
```
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        void getVolume() {
                System.out.println("Volume is : " + width * height * depth);
        }
}

public class MatchBox extends Box {

        double weight;
        MatchBox() {
        }
        MatchBox(double w, double h, double d, double m) {
                super(w, h, d);
                weight = m;
        }
        public static void main(String args[]) {
                MatchBox mb1 = new MatchBox(10, 10, 10, 10);
                mb1.getVolume();
                System.out.println("width of MatchBox 1 is " + mb1.width);
                System.out.println("height of MatchBox 1 is " + mb1.height);
                System.out.println("depth of MatchBox 1 is " + mb1.depth);
                System.out.println("weight of MatchBox 1 is " + mb1.weight);
        }
}
```

Types of Inheritance

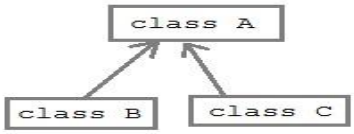
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance



Simple Inheritance



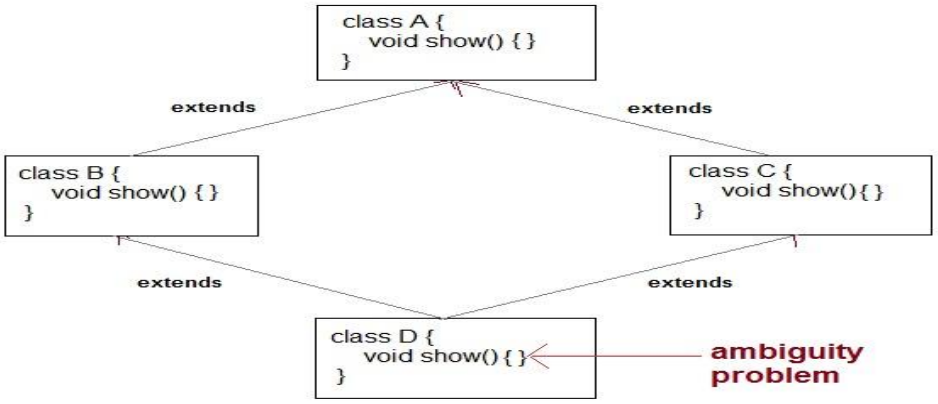
Multilevel inheritance



Heirarchical inheritance

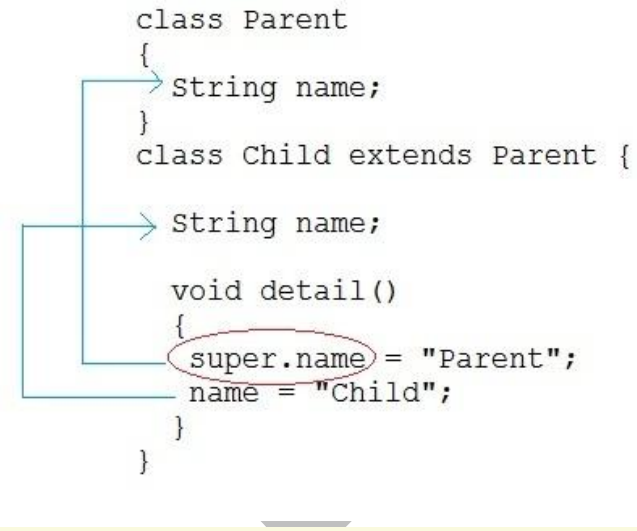
Why Multiple Inheritance isn't supported in Java

1. To remove ambiguity
2. To provide more maintenable and clear design



super keyword

In java, super keyword is used to refer to immediate parent class of a class. In other words, super keyword is used by a subclass whenever it need to refer to its immediate super class




```
class Vehicle {

    // Instance fields
    int noOfTyres; // no of tyres
    private boolean accessories; // check if accessorees present or not
    protected String brand; // Brand of the car
    // Static fields
    private static int counter; // No of Vehicle objects created
```

```
        // Constructor
        Vehicle() {
                System.out.println("Constructor of the Super class called");
                noOfTyres = 5;
                accessories = true;
                brand = "X";
                counter++;
        }
        // Instance methods
        public void switchOn() {
                accessories = true;
        }
        public void switchOff() {
                accessories = false;
        }
        public boolean isPresent() {
                return accessories;
        }
        private void getBrand() {
                System.out.println("Vehicle Brand: " + brand);
        }
        // Static methods
        public static void getNoOfVehicles() {
                System.out.println("Number of Vehicles: " + counter);
        }
}

class Car extends Vehicle {

        private int carNo = 10;
        public void printCarInfo() {
                System.out.println("Car number: " + carNo);
                System.out.println("No of Tyres: " + noOfTyres); // Inherited.
                //  System.out.println("accessories: "    + accessories); // Not Inherited.
                System.out.println("accessories: " + isPresent()); // Inherited.
                //       System.out.println("Brand: "     + getBrand());  // Not Inherited.
                System.out.println("Brand: " + brand); // Inherited.
                //  System.out.println("Counter: "    + counter);     // Not Inherited.
                getNoOfVehicles(); // Inherited.
        }
```

```
}

public class VehicleDetails { // (3)

        public static void main(String[] args) {
                new Car().printCarInfo();
        }
}
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
```

```
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

```
k: 3
```

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the versio n of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden function, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**. Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded – not
// overridden.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
```

```
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// overload show()
void show(String msg) {
System.out.println(msg + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show() in B
subOb.show(); // this calls show() in A
}
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

Final Keyword

The **final keyword** in java is used to restrict the user. The final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1) final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike{
  final int speedlimit=90;//final variable
  void run(){
   speedlimit=400;
  }
  public static void main(String args[]){
  Bike obj=new  Bike();
  obj.run();
  }
}//end of class
```

Output:Compile Time Error

---

2) final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda honda= new Honda();
  honda.run();
  }
}
```

Output:Compile Time Error

---

3) final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}
class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
```

```
    public static void main(String args[]){
      Honda honda= new Honda();
      honda.run();
      }
    }
```

Compile Error

Abstract Classes and Interfaces

Abstract Class in java

Java Abstract classes are used to declare common characteristics of subclasses. An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class. Abstract classes are declared with the abstract keyword. Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.

Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform. An abstract class can include methods that contain no implementation. These are called abstract methods. The abstract method declaration must then end with a semicolon rather than a block. If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract. Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method.

A class abstract Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not be meaningful.

```
abstract class Vehicle {

        int numofGears;
        String color;
        abstract boolean hasDiskBrake();
        abstract int getNoofGears();
}
```

Example of a shape class as an abstract class

```
abstract class Shape {
```

```
        public String color;
        public Shape() {
        }
        public void setColor(String c) {
                color = c;
        }
        public String getColor() {
                return color;
        }
        abstract public double area();
}
```

We can also implement the generic shapes class as an abstract class so that we can draw lines, circles, triangles etc. All shapes have some common fields and methods, but each can, of course, add more fields and methods. The abstract class guarantees that each shape will have the same set of basic properties. We declare this class abstract because there is no such thing as a generic shape. There can only be concrete shapes such as squares, circles, triangles etc.

```
public class Point extends Shape {

        static int x, y;
        public Point() {
                x = 0;
                y = 0;
        }
        public double area() {
                return 0;
        }
        public double perimeter() {
                return 0;
        }
        public static void print() {
                System.out.println("point: " + x + "," + y);
        }
        public static void main(String args[]) {
                Point p = new Point();
                p.print();
        }
}
```

Output

point: 0, 0

Notice that, in order to create a Point object, its class cannot be abstract. This means that all of the abstract methods of the Shape class must be implemented by the Point class.

The subclass must define an implementation for every abstract method of the abstract superclass, or the subclass itself will also be abstract. Similarly other shape objects can be created using the generic Shape Abstract class.
A big Disadvantage of using abstract classes is not able to use multiple inheritance. In the sense, when a class extends an abstract class, it can't extend any other class.

Java Interface

In Java, this multiple inheritance problem is solved with a powerful construct called **interfaces**. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final). Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated.

Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.

If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.
Example 1: Below is an example of a Shape interface

```
interface Shape {

        public double area();
        public double volume();
}
```

Below is a Point class that implements the Shape interface.

```
public class Point implements Shape {

        static int x, y;
        public Point() {
                x = 0;
                y = 0;
```

```
        }
        public double area() {
                return 0;
        }
        public double volume() {
                return 0;
        }
        public static void print() {
                System.out.println("point: " + x + "," + y);
        }
        public static void main(String args[]) {
                Point p = new Point();
                p.print();
        }
}
```

Similarly, other shape objects can be created by interface programming by implementing generic Shape Interface.

Example 2: Below is a java interfaces program showing the power of interface programming in java
Listing below shows 2 interfaces and 4 classes one being an abstract class.

Note: The method *toString* in class *A1* is an overridden version of the method defined in the class named **Object**. The classes *B1* and *C1* satisfy the interface contract. But since the class **D1** does not define all the methods of the implemented interface *I2*, the class D1 is declared abstract. Also,i1.methodI2() produces a compilation error as the method is not declared in *I1* or any of its super interfaces if present. Hence a downcast of interface reference I1 solves the problem as shown in the program. The same problem applies to i1.methodA1(), which is again resolved by a downcast.

When we invoke the toString() method which is a method of an Object, there does not seem to be any problem as every interface or class extends Object and any class can override the default toString() to suit your application needs. ((C1)o1).methodI1() compiles successfully, but produces a ClassCastException at runtime. This is because B1 does not have any relationship with C1 except they are "siblings". You can't cast siblings into one another.
When a given interface method is invoked on a given reference, the behavior that results will be appropriate to the class from which that particular object was instantiated. This is runtime polymorphism based on interfaces and overridden methods.

```
interface I1 {
```

```
	void methodI1(); // public static by default
}

interface I2 extends I1 {

	void methodI2(); // public static by default
}

class A1 {

	public String methodA1() {
		String strA1 = "I am in methodC1 of class A1";
		return strA1;
	}
	public String toString() {
		return "toString() method of class A1";
	}
}

class B1 extends A1 implements I2 {

	public void methodI1() {
		System.out.println("I am in methodI1 of class B1");
	}
	public void methodI2() {
		System.out.println("I am in methodI2 of class B1");
	}
}

class C1 implements I2 {

	public void methodI1() {
		System.out.println("I am in methodI1 of class C1");
	}
	public void methodI2() {
		System.out.println("I am in methodI2 of class C1");
	}
}
```

```
// Note that the class is declared as abstract as it does not
// satisfy the interface contract
abstract class D1 implements I2 {

	public void methodI1() {
	}
	// This class does not implement methodI2() hence declared abstract.
}

public class InterFaceEx {

	public static void main(String[] args) {
		I1 i1 = new B1();
		i1.methodI1(); // OK as methodI1 is present in B1
		// i1.methodI2(); Compilation error as methodI2 not present in I1
		// Casting to convert the type of the reference from type I1 to type I2
		((I2) i1).methodI2();
		I2 i2 = new B1();
		i2.methodI1(); // OK
		i2.methodI2(); // OK
		// Does not Compile as methodA1() not present in interface reference I1
		// String var = i1.methodA1();
		// Hence I1 requires a cast to invoke methodA1
		String var2 = ((A1) i1).methodA1();
		System.out.println("var2 : " + var2);
		String var3 = ((B1) i1).methodA1();
		System.out.println("var3 : " + var3);
		String var4 = i1.toString();
		System.out.println("var4 : " + var4);
		String var5 = i2.toString();
		System.out.println("var5 : " + var5);
		I1 i3 = new C1();
		String var6 = i3.toString();
		System.out.println("var6 : " + var6); // It prints the Object toString() method
		Object o1 = new B1();
		// o1.methodI1(); does not compile as Object class does not define
		// methodI1()
		// To solve the probelm we need to downcast o1 reference. We can do it
		// in the following 4 ways
		((I1) o1).methodI1(); // 1
```

```
                ((I2) o1).methodI1(); // 2
                ((B1) o1).methodI1(); // 3
                /*
                 *
                 * B1 does not have any relationship with C1 except they are "siblings".
                 *
                 * Well, you can't cast siblings into one another.
                 *
                 */
                // ((C1)o1).methodI1(); Produces a ClassCastException
        }
}
```

Output

```
I am in methodI1 of class B1
I am in methodI2 of class B1
I am in methodI1 of class B1
I am in methodI2 of class B1
var2 : I am in methodC1 of class A1
var3 : I am in methodC1 of class A1
var4 : toString() method of class A1
var5 : toString() method of class A1
var6 : C1@190d11
I am in methodI1 of class B1
I am in methodI1 of class B1
I am in methodI1 of class B1
```

Possible Questions

Part - B (2 Marks)

1. Define Instance variable.
2. What is Instance method.
3. What is Class variable.
4. What is Class method

Part - C (6 Marks)

1. Define Constructor and explain it with an example.
2. Write a note on how to declare Object and accessing members of a Class.
3. Describe about Garbage collection and This Reference
4. Explain with an example program Method Overloading and Method Overriding.
5. Write note on Constructor Overloading.
6. Explain Inheritance and its types with a sample program.
7. Explain Abstract classes with an example.
8. What is meant by interface and Explain how to implement interface a class with a sample program.

Questions	opt1	opt2	opt3	opt4	answer
It takes no parameters	Default Constructors	Copy Constructors	Parameter Constructor	Function	Default Constructors
It is required when objects are required to perform a similar task	Method Overriding	Polymorphism	Static Binding	Method Overloading	Method Overloading
It is used to refer to the current object	this reference	that reference	dot	Arrow	this reference
The data or variables,defined within a class are called	Variables	Class variables	Data variables	Instance Variables	Instance Variables
Class is a _____Construct	Hierarchical	Logical	Physical	Hybrid	Logical
To access instance variables of an object_____operator is used	Dot Operator	Logical operator	Relational Operator	Boolean Operator	Dot Operator
Variables declared as static are_____variables	Member variables	Instance	Global	Local	Global
It is used to initialize the member variables when we create an object	Constructors	destructors	Overloading	Overriding	Constructors
What is the printout of the following code:	x is 5 and y is 6	x is 6.0 and y is 6.0	x is 6 and y is 6	x is 5.5 and y is 5	x is 5.5 and y is 5
A_____ variable is known only in the method that declares the variable.	Local	Global	Static	Auto	Local
To declare a constant MAX_LENGTH inside a method with value 99.98, you	final MAX_LENGTH	final float MAX_LENGTH	double MAX_LENGTH	final double MAX_LENGTH	final double MAX_LENGTH
Which of the following is a constant, according to Java naming conventions?	MAX_VALUE	Test	read	ReadInt	MAX_VALUE
The _____ method parses a string s to a double value.	double.ParseDou	Double.parsedou	double.ParseDou	Double.parseDou	Double.parseDou
The _____ method returns a raised to the power of	Math.power(a,b)	Math.exponent(a	Math.pow(a,b)	None of the above	Math.pow(a,b) ;
If a program compiles fine, but it produces incorrect result, then the	compilation error	runtime error	logic error	Syntax error	logic error
Analyze the following code: boolean even = false; if (even = true) {	The program has a syntax error.	The program has a runtime error.	The program runs fine, but	The program runs fine and	The program runs fine and
The number used to refer to a particular element of an array is called the	Pointer	Index	0	1	Index
_____ is an object that contains elements of same data type.	Array	Structure	Class	Object	Array
What is the representation of the third element in an array called a?	a[2]	a(2)	a[3]	a(3)	a[2]
Which of the following is correct?	int[] a = new int[2];	int a[] = new int[2];	int[] a = new int(2);	int a() = new int[2];	int[] a = new int[2];
_____ is a keyword	import	loop	export	package	import
It is used to refer to the current object	this reference	that reference	dot	Arrow	this reference
Code Reusability is characterized by	baseclass	Subclass	Derived class	Inheritance	Inheritance
We can use the ____keyword from any method or constructor to refer to the	this	try	new	throw	this

_____ is used to extend a class by creating a new class	constructors	method overloading	inheritance	overriding	inheritance
When you extends a class, you can change the behavior of a method in the	method overriding.	object refernce	method overloading	polymorphism	method overriding.
The _____ operator creates a single instances of a named class and returns a	dot	new	super	this	new
_____ initializes an object	overloading	constructors	overriding	destructor	constructors
To add a finalizer to a class, you simply define the _____ method	finalize( )	stop()	exit()	break()	finalize( )
the new operator dynamically _____memory for an object.	free	allocates	delete	new	allocates
Java supports a concept called _____ which is just opposite to initialization.	free	finalization	delete	new	finalization
A class that cannot be subclassed is called as _____ class.	abstract	final	static	methods	final
_____ enables an object to initialize itself when it is created	Destructor	constructor	overloading	overriding	constructor
Subclass constructors can call superclass constructors via the _____ keyword	final	protected	inherit	super	super
The _____ is special because its name is the same as the class name.	Destructor	static	constructor	free	constructor
A constructor that accepts no parameters is called the _____ constructor	Copy	default	multiple	multilevel	default
Constructors are invoked automatically when the _____ are created	Data	classes	objects	methods	objects
Constructors cannot be _____	Inherited	destroyed	both Inherited and destroyed	constructor	Inherited
The constructors that can take arguments are called _____ constructors	Copy	multiple	parameterized	destructor	parameterized
static methods will not refer the _____	this	dot	new	public	this
a _____ statement causes control to be transferred directly to the conditional	continue	return	jump	goto	continue
a method in a subclass has the same name and type signature as a method in its	override	overload	function	final	override
_____ dispatch is the mechanism by which a call to an overridden method is	Static method	Dynamic method	overload	finalized	Dynamic method
Once you have an object, you can call its methods and access its fields, by using	object reference	class	variables	data types	object reference
Which of these keywords is used to define interfaces in Java?	interface	Interface	intf	Intf	interface
Which of these can be used to fully abstract a class from its implementation?	Objects	Packages	Interfaces	class	Interfaces
Which of these access specifiers can be used for an interface?	Public	Protected	private	All	Public
Which of these keywords is used by a class to use an interface defined	import	Import	implements	Implements	implements
Which of the following is correct way of implementing an interface salary by class	class manager extends salary {}	class manager implements	class manager imports salary {}	manager extends salary{}	class manager implements

In Java, declaring a class abstract is useful	To prevent developers from	When it doesn't make sense to	When default implementations	To force developers to	When it doesn't make sense to
Runnable is a _____ .	class	abstract class	interface	vaiable	interface
Command used to execute java program is _____	javac	java	run	execute	javac
The java compiler _____	creates executable	translates java code into	creates classes	produces java interpreter	translates java code into
Java uses _____ to represent characters	ASCII code	unicode	byte code	bitcode	unicode
Which is not supported in java?	abstraction	polymorphism	encapsulation	global variables	global variables
Java programs are _____	platform-dependent	interpreter-dependent	platform-independent	interpreter-independent	platform-independent
The order of the three top level elements of the java source file are _____	import,package,class	class,import,package	package,import,class	random order	package,import,class
What is byte code in the context of Java?	The type of code generated by a	It is the code written within	The type of code generated by a	It is another name for a Java	The type of code generated by a
You read the following statement in a Java program that compiles and executes.	depth must be an int	dive must be a method	dive must be the name of an	submarine must be the name of a	dive must be a method
What is garbage collection in the context of Java?	The operating system	Any package imported in a	When all references to an	The JVM checks the output of any	When all references to an

KARPAGAM ACADEMY OF HIGHER EDUCATION
CLASS: I B.Sc IT COURSE NAME: Programming in Java
COURSE CODE: 19ITU201 UNIT: III (Exception Handling) BATCH-2019-2022
SYLLABUS

Fundamentals – Hierarchy of Classes – Types of Exceptions-Exception Class – Uncaught Exceptions – Handling Exceptions – User Defined Exceptions. Multithreaded Programming: The Java Thread Model – Runnable Interface - Thread Class – Thread Creation – Thread's Life Cycle – Thread Scheduling -Synchronization and Deadlock. Packages and Access Modifiers: Package Declaration – The CLASSPATH variable - import statement – The Java Language Packages - Access Protection.

Fundamentals

- An *exception* is an abnormal condition that arises in a code sequence at run time
- A Java exception is an object that describes an exceptional 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
- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code
- Java exception handling is managed by via five keywords: **try, catch, throw, throws,** and **finally**
- Program statements to monitor are contained within a **try** block
- If an exception occurs within the **try** block, it is thrown
- Code within **catch** block catch the exception and handle it
- System generated exceptions are automatically thrown by the Java run-time 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
- 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

}
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Throwable has two subclasses, they are
  - Exception (to handle exceptional conditions that user programs should catch)
    - An important subclass of Exception is **RuntimeException**, that includes division by zero and invalid array indexing
  - Error (to handle exceptional conditions that are not expected to be caught under normal circumstances). i.e. stack overflow

Uncaught Exceptions

- If an exception is not caught by user program, then execution of the program stops and it is caught by the default handler provided by the Java run-time system
- Default handler prints a stack trace from the point at which the exception occurred, and terminates the program

Ex:

```
class Exc0 {

  public static void main(String args[]) {

    int d = 0;

    int a = 42 / d;

  }

}
```

Output:

java.lang.ArithmeticException: / by zero

at Exc0.main(Exc0.java:4)

Exception in thread "main"

Using try and catch

- Handling an exception has two benefits,
  - It allows you to fix the error
  - It prevents the program from automatically terminating
- The **catch** clause should follow immediately the **try** block
- Once an exception is thrown, program control transfer out of the **try** block into the catch block
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism

Example

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

Output:

Division by zero.

After catch statement.

Using try and catch

- A **try** and **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement
- we cannot use **try** on a single statement

```
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<10; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

Multiple catch Clauses

- If more than one can occur, then we use multiple catch clauses
- 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

EXAMPLE:

```
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.");
  }
}
```

Example

If no command line argument is provided, then output will be:

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks

- If any command line argument is provided, then we will see the following output:

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException

After try/catch blocks.

Caution

- Remember that, exception subclass must come before any of of their superclasses
- Because, a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass
- For example, ArithmeticException is a subclass of Exception
- Moreover, unreachable code in Java generates error

Example

```
/*  This program contains an error.

    A subclass must come before its superclass in
    a series of catch statements. If not,
    unreachable code will be created and a
    compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Nested try Statements

- A **try** statement can be inside the block of another try
- Each time a **try** statement is entered, the context of that exception is pushed on the stack

- If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match
- If a method call within a **try** block has **try** block within it, then then it is still nested **try**

Example

```
// An example nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command line arg is used,
           then an divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command line args are used
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

Output

- When no parameter is given:

Divide by 0: java.lang.ArithmeticException: / by zero

- When one parameter is given

  a = 1

  Divide by 0: java.lang.ArithmeticException: / by zero

- When two parameters are given

a = 2

Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException

throw

- It is possible for your program to to throw an exception explicitly
  throw *TrrowableInstance*
- Here, *TrrowableInstance* must be an object of type **Throwable** or a subclass **Throwable**
- There are two ways to obtain a **Throwable** objects:
  - Using a parameter into a catch clause
  - Creating one with the **new** operator

Example

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // re-throw the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

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

Throws

- 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

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

- It is not applicable for **Error** or **RuntimeException,** or any of their subclasses

Example: incorrect program

```
// 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();
  }
}
```

Example: corrected version

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:
Inside throwOne.
Caught java.lang.IllegalAccessException: demo

finally

- It is used to handle premature execution of a method (i.e. a method open a file upon entry and closes it upon exit)
- **finally** creates a block of code that will be executed after **try/catch** block has completed and before the code following the **try/catch** block
- **finally** clause will execute whether or not an exception is thrown

Example

```
// Demonstrate finally.
class FinallyDemo {
  // Through an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }

  // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
  }

  public static void main(String args[]) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

Output
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
Java’s Built-in Errors

- class java.lang.**Throwable** (implements java.io.Serializable)
  - class java.lang.**Error**
    - class java.lang.**LinkageError**
      - class java.lang.**ClassCircularityError**
      - class java.lang.**ClassFormatError**
        - class java.lang.**UnsupportedClassVersionError**
      - class java.lang.**ExceptionInInitializerError**
      - class java.lang.**IncompatibleClassChangeError**
        - class java.lang.**AbstractMethodError**
        - class java.lang.**IllegalAccessError**
        - class java.lang.**InstantiationError**
        - class java.lang.**NoSuchFieldError**
        - class java.lang.**NoSuchMethodError**
      - class java.lang.**NoClassDefFoundError**
      - class java.lang.**UnsatisfiedLinkError**
      - class java.lang.**VerifyError**
    - class java.lang.**ThreadDeath**
    - class java.lang.**VirtualMachineError**
      - class java.lang.**InternalError**
      - class java.lang.**OutOfMemoryError**
      - class java.lang.**StackOverflowError**
      - class java.lang.**UnknownError**

Java's Built-in Exceptions:

- class java.lang.**Exception**
  - class java.lang.**ClassNotFoundException**
  - class java.lang.**CloneNotSupportedException**
  - class java.lang.**IllegalAccessException**
  - class java.lang.**InstantiationException**
  - class java.lang.**InterruptedException**
  - class java.lang.**NoSuchFieldException**
  - class java.lang.**NoSuchMethodException**
  - class java.lang.**RuntimeException**
    - class java.lang.**ArithmeticException**
    - class java.lang.**ArrayStoreException**
    - class java.lang.**ClassCastException**
    - class java.lang.**IllegalArgumentException**
      - class java.lang.**IllegalThreadStateException**
      - class java.lang.**NumberFormatException**
    - class java.lang.**IllegalMonitorStateException**
    - class java.lang.**IllegalStateException**
    - class java.lang.**IndexOutOfBoundsException**
      - class java.lang.**ArrayIndexOutOfBoundsException**
      - class java.lang.**StringIndexOutOfBoundsException**
    - class java.lang.**NegativeArraySizeException**
    - class java.lang.**NullPointerException**
    - class java.lang.**SecurityException**
    - class java.lang.**UnsupportedOperationException**

MULTITHREADING

Introduction

Java environment has been built around the multithreading model. In fact all Java class libraries have been designed keeping multithreading in mind. If a thread goes off to sleep for some time, the rest of the program does not get affected by this. Similarly, an animation loop can be fired that will not stop the working of rest of the system.

At a point of time a thread can be in any one of the following states – new, ready, running, inactive and finished. A thread enters the new state as soon as it is created. When it is started (by invoking start() method), it is ready to run. The start() method in turn calls the run() method which makes the thread enter the running state. While running, a thread might get blocked because some resource that it requires is not available, or it could be suspended on purpose for some reason (like put off to sleep by the programmer). In such a case the thread enters the state of being inactive. A thread can also be stopped purposely because its time has expired, then it enters the state of ready to run once again.

A thread that is in running state can be stopped once its job has finished. A thread that is ready to run can also be stopped. A thread that is stopped enters the finished state. A thread that is in inactive state can either be resumed, in which case it enters the ready state again, or it can be stopped in which case it enters the finished state.

Thread Priorities

In multithreading environment, one thread might require the attention of the CPU more quickly than other. In such a case that thread is said to be of high priority. Priority of a thread determines the switching from one thread to another. In other words, priority determines how a thread should behave with respect to the other threads.

The word priority should not be confused with the faster running of a thread. A high priority thread does not run any faster than the low priority thread. A thread can voluntarily leave the control by explicitly stopping, sleeping or blocking on pending I/O or it can pre-empted by the system to do so. In the first case the processor examines all threads and assigns the control to the thread having the highest priority. In the second case, a low priority thread that is not ready to leave the control is simply pre-empted by the higher priority thread no matter what it is doing. This is known as pre-emptive multi-tasking. It is advisable that in case two threads have the same priority, they must explicitly surrender the control to their peers.

Synchronization

Multithreading produces asynchronous behavior among the programs. This means that all threads run as independent units without affecting each other. But sometimes it becomes necessary to synchronize these threads. For example, synchronization must be provided when two threads share the same variable or data structure like an array. In such a case there must be way by which they should not come in each other's way.

Messaging

Since there are more than one thread in a multithreaded environment, inter-process communication becomes imperative. A thread should be able to communicate with the other threads. Threads can talk to each other by using method such as wait().

Java - Multithreading

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

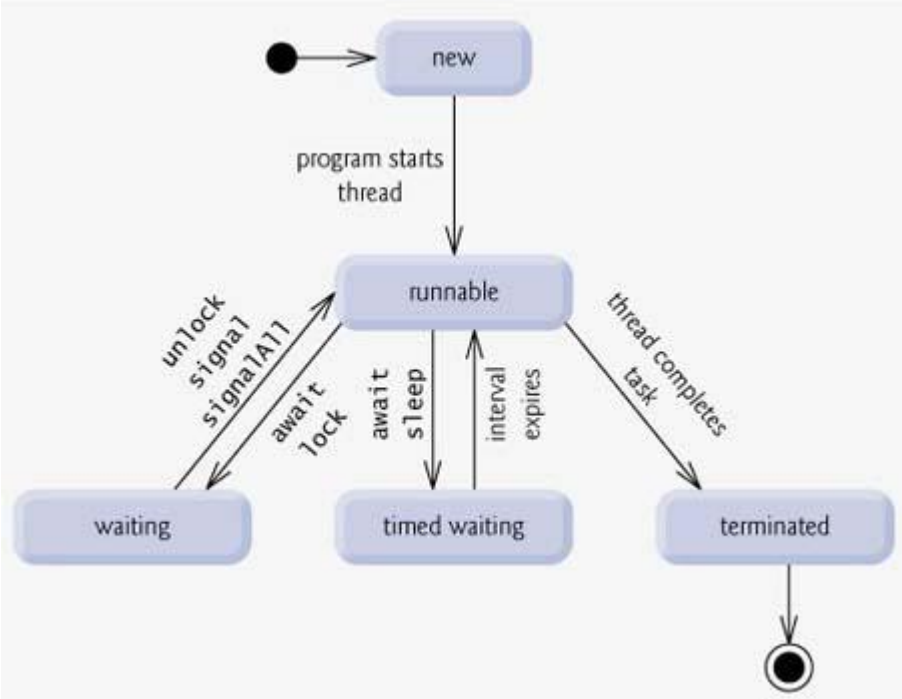
A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

I need to define another term related to threads: process: A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

Multithreading enables to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above mentioned stages are explained here:

New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
Waiting: Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Creating a Thread

Java defines two ways in which this can be accomplished:

Implement the Runnable interface.
Extend the Thread class, itself.

Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the Runnable interface.

To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

```
public void run( )
```

we will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread. The start( ) method is shown here:

```
void start( );
```

Example:

Here is an example that creates a new thread and starts it running:

```
// Create a new thread.
class NewThread implements Runnable {
  Thread t;
  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        // Let the thread sleep for a while.
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
   System.out.println("Exiting child thread.");
  }
}

class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // create a new thread
    try {
      for(int i = 5; i > 0; i--) {
       System.out.println("Main Thread: " + i);
       Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

This would produce following result:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
```

```
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Create Thread by Extending Thread

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

Example:

Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
                                // Let the thread sleep for a while.
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
```

```
}

class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

This would produce following result:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Thread Methods

Following is the list of important medthods available in the Thread class.

SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.

3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread

SN	Methods with Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}

// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
         guess = (int) (Math.random() * 100 + 1);
         System.out.println(this.getName()
                  + " guesses " + guess);
         counter++;
      }while(guess != number);
      System.out.println("** Correct! " + this.getName()
                  + " in " + counter + " guesses.**");
   }
}
```

```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
  public static void main(String [] args)
  {
    Runnable hello = new DisplayMessage("Hello");
    Thread thread1 = new Thread(hello);
    thread1.setDaemon(true);
    thread1.setName("hello");
    System.out.println("Starting hello thread...");
    thread1.start();

    Runnable bye = new DisplayMessage("Goodbye");
    Thread thread2 = new Thread(hello);
    thread2.setPriority(Thread.MIN_PRIORITY);
    thread2.setDaemon(true);
    System.out.println("Starting goodbye thread...");
    thread2.start();

    System.out.println("Starting thread3...");
    Thread thread3 = new GuessANumber(27);
    thread3.start();
    try
    {
      thread3.join();
    }catch(InterruptedException e)
    {
      System.out.println("Thread interrupted.");
    }
    System.out.println("Starting thread4...");
    Thread thread4 = new GuessANumber(75);

          thread4.start();
    System.out.println("main() is ending...");
  }
}
```

This would produce following result. we can try this example again and again and we would get different result every time.

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
```

```
Hello
Hello
Hello
Hello
Hello
Thread-2 guesses 27
Hello
** Correct! Thread-2 in 102 guesses.**
Hello
Starting thread4...
Hello
Hello
.....remaining result produced
```

Java Thread Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

The process by which this synchronization is achieved is called *thread synchronization*.

The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

This is the general form of the synchronized statement:

```
synchronized(object) {
  // statements to be synchronized
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Here is an example, using a synchronized block within the run( ) method:

```
// File Name : Callme.java
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
```

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

  // File Name : Caller.java
  class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
      target = targ;
      msg = s;
      t = new Thread(this);
      t.start();
    }

    // synchronize calls to call()
    public void run() {
      synchronized(target) { // synchronized block
        target.call(msg);
      }
    }
  }
  // File Name : Synch.java
  class Synch {
    public static void main(String args[]) {
      Callme target = new Callme();
      Caller ob1 = new Caller(target, "Hello");
      Caller ob2 = new Caller(target, "Synchronized");
      Caller ob3 = new Caller(target, "World");

      // wait for threads to end
      try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
      } catch(InterruptedException e) {
        System.out.println("Interrupted");
      }
    }
  }
```

This would produce following result:

```
[Hello]
[World]
[Synchronized]
```

Java - Thread Deadlock

A special type of error that we need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Example:

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, A and B, with methods foo( ) and bar( ), respectively, which pause briefly before trying to call a method in the other class.

The main class, named Deadlock, creates an A and a B instance, and then starts a second thread to set up the deadlock condition. The foo( ) and bar( ) methods use sleep( ) as a way to force the deadlock condition to occur.

```
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entered A.foo");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrupted");
    }
    System.out.println(name + " trying to call B.last()");
    b.last();
  }
  synchronized void last() {
    System.out.println("Inside A.last");
  }
}
class B {
  synchronized void bar(A a) {
```

```
      String name = Thread.currentThread().getName();
      System.out.println(name + " entered B.bar");
      try {
        Thread.sleep(1000);
      } catch(Exception e) {
        System.out.println("B Interrupted");
      }
      System.out.println(name + " trying to call A.last()");
      a.last();
    }
    synchronized void last() {
      System.out.println("Inside A.last");
    }
  }
  class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Deadlock() {
      Thread.currentThread().setName("MainThread");
      Thread t = new Thread(this, "RacingThread");
      t.start();
      a.foo(b); // get lock on a in this thread.
      System.out.println("Back in main thread");
    }
    public void run() {
      b.bar(a); // get lock on b in other thread.
      System.out.println("Back in other thread");
    }
    public static void main(String args[]) {
      new Deadlock();
    }
  }
```

Here is some output from this program:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Deadlock Example:

Following is the depiction of a dead lock:

```
// File Name ThreadSafeBankAccount.java
public class ThreadSafeBankAccount
```

```
{
  private double balance;
  private int number;
  public ThreadSafeBankAccount(int num, double initialBalance)
  {
    balance = initialBalance;
    number = num;
  }
  public int getNumber()
  {
    return number;
  }
  public double getBalance()
  {
    return balance;
  }
  public void deposit(double amount)
  {
    synchronized(this)
    {
     double prevBalance = balance;
     try
     {
       Thread.sleep(4000);
     }catch(InterruptedException e)
     {}
     balance = prevBalance + amount;
    }
  }
  public void withdraw(double amount)
  {
    synchronized(this)
    {
              double prevBalance = balance;
      try
      {
        Thread.sleep(4000);
      }catch(InterruptedException e)
      {}
      balance = prevBalance - amount;
    }
  }
}

// File Name LazyTeller.java
public class LazyTeller extends Thread
```

```
{
  private ThreadSafeBankAccount source, dest;
  public LazyTeller(ThreadSafeBankAccount a,
                    ThreadSafeBankAccount b)
  {
    source = a;
    dest = b;
  }
  public void run()
  {
    transfer(250.00);
  }
  public void transfer(double amount)
  {
    System.out.println("Transferring from "
       + source.getNumber() + " to " + dest.getNumber());
    synchronized(source)
    {
       Thread.yield();
       synchronized(dest)
       {
         System.out.println("Withdrawing from "
               + source.getNumber());
         source.withdraw(amount);
         System.out.println("Depositing into "
               + dest.getNumber());
         dest.deposit(amount);
       }
     }
  }
}
public class DeadlockDemo
{
  public static void main(String [] args)
  {
    System.out.println("Creating two bank accounts...");
    ThreadSafeBankAccount checking =
              new ThreadSafeBankAccount(101, 1000.00);
    ThreadSafeBankAccount savings =
              new ThreadSafeBankAccount(102, 5000.00);

    System.out.println("Creating two teller threads...");
    Thread teller1 = new LazyTeller(checking, savings);
    Thread teller2 = new LazyTeller(savings, checking);
    System.out.println("Starting both threads...");
    teller1.start();
```

```
    teller2.start();
  }
}
```

This would produce following result:

```
Creating two bank accounts...
Creating two teller threads...
Starting both threads...
Transferring from 101 to 102
Transferring from 102 to 101
```

The problem with the LazyTeller class is that it does not consider the possibility of a race condition, a common occurrence in multithreaded programming.

After the two threads are started, teller1 grabs the checking lock and teller2 grabs the savings lock. When teller1 tries to obtain the savings lock, it is not available. Therefore, teller1 blocks until the savings lock becomes available. When the teller1 thread blocks, teller1 still has the checking lock and does not let it go.

Similarly, teller2 is waiting for the checking lock, so teller2 blocks but does not let go of the savings lock. This leads to one result: deadlock!

Deadlock Solution Example:

Here transfer() method, in a class named OrderedTeller, in stead of arbitrarily synchronizing on locks, this transfer() method obtains locks in a specified order based on the number of the bank account.

```
// File Name ThreadSafeBankAccount.java
public class ThreadSafeBankAccount
{
  private double balance;
  private int number;
  public ThreadSafeBankAccount(int num, double initialBalance)
  {
    balance = initialBalance;
    number = num;
  }
  public int getNumber()
  {
    return number;
  }
  public double getBalance()
  {
    return balance;
```

```
    }
    public void deposit(double amount)
    {
      synchronized(this)
      {
       double prevBalance = balance;
       try
       {
         Thread.sleep(4000);
       }catch(InterruptedException e)
       {}
       balance = prevBalance + amount;
      }
    }
    public void withdraw(double amount)
    {
      synchronized(this)
      {
                double prevBalance = balance;
        try
        {
          Thread.sleep(4000);
        }catch(InterruptedException e)
        {}
        balance = prevBalance - amount;
      }
    }
  }
  // File Name OrderedTeller.java
  public class OrderedTeller extends Thread
  {
    private ThreadSafeBankAccount source, dest;
    public OrderedTeller(ThreadSafeBankAccount a,
                         ThreadSafeBankAccount b)
    {
      source = a;
      dest = b;
    }
    public void run()
    {
      transfer(250.00);
    }
    public void transfer(double amount)
    {
       System.out.println("Transferring from " + source.getNumber()
          + " to " + dest.getNumber());
```

```
        ThreadSafeBankAccount first, second;
        if(source.getNumber() < dest.getNumber())
        {
          first = source;
          second = dest;
        }
        else
        {
          first = dest;
          second = source;
        }
        synchronized(first)
        {
          Thread.yield();
          synchronized(second)
          {
            System.out.println("Withdrawing from "
                    + source.getNumber());
            source.withdraw(amount);
            System.out.println("Depositing into "
                    + dest.getNumber());
            dest.deposit(amount);
          }
        }
      }
    }

    // File Name DeadlockDemo.java
    public class DeadlockDemo
    {
      public static void main(String [] args)
      {
        System.out.println("Creating two bank accounts...");
        ThreadSafeBankAccount checking =
                new ThreadSafeBankAccount(101, 1000.00);
        ThreadSafeBankAccount savings =
                new ThreadSafeBankAccount(102, 5000.00);

        System.out.println("Creating two teller threads...");
        Thread teller1 = new OrderedTeller(checking, savings);
        Thread teller2 = new OrderedTeller(savings, checking);
        System.out.println("Starting both threads...");
        teller1.start();
        teller2.start();
      }
    }
```

This would remove deadlock problem and would produce following result:

```
Creating two bank accounts...
Creating two teller threads...
Starting both threads...
Transferring from 101 to 102
Transferring from 102 to 101
Withdrawing from 101
Depositing into 102
Withdrawing from 102
Depositing into 101
```

Interface Runnable

public interface Runnable

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if we are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class

void	run() When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

run

public void run()

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

The general contract of the method run is that it may take any action whatsoever

Starting a Thread Using the Thread Class or the Runnable Interface

We can start a thread in Java by either implementing the java.lang.Runnable interface or by extending the java.lang.Thread class

```
1. public MyClass implements MyInterface{
2. }

3. public MyThread extends Thread implements MyInterface {
4.   MyClass myObject;
5.
6.   // Provide delegation methods for myObject here
7. }
```

the class MyThread has to delegate to a local object myObject in order to reuse the behavior for MyInterface. If MyInterface had 30 methods, we would have to provide wrapper methods for 30 methods in MyThread.

On the other hand, implementing the Runnable interface only requires an implementation for the run() method. Since Runnable is an interface, we can still extend from another class:

```
1. public MyThread extends MyClass implements Runnable {
2. }
```

In general, implementing the Runnable interface presents a more flexible choice. Another thing to keep in mind is that the Thread class provides implementation for several other methods (besides run()) that we may never use. This overhead can also be avoided by using the Runnable interface.

Java's Abstract Windowing Toolkit provides many of the user interface objects we find in the Windows environment. These are called "Components" of the Java AWT. The applet below contains most of the components we will use to create a graphical user interface (GUI) for our applets. It simply initializes and creates the components but does not handle any of the events they trigger.

PACKAGES

Programs are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts. A top level type is accessible outside the package that declares it only if the type is declared public.

The naming structure for packages is hierarchical . The members of a package are class and interface types, which are declared in compilation units of the package, and subpackages, which may contain compilation units and subpackages of their own.

A package can be stored in a file system or in a database. Packages that are stored in a file system have certain constraints on the organization of their compilation units to allow a simple implementation to find classes easily.

A package consists of a number of compilation units. A compilation unit automatically has access to all types declared in its package and also automatically imports all of the public types declared in the predefined package java.lang.

For small programs and casual development, a package can be unnamed or have a simple name, but if code is to be widely distributed, unique package names should be chosen. This can prevent the conflicts that would otherwise occur if two development groups happened to pick the same package name and these packages were later to be used in a single program.

Package Members

The members of a package are subpackages and all the top level class and top level interface types declared in all the compilation units of the package.

For example, in the Java Application Programming Interface:

- The package java has sub-packages awt, applet, io, lang, net, and util, but no compilation units.
- The package java.awt has a subpackage named image, as well as a number of compilation units containing declarations of class and interface types.

If the fully qualified name of a package is P, and Q is a subpackage of P, then P.Q is the fully qualified name of the subpackage.

A package may not contain two members of the same name, or a compile-time error results.

Here are some examples:

- Because the package java.awt has a subpackage image, it cannot (and does not) contain a declaration of a class or interface type named image.
- If there is a package named mouse and a member type Button in that package (which then might be referred to as mouse.Button), then there cannot be any package with the fully qualified name mouse.Button or mouse.Button.Click.
- If com.sun.java.jag is the fully qualified name of a type, then there cannot be any package whose fully qualified name is either com.sun.java.jag or com.sun.java.jag.scrabble.

The hierarchical naming structure for packages is intended to be convenient for organizing related packages in a conventional manner, but has no significance in itself other than the prohibition against a package having a subpackage with the same simple name as a top level type declared in that package. There is no special access relationship between a package named oliver and another package named oliver.twist, or between packages named evelyn.wood and evelyn.waugh. For example, the code in a package

named oliver.twist has no better access to the types declared within package oliver than code in any other package.

Package Declarations

A package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs.

Named Packages

A package declaration in a compilation unit specifies the name of the package to which the compilation unit belongs.

PackageDeclaration:
package PackageName ;

The package name mentioned in a package declaration must be the fully qualified name of the package.

Unnamed Packages

A compilation unit that has no package declaration is part of an unnamed package.

Note that an unnamed package cannot have subpackages, since the syntax of a package declaration always includes a reference to a named top level package.

As an example, the compilation unit:

```
class FirstCall {
        public static void main(String[] args) {
                System.out.println("Mr. Watson, come here. "
                                                + "I want you.");
        }
}
```

defines a very simple compilation unit as part of an unnamed package.

An implementation of the Java platform must support at least one unnamed package; it may support more than one unnamed package but is not required to do so. Which compilation units are in each unnamed package is determined by the host system.

In implementations of the Java platform that use a hierarchical file system for storing packages, one typical strategy is to associate an unnamed package with each directory; only one unnamed package is observable at a time, namely the one that is associated with the "current working directory." The precise meaning of "current working directory" depends on the host system.

Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development.

Observability of a Package

A package is observable if and only if either:

- A compilation unit containing a declaration of the package is observable.
- A subpackage of the package is observable.

One can conclude from the rule above and from the requirements on observable compilation units, that the packages java, java.lang, and java.io are always observable.

Scope of a Package Declaration

The scope of the declaration of an observable top level package is all observable compilation units. The declaration of a package that is not observable is never in scope. Subpackage declarations are never in scope.

It follows that the package java is always in scope.

Package declarations never shadow other declarations.

Import Declarations

An import declaration allows a named type to be referred to by a simple name that consists of a single identifier. Without the use of an appropriate import declaration, the only way to refer to a type declared in another package is to use a fully qualified name.

ImportDeclaration:
SingleTypeImportDeclaration
TypeImportOnDemandDeclaration

A single-type-import declaration imports a single named type, by mentioning its canonical name. A type-import-on-demand declaration imports all the accessible types of a named type or package as needed.

The scope of a type imported by a single-type-import declaration or type-import-on-demand declaration is all the class and interface type declarations in the compilation unit in which the import declaration appears.

An import declaration makes types available by their simple names only within the compilation unit that actually contains the import declaration. The scope of the entities(s) it introduces specifically does not include the package statement, other import declarations in the current compilation unit, or other compilation units in the same package..

Automatic Imports

Each compilation unit automatically imports all of the public type names declared in the predefined package java.lang, as if the declaration:

```
import java.lang.*;
```

appeared at the beginning of each compilation unit, immediately following any package statement.

Importing a Package Member

To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one.

```
import graphics.Rectangle;
```

Now you can refer to the Rectangle class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the graphics package. But if you use many types from a package, you should import the entire package.

Importing an Entire Package

To import all the types contained in a particular package, use the import statement with the asterisk (*) wildcard character.

```
import graphics.*;
```

Now you can refer to any class or interface in the graphics package by its simple name.

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the graphics package that begin with A.

```
import graphics.A*;     //does not work
```

Instead, it generates a compiler error. With the import statement, you generally import only a single package member or an entire package.

Another, less common form of import allows you to import the public nested classes of an enclosing class. For example, if the graphics.Rectangle class contained useful nested classes, such as Rectangle.DoubleWide and Rectangle.Square, you could import Rectangle and its nested classes by using the following two statements.

```
import graphics.Rectangle;
```

```
import graphics.Rectangle.*;
```

Be aware that the second import statement will not import Rectangle.
Another less common form of import, the static import statement, will be discussed at the end of this section.
For convenience, the Java compiler automatically imports three entire packages for each source file: (1) the package with no name, (2) the java.lang package, and (3) the current package (the package for the current file).

The Static Import Statement

There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The static import statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.

The java.lang.Math class defines the PI constant and many static methods, including methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many more. For example,

```
public static final double PI 3.141592653589793
public static double cos(double a)
```

Ordinarily, to use these objects from another class, you prefix the class name, as follows.

```
double r = Math.cos(Math.PI * theta);
```

You can use the static import statement to import the static members of java.lang.Math so that you don't need to prefix the class name, Math. The static members of Math can be imported either individually:

```
import static java.lang.Math.PI;
```

or as a group:

```
import static java.lang.Math.*;
```

Once they have been imported, the static members can be used without qualification. For example, the previous code snippet would become:

```
double r = cos(PI * theta);
```

Obviously, you can write your own classes that contain constants and static methods that you use frequently, and then use the static import statement. For example,

```
import static mypackage.MyConstants.*;
```

Access Protection

Global variables are a classic cause of bugs in most programming languages. Some unknown function can change the value of a variable when the programmer isn't expecting it to change. This plays all sorts of havoc.

Java allows to protect variables from external modification. For example, in the Car class we'd like to make sure that no block of code in some other class is allowed to make the speed greater than the maximum speed. We want a way to make the following illegal:

```
Car c = new Car("New York A234 567", 100.0);
c.speed = 150.0;
```

This code violates the constraints we've placed on the class. We want to allow the compiler to enforce these constraints.

A class presents a picture of itself to the world. (This picture is sometimes called an *interface*, but the word *interface* has a more specific meaning in Java.) This picture says that the class has certain methods and certain fields. Everything else about the class including the detailed workings of the class's methods is hidden. As long as the picture the class shows to the world doesn't change, the programmer can change how the class implements that picture. Among other advantages this allows the programmer to change and improve the algorithms a class uses without worrying that some piece of code depends in unforeseen ways on the details of the algorithm used. This is called encapsulation.

Another way to think about encapsulation is that a class signs a contract with all the other classes in the program. This contract says that a class has methods with unambiguous names which take particular types of arguments and return a particular type of value. The contract may also say that a class has fields with given names and of a given type. However the contract does not specify how the methods are implemented. Furthermore, it does not say that there aren't other private fields and methods which the class may use. A contract guarantees the presence of certain methods and fields. It does not exclude all other methods and fields. This contract is implemented through access protection. Every class, field and method in a Java program is defined as either public, private, protected or unspecified.

Possible Questions
Part – B(2 Marks)

1. Define Exception
2. What is meant by uncaught exception?
3. Define runnable interface
4. List the steps involved in thread’s life cycle

5. Define package

Part – C(6 Marks)

1. Define an exception. Sketch the hierarchy of Exception Class. List out the types of exceptions and explain them.
2. Write the syntax to declare and create a package and explain the compilation and running procedure of a package with example
3. What is synchronization? When do we use it?
4. What is the goal of package designing?
5. What is the function of StackTrace?
6. Explain the "try-catch" construct used to capture and handle exception with an example program
7. Discuss the use of "finally" block in Java program with an example
8. Explain the creation of Thread using i)Thread Class ii) Runnable Interface. Give example for each
9. Name the 5 states of a thread. Explain each of them.
10. Write short notes on a) Thread Life Cycle b) Thread Scheduling
11. Give the steps for creating a thread using "Runnable" interface.

Questions	opt1	opt2	opt3	opt4	answer
_____ is an explicit specification of a set of methods	Interface	package	Statement	None	Interface
_____ are containers for classes that are used to keep the class namespace compartmenterised.	Interface	package	Statement	None	Package
All of the Java “built-in” classes included in the java distribution are stored in a package called _____.	Header	Java	Package	Files	Package
_____ are the means of encapsulation and containing the namespace and scope of variables and	Class	Package	Classes and Package	None	Class and Package
_____ act as containers for classes and other packages.	Container	Classes	Java	Packages	Packages
_____ act as containers for data and code	Container	Classes	Java	Packages	Classes
Java’s _____ are designed to support dynamic method resolution at runtime.	Interface	Class	Package	None	Class
An _____ is a condition that is caused by a runtime error in the program	throw	exception	handle	catch	exception
Exception can be generated by the _____ or manually by the code	Throwable class	Java runtime system	object	catch	Java runtime system
All exception types are subclasses of the built_in class _____	Throwable	RuntimeException	StackTree	LocalizedMessage	Throwable
All exception classes are divided into _____ groups	3	4	2	6	2
The _____ defines the exceptions which are not expected to be caught	java.lang.Error	java.lang.Math	java.lang.Throwable	java.lang.IOException	java.lang.Error
When an exception occurs within a java method, the method creates an exception object and hands it over to the runtime	catching the exception	throwing an exception	handle the exception	get the exception	throwing an exception
When java method throws an exception the java runtime system searches all the methods in the call stack to find one that	catching the exception	throwing an exception	handle the exception	get the exception	catching the exception
Exception performs _____ tasks	3	4	5	2	4
Unchecked exceptions are extensions of _____	throws	catch	RuntimeException	Error	RuntimeException
Checked exceptions are extensions of _____	throws	catch	Exception	Error	Exception
Each of Exception's predefined class provide _____ constructors	3	4	5	2	2
The errors are printed by _____	Stack Trace	StackTree	Message	Error	Stack Trace
AWT includes a very simple plain text,multiline editor called _____	Label	TextField	TextArea	Option.	TextArea
_____ class is a button that is used to toggle the state of a check mark.	Label	Option	CheckBox	Button	CheckBox

_____ class is at the top of the exception class hierarchy.	Exception	Error	Throws	Throwable	Throwable
_____subclass of throwable defines exceptions that are not expected to be caught under normal circumstances	Exception	Error	Throws	Throwable	Error
The _____ class is used for exceptional conditions that the user programs should catch	Exception	Error	Throws	Throwable	Exception
The two subclass of throwable class are _____	Exception and Error	Exception and handler	throw and throwable	try and catch	Exception and Error
The _____ Keyword is used to specify a block of code that should be guarded against all exceptions	Catch	try	exception	block of code	try
_____ specifies the type of exception to be caught.	Catch	try	exception	block of code	Catch
_____ keyword is used to identify the list of possible exceptions that a method might throw	throw	try	catch	Throwable	throw
Certain block of code necessarily has to be run no matter of what exceptions occurs. These codes are identified using	throw	final	finally	try and catch	finally
There are _____ ways of creating Throwable object	3	4	5	2	2
_____ is an important subclass of exception	RuntimeExceptio n	AarithmeticExcep tion	NullException	Subclasses of Throwable	RuntimeExceptio n
What is the mechanisam defind by java for the Resources to be used by only one Thread at a time?	priority	parameters	arguments	Synchronisation	Synchronisation
Garbage collector thread belongs to which priority?	high-priority	low-priority	middle-priority	highest-priority	low-priority
When a Java program starts up, ____ thread begins running immediately	program	main	function	input	main
The ____ method causes the thread from which it is called to suspend execution for the specified period of milliseconds	wait()	notify()	sleep( )	run()	sleep( )
To implement Runnable, a class need only implement a single method called ____	wait()	notify()	sleep( )	run()	run()
A ____ is an object that is used as a mutually exclusive lock to achieve synchronization	monitor	thread	process	applet	monitor
Which of these packages contain classes and interfaces used for input & output operations of a program?	java.util	java.lang	java.io	java.util.date	java.io
A package is a collection of _____	classes	interfaces	editing tools	classes and interfaces	classes and interfaces
For which purpose packages are used in java?	categorizes data	organizing java classes into namespaces	for faster compilation	organize package	organizing java classes into namespaces
In a java program, package declaration _____ import statements	must precede	must succeed	may precede or succeed	prdessor	must precede
_____ package is used by compiler itself. So it does not need to be imported for use.	java.math	java.awt	java.applet	java.lang	java.lang
A class can be converted to a thread by implementing the interface_____	Thread	Runnable	Start	Yield	Runnable

Which of the following classes are not available in the java.lang package?	Stack	Object	Math	String	Stack
A thread can make a second thread ineligible for execution by calling the _____ method on the second thread.	second()	suspend()	append()	yield()	append()
When we implement the Runnable interface, we must define the method	run()	start()	init()	main()	run()
The methods wait() and notify() are defined in?	java.lang.String	java.lang.Object	java.lang.Runnable	java.lang.Thread	java.lang.Object
How many ways are there to access package from another package?	3	2	1	5	3
The life cycle of the thread is controlled by ?	JDK	JVM	JRE	J2SDK	JVM
In how many states Threads can be explained ?	4	5	3	2	5
In which state the thread is still alive, but is currently not eligible to run?	Non-Runnable	Terminated	Runnable	Running	Non-Runnable
In Which state after invocation of start() method, but the thread Scheduler has not selected it to be the running thread?	Running	Runnable	Terminated	Non-Runnable	Runnable
These two ways are used to? (i) By extending Thread class (ii) By implementing Runnable interface.	Joining a thread	Naming a thread	Create a thread	sleeping a thread	Create a thread
Which method is used in thread class to starts the execution of the thread.JVM calls the run() method on the thread?	public void start()	public void run()	public void stop()	public coid suspend()	public void start()
Which method is used in thread class to tests if the current thread has been interrupted?	public static boolean interrupted()	public boolean isInterrupted()	public void interrupt()	public boolean isAlive()	public static boolean interrupted()
Which method in thread class causes the currently executing thread object to temporarily pause and allow other threads	public boolean isAlive()	public int getId()	public void yield()	public boolean isDaemon()	public void yield()
How many methods does a thread class provides for sleeping a thread?	3	1	4	2	2
Which method waits for a thread to die?	stop()	start()	terminate()	join()	join()
In Naming a thread which method is used to change the name of a thread?	public String getName()	public void setName(String name)	public void getName()	public String setName(String name)	public void setName(String name)
Default priority value of a thread class for NORM_PRIORITY is?	1	10	5	4	5

KARPAGAM ACADEMY OF HIGHER EDUCATION
CLASS: I B.Sc IT COURSE NAME: Programming in Java
COURSE CODE: 19ITU201 UNIT: IV (Strings, Collections, Utilities) BATCH-2019-2022
SYLLABUS

Strings: Creation – Operation on strings - Character Extraction Methods – Comparison –Searching and Modifying –Data Conversion and valueOf() Methods – Changing case of characters - String Buffer Class and its methods. Collection and Utilities: Collection of Objects – Core Interfaces and Classes – Iterators – List, Set, Map Implementations.

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

In this case, "Hello world!" is a *string literal*—a series of characters in the code that is enclosed in double quotes. Whenever it encounters a string literal in the code, the compiler creates a String object with its value—in this case, Hello world!.

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
String helloString = new String(helloArray);
System.out.println(helloString);
```

The last line of this code snippet displays hello.

Methods used to obtain information about an object are known as *accessor methods*. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object. After the following two lines of code have been executed, len equals 17:

```
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
```

A *palindrome* is a word or sentence that is symmetric—it is spelled the same forward and backward, ignoring case and punctuation. Here is a short and inefficient program to reverse a palindrome string. It invokes the String method charAt(i), which returns the i^{th} character in the string, counting from 0.

```
public class StringDemo {
   public static void main(String[] args) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      char[] tempCharArray = new char[len];
      char[] charArray = new char[len];

      // put original string in an array of chars
      for (int i = 0; i < len; i++) {
         tempCharArray[i] = palindrome.charAt(i);
      }
```

```
        // reverse array of chars
        for (int j = 0; j < len; j++) {
            charArray[j] = tempCharArray[len - 1 - j];
        }

        String reversePalindrome =  new String(charArray);
        System.out.println(reversePalindrome);
    }
}
```

Running the program produces this output:
doT saw I was toD

To accomplish the string reversal, the program had to convert the string to an array of characters (first for loop), reverse the array into a second array (second for loop), and then convert back to a string. The String class includes a method, getChars(), to convert a string, or a portion of a string, into an array of characters so we could replace the first for loop in the program above with

```
palindrome.getChars(0, len, tempCharArray, 0);
```

Creating Format Strings
The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows to create a formatted string that can reuse, as opposed to a one-time print statement. For example, instead of

```
System.out.printf("The value of the float variable is %f, while the value of the " +
                  "integer variable is %d, and the string is %s", floatVar, intVar, stringVar);
```

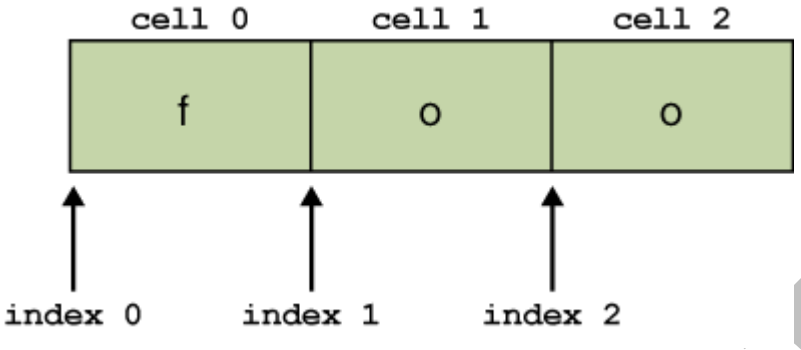
can write

```
String fs;
fs = String.format("The value of the float variable is %f, while the value of the " +
                   "integer variable is %d, and the string is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String Literals
The most basic form of pattern matching supported by this API is the match of a string literal. For example, if the regular expression is foo and the input string is foo, the match will succeed because the strings are identical. Try this out with the test harness:

Enter your regex: foo
Enter input string to search: foo
I found the text "foo" starting at index 0 and ending at index 3.

This match was a success. Note that while the input string is 3 characters long, the start index is 0 and the end index is 3. By convention, ranges are inclusive of the beginning index and exclusive of the end index, as shown in the following figure:



The string literal "foo", with numbered cells and index values.

Each character in the string resides in its own *cell*, with the index positions pointing between each cell. The string "foo" starts at index 0 and ends at index 3, even though the characters themselves only occupy cells 0, 1, and 2.

Enter your regex: foo
Enter input string to search: foofoofoo
I found the text "foo" starting at index 0 and ending at index 3.
I found the text "foo" starting at index 3 and ending at index 6.
I found the text "foo" starting at index 6 and ending at index 9.

Concatenating Strings

The String class includes a method for concatenating two strings:

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end.

concat() method with string literals, as in:

"My name is ".concat("Rumplestiltskin");

Strings are more commonly concatenated with the + operator, as in
"Hello," + " world" + "!"
which results in

"Hello, world!"

The + operator is widely used in print statements. For example:

String string1 = "saw I was ";

```
System.out.println("Dot " + string1 + "Tod");
```

which prints

Dot saw I was Tod

Such a concatenation can be a mixture of any objects. For each object that is not a String, its toString() method is called to convert it to a String.

The Java programming language does not permit literal strings to span lines in source files, so + must used to concatenation operator at the end of each line in a multi-line string. For example,

```
String quote = "Now is the time for all good " +
    "men to come to the aid of their country.";
```

Breaking strings between lines using the + concatenation operator is, once again, very common in print statements.

String Conversion and tostring()

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**. **valueOf( )** is overloaded for all the simple types and for type **Object**. For the simple types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf( )** calls the **toString( )** method on the object. We will look more closely at **valueOf( )** later in this chapter. Here, let's examine the **toString( )** method, because it is the means by which you can determine the string representation for objects of classes that you create.

The **toString( )** method has this general form:

String toString( )

To implement **toString( )**, simply return a **String** object that contains the human-readable string that appropriately describes an object of the class.

By overriding **toString( )** for classes that allow the resulting strings to be fully integrated into Java's programming environment. For example, they can be used in **print( )** and **println( )** statements and in concatenation expressions. The following program demonstrates this by overriding **toString( )** for the **Box** class:

```
// Override toString() for Box class.
class Box {
double width;
```

```
double height;
double depth;
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " +
depth + " by " + height + ".";
}
}
class toStringDemo {
public static void main(String args[]) {
Box b = new Box(10, 12, 14);
String s = "Box b: " + b; // concatenate Box object
System.out.println(b); // convert Box to string
System.out.println(s);
}
}
```

The output of this program is shown here:

Dimensions are 10 by 14 by 12.
Box b: Dimensions are 10 by 14 by 12.

Character extraction methods

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Each is examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt( )

To extract a single character from a **String**, we can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int *where*)

Here, *where* is the index of the character that we want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the v0alue "**b**" to **ch**.

getChars( )

If we need to extract more than one character at a time, we can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target.* The index within *target* at which the substring will be copied is passed in *targetStart.* Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring. The following program demonstrates **getChars( )**:

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

Here is the output of this program:

demo

getBytes( )

There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( )

Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when we are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( )

If we want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

Java String comparison

Java String comparison with the equals method

if (string1.equals(string2))

This Java String equals method looks at the two Java Strings, and if they contain the exact same string of characters, they are considered equal.

Taking a look at a quick Java String comparison example with the equals method, if the following test were run, the two strings would not be considered equal because the characters are not the exactly the same (the case of the characters is different):

```
String string1 = "foo";
String string2 = "FOO";

if (string1.equals(string2))
{
  // this line will not print because the
  // java string equals method returns false:
  System.out.println("The two strings are the same.")
}
```

But, when the two strings contain the *exact same* string of characters, the String equals method will return true, as in this example:

```
String string1 = "foo";
String string2 = "foo";
```

```
// test for equality with the java string equals method
if (string1.equals(string2))
{
  // this line WILL print
  System.out.println("The two strings are the same.")
}
```

Java String comparison with the equalsIgnoreCase method

In some Java String comparison tests user wants to ignore whether the strings are *uppercase* or *lowercase*. When we want to test our strings for equality in this case-insensitive manner, use the equalsIgnoreCase method of the Java String class, like this:

```
String string1 = "foo";
String string2 = "FOO";

// java string compare while ignoring case
if (string1.equalsIgnoreCase(string2))
{
  // this line WILL print
  System.out.println("Ignoring case, the two strings are the same.")
}
```

Java String comparison with the compareTo method

There is also a third, less common way to compare Java strings, and that's with the String class compareTo method. If the two Java strings are exactly the same, the compareTo method will return a value of 0 (zero). Here's a quick example of what this String comparison approach looks like:

```
String string1 = "foo bar";
String string2 = "foo bar";

// java string compare example
if (string1.compareTo(string2) == 0)
{
  // this line WILL print
  System.out.println("The two strings are the same.")
```

```
}
```

Operations on string

The **String** class provides two methods that allows to search a string for a specified character or substring:

- **indexOf( )** Searches for the first occurrence of a character or substring.
- **lastIndexOf( )** Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

To search for the first occurrence of a character, use int indexOf(int *ch*)
To search for the last occurrence of a character, use int lastIndexOf(int *ch*)

Here, *ch* is the character being sought. To search for the first or last occurrence of a substring, use

int indexOf(String *str*)
int lastIndexOf(String *str*)

Here, *str* specifies the substring.
we can specify a starting point for the search using these forms:

int indexOf(int *ch*, int *startIndex*)
int lastIndexOf(int *ch*, int *startIndex*)
int indexOf(String *str*, int *startIndex*)
int lastIndexOf(String *str*, int *startIndex*)

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.
The following example shows how to use the various index methods to search inside of **String**s:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
public static void main(String args[]) {

    String s = "Now is the time for all good men " +
    "to come to the aid of their country.";
    System.out.println(s);
    System.out.println("indexOf(t) = " +
    s.indexOf('t'));
```

```
System.out.println("lastIndexOf(t) = " +
s.lastIndexOf('t'));
System.out.println("indexOf(the) = " +
s.indexOf("the"));
System.out.println("lastIndexOf(the) = " +
s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " +
s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " +
s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " +
s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +
s.lastIndexOf("the", 60));
}
}
```

Here is the output of this program:

Now is the time for all good men to come to the aid of their country.

indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55

substring()

In this example we are taking a sub string from a given string.

In this example we are creating an string object .We initialize this string object as "Rajesh Kumar". We are taking sub string by use of **substring()** method.

The methods used:
substring(int i):

This method is used to find all sub string after index i.

substring(int start,int end):

This is used to find the substring between start and end point.

The code of the program is given below:

```
public class SubstringExample1{
   public static void main(String[] args){
      String string = "Rajesh kumar";
      System.out.println("String : " + string);
      String substring = string.substring(3);
      System.out.println("String after 3rd index:
 " + substring);
      substring = string.substring(1, 2);
      System.out.println("Substring (1,2): " +
substring);
   }
}
```

Replace() method

This program describes how to replace all the words in a String. We are going to use ***replaceAll()*** method of **String** class in Java.

```
public class ReplaceDemo {
  public static void main(String[] args) {
    String str = "Her name is Tamana and Tamana is a good girl.";
    String strreplace = "Sonia";
    String result = str.replaceAll("Tamana", strreplace);
    System.out.println(result);
    }
}
```

Output of the program:

```
C:\unique>javac ReplaceDemo.java

C:\unique>java ReplaceDemo
Her name is Sonia and Sonia is a good girl.

C:\unique>
```

Trim String Example

In this section, you will learn how to remove the blank spaces. For removing the blank spaces use **trim()** method that removes the blank spaces and shows only string.

Description of code:

trim():

This method removes the blank spaces from both ends of the given string (Front and End).

Here is the code of program:

```
import java.lang.*;

public class StringTrim{
  public static void main(String[] args) {
    System.out.println("String trim example!");
    String str = "     RoseIndia";
    System.out.println("Given String :" + str);
    System.out.println("After trim :" +str.trim());
  }
}
```

Output of program:

```
C:\vinod\Math_package>javac
StringTrim.java

C:\vinod\Math_package>java
StringTrim
String trim example!
Given String :  RoseIndia
After trim :RoseIndia
```

String Buffer class

The StringBuffer Class

Once a String object is instantiated, it cannot change in size or content. Any change yields a new String object and the old one is discarded. Strings created with the StringBuffer class, however, are dynamic.

Once created, characters can change and new characters can be inserted or deleted. Although these tasks are possible through the creative use of substringing and concatenation with String objects, there are performance benefits in using StringBuffer objects when such manipulations are frequent.

The StringBuffer class is part of the java.lang package. The most common methods are,

Methods of the StringBuffer Class
Method Description
Constructors
StringBuffer() constructs a StringBuffer object with no characters in it and an initial capacity of 16 characters; returns a reference to the new object

StringBuffer(int length) constructs a StringBuffer object with no characters in it and an initial capacity specified by length; returns a reference to the new object

StringBuffer(String s) constructs a StringBuffer object so that it represents the same sequence of characters as the string s; returns a reference to the new object.

StringBuffer is a peer class of String that provides much of the functionality of the strings. String represents fixed-length, immutable character sequences. In contrast StringBuffer represents growable and writable character sequences. The StringBuffer provides 3 constructors which create, initialize and set the initial capacity of StringBuffer objects. This class provides many methods. For example the length() method gives the current length i.e. how many characters are there in the string, while the total allocated capacity can be found by the capacity() method.

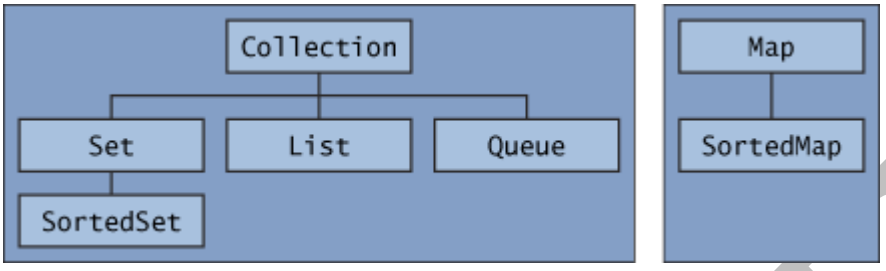
```
public class StringBufferDemo {

  public static void main(String[] args) {

    StringBuffer sb =new StringBuffer("Hello");
    System.out.println("buffer= " +sb);
    System.out.println("length= " +sb.length());
    System.out.println("capacity= " +sb.capacity());

    //appending and inserting into StringBuffer.
    String s;
    int a = 42;
    StringBuffer sb1= new StringBuffer(40);
    s= sb1.append("a=").append(a).append("!").toString();
    System.out.println(s);
    StringBuffer sb2 = new StringBuffer("I JAVA!");
    sb2.insert(2, "LIKE");
    System.out.println(sb2);

  }

}
```

Output Screen

```
buffer= Hi Rohit
length= 8
capacity= 24
a=42!
I LIKEJAVA!
```

Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. See also The Set Interface section.

List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties. Also see The Queue Interface section.

Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.

The last two core collection interfaces are merely sorted versions of Set and Map:

SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls. Also see The SortedSet Interface section.

SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories. Also see The SortedMap Interface section.

Modifier and Type	Method and Description

static <E> List<E>	asList(E first, E[] rest) Returns an unmodifiable list containing the specified first element and backed by the specified array of additional elements.
static <E> List<E>	asList(E first, E second, E[] rest) Returns an unmodifiable list containing the specified first and second element, and backed by the specified array of additional elements.
static List<Character>	charactersOf(CharSequence sequence) Returns a view of the specified CharSequence as a List<Character>, viewing sequence as a sequence of Unicode code units.
static ImmutableList<Character>	charactersOf(String string) Returns a view of the specified string as an immutable list of Character values.
static <E> ArrayList<E>	newArrayList() Creates a *mutable*, empty ArrayList instance (for Java 6 and earlier).
static <E> ArrayList<E>	newArrayList(E... elements) Creates a *mutable* ArrayList instance containing the given elements.
static <E> ArrayList<E>	newArrayList(Iterable<? extends E> elements) Creates a *mutable* ArrayList instance containing the given elements; a very thin shortcut for creating an empty list then calling Iterables.addAll(java.util.Collection<T>, java.lang.Iterable<? extends T>).
static <E> ArrayList<E>	newArrayList(Iterator<? extends E> elements) Creates a *mutable* ArrayList instance containing the given elements; a very thin shortcut for creating an empty list and then calling Iterators.addAll(java.util.Collection<T>, java.util.Iterator<? extends T>).
static <E> ArrayList<E>	newArrayListWithCapacity(int initialArray Size) Creates an ArrayList instance backed by an array with the specified initial size; simply delegates to ArrayList.ArrayList(int).

	newArrayListWithExpectedSize(int estimatedSize) Creates an ArrayList instance to hold estimatedSize elements, *plus* an unspecified amount of padding; you almost certainly mean to call newArrayListWithCapacity(int) (see that method for further advice on usage).
static <E> ArrayList<E>	
static <E> CopyOnWriteArrayList<E>	newCopyOnWriteArrayList() Creates an empty CopyOnWriteArrayList instance.
static <E> CopyOnWriteArrayList<E>	newCopyOnWriteArrayList(Iterable<? extends E> elements) Creates a CopyOnWriteArrayList instance containing the given elements.
static <E> LinkedList<E>	newLinkedList() Creates a *mutable*, empty LinkedList instance (for Java 6 and earlier).
static <E> LinkedList<E>	newLinkedList(Iterable<? extends E> elements) Creates a *mutable* LinkedList instance containing the given elements; a very thin shortcut for creating an empty list then calling Iterables.addAll(java.util.Collection<T>, java.lang.Iterable<? extends T>).
static <T> List<List<T>>	partition(List<T> list, int size) Returns consecutive sublists of a list, each of the same size (the final list may be smaller).
static <T> List<T>	reverse(List<T> list) Returns a reversed view of the specified list.
static <F,T> List<T>	transform(List<F> fromList, Function<? super F,? extends T> function) Returns a list that applies function to each element of fromList.

- ***Class Sets***

@GwtCompatible(emulated=true)
public final class Sets extends Object

Nested Classes

Modifier and Type	Class and Description
static class	Sets.SetView<E> An unmodifiable view of a set which may be backed by other sets; this view will change as the backing sets do.

- **Method Summary**

Methods

Modifier and Type	Method and Description
static <B> Set<List<B>>	cartesianProduct(List<? extends Set<? extends B>> sets) Returns every possible list that can be formed by choosing one element from each of the given sets in order; the "n-ary Cartesian product" of the sets.
static <B> Set<List<B>>	cartesianProduct(Set<? extends B>... sets) Returns every possible list that can be formed by choosing one element from each of the given sets in order; the "n-ary Cartesian product" of the sets.
static <E extends Enum<E>> EnumSet<E>	complementOf(Collection<E> collection) Creates an EnumSet consisting of all enum values that are not in the specified collection.
static <E extends Enum<E>> EnumSet<E>	complementOf(Collection<E> collection, Class<E> type) Creates an EnumSet consisting of all enum values that are not in the specified collection.
static <E> Sets.SetView<E>	difference(Set<E> set1, Set<?> set2) Returns an unmodifiable **view** of the difference of two sets.
static <E> NavigableSet<E>	filter(NavigableSet<E> unfiltered, Predicate<? super E> predicate) Returns the elements of a NavigableSet, unfiltered, that satisfy a predicate.
static <E> Set<E>	filter(Set<E> unfiltered, Predicate<? super E> predicate) Returns the elements of unfiltered that satisfy a predicate.
static <E> SortedSet<E>	filter(SortedSet<E> unfiltered, Predicate<? super E> predicate) Returns the elements of a SortedSet, unfiltered, that satisfy a predicate.
static <E extends Enum<E>> ImmutableSet<E>	immutableEnumSet(E anElement, E... otherElements) Returns an immutable set instance containing the given enum elements.
static <E extends Enum<E>> ImmutableSet<E>	immutableEnumSet(Iterable<E> elements) Returns an immutable set instance containing the given enum elements.
static <E> Sets.SetView<E>	intersection(Set<E> set1, Set<?> set2) Returns an unmodifiable **view** of the intersection

	of two sets.
static <E> Set<E>	newConcurrentHashSet() Creates a thread-safe set backed by a hash map.
static <E> Set<E>	newConcurrentHashSet(Iterable<? extends E> elements) Creates a thread-safe set backed by a hash map and containing the given elements.
static <E> CopyOnWriteArraySet<E>	newCopyOnWriteArraySet() Creates an empty CopyOnWriteArraySet instance.
static <E> CopyOnWriteArraySet<E>	newCopyOnWriteArraySet(Iterable<? extends E> elements) Creates a CopyOnWriteArraySet instance containing the given elements.
static <E extends Enum<E>> EnumSet<E>	newEnumSet(Iterable<E> iterable, Class<E> elementType) Returns a new EnumSet instance containing the given elements.
static <E> HashSet<E>	newHashSet() Creates a *mutable*, empty HashSet instance.
static <E> HashSet<E>	newHashSet(E... elements) Creates a *mutable* HashSet instance containing the given elements in unspecified order.
static <E> HashSet<E>	newHashSet(Iterable<? extends E> elements) Creates a *mutable* HashSet instance containing the given elements in unspecified order.
static <E> HashSet<E>	newHashSet(Iterator<? extends E> elements) Creates a *mutable* HashSet instance containing the given elements in unspecified order.
static <E> HashSet<E>	newHashSetWithExpectedSize(int expectedSize) Creates a HashSet instance, with a high enough "initial capacity" that it *should* hold expectedSize elements without growth.
static <E> Set<E>	newIdentityHashSet() Creates an empty Set that uses identity to determine equality.
static <E> LinkedHashSet<E>	newLinkedHashSet() Creates a *mutable*, empty LinkedHashSet instance.
static <E> LinkedHashSet<E>	newLinkedHashSet(Iterable<? extends E> elements) Creates a *mutable* LinkedHashSet instance

	containing the given elements in order.
static <E> LinkedHashSet<E>	newLinkedHashSetWithExpectedSize(int expectedSize) Creates a LinkedHashSet instance, with a high enough "initial capacity" that it *should* hold expectedSize elements without growth.
static <E> Set<E>	newSetFromMap(Map<E,Boolean> map) Returns a set backed by the specified map.
static <E extends Comparable> TreeSet<E>	newTreeSet() Creates a *mutable*, empty TreeSet instance sorted by the natural sort ordering of its elements.
static <E> TreeSet<E>	newTreeSet(Comparator<? super E> comparator) Creates a *mutable*, empty TreeSet instance with the given comparator.
static <E extends Comparable> TreeSet<E>	newTreeSet(Iterable<? extends E> elements) Creates a *mutable* TreeSet instance containing the given elements sorted by their natural ordering.
static <E> Set<Set<E>>	powerSet(Set<E> set) Returns the set of all possible subsets of set.
static <E> Sets.SetView<E>	symmetricDifference(Set<? extends E> set1, Set<? extends E> set2) Returns an unmodifiable **view** of the symmetric difference of two sets.
static <E> NavigableSet<E>	synchronizedNavigableSet(NavigableSet<E> navigableSet) Returns a synchronized (thread-safe) navigable set backed by the specified navigable set.
static <E> Sets.SetView<E>	union(Set<? extends E> set1, Set<? extends E> set2) Returns an unmodifiable **view** of the union of two sets.
static <E> NavigableSet<E>	unmodifiableNavigableSet(NavigableSet<E> set) Returns an unmodifiable view of the specified navigable set.

Class Maps

Modifier and Type	Method and Description
static <A,B> Converter<A,B>	asConverter(BiMap<A,B> bimap) Returns a Converter that converts values using bimap.get(), and whose inverse view converts values using bimap.inverse().get().

static <K,V> NavigableMap<K,V>	asMap(NavigableSet<K> set, Function<? super K,V> function) Returns a view of the navigable set as a map, mapping keys from the set according to the specified function.
static <K,V> Map<K,V>	asMap(Set<K> set, Function<? super K,V> function) Returns a live Map view whose keys are the contents of set and whose values are computed on demand using function.
static <K,V> SortedMap<K,V>	asMap(SortedSet<K> set, Function<? super K,V> function) Returns a view of the sorted set as a map, mapping keys from the set according to the specified function.
static <K,V> MapDifference<K,V>	difference(Map<? extends K,? extends V> left, Map<? extends K,? extends V> right) Computes the difference between two maps.
static <K,V> MapDifference<K,V>	difference(Map<? extends K,? extends V> left, Map<? extends K,? extends V> right, Equivalence<? super V> valueEquivalence) Computes the difference between two maps.
static <K,V> SortedMapDifference<K,V>	difference(SortedMap<K,? extends V> left, Map<? extends K,? extends V> right) Computes the difference between two sorted maps, using the comparator of the left map, or Ordering.natural() if the left map uses the natural ordering of its elements.
static <K,V> BiMap<K,V>	filterEntries(BiMap<K,V> unfiltered, Predicate<? super Map.Entry<K,V>> entryPredicate) Returns a bimap containing the mappings in unfiltered that satisfy a predicate.
static <K,V> Map<K,V>	filterEntries(Map<K,V> unfiltered, Predicate<? super Map.Entry<K,V>> entryPredicate) Returns a map containing the mappings

	in unfiltered that satisfy a predicate.
static <K,V> NavigableMap<K,V>	filterEntries(NavigableMap<K,V> unfiltered, Predicate<? super Map.Entry<K,V>> entryPredicate) Returns a sorted map containing the mappings in unfiltered that satisfy a predicate.
static <K,V> SortedMap<K,V>	filterEntries(SortedMap<K,V> unfiltered, Predicate<? super Map.Entry<K,V>> entryPredicate) Returns a sorted map containing the mappings in unfiltered that satisfy a predicate.
static <K,V> BiMap<K,V>	filterKeys(BiMap<K,V> unfiltered, Predicate<? super K> keyPredicate) Returns a bimap containing the mappings in unfiltered whose keys satisfy a predicate.
static <K,V> Map<K,V>	filterKeys(Map<K,V> unfiltered, Predicate<? super K> keyPredicate) Returns a map containing the mappings in unfiltered whose keys satisfy a predicate.
static <K,V> NavigableMap<K,V>	filterKeys(NavigableMap<K,V> unfiltered, Predicate<? super K> keyPredicate) Returns a navigable map containing the mappings in unfiltered whose keys satisfy a predicate.
static <K,V> SortedMap<K,V>	filterKeys(SortedMap<K,V> unfiltered, Predicate<? super K> keyPredicate) Returns a sorted map containing the mappings in unfiltered whose keys satisfy a predicate.
static <K,V> BiMap<K,V>	filterValues(BiMap<K,V> unfiltered, Predicate<? super V> valuePredicate) Returns a bimap containing the mappings in unfiltered whose values satisfy a predicate.
static <K,V> Map<K,V>	filterValues(Map<K,V> unfiltered, Predicate<? super V> valuePredicate) Returns a map containing the mappings in unfiltered whose values satisfy a predicate.
static	filterValues(NavigableMap<K,V> unfilt

<K,V> NavigableMap<K,V>	ered, Predicate<? super V> valuePredicate) Returns a navigable map containing the mappings in unfiltered whose values satisfy a predicate.
static <K,V> SortedMap<K,V>	filterValues(SortedMap<K,V> unfiltered, Predicate<? super V> valuePredicate) Returns a sorted map containing the mappings in unfiltered whose values satisfy a predicate.
static ImmutableMap<String,String>	fromProperties(Properties properties) Creates an ImmutableMap<String, String> from a Properties instance.
static <K,V> Map.Entry<K,V>	immutableEntry(K key, V value) Returns an immutable map entry with the specified key and value.
static <K extends Enum<K>,V> ImmutableMap<K,V>	immutableEnumMap(Map<K,? extends V> map) Returns an immutable map instance containing the given entries.
static <K,V> ConcurrentMap<K,V>	newConcurrentMap() Returns a general-purpose instance of ConcurrentMap, which supports all optional operations of the ConcurrentMap interface.
static <K extends Enum<K>,V> EnumMap<K,V>	newEnumMap(Class<K> type) Creates an EnumMap instance.
static <K extends Enum<K>,V> EnumMap<K,V>	newEnumMap(Map<K,? extends V> map) Creates an EnumMap with the same mappings as the specified map.
static <K,V> HashMap<K,V>	newHashMap() Creates a *mutable*, empty HashMap instance.
static <K,V> HashMap<K,V>	newHashMap(Map<? extends K,? extends V> map) Creates a *mutable* HashMap instance with the same mappings as the specified map.
static <K,V> HashMap<K,V>	newHashMapWithExpectedSize(int expectedSize) Creates a HashMap instance, with a high enough "initial capacity" that it *should*

	hold expectedSize elements without growth.
static <K,V> IdentityHashMap<K,V>	newIdentityHashMap() Creates an IdentityHashMap instance.
static <K,V> LinkedHashMap<K,V>	newLinkedHashMap() Creates a *mutable*, empty, insertion-ordered LinkedHashMap instance.
static <K,V> LinkedHashMap<K,V>	newLinkedHashMap(Map<? extends K,? extends V> map) Creates a *mutable*, insertion-ordered LinkedHashMap instance with the same mappings as the specified map.
static <K extends Comparable,V> TreeMap<K,V>	newTreeMap() Creates a *mutable*, empty TreeMap instance using the natural ordering of its elements.
static <C,K extends C,V> TreeMap<K,V>	newTreeMap(Comparator<C> comparator) Creates a *mutable*, empty TreeMap instance using the given comparator.
static <K,V> TreeMap<K,V>	newTreeMap(SortedMap<K,? extends V> map) Creates a *mutable* TreeMap instance with the same mappings as the specified map and using the same ordering as the specified map.
static <K,V> BiMap<K,V>	synchronizedBiMap(BiMap<K,V> bimap) Returns a synchronized (thread-safe) bimap backed by the specified bimap.
static <K,V> NavigableMap<K,V>	synchronizedNavigableMap(NavigableMap<K,V> navigableMap) Returns a synchronized (thread-safe) navigable map backed by the specified navigable map.
static <K,V> ImmutableMap<K,V>	toMap(Iterable<K> keys, Function<? super K,V> valueFunction) Returns an immutable map whose keys are the distinct elements of keys and whose value for each key was computed by valueFunction.
static <K,V> ImmutableMap<K,V>	toMap(Iterator<K> keys, Function<? super K,V> valueFunction)

	Returns an immutable map whose keys are the distinct elements of keys and whose value for each key was computed by valueFunction.
static <K,V1,V2> Map<K,V2>	transformEntries(Map<K,V1> fromMap , Maps.EntryTransformer<? super K,? super V1,V2> transformer) Returns a view of a map whose values are derived from the original map's entries.
static <K,V1,V2> NavigableMap<K,V2 >	transformEntries(NavigableMap<K,V1> fromMap, Maps.EntryTransformer<? super K,? super V1,V2> transformer) Returns a view of a navigable map whose values are derived from the original navigable map's entries.
static <K,V1,V2> SortedMap<K,V2>	transformEntries(SortedMap<K,V1> fro mMap, Maps.EntryTransformer<? super K,? super V1,V2> transformer) Returns a view of a sorted map whose values are derived from the original sorted map's entries.
static <K,V1,V2> Map<K,V2>	transformValues(Map<K,V1> fromMap , Function<? super V1,V2> function) Returns a view of a map where each value is transformed by a function.
static <K,V1,V2> NavigableMap<K,V2 >	transformValues(NavigableMap<K,V1> fromMap, Function<? super V1,V2> function) Returns a view of a navigable map where each value is transformed by a function.
static <K,V1,V2> SortedMap<K,V2>	transformValues(SortedMap<K,V1> fro mMap, Function<? super V1,V2> function) Returns a view of a sorted map where each value is transformed by a function.
static <K,V> ImmutableMap<K,V>	uniqueIndex(Iterable<V> values, Function<? super V,K> keyFunction) Returns an immutable map for which the Map.values() are the given elements in the given order, and each key is the product of invoking a supplied function on its corresponding value.

static <K,V> ImmutableMap<K,V>	uniqueIndex(Iterator<V> values, Function<? super V,K> keyFunction) Returns an immutable map for which the Map.values() are the given elements in the given order, and each key is the product of invoking a supplied function on its corresponding value.
static <K,V> BiMap<K,V>	unmodifiableBiMap(BiMap<? extends K,? extends V> bimap) Returns an unmodifiable view of the specified bimap.
static <K,V> NavigableMap<K,V>	unmodifiableNavigableMap(Navigable Map<K,V> map) Returns an unmodifiable view of the specified navigable map.

Possible Questions

Part - B(2 Marks)

1. What is the purpose of using valueOf() methods
2. How will you create string in java?
3. Define StringBuffer class

Part - C(6 Marks)

1. How will you create a string in Java? List out the various constructors provided with String class
2. What is the method used to find the number of characters in a string. Give an example
3. How is the Map implementation useful in Java. Explain in detail with examples.
4. Define the constructors used in StringBuffer class. Describe in detail the methods of StringBuffer class.
5. Write in detail about the SET implementations in Java with example for each
6. What is the function of substring() method? Give an example
7. What are lists? Describe in detail about the List implementations with example.
8. Spot out the methods used to compare strings. Explain in detail each method with example.
9. What are Iterators? Explain in detail about the constructors and methods for Iterator Interface.
10. Explain the methods used to extract characters from the given string. Discuss each of them with example

Questions	opt1	opt2	opt3	opt4	answer
A built_in class which encapsulates the data structure of a string is _____	java io	String	Character	StringBuffer	String
The instances of the class String is created using _____	new	free	object	try	new
To extract a single character from a string , the _____ method is used.	charAt	Stringto	charone	indexOf	charAt
To get the substring from a string _____ method is used.	getchars	substr	extract	substring	getchars
The _____ method compares the characters inside the string.	= =	equivalent	equals	lastIndexOf	equals
The _____ operator compares two objects references to see if they refer to	= =	equivalent	equals	equalto	= =
The String method _____ can be used to determine ordering.	StringTo	CompareTo	Compare	CompareOf	CompareTo
If the integer result of CompareTo is negative, then the string is _____	Equal	Less	Greater	compare	Less
If the integer result of CompareTo is positive, then the string is _____	Equal	Less	Greater	lesser	Greater
The search for a certain character or substring is done using _____ &	index & indexof	index & lastindex	indexof & lastindexof	compareTo	indexof & lastindexof
The replace method takes _____ characters as parameters.	1	2	3	4	2
_____ represents fixed length immutable character sequences.	String	Characters	Variable	Identifier	String
The length of a string by calling the ____ method	strlen()	len()	length( )	none	length( )
The character at a specified index within a string by calling ____	charAt( )	chatat()	char()	character()	charAt( )
_____ is a sequence of characters	Variable	String	Values	stringbuffer	String
A built-in class which encapsulates the data structure of a string is _____	jav io	String	Character	int	String
The instances of the class String is created using _____	new	free	object	methods	new
To extract a single character from a string , the _____ method is used.	charAt	Stringto	charone	replace	charAt
To get the substring from a string _____ method is used.	getchars	substr	extract	substring	getchars
The _____ method compares the characters inside the string.	= =	equivalent	equals	equalto	equals
The _____ operator compares two objects references to see if they refer to the exact same instance	= =	equivalent	equals	compare	= =
The String method _____ can be used to determine ordering.	StringTo	CompareTo	Compare	CompareOf	CompareTo
If the integer result of CompareTo is negative, then the string is _____ than the parameter	Equal	Less	Greater	leser	Less
If the integer result of CompareTo is positive, then the string is _____ than the parameter	Equal	Less	Greater	greaterthan	Greater

The search for a certain character or substring is done using _____ &	index & indexof	index & lastindex	indexof & lastindexof	all	indexof & lastindexof
The replace method takes _____ characters as parameters.	1	2	3	4	2
_____ represents fixed length immutable character sequences.	String	Characters	Variable	Identifier	String
The append method on StringBuffer is most often called through the _____ operator.	-	+	add	+=	+
A group of Character is Called _____	function	arrays	data types	strings	
Suppose that you would like to create an instance of a new `Map` that has an iteration order that is the same as the	TreeMap	HashMap	LinkedHashMap	The answer depends on the implementation	LinkedHashMap
Which class does not override the `equals()` and `hashCode()` methods, inheriting them directly from class	java.lang.String	java.lang.Double	java.lang.StringBuffer	java.lang.Character	java.lang.StringBuffer
Which collection class allows you to grow or shrink its size and provides indexed access to its elements, but whose	java.util.HashSet	java.util.LinkedHashSet	java.util.List	java.util.ArrayList	java.util.ArrayList
You need to store elements in a collection that guarantees that no duplicates are stored and all elements can	java.util.Map	java.util.Set	java.util.List	java.util.Collection	java.util.Set
Which interface does `java.util.Hashtable` implement?	Java.util.Map	Java.util.List	Java.util.HashTable	Java.util.Collection	Java.util.Map
Which interface provides the capability to store objects using a key-value pair?	Java.util.Map	Java.util.Set	Java.util.List	Java.util.Collection	Java.util.Map
Which collection class allows you to associate its elements with key values, and allows you to retrieve objects in	java.util.ArrayList	java.util.LinkedHashMap	java.util.HashMap	java.util.TreeMap	java.util.LinkedHashMap
Which collection class allows you to access its elements by associating a key with an element's value, and provides	java.util.SortedMap	java.util.TreeMap	java.util.TreeSet	java.util.Hashtable	java.util.Hashtable
Which of these packages contain all the collection classes?	java.lang	java.util	java.net	java.awt	java.util
Which of these classes is not part of Java's collection framework?	Maps	Array	Stack	Queue	Queue
Which of these interface is not a part of Java's collection framework?	List	Set	SortedMap	SortedList	SortedList
Which of these methods deletes all the elements from invoking collection?	clear()	reset()	delete()	refresh()	clear()
What is Collection in Java?	A group of objects	A group of classes	A group of interfaces	A group of interfaces	A group of objects
Which of these interface declares core method that all collections will have?	set	EventListner	Comparator	Collection	Collection
Which of these interface handle sequences?	Set	List	Comparator	Collection	List
Which of these interface must contain a unique element?	Set	List	Array	Collection	Set
Which of these is Basic interface that all other interface inherits?	Set	Array	List	Collection	Collection
Which of these is an incorrect form of using method max() to obtain maximum element?	max(Collection c)	max(Collection c, Comparator comp)	max(Comparator comp)	max(List c)	max(Comparator comp)
Which of these methods sets every element of a List to a specified object?	set()	fill()	Complete()	add()	fill()
Which of these methods can randomize all elements in a list?	rand()	randomize()	shuffle()	ambigous()	shuffle()

Which of these methods can convert an object into a List?	SetList()	ConvertList()	singletonList()	CopyList()	singletonList()
Which of these is true about unmodifiableCollection() method?	unmodifiableCollection() returns a collection that	unmodifiableCollection() method is available only	unmodifiableCollection() is defined in	unmodifiableCollection() method is available only	unmodifiableCollection() method is available only
Which of these is static variable defined in Collections?	EMPTY_SET	EMPTY_LIST	EMPTY_MAP	All	All

KARPAGAM ACADEMY OF HIGHER EDUCATION
CLASS: I B.Sc IT COURSE NAME: Programming in Java
COURSE CODE: 19ITU201 UNIT: V (Input Output Classes, Applets) BATCH-2019-2022
SYLLABUS

I/O Operations –Hierarchy of Classes – File class – Input Stream, Output Stream, FilterInputStream, FilterOutputStream, Reader and Writer classes – Random Access File class – Stream Tokenizer. Applets: Basics – Life Cycle –Methods –Graphics Class- Color, Font, and Font Metrics Class – Using the Status window – Passing parameters to Applets – getDocumentBase() and getCodeBase(). AWT Components: AWT Classes – Basic Component and Container Classes – Frame Window in an Applet.

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in java** by Java I/O API.

In java, 3 streams are created for us automatically. All these streams are attached with console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and error** message to the console.

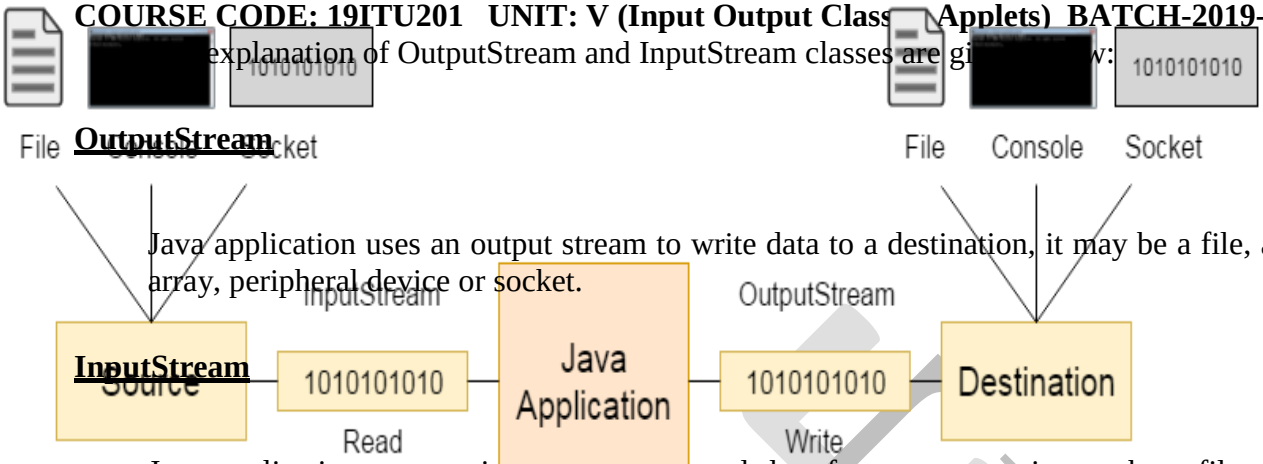
```
System.out.println
("simple
message");
System.err.println(
"error message");
```

Let's see the code to get **input** from console.

```
int i=System.in.read();//returns ASCII
code of 1st character
System.out.println((char)i);//will print
the character
```

OutputStream vs InputStream

explanation of OutputStream and InputStream classes are gi w:



OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

OutputStream class

OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stre
2) public void write(byte[])throws IOException	is used to write an array of byte to the cu output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

InputStream class

InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

Character Stream Vs Byte Stream in Java I/O Stream

A stream is a method to sequentially access a file. I/O Stream means an input source or output destination representing different types of sources e.g. disk files.The java.io package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text.

Stream – A sequence of data.
Input Stream: reads data from source.
Output Stream: writes data to destination.

Character Stream

In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source and write to destination.

Byte Stream

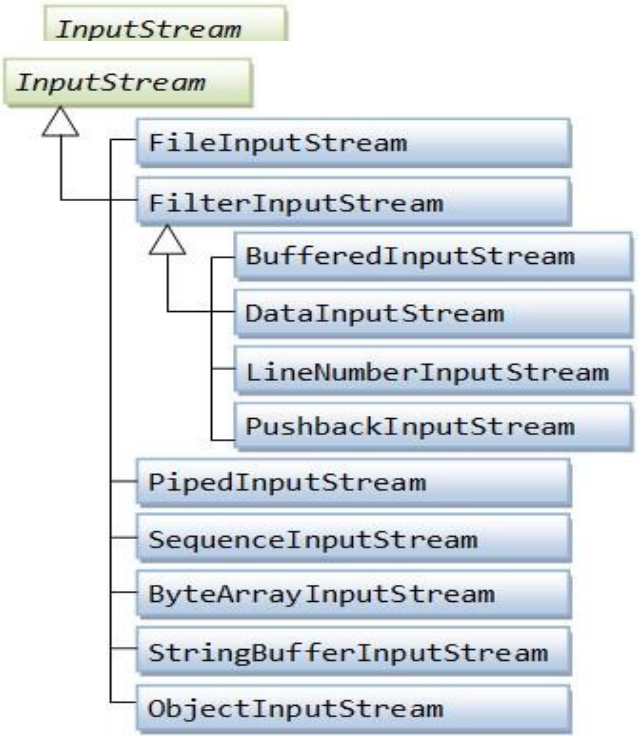
Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

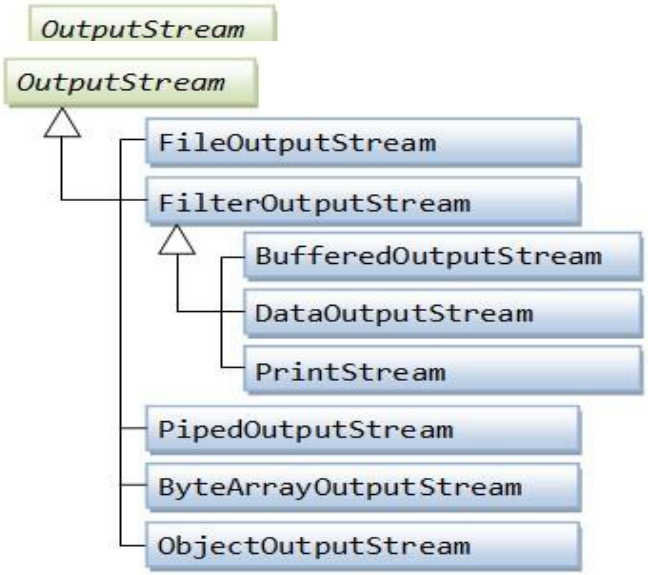
// Java Program illustrating the Byte Stream to copy

// contents of one file to another file.

importjava.io.*; publicclassBStream{
publicstaticvoidmain(String[] args) throwsIOException { FileInputStream sourceStream =

```
null;
FileOutputStream targetStream = null; try     {
sourceStream = newFileInputStream("sorcefile.txt"); targetStream = newFileOutputStream
("targetfile.txt");
// Reading source file and writing content to target
// file byte by byte inttemp;
while((temp = sourceStream.read()) != -1) targetStream.write((byte)temp);}
finally { if(sourceStream != null)
sourceStream.close(); if(targetStream != null)
targetStream.close();   }         }}
```





When to use Character Stream over Byte Stream?

- In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

When to use Byte Stream over Character Stream?

- Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.

File I/O

In Java, we can read data from files and also write data in files.

We do these using streams. Java has many input and output streams that are used to read

and write data. Same as a continuous flow of water is called water stream, in the same way input and output flow of data is called stream.

Stream

Java

Strea

Let's

Byte

It is

We c

are **F**

Byte Stream class	Description
BufferedInputStream	handles buffered input stream
BufferedOutputStream	handles buffered output stream
FileInputStream	used to read from a file
FileOutputStream	used to write to a file
InputStream	Abstract class that describe input stream

Character Stream

It is used in the input and output of characters.

For input and output of characters, we have Character stream classes. Two most commonly used Character stream classes are **FileReader** and **FileWriter**. Below is the list of some Character Stream classes.

FileWriter	used to write to a file
InputStreamReader	translate input from byte to character
OutputStreamReader	translate character to byte output
Reader	Abstract class that describe input stream
Writer	Abstract class that describe output stream

AWT Components

The applet is implemented as a button that brings up the window showing the components. The window is necessary because the program includes a menu, and menus can be used only in windows. Here, for the curious, is the source code for the window that displays the components. The program has a main() method so it can run as an application. The Applet Button class provides an applet framework for the window. AppletButton is a highly configurable applet that's discussed on the following pages: Deciding Which Parameters to Support and Writing the Code to Support Parameters.

The Basic Controls: Buttons, Checkboxes, Choices, Lists, Menus, and Text Fields

The Button, Checkbox, Choice, List, Menu Item, and Text Field classes provide basic controls. These are the most common ways that users give instructions to Java programs. When a user activates one of these controls -- by clicking a button or by pressing Return in a text field, for example -- it posts an event (ACTION_EVENT). An object that contains the control can react to the event by implementing the action() method.

Other Ways of Getting User Input: Sliders, Scrollbars, and Text Areas

When the basic controls aren't appropriate, we can use the Scrollbar and Text Area classes to get user input. The Scrollbar class is used for both slider and scrollbar functionality.

The TextArea class simply provides an area to display or allow editing of several lines of text.

Creating Custom Components: Canvases

The Canvas class lets we write custom Components. With Canvas subclass, we can draw custom graphics to the screen -- in a paint program, image processor, or game, for example -- and implement any kind of event handling.

Labels

A Label simply displays an unselectable line of text.

Containers: Windows and Panels

The AWT provides two types of containers, both implemented as subclasses of the Container api class (which is a Component subclass). The Window subclasses -- Dialog, File Dialog, and Frame -- provide windows to contain components. Frames create normal, full-fledged windows, as opposed to the windows that Dialogs create, which are dependent on Frames and can be modal. Panels group components within an area of an existing window.

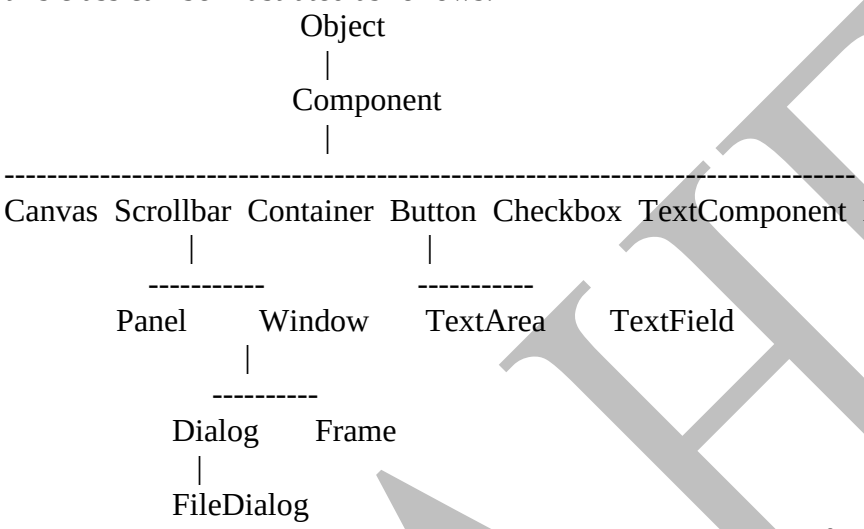
The AWT Classes

There are four main classes in AWT:

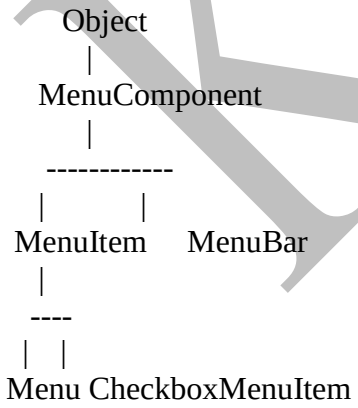
- the Component class - this class implements interface components such as menus, buttons, lists etc.;

- the Container class - this extends components to include higher level objects such as Dialog and Window;
- the Graphics class - this defines the methods for performing graphical operations on components;
- the LayoutManager - this defines methods for positioning and sizing objects within a container.

The java.awt.Component class is, therefore, fundamental to the AWT in Java. The structure of this class can be illustrated as follows:



The only exception to the classes shown in this diagram are for Menus and Menu items. This difference can be explained by again looking at Word running under WindowsNT and a Macintosh. There are pronounced differences in the ways that different native platforms implement menus. In some systems it is possible to set the background colour of a menu, in other it is not. Menus, therfore, form part of java.awt.MenuComponent rather than java.awt.Component:

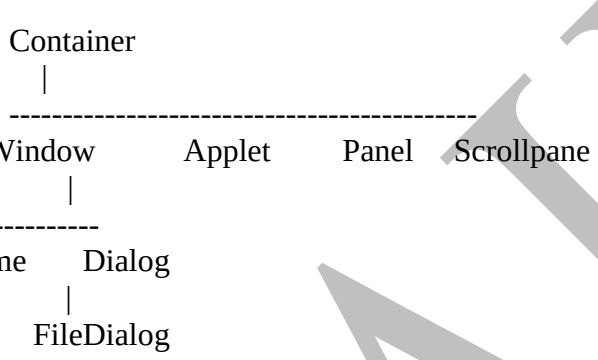


Each component has a corresponding native peer so that it can be implemented on particular platforms. They also have a number of attributes that can be summarised as follows:

- a graphical image;

- background colour;
- a location;
- actual size;
- minimum, maximum and prefered sizes.
- font metrics (discussed in the section on text);
- a parent container (see below);

As the names suggests, the container class provides a means of "grouping" multiple components. For instance, an applet may *contain* a number of buttons. Components can be added to a container. This can be thought of as a list. The order of the list determine the front to back order in which the components are presented on the screen. This is important is one component is not to obscure another. If no index is specified when adding a component to a container, it will be added to the end of the list which represents the bottom of the stacking order. There are a number of different subclasses to Container.



Note that a Frame can have a menubar but that an Applet may not. A window can have no menu or border and so Frames and Dialogs are used more frequently. For instance, the following code creates a Frame with the title "Warning". The size of the frame is defined in terms of two constants (width and height), these can be thought of as pixels.

```
/*
* A frame
*
* Author: Chris Johnson (johnson@dcs.gla.ac.uk)
* Last revision date: 11/10/98
*
* Produces a warning window on the screen
*
* Beware - there is no way of closing the frame!
* see later section on event handling...
*/

import java.awt.*;

public class SimpleWarningFrame extends Frame {

static private final int frame_height = 150;
static private final int frame_width = 250;
```

```
public SimpleWarningFrame () {
      setBackground(Color.red);
      setForeground(Color.black);
            setTitle("Warning");
      resize(frame_width, frame_height);
}

  public static void main (String[] args){

      Frame f= new SimpleWarning();
      f.show();
  }
}
```

Containers simply provide a grouping mechanism for interface objects. LayoutManagers provide means of positioning and sizing these objects. This class will be discussed in later sections.

Frames

Java's Abstract Windowing Toolkit provides windows containers that allow we to create separate windows for our applications. When used with a Web browser, they can run outside the main window (unlike panels which are nested within the applet window.)

Frames have a title bar and if they are created by an applet, they should be destroyed BEFORE we close the applet that created them so that we can reclaim the resources they were using. This is how we create a frame with a title:

Frame window = new Frame("This is the Frames's Title Bar!");

There are several things we must be concerned with after a frame is created in order for it to be visible on the screen:

- Choosing a layout manager.
- Resizing the frame: Unlike a panel, a frame must be given a size before we can see it.
- window.resize(300,200);
- Making the frame visible
- window.show();

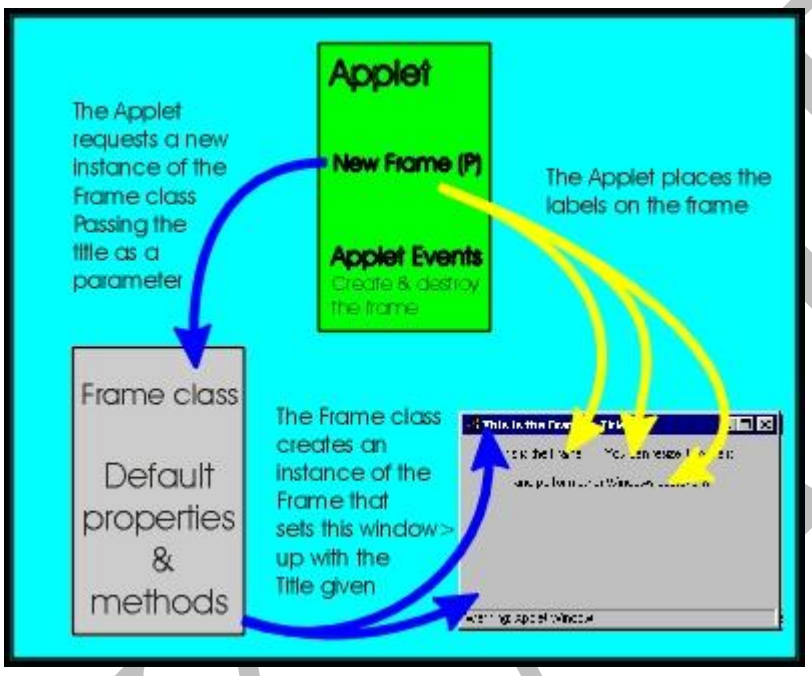
If we want to hide the frame again, in order to show a different frame for example, use the hide method.

window.hide();

Once a frame has been created and the show method has been called, it can be resized, maximized, and minimized just like any other window. When we are finished with the frame, we should always use the dispose method to get rid of it.

window.dispose();

Notice that the frame can be minimized, resized or maximized but not closed by clicking on the X button or control icon on the top left and right of the frame. The applet's event handling routines cannot detect or deal with events that occur to the frame. This stems from the object oriented nature of Java.



The new frame inherits handlers for the Maximize, Minimize and Resize events from the Frame class but no others. Any other frame events or actions would need to be handled by a class that extends the Frame class. We will write this type of class in the next section.

The code for the Applet is listed below

import java.applet.*; import java.awt.*; public class frames extends Applet { Frame window = new Frame("This is the Frame's Title Bar!");	Import all the facilities of the AWT and applet that Java has to offer. Create an applet called frames. Create an instance of the frame class, initializing the title bar. Create an instance of the button class with a

Button btn = new Button("Create a new Frame");	label.

public void init()
{
add(new Label("Hit this button to"));
add(btn);
add(new Label("."));
add(new Label("The new Frame is independent of the applet."));
add(new Label("You can maximize and minimize it by using"));
add(new Label("the buttons on the top right or the control icon."));
add(new Label("on the top left. You will not be able to close it."));
add(new Label("You must use the applet's button to do that."));
add(new Label("In order to handle Frame events you need to "));
add(new Label("create a separate class for it."));
window.setLayout(new FlowLayout());
window.add(new Label("This is the Frame."));
window.add(new Label("You can resize it, move it"));
window.add(new Label("and perform other Windows operations."));
}
	The init method adds a label The button created above is added to the applet.
	
	Labels are added to explain the behavior of the frame.
	
	The layout for the frame named window is set for FlowLayout. The default FlowLayout is center, top to bottom. Using a layout manager for frames is ***required.***
	
	Labels are added to the newly created frame.

public boolean action(Event evt, Object whatAction)
{
 	
if((evt.target instanceof Button))
{
String buttonLabel = (String) whatAction;
if (buttonLabel == "Destroy the Frame")
{
window.hide();
window.dispose();
btn.setLabel("Create a new Frame");
return true;
}
if (buttonLabel == "Create a new Frame")
{
window.resize(300,200);
window.show();
btn.setLabel("Destroy the Frame");
	When an action takes place this method tests for it.
	
	If the action was an instance of Button, the string on the button is stored in buttonLabel
	
	If the string on the button is "Destroy the Frame", hide the frame named window and dispose of it. Change the label on btn to "Create a new Frame" and return true.
	
	If the string on the button is "Create a new Frame", resize to 300x200 and show the frame named window. Change the label on btn to "Destroy

return true; } } return false; } }	the Frame" and return true otherwise return false.

Possible Questions

Part – B(2 Marks)

1. List the life cycle of applet
2. What is Color class
3. What is Font class
4. What is Font Metrics class

Part – C(6 Marks)

5. Give the constructor of Color class in an applet.
6. What is the use of layout manager in container class?
7. Describe in detail FileInputStream and FileOutputStream to define byte input and output streams connected to files.
8. Write a program to read data from a text file using FileInputStream.
9. Write the execution procedure to run an Applet and implement it using a sample Java program.
10. What is the use of Graphics class in an applet?
11. Why do we need random access files? Explain their operation in detail with their constructors and methods
12. Write a Java program to append names to an already existing file.
13. Discuss in detail about the various Basic Component Classes in AWT with example
14. Sketch out the uses of Reader and Writer Classes. Describe their methods and give an example Java program.
15. Explain the various methods in an applet life cycle and describe its operation in detail.
16. How will you create a frame window in an Applet. Explain with an example program
17. Which class is used to encapsulate Fonts in Java. Describe it in detail and write an applet program to demonstrate the use of fonts.

18. What are the various Container Classes in Java. Explain each with their constructors and methods

Questions	opt1	opt2	opt3	opt4	answer
The concept of reading and writing data as _____ of either bytes or	stream	file	java.io	reader	stream
Java also uses the _____ class to manipulate files	stream	File	String	Array	File
To support input and output package _____is used	java.util	java.awt	java.lang	java.io	java.io
_____ support 8_bit input and output operations	ByteStreams	InputStream	OutputStream	Writer	ByteStreams
_____ support 16_bit Unicode character input and output	ByteStreams	InputStream	OutputStream	Character streams	Character streams
Streams can be chained with _____ to provide enhanced	DataInput	DataOutput	filters	serializable	filters
_____ class in java does not specify how information is retrieved from or	stream	File	String	Array	File
The _____ class also defines platform_dependent constants that can be	stream	File	java.io	reader	File
The method to check for directory is _____	isFile()	isDirectory()	File	String	isDirectory()
_____ returns file size in bytes	long float()	long length()	boolean delete()	boolean mkdir()	long length()
_____ class defines Java's model of streaming byte input	ByteStreams	InputStream	OutputStream	Character streams	InputStream
InputStream suports certain methods, all of which throw an IOException on error	ByteStreams	InputStream	OutputStream	Character streams	InputStream
The _____ class define byte input streams that are connected to files	InputStream	OutputStream	FileInputStream	FileOutputStream	FileInputStream
The _____ class define byte output streams that are connected to files	InputStream	OutputStream	FileInputStream	FileOutputStream	FileOutputStream
The FileInputStream class provides an implementation for the _____	read()	write()	update()	replace()	read()
The FileOutputStream class provides an implementation for the _____	read()	write()	update()	replace()	write()
_____ is an implementation of an input stream that uses a byte array as the	InputStream	OutputStream	ByteArrayInputStream	ByteArrayOutputStream	ByteArrayInputStream
_____ is an implementation of an output stream that uses a byte array as the	InputStream	OutputStream	ByteArrayInputStream	ByteArrayOutputStream	ByteArrayOutputStream
Methods of DataOutputStream for writing are named _____	readX()	writeX()	updateX()	replaceX()	writeX()
DataOutputStream classes implement _____ interfaces	InputStream	DataOutput	OutputStream	DataInput	DataOutput
DataInputStream classes implement _____ interfaces	InputStream	DataOutput	OutputStream	DataInput	DataInput
The method _____ is used to write string value	readChars()	writeChars()	read()	write()	writeChars()
The _____ class provides a buffered stream of input	DataInputStream	DataOutputStream	BufferedInputStream	BufferedOutputStream	BufferedInputStream
The _____ class maintains a buffer that is written to when you write to the	DataInputStream	DataOutputStream	BufferedInputStream	BufferedOutputStream	BufferedOutputStream

The _____ class is designed primarily for printing output data as text	print	primtln	PrintStream	write	PrintStream
The method provided by the Reader class is _____	skip()	write()	flush()	writeX()	skip()
The method provided by the Writer class is _____	read()	flush()	reset()	skip()	flush()
_____ some input implies reducing it to a simpler stream of tokens	length	tokenizing	Stream	Exception	tokenizing
DataInput is _____	an abstract class	used to read primitive data	an interface that defines method	an interface that defines method	an interface that defines method
Which of the following statements are valid?	new DataInputStream	new DataInputStream	new DataInputStream	new DataInputStream	new DataInputStream
_____ are small applicationsthat are accessed on an internet server	utilities	networks	applets	bean	applets
The compiled applet is tested using _____	word	dos	notepad	applet viewer	applet viewer
The _____ tag is used to start an applet from both HTML and JDK applet	Html	JDK	applet	title	applet
_____ method gets called first	paint	start	init	update	init
Applet basically is a Java class defined in the _____ package of JDK	java.awt	java.lang	java.applet	java.util	java.applet
The Applet class which is in the java.applet package inherits the	Container	Componenet	Panel	List	Panel
The Panel class inherits the properties of the _____ class in the java.awt	Container	Componenet	Panel	List	Container
The container class inherits the properties of the _____ class	Container	Componenet	Panel	List	Componenet
An _____ is a window based event driven program	Html	JDK	applet	title	applet
The _____ and _____ method executes only once	stop() and destroy()	start() and stop()	init() and paint()	init() and destroy()	init() and destroy()
Immediately after calling init() methodthe browser calls the	stop()	start()	init()	destroy()	start()
The _____ method also called when the user returns to an HTML page that	paint()	init()	destroy()	start()	start()
The _____ methodis called each time your applet's output is redrawn	stop()	start()	init()	paint()	paint()
The _____ method acalled when the user moves from the HTML page that	paint()	init()	stop()	destroy()	stop()
The _____ method that is used to release additional resource	paint()	init()	destroy()	start()	destroy()
There are _____ main methods defined in java.awt.Component	2	4	5	3	3
The _____ method is defined by the AWT and is usually called by the applet	paint()	init()	stop()	repaint()	repaint()
_____ class cannot be created directly by using constructors	Panel	Container	Componenet	Graphics	Grapahics
In java color is encapsulated by the _____ class	Container	Componenet	Graphics	Color	Color

Color class also defines _____ common colors as constants	10	13	12	14	13
Methods of _____ class can also be used in the Graphiocs class methods to	Container	Componenet	Panel	List	Componenet
There are _____ common terms that are used when describing fonts	2	4	5	3	5
The java.applet package defines _____ inetrfaces	2	4	5	3	3
The user cannot have their HTML document,applet code,data and web	2	4	5	3	4
The loop() method plays the audio clip automatically while _____ plays it	paint()	play()	init()	start()	play()
The audio clip can be stopped by calling the _____ method	paint()	init()	stop()	repaint()	stop()
The _____ interface provides the inter_communication between an applet	AppletContext	AppletStub	getApplet	showDocument	AppletStub
The _____ inetface gives the information about the applet's execution	AppletStub	getApplet	AppletContext	showDocument	AppletContext
The setBackground() is the part of the class _____	Graphics	AppletStub	Component	Container	Component
If you want to assign a vlaue 99 to a variable called number, which of the	number=99	param = number value=99	param name = number value=99	param number =99	param name = number value=99

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

INFORMATION TECHNOLOGY
Second Semester
FIRST INTERNAL EXAMINATION - December 2019

PROGRAMMING IN JAVA

Class & Section: I B.Sc IT **Duration: 2 hours**
Date & Session: 18.12.19 (FN) **Maximum marks: 50 marks**
Sub.Code: I9ITU201

PART- A (20 * 1= 20 Marks)
Answer ALL the Questions

1. Java is a _____ language
   a. structured programming c. procedural oriented
   b. object oriented d. machine
2. OOPS follows_____ approach in program design
   a. bottom_up b. top_down c. middle d. top
3. Objects take up _____in the memory
   a. Space b. Address c. Memory d. bytes
4. _____is a collection of objects of similar type
   a. Objects b. methods c. classes d. messages
5. The wrapping up of data & function into a single unit is known as _____
   a. Polymorphism c. functions
   b. encapsulation d. data members
6. _____refers to the act of representing essential features without including the background details or explanations
   a. Encapsulation b. inheritance c. Dynamic binding d. Abstraction
7. Attributes are sometimes called_____
   a. data members b. methods c. messages d. functions
8. The functions operate on the datas are called_____
   a. Methods b. data members c. messages d. classes
9. _____is the process by which objects of one class acquire the properties of objects of another class
   a. Polymorphism b. encapsulation c. data binding d. Inheritance
10. Class is a _____Construct
    a. Hierarchical b. Logical c. Physical d. Hybrid
11. To access instance variables of an object_____operator is used
    a. Dot Operator c. Relational Operator
    b. Logical operator d. Boolean Operator
12. Variables declared as static are_____variables
    a. Member variables b. Instance c. class d. Local

13. It takes no parameters
    a. Default Constructors
    b. Copy Constructors
    c. Parameter Constructor
    d. Function
14. It is required when objects are required to perform a similar task
    a. Method Overriding
    b. Polymorphism
    c. Static Binding
    d. Method Overloading
15. It is used to refer to the current object
    a. this reference b. that reference c. dot d. Arrow
16. The data or variables, defined within a class are called
    a. Variables
    b. Class variables
    c. Data variables
    d. Instance Variable
17. The _____ operator creates a single instances of a named class and returns a reference to that object
    a. dot b. new c. super d. this
18. _____ initializes an object
    a. overloading b. constructors c. overriding d. destructor
19. A constructor that accepts no parameters is called the _____ constructor
    a. Copy b. default c. multiple d. multilevel
20. Constructors are invoked automatically when _____ are created
    a. Data b. classes c. objects d. methods

PART B (3 * 2 = 6 Marks)
Answer ALL the Questions

21. List the types of Java program
22. Define Type casting
23. Define constructor

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain in detail about the features and architecture of JAVA

(OR)

    b. Explain in detail about Object Oriented Programming concepts with example
25. a. Explain Java tokens

(OR)

    b. What is a class? Explain in detail how you will define a class with syntax and example
26. a. Write in detail about i) Instance variables ii) Instance methods
    iii) Class variables iv) Class Methods.

(OR)

    b. Explain the operators available in java with neat example for each.

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

INFORMATION TECHNOLOGY
Second Semester
FIRST INTERNAL EXAMINATION - December 2019

PROGRAMMING IN JAVA

Class & Section: I B.Sc IT **Duration: 2 hours**
Date & Session: 18.12.19 (FN) **Maximum marks: 50 marks**
Sub.Code: I9ITU201

PART- A (20 * 1= 20 Marks)
Answer ALL the Questions

1. Java is a _____ language
   a. object oriented
2. OOPS follows_____ approach in program design
   a. bottom_up
3. Objects take up _____in the memory
   a. Space
4. _____is a collection of objects of similar type
   a. classes
5. The wrapping up of data & function into a single unit is known as _____
   a. encapsulation
6. _____refers to the act of representing essential features without including the background details or explanations
   a. Abstraction
7. Attributes are sometimes called_____
   a. data members
8. The functions operate on the datas are called_____
   a. Methods
9. _____is the process by which objects of one class acquire the properties of objects of another class
   a. Inheritance
10. Class is a _____Construct
    a. Logical
11. To access instance variables of an object_____operator is used
    a. Dot Operator
12. Variables declared as static are_____variables
    a. Member variables b. Instance **c. class** d. Local
13. It takes no parameters
    a. Default Constructors
14. It is required when objects are required to perform a similar task

a. Method Overloading

15. It is used to refer to the current object
    a. this reference
16. The data or variables, defined within a class are called
    a. Instance Variable
17. The _____ operator creates a single instances of a named class and returns a reference to that object
    a. new
18. _____ initializes an object
    a. **b. constructors**
19. A constructor that accepts no parameters is called the _____ constructor
    a. default
20. Constructors are invoked automatically when _____ are created
    a. objects

PART B (3 * 2 = 6 Marks)
Answer ALL the Questions

21. List the types of Java program
    1. Application Program
    2. Applet Program
22. Define Type casting

    When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.
23. Define constructor

    A **java constructor** has the same name as the name of the class to which it belongs. Constructor's syntax does not include a return type, since constructors never return a value. Constructors may include parameters of various types. When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition.

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain in detail about the features and architecture of JAVA

Features of Java

Java changes the passive nature of the Internet and World Wide Web by enabling architecturally neutral code to be dynamically loaded and run on a heterogeneous network of machines. It is also a leading programming language for wireless technology and real-time systems. ·

Sun Microsystems officially describes Java as a programming language with the following attributes:

- Compiled and Interpreted
- Platform independent and Portable
- Object oriented
- Robust and Secure
- Distributed
- Multithreaded
- Dynamic

Compiled and Interpreted

Java is both a compiled and an interpreted language. Java translates source code into bytecode instructions. Java interpreter generates machine code that can directly be executed by the particular machine that is running the Java program.

Platform Independent and Portable

Java programs once written can be run anywhere anytime .Java's portability is one of the major reasons for its popularity. A program written in Java can easily be moved from one computer system to another. The Java programmer need not make any alterations in the code for using it on a computer having a different operating system, processor and system resources.

This feature has made Java a popular language for the Internet.

Object Oriented

Java is clean, usable, pragmatic approach to object orientation. The object model in java is simple and easy to extend, while simple types, such as integers are kept as high performance non objects.

Robust and Secure

The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably on a variety of systems. Accordingly, 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 life cycle.

Further, it also checks your code at runtime. ln fact, many hard-to-track-down bugs that often tum up in hard-to-reproduce runtime situations arc simply impossible to occur in Java.

For a language that is widely used for programming on the Internet, security becomes a crucial issue. Java systems safeguard the memory by ensuring that no viruses are communicated with an applet. As there are no pointers in Java, the programs are not allowed to gain access to memory locations without proper authorization.

Distributed

Java is a distributed language; it can be used for creating applications that can be run on networks. It can share both data and programs and Java applications can easily access remote objects on Internet.

Multithreaded

The word Multithreaded implies handling multiple tasks simultaneously. Java supports multithreaded programs. i.e. the user need not wait for the application to execute one task completely before starting the other. For example. one can Listen 10 sound clip while browsing a page and at the same time download an applet from a remote computer.

A multithreaded application can have several threads of execution running independently and simultaneously. These threads may communicate and cooperate and will appear to be a single stream to the user.

Dynamic

Java was designed to adapt in a constantly evolving environment It is capable of incorporating new functionality whether it comes from local system, local network or the Internet. Java dynamically links new class libraries and methods at runtime. This gives Java programs a high level of flexibility during execution.

(OR)

b. Explain in detail about Object Oriented Programming concepts with example

Object Oriented Paradigm and Concepts

1) **Object**

In object oriented programming, the object is the basic unit; the focus is mainly on data and behaviors. The purpose of object oriented programming is to combine data and behavior into a package, just as objects in the real world do.

2) **Class**

Classes are the base-structures or blueprints or templates from which objects are created. These structures define all the properties and behavior an object will possess.

3) **Data and Behavior**

In OOP, the properties used to describe an object are known as data. Data generally defines how an object looks like.

The behaviors are implemented as functions called methods.

For example, Mobile Phone

Data defines size, color, screen size of the mobile phone whereas the behavior describes making calls, sending messages and taking pictures etc.

These data and methods combined together into single, self contained unit called object.

4) **Abstraction**

Abstraction enables us to focus only on essential and ignore the non-essential. I other words exposing only the necessary details and ignore the unnecessary.

For example,

1) To drive a car it is not mandatory that one has to be aware of internal workings of a car engine
2) Coimbatore to Salem, what's the route map.
   Coimbatore→ Avinashi→ Perundurai→Salem. Only the major towns are focused and the small villages, houses, trees in between them are ignored.

5) **Encapsulation**

Capsules may be used when more mixes of sensitive drugs needs to be taken, but those drugs can't be viewed from outside world.Similarly encapsulation or information hiding permits objects to operate as complete independent, self contained package of data and methods. It hides the data and method implementation from the outside world.

6) **Inheritance**

Inheritance allows the new class to automatically inherit the data and methods of another class. It also allows adding new data and methods to the inherited ones. This dynamically increases the proficiency.

7) **Message Passing**

Communication among the objects can be made through message passing, any object can send message to any other object.

8) **Polymorphism**

Polymorphism is a feature that allows one interface to be used for a general class of actions. For example, a single button of a mobile phone is used to call, take pictures, send messages etc.

Polymorphism achieves extensibility.

25. a. Explain Java tokens

Java Tokens

A token is the smallest element in a program that is meaningful to the compiler. These tokens define the structure of the language. The Java token set can be divided into five categories: Identifiers, Keywords, Literals, Operators, and Separators.

1. Identifiers

Identifiers are names provided by you. These can be assigned to variables, methods, functions, classes etc. to uniquely identify them to the compiler.

2. Keywords

Keywords are reserved words that have a specific meaning for the compiler. They cannot be used as identifiers. Java has a rich set of keywords. Some examples are: boolean, char, if, protected, new, this, try, catch, null, threadsafe etc.

3. Literals

Literals are variables whose values remain constant throughout the program. They are also called Constants. Literals can be of four types. They are:

a. String Literals

String Literals are always enclosed in double quotes and are implemented using the java.lang.String class. Enclosing a character string within double quotes will automatically create a new String object. For example,`String s = "this is a string";`. String objects are immutable, which means that once created, their values cannot be changed.

b. Character Literals

These are enclosed in single quotes and contain only one character.

c. Boolean Literals

They can only have the values `true` or `false`. These values do not correspond to 1 or 0 as in C or C++.

d. Numeric Literals

Numeric Literals can contain integer or floating point values.

4. Operators

An operator is a symbol that operates on one or more operands to produce a result.

5. Separators

Separators are symbols that indicate the division and arrangement of groups of code. The structure and function of code is generally defined by the separators. The separators used in Java are as follows:

parentheses ( )

Used to define precedence in expressions, to enclose parameters in method definitions, and enclosing cast types.

braces { }

Used to define a block of code and to hold the values of arrays.

brackets [ ]

Used to declare array types.

semicolon ;

Used to separate statements.

comma ,

Used to separate identifiers in a variable declaration and in the `for` statement.

period .

Used to separate package names from classes and subclasses and to separate a variable or a method from a reference variable.

(OR)

b. What is a class? Explain in detail how you will define a class with syntax and example

Introduction to classes

A class is a template or a prototype defines a type of object. A class is to an object what a blueprint is to a house. A class is a collection of data variables and methods that define a particular entity. A class can be either user-defined or provided by one of the built in java packages.

Defining a Class

The class is defined using a keyword **class** followed by a user defined class name. The body of the class is contained in the block that is defined by curly braces{}

```
class classname
{
        [variable declarations;]
        [method declarations;]
}
```

The data or variables defined within a classes are called instance variables. The code is contained within methods, these are also called members of the class.

For example

```
class exampleclass
{
        char cc;
        int f1;
        double dd;
        void examplemethod1()
        {
        System.out.println("Hello world");
        }
        void examplemethod2()
        {
        System.out.println("Hai World");
        }
        }
```

A class is an encapsulated collection of data, and methods to operate on data. A class definition typically includes the following

1. Access Modifier
2. The class keyword
3. Instance fields
4. Constructors
5. Instance methods
6. Class fields
7. Class method

26. a. Write in detail about i) Instance variables ii) Instance methods
iii) Class variables iv) Class Methods.

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . *ObjectReference.VariableName*.

```
import java.io.*;

public class Employee{
  // this instance variable is visible for any child class.
  public String name;

  // salary  variable is visible in Employee class only.
  private double salary;

  // The name variable is assigned in the constructor.
  public Employee (String empName){
    name = empName;
  }

  // The salary variable is assigned a value.
  public void setSalary(double empSal){
    salary = empSal;
  }

  // This method prints the employee details.
  public void printEmp(){
    System.out.println("name  : " + name );
    System.out.println("salary :" + salary);
  }

  public static void main(String args[]){
    Employee empOne = new Employee("Ransika");
    empOne.setSalary(1000);
    empOne.printEmp();
```

```
    }
}
```

Class/static variables:

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name . *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

```
import java.io.*;
public class Employee{
  // salary  variable is a private static variable
  private static double salary;

  // DEPARTMENT is a constant
  public static final String DEPARTMENT = "Development ";

  public static void main(String args[]){
    salary = 1000;
    System.out.println(DEPARTMENT+"average salary:"+salary);
  }
}
```

Instance Methods

A java method is equivalent to a function, procedure, or subroutine in other languages except that it must be defined inside a class definition. Instance methods are the foundation of encapsulation and provide a consistent interface to the class.

Adding methods to the class

Methods are declared inside the body of the class but immediately after the declaration of the instance and class variables. The general form of a method declaration is

```
returntype methodname(parameter_list)
```

{
Method body;
}
A returntype can be a primitive type such as int, or a class type such as string or void.

A methodname begin with a lowercase letter and according to java convention, compund words in the method name should begin with uppercase letters.

The method body must be enclosed in curly braces.

An optional parameter_list/argument_list must be inside parenthesis, seperated by commas.
For example

```
String gettitle()
        {
                return title;
        }
void printdetails()
        {
                System.out.println("Title is:"+title);
        }
```

(OR)

b. Explain the operators available in java with neat example for each.

Java Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:
Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2

%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

The Relational Operators:

There are following relational operators supported by Java language
Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.
Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100
b = 0000 1101
---
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

The following table lists the bitwise operators:
Assume integer variable A holds 60 and variable B holds 13 then:
Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators:

There are following assignment operators supported by Java language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the	C /= A is equivalent to C = C / A

	result to left operand	
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {

   public static void main(String args[]){
      int a , b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

INFORMATION TECHNOLOGY
Second Semester
SECOND INTERNAL EXAMINATION - February 2020

PROGRAMMING IN JAVA

Class & Section: I B.Sc IT **Duration: 2 hours**
Date & Session: 4.2.2020 (FN) **Maximum marks: 50 marks**
Sub.Code: I9ITU201

PART- A (20 * 1= 20 Marks)
Answer ALL the Questions

1. A method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to____ the method in the superclass
   a. override b. overload c. function d. final
2. _____ dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
   a. Static method c. overload
   b. Dynamic method d. finalized
3. Once you have an object, you can call its methods and access its fields, by using the _____
   a. object reference c. variables
   b. class d. data types
4. Which of these keywords is used to define interfaces in Java?
   a. interface b. Interface c. intf d. Intf
5. Which of these can be used to fully abstract a class from its implementation?
   a. Objects b. Packages c. Interfaces d. class
6. Which of these keywords is used by a class to use an interface defined previously?
   a. import b. Import c. implements d. Implements
7. Runnable is a _____ .
   a. class b. abstract class c. interface d. variable
8. _____ act as containers for classes and other packages.
   a. Container b. Classes c. Java d. Packages
9. An _____ is a condition that is caused by a runtime error in the program
   a. throw b. exception c. handle d. catch
10. Exception can be generated by the _____ or manually by the code
    a. Throwable class c. object
    b. Java runtime system d. catch
11. When an exception occurs within a java method, the method creates an exception object and hands it over to the runtime systewm is called ____
    a. catching the exception c. handle the exception
    b. throwing an exception d. get the exception

12. When java method throws an exception the java runtime system searches all the methods in the call stack to find one that can handle this type of exception is known as _____
    a. catching the exception
    b. throwing an exception
    c. handle the exception
    d. get the exception
13. Unchecked exceptions are extensions of _____
    a. throws
    b. catch
    c. RuntimeException
    d. Error
14. Checked exceptions are extensions of _____
    a. throws
    b. catch
    c. Exception
    d. Error
15. Which method is used in thread class to tests if the current thread has been interrupted?
    a. public static boolean interrupted()
    b. public boolean isInterrupted()
    c. public void interrupt()
    d. public boolean isAlive()
16. Which method in thread class causes the currently executing thread object to temporarily pause and allow other threads to execute?
    a. public boolean isAlive()
    b. public int getId()
    c. public void yield()
    d. public boolean isDaemon()
17. How many methods does a thread class provides for sleeping a thread?
    a. 3
    b. 1
    c. 4
    d. 2
18. Which method waits for a thread to die?
    a. stop()
    b. start()
    c. terminate()
    d. join()
19. In Naming a thread which method is used to change the name of a thread?
    a. public String getName()
    b. public void setName(String name)
    c. public void getName()
    d. public String setName(String name)
20. Default priority value of a thread class for NORM_PRIORITY is?
    a. 1
    b. 10
    c. 5
    d. 4

PART B (3 * 2 = 6 Marks)
Answer ALL the Questions

21. What is Inheritance?
22. Define Multithreading
23. Define Package

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. What is Inheritance? Describe the various forms of inheritance in Java.

(OR)

    b. Why is it necessary to implement an interface? Give its syntax and explain with example.
25. a. Write short note on Abstract class and methods with example.

(OR)

    b. Write short note super reference with example.
26. a. What is package? Discuss about the creation and importing package with example.

(OR)

    b. Explain the “try-catch” construct used to capture and handle exception with an example program

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

INFORMATION TECHNOLOGY
Second Semester
SECOND INTERNAL EXAMINATION - February 2020

PROGRAMMING IN JAVA

Class & Section: I B.Sc IT **Duration: 2 hours**
Date & Session: 4.2.2020 (FN) **Maximum marks: 50 marks**
Sub.Code: I9ITU201

PART- A (20 * 1= 20 Marks)
Answer ALL the Questions

1. A method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to____ the method in the superclass
   override
2. _____ dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
   Dynamic method
3. Once you have an object, you can call its methods and access its fields, by using the _____
   object reference a.
4. Which of these keywords is used to define interfaces in Java?
   interface
5. Which of these can be used to fully abstract a class from its implementation?
   Interfaces
6. Which of these keywords is used by a class to use an interface defined previously?
   implements
7. Runnable is a _____ .
   interface
8. _____ act as containers for classes and other packages.
   Packages
9. An _____ is a condition that is caused by a runtime error in the program
   exception
10. Exception can be generated by the _____ or manually by the code
    Java runtime system
11. When an exception occurs within a java method, the method creates an exception object and hands it over to the runtime systewm is called ____
    throwing an exception
12. When java method throws an exception the java runtime system searches all the methods in the call stack to find one that can handle this type of exception is known as _____
    catching the exception
13. Unchecked exceptions are extensions of _____

RuntimeException

14. Checked exceptions are extensions of _____
    Exception
15. Which method is used in thread class to tests if the current thread has been interrupted?
    public static boolean interrupted()
16. Which method in thread class causes the currently executing thread object to temporarily pause and allow other threads to execute?
    public void yield()
17. How many methods does a thread class provides for sleeping a thread?
    2
18. Which method waits for a thread to die?
    join()
19. In Naming a thread which method is used to change the name of a thread?
    public void setName(String name)
20. Default priority value of a thread class for NORM_PRIORITY is?
    5

PART B (3 * 2 = 6 Marks)
Answer ALL the Questions

21. What is Inheritance?
    Inheritance provided a mechanism that allowed a class to inherit property of another class. When a class extends another class it inherits all non private members including fields and methods. Inheritance in java can be best understood in terms of parent and child relationship, also known as super class(parent) and sub class(child).
22. Define Multithreading
    Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
23. Define Package
    Programs are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts.

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. What is Inheritance? Describe the various forms of inheritance in Java.

Inheritance is one of the key features of object oriented programming. Inheritance provided a mechanism that allowed a class to inherit property of another class. When a class extends another class it inherits all non private members including fields and methods. Inheritance in java can be best understood in terms of parent and child relationship, also known as super class(parent) and sub class(child).

extends and implements keywords are used in inheritance in java.

<u>Purpose of Inheritance</u>

1. To promote code reuse

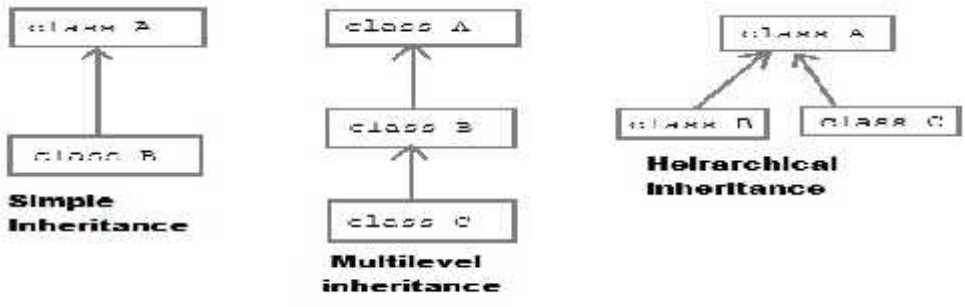
2. To use polymorphism

For example

```
class Box {

        double width;
        double height;
        double depth;
        Box() {
        }
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        void getVolume() {
                System.out.println("Volume is : " + width * height * depth);
        }
}

public class MatchBox extends Box {

        double weight;
        MatchBox() {
        }
        MatchBox(double w, double h, double d, double m) {
                super(w, h, d);
                weight = m;
        }
        public static void main(String args[]) {
                MatchBox mb1 = new MatchBox(10, 10, 10, 10);
                mb1.getVolume();
                System.out.println("width of MatchBox 1 is " + mb1.width);
                System.out.println("height of MatchBox 1 is " + mb1.height);
                System.out.println("depth of MatchBox 1 is " + mb1.depth);
                System.out.println("weight of MatchBox 1 is " + mb1.weight);
        }
}
```

Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance



(OR)

b. Why is it necessary to implement an interface? Give its syntax and explain with example.

Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final). Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated.

Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.

```
interface Shape {

        public double area();
        public double volume();
}
```

Below is a Point class that implements the Shape interface.

```
public class Point implements Shape {

        static int x, y;
        public Point() {
                x = 0;
                y = 0;
        }
        public double area() {
                return 0;
        }
        public double volume() {
                return 0;
        }
        public static void print() {
                System.out.println("point: " + x + "," + y);
        }
        public static void main(String args[]) {
                Point p = new Point();
                p.print();
        }
```

```
}
```

25. a. Write short note on Abstract class and methods with example.

Java Abstract classes are used to declare common characteristics of subclasses. An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class. Abstract classes are declared with the abstract keyword.

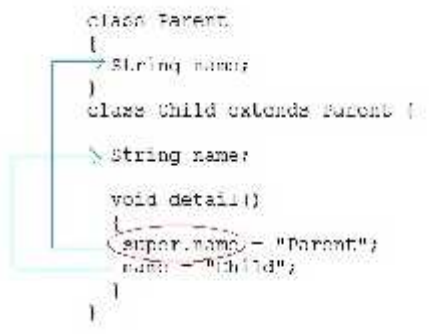
Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods.

```
abstract class Shape {
public String color;
public Shape() {
}
public void setColor(String c) {
        color = c;
}
public String getColor() {
        return color;
}
abstract public double area();
}
```

(OR)

b. Write short note super reference with example.

In java, super keyword is used to refer to immediate parent class of a class. In other words, super keyword is used by a subclass whenever it need to refer to its immediate super class




```
class Vehicle {

    // Instance fields
    int noOfTyres; // no of tyres
    private boolean accessories; // check if accessorees present or not
    protected String brand; // Brand of the car
    // Static fields
    private static int counter; // No of Vehicle objects created
    // Constructor
    Vehicle() {
        System.out.println("Constructor of the Super class called");
        noOfTyres = 5;
```

```
                accessories = true;
                brand = "X";
                counter++;
        }
        // Instance methods
        public void switchOn() {
                accessories = true;
        }
        public void switchOff() {
                accessories = false;
        }
        public boolean isPresent() {
                return accessories;
        }
        private void getBrand() {
                System.out.println("Vehicle Brand: " + brand);
        }
        // Static methods
        public static void getNoOfVehicles() {
                System.out.println("Number of Vehicles: " + counter);
        }
}

class Car extends Vehicle {

        private int carNo = 10;
        public void printCarInfo() {
                System.out.println("Car number: " + carNo);
                System.out.println("No of Tyres: " + noOfTyres); // Inherited.
                //  System.out.println("accessories: "    + accessories); // Not Inherited.
                System.out.println("accessories: " + isPresent()); // Inherited.
                //        System.out.println("Brand: "     + getBrand());  // Not Inherited.
                System.out.println("Brand: " + brand); // Inherited.
                //  System.out.println("Counter: "    + counter);     // Not Inherited.
                getNoOfVehicles(); // Inherited.
        }
}

public class VehicleDetails { // (3)

        public static void main(String[] args) {
                new Car().printCarInfo();
        }
}
```

26. a. What is package? Discuss about the creation and importing package with example.

Programs are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts.

The members of a package are subpackages and all the top level class and top level interface types declared in all the compilation units of the package.

Package Declarations

A package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs.

Named Packages

A package declaration in a compilation unit specifies the name of the package to which the compilation unit belongs.

PackageDeclaration:
package PackageName ;

The package name mentioned in a package declaration must be the fully qualified name of the package.

Importing a Package Member

To import a specific member into the current file, put an `import` statement at the beginning of the file before any type definitions but after the `package` statement, if there is one.

```
import graphics.Rectangle;
```

Now you can refer to the `Rectangle` class by its simple name.

```
Rectangle myRectangle = new Rectangle();
```

This approach works well if you use just a few members from the `graphics` package. But if you use many types from a package, you should import the entire package.

Importing an Entire Package

To import all the types contained in a particular package, use the `import` statement with the asterisk `(*)` wildcard character.

```
import graphics.*;
```

(OR)

b. Explain the "try-catch" construct used to capture and handle exception with an example program

- An *exception* is an abnormal condition that arises in a code sequence at run time
- A Java exception is an object that describes an exceptional 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
- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code
- Java exception handling is managed by via five keywords: **try, catch, throw, throws,** and **finally**
- Program statements to monitor are contained within a **try** block

- If an exception occurs within the **try** block, it is thrown
- Code within **catch** block catch the exception and handle it
- System generated exceptions are automatically thrown by the Java run-time 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
- 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

}
```

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```